Assignment 2 Report - A Calculator in Java

Dean Tsankov

September 7, 2024

Introduction

In this report I will present the implementation of a calculator that can calculate mathematical expressions described using reverse Polish notation. This is done in order to show how a stack can be used. In this case unlike previous coding courses here I will create the stack explicitly.

I will show two ways to implement a stack:

- Static Stack - the size of the stack array is constant and cannot be changed.
- Dynamic Stack - the size of the stack array can change to accommodate the amount of elements.

Implementing the stack

A stack is a general data structure that allows two basic operations - push and pop. When an item is pushed on the stack it is placed at its top. A pop operation will remove, and return, the item at the top of the stack, if the stack is not empty. The item below the removed item takes the top spot of the stack.

The static stack

The first implementation will be a fixed sized stack where the size is given on initialization of the stack. It should allocate an array of this size and keep track of a stack pointer (an index).

During the process of coding the stack some questions I had to consider in order for it to work properly were:

- Does the pointer point to the location above the top of the stack or does it point to the top of the stack? - Points to the top of the stack itself.

- What is the value of the pointer when the stack is empty? - I chose to use the value `-1` since using `0` would still be a valid index and may yield unwanted results.
- What should you do when a program tries to push a value on a full stack (stack overflow)? - In my code this is just an invalid operation and nothing happens, but a good practice is to throw an exception and let the rest of the program know that something went wrong.
- What should happen when someone pops an item from an empty stack? - Again this is an invalid operation and the same holds as for the question above.

In the following code showcase I have already introduced a generic variable type in the implementation which allows me to use this data structure not only with integers, but with other object types as well:

```
public class StaticStack<T> extends Stack<T>{
  T[] stack;
  int top;

  public StaticStack(int size) {
      stack = (T[]) new Object[size];
      top = -1;
  }
  (...)
```

The class has two parameters a generic array, and an index which refers to the most recently added top element. In the initialization I create the array and also set the top index as `-1` since we have wet to have any elements in the stack. Later I mention more about the class extension.

```
  (...)
  public void push(T val) {
      if (top < stack.length-1) {
          top += 1;
          stack[top] = val;
      }
  }
  (...)
```

When an element is pushed I do a check to see if this element would exceed the set size of the stack. If not then the top index is updated and the value is set.

```
(...)
public T pop() {
    T val = null;
    if (top >= 0) {
        val = stack[top];
        stack[top] = null;
        top -= 1;
    }
    return val;
}
(...)
```

If we pop an element again depending on whether there still are elements in the stack the corresponding value is set to be returned, otherwise `null` gets returned. And the top index also gets updated accordingly.

An abstract class

Above also my `StaticStack` class extends an abstract class named `Stack`. This allows me to abstractly declare a Stack object and not bother a developer with the way it was implemented since this abstract class guarantees that there will be methods for push and pop operations. Below is the way the abstract class was written:

```
public abstract class Stack<T> {
  T[] stack;

  int size;
  int top;

  public abstract void push(T value);

  public abstract T pop();

  }
```

The dynamic stack

A more complex solution is required to handle a stack that can grow as we add more items. In a push operation, if we have already used all of our array that would yield an index out of range exception. What we should do is extend the size of the stack by allocating a new larger array and to copy the items from the original array to the new one. One question is how much larger the new stack should be, should we increase by only one item, a fixed

amount or some other solution? - I chose to increase the stack by a fixed amount, which is a private parameter named `stackStep`.

Here is my code for the dynamic stack implementation (it again extends the abstract Stack class and uses a generic type) :

```
public class DynamicStack<T> extends Stack<T>{
  T[] stack;
  int top;
  int size = 1;
  private int stackStep = 4;
  private int popfrequency = 0;
  public DynamicStack() {
      stack = (T[]) new Object[size];
      top = -1;
  }
  (...)
```

The initialization itself follows similar steps as the one for the static stack, with the addition of two private parameters, whose function I iterate more on below.

```
  (...)
  public void push(T val) {
      popfrequency = 0;
      top+=1;
      if (top == size) {
          T[] newStack = (T[]) new Object[size + stackStep];
          for (int i = 0; i < size; i++) {
              newStack[i] = stack[i];
          }
          size += stackStep;
          stack = newStack;
      }
      stack[top] = val;
  }
  (...)
```

For the push I do a size comparison which dictates when a stack array size increase should happen. The algorithm for this is as explained: creating a larger array and copying the contents of the smaller one into it.

```
  (...)
  public T pop() {
      popfrequency++;
      if (popfrequency > stackStep) {
          size -= stackStep;
```

```
            T[] newStack = (T[]) new Object[size];
            for (int i = 0; i < size; i++) {
                newStack[i] = stack[i];
            }
            stack = newStack;
            popfrequency = 0;
        }
        T val = null;
        if (top >= 0) {
            val = stack[top];
            stack[top] = null;
            top -= 1;
        }
        return val;
    }
    (...)
```

The first part of the code segment deals with the dynamic aspect of the stack while the latter is almost the same as the pop of the static stack. Along with extending the stack array if there are too many items, I had to come up with a mechanism to shrink the same if there were no longer that many items. I do this by introducing the private variable `popfrequency` which keeps track of whether there have been a lot of pops happening. If these pops are more than the stack size increase step then we can apply a similar algorithm to copy the contents of the current large array into a smaller array, and thus minimize the unnecessary use of resources.

Does it work

After trying both of the stack implementations with the provided simple test code the result for both is the same:

```
pop: 31
pop: 30
...
pop: 2
pop: 1
pop: 0
```

Figure 1: Console output

The HP-35 style calculator

Implementing the `+, -, *` operations for the calculator using the provided code skeleton proved to be fairly simple.

Here I present only the code regarding the calculator operations:

```
(...)
switch (input) {
    case "+":
        stack.push(stack.pop() + stack.pop());
        break;
    case "-":
        stack.push(-(stack.pop() - stack.pop()));
        break;
    case "*":
        stack.push(stack.pop() * stack.pop());
        break;
    case "":
        run = false;
        break;
    default:
        Integer nr = Integer.parseInt(input);
        stack.push(nr);
        break;
}
(...)
```

Regarding the subtraction operation since it is the only one of the three which is not commutative i.e. the order of the operands matters I reverse the the result, because when popping the items from the stack we get them with switched places.

Conclusion

To check how well my implementation of both the `Stack` and the mathematical operations work I enter the input:

```
HP-35 pocket calculator
> 4
> 2
> 3
> *
> 4
> +
> 4
> *
> +
> 2
> -
```

and receive:

```
the result is:  42
I love reversed polish notation, don't you?
```

Which is of course *"The Answer to the Ultimate Question of Life, The Universe, and Everything"* as in reference to "The Hitchhiker's Guide to the Galaxy". In essence the calculator works correctly.