PAL Worksheet The Stack Class CSC20

Topics: Stack Class

The stack and queue are some of the simplest collections in Java. They are easy to implement and widely used in all kinds of applications. The Stack is a class from the Collection framework and implements the List Interface. The Queue Interface inherits from the Collection Interface and may be implemented as a LinkedList.

Let's go over some of the basic design and operations of a stack and queue first.

We can think of a stack like a vertical laundry basket, in which you pile dirty clothes one on top of another. When the basket is full and needs to be emptied, we take out the topmost item first, followed by the one underneath, and so on until we reach the item on the very bottom, which had been thrown in before all the others.

We call this design **'Last In First Out'** (LIFO) because the item which was dropped in last, came out first as the top of the stack. We add items to a stack via a push() method and remove the most recently pushed item via pop(). Remember that once we pop() an item, it's removed from the stack. If we wish to just see the first item at the top of the stack, we can use peek() which accesses the first item without removal. To check the size of a stack, we use size() and isEmpty(), which returns a boolean.

Let's see some of these in action:

```
Stack<Integer> myStack = new Stack<Integer>();

// add items to the stack
myStack.push(2);
myStack.push(4);
myStack.push(6);

// retrieve most recently added item
Integer topItem = myStack.pop() // this has value 6

// peek at top of stack without removing it
Integer nextItem = myStack.peek() // this has value 4
```

By contrast, a queue is like a stack in which items are retrieved from the bottom of the pile first. Its design is called ‘First In First Out’ (FIFO). The analogy of a queue is an actual queue at a restaurant which serves customers on a first-come, first-serve basis. The customers who entered the queue first also leave the queue first.

Where are stacks useful? Think of your email, where you end up replying to the most recent emails first simply because they’re at the top of the stack, and not the ones which are down below and arrived earlier. Stacks are also a staple feature in operating systems.

In this worksheet, we will go over Stack features and Stack exercises. The next worksheet will cover Queue features, exercises and their close relationship with the Stack data structure.

Let’s now try a common problem which illustrates how it can be helpful to use a stack.

Write a method which when given an input string, uses a stack to determine if it is a palindrome, i.e. is the same even when read in reverse. An example would be ‘racecar’. Here is how we could solve it with a stack.

```
public static boolean palindrome(String s) {

     Stack<Character> stack = new Stack<>();
     char[] charArray = s.toCharArray();

     // push all characters onto stack
     for (char c : charArray) {
          stack.push(c);
     }

     // pop all off stack and compare with original string
     for (char c : charArray) {
          char top = stack.pop();

          // if not the same, return false
          if (top != c) {
               return false;
          }
     }

     // else return true
     return true;
}
```

The main idea here is that after pushing (a string) onto a stack, the stack will return it in *reverse* order. And that is precisely what we want to do when checking for a palindrome – compare the original string against itself, reversed!

For example, given 'bald', we wish to compare 'bald' with 'dlab', character by character until we find a mismatch. If there is a mismatch, we can declare the string not a palindrome. But if we get through the entire comparison, then all characters are the same and it's a palindrome.

Let's now move onto comparing stacks and queues for common operations like search, insertion, sort, and reversal.

Search

Searching a stack means using pop() or remove() to remove the top/front element repeatedly so we can examine all the items and compare to our search value. However, this depletes the collection and may leave it empty. We need to be mindful of restoring the stack to its original state after the search.

For a stack, there is no operation to simply attach a freshly removed item down at the bottom, so that it can be used for restoration. To implement stack restoration, it is advisable to use an auxiliary storage data structure like an empty temporary stack, which can store (and therefore save) the popped elements from our original stack.

And as we learned earlier, retrieving elements from the temporary stack provides them in reversed order, so when we push them back on to the original stack, they will be restored in the proper order. See below:

```
public static void stackSwap(Stack<Integer> source,
Stack<Integer> target){

     // transfer elements from source stack to target stack
     while(!source.isEmpty()){
          target.push(source.pop());
     }}
```

Q1) Write a method *findInStack* which takes in an Integer stack and an Integer value, then verifies if the value exists in the stack or not. This method returns a *boolean* result *true* if the value is found in the stack and false otherwise. Make sure the stack is restored to its original condition

Insertion

Insertion of an element into a stack would likely require another stack to help it store removed elements before returning them.

Try doing a stack insertion problem, following this theme.

Q2) Write a method *insertStack* which takes in an Integer stack, an Integer value, and a position and then inserts the value into the stack at the given position (ascending from top to bottom). You may presume that the stack positions are ordered from 1 to the size of the stack and that the input positions given are greater than 0 and less than or equal to the stack size. Use an auxiliary stack to help with this operation. This method returns void. Given a stack s =[3,6,8,1] a call to insertStack(s, 5, 2) would result in s =[3,5,6,8,1]

Reversal

As you can probably guess by now, reversal is an operation stacks are very helpful with. We have seen how a string or array can easily be reversed by pushing all elements onto a stack, then simply popping them off for the reversed sequence. Try it yourself now.

Q3) Write a method called *reverseStack* which takes in an Integer stack and returns its elements in reversed order. It returns a new Stack that holds the elements of the input stack in the reverse order. The input stack needs to be restored to its original condition. Use one auxiliary stack only and a new Stack that holds the elements in reverse order.

Sort

Sorting a stack (i.e. in ascending order from top to bottom) can be done with the help of an auxiliary stack which, from creation, is always kept ordered. In this algorithm, we pop the top element from the original stack, and before placing in the auxiliary stack, remove all elements in the auxiliary stack unless they are greater than it. Then, place it in the auxiliary stack, knowing that the auxiliary stack remains ordered. Repeat this process until the original stack is empty and the auxiliary stack contains all the elements in order. Additional auxiliary data structures like an additional stack or queue may be used as needed.

Q4) Write a method *sortStack* which takes an unordered Integer stack and returns an ordered stack in which the Integers are in ascending order, top to bottom. For example, take [6, 2, 8, 4] and return [2, 4, 6, 8]. This method has a void return type. The original stack gets sorted.