ITI 1121. Introduction to Computing II

Stack: applications

by

Marcel Turcotte

Version February 12, 2020

Preamble

Preamble

Overview

Overview

Stack: applications

We look at several examples of the use of stacks, including evaluating arithmetic expressions, saving command history, and running Java programs.

General objective :

- This week you will be able to apply the stacks for algorithm design.

Preamble

Learning objectives

Learning objectives

- **Justify** the role of a stack in solving a computer problem.
- **Design** a computer program requiring the use of a stack.

Readings:

- Pages 159-176 of E. Koffman and P. Wolfgang.

Preamble

Plan

Plan

Applications

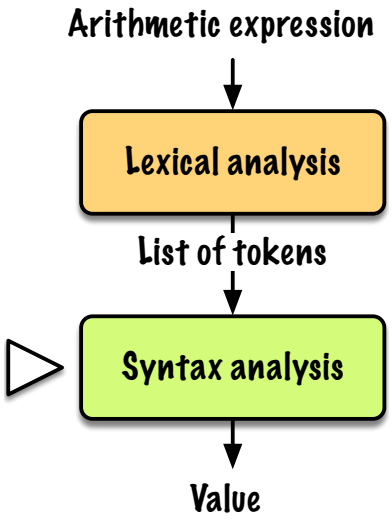
Applications

Evaluating an arithmetic expression

Application : Evaluating an arithmetic expression

Architecture of our application

- Clear separation of concerns: **lexical analysis** and **syntactic analysis**
- **Lexical analysis** takes a **string of characters** as input and cuts it into chunks called **tokens**.
  - **Input**: $\cdot 1 \cdot + \cdot \cdot 2 \times 33 \cdot \cdot \cdot - 4 \cdot$
  - **Output**: $[1, +, 2, \times, 33, -, 4]$
- Our **syntax analysis** takes a sequence of **tokens** as input and returns the **value** of the expression.
  - **Input**: $[1, +, 2, \times, 33, -, 4]$
  - **Output**: 63



StringTokenizer

Java has a **lexical analyzer**!

```
StringTokenizer st;
st = new StringTokenizer(" 1 +  2    * 33 - 4");

while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

```
1
+
2
*
33
-
4
```

StreamTokenizer is more versatile!

Scan

Please take a few minutes to analyze this example. **What do you think?**

```
public static int scan(String expression) {

    StringTokenizer st; String op; int l,r;

    st = new StringTokenizer(expression);

    l = Integer.parseInt(st.nextToken());

    while (st.hasMoreTokens()) {
        op = st.nextToken();
        r = Integer.parseInt(st.nextToken());
        l = eval(l,op,r);
    }

    return l;
}
```

```
    private static int eval(int l, String op, int r) {
        int result;
        switch (op) {
            case "+":
                result = l + r;
                break;
            case "-":
                result = l - r;
                break;
            case "/":
                result = l / r;
                break;
            case "*":
                result = l * r;
                break;
            default:
                System.exit(-1);
        }
        return result;
    }
```

Exercises

- What does **scan("3 * 12 + 4")** return?
- What does **scan("3 + 12 * 4")** return?
- What do you think?

Discussion

- The **scan** algorithm evaluates operations from left to right, regardless of the **priority** of the operations. **Scan** doesn't process the **parentheses**.

There are two solutions:

- Use a new representation for the expressions
- Use a more complex algorithm

$\Rightarrow$ Both of these solutions require the use, implicit or explicit, of a **stack**!

Representations. There are three ways to represent an expression: $l \diamond r$, where $\diamond$ is an operator.

infix: The infix notation corresponds to the usual notation, the operator is sandwiched between its operands: $l \diamond r$.

post-fixed: In post-fixed notation, the operands are placed in front of the operator, $l\ \ r\ \ \diamond$. This notation is also called *Reverse Polish Notation* or **RPN**, it is the notation used by some scientific calculators (such as HP-35 from Hewlett-Packard or Texas Instruments TI-89 using RPN Interface by Lars Frederiksen*) and the languages **PostScript** and **PDF**.

- $7 - (3 - 2) \rightarrow 7\ 3\ 2\ -\ -$
- $(7 - 3) - 2 \rightarrow 7\ 3\ -\ 2\ -$

pre-fixed The third notation is to place the operator first followed by its operands, $\diamond\ \ l\ \ r$. The programming language **Lisp** uses a combination of parentheses and prefix notation, `(- 7 (* 3 2))`.

*www.calculator.org/rpn.html

From infix to postfix

- Successively transform, **one by one**, each subexpression **following the normal order of evaluation** of an infixed expression.
- An infixed subexpression $l \diamond r$ becomes $l\ r\ \diamond$,
  where l and r are themselves subexpressions and $\diamond$ is an operator.

Evaluating a postfixed expression (mentally)

Until the end of the expression is reached:

1. Read from **left to right** up to the **first operator**;
2. **Apply** the operator to the (2) operands preceding it.;
3. **Replace** the operator and its (2) operands by the result.

When the end of the expression is reached, we have the result.

Evaluating a postfix expression (mentally)

A few **exercises**:

- $9 \quad 3 \quad / \quad 10 \quad 2 \quad 3 \quad * \quad - \quad +$
- $9 \quad 2 \quad 4 \quad * \quad 5 \quad - \quad /$

9 3 / 10 2 3 * − +

9 2 4 * 5 − /

Remarks

The **order of the operands is the same** for the two notations, postfix and infix, however the **places where the operators are inserted differ**.

- $2 + (3 * 4) \rightarrow 2 \quad 3 \quad 4 \quad * \quad +$
- $(2 + 3) * 4 \rightarrow 2 \quad 3 \quad + \quad 4 \quad *$

To evaluate an infixed expression, the **operator priority** as well as the **parentheses** must be taken into account.

- In the case of postfix notation, **these concepts are represented within the notation**.

9 3 / 10 2 3 * − +

9 2 4 * 5 − /

Exercises

- Give the **content of the stack** for each iteration of the algorithm. :
  - $9 \quad 3 \quad / \quad 10 \quad 2 \quad 3 \quad * \quad - \quad +$
  - $9 \quad 2 \quad 4 \quad * \quad 5 \quad - \quad /$
- Modify the algorithm so that it constructs an expression **infix** from an expression **postfix** given as input.

Applications

Discussion on the usefulness of abstract data types

Discussion

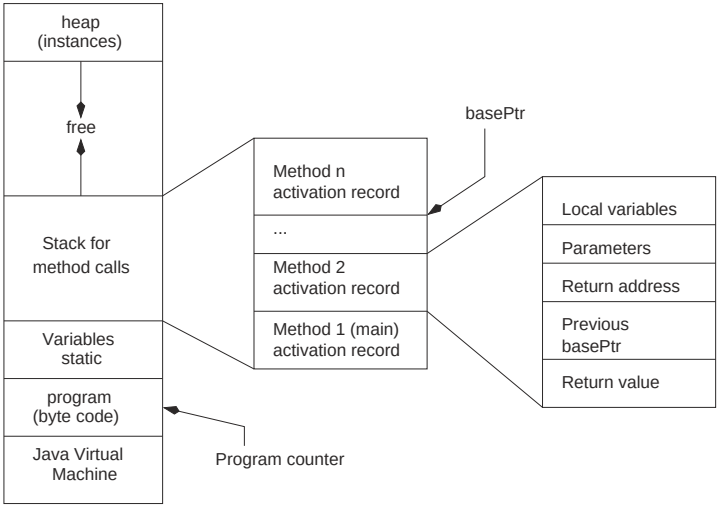
- Now please answer the question asked earlier: “One of the proposed implementations uses an array, **why don’t we just use an array** for algorithm design? **What are the advantages**? ”

Applications

Memory management

Application : Memory management during program execution

Memory representation and program interpretation



When a method is called

The Java Virtual Machine (**JVM**) must:

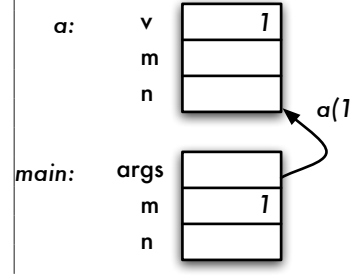
1. Create a new **activation frame** [working memory] (the return value, previous basePtr value and the return address have a fixed size, the size of local variables and parameters depends on the method);
2. **Save** the current value of basePtr, in the space "previous value of basePtr", point basePtr to the base of the current block;
3. **Save** the value of locationCounter in the space designated by "return address", make locationCounter point to the first instruction of the called method;
4. **Copy the actual parameter values in the region designated by "parameter"**;
5. Initialize the textbflocal variables;
6. **Start** execution at the instruction pointed to by locationCounter.

When the execution of a method ends

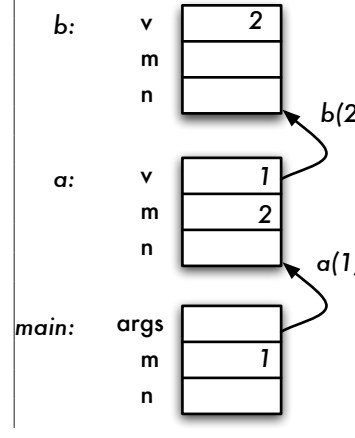
1. The method **saves the return value** at the location indicated by “return value”.;
2. **Returns control** to the calling method, i.e., resets the locationCounter and basePtr values;
3. **Removes the current activation block**;
4. **Resumes** execution at the location designated by locationCounter.

```
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```

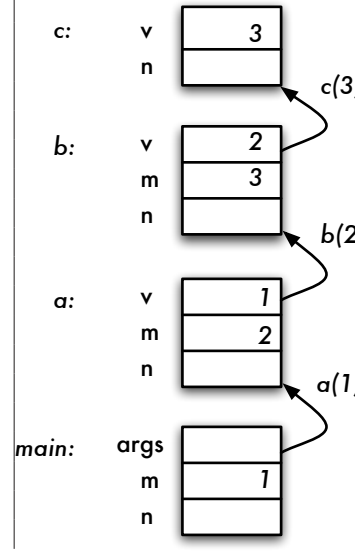
```java
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



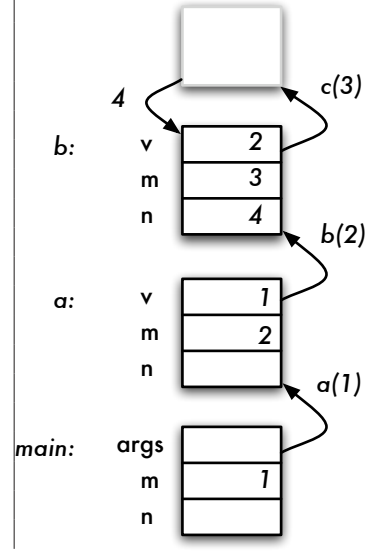
```java
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



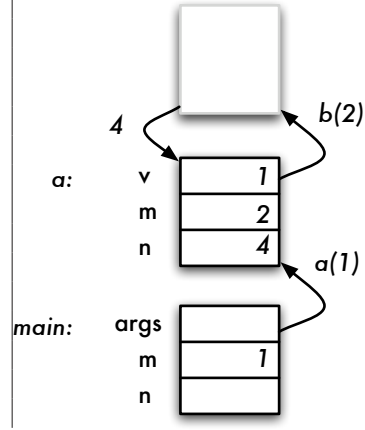
```java
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



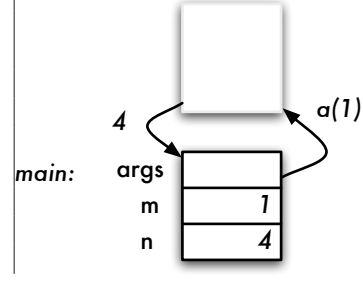
```java
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



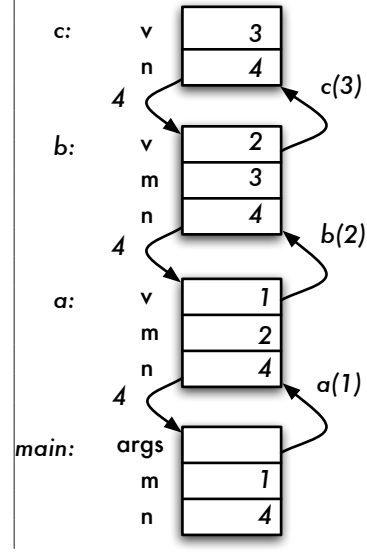
```java
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



```java
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



```java
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



Prologue

Summary

- A **a stack** is used when one wishes to process the elements **in reverse order**.

Next module

- **Stack** : linked elements

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures: Abstraction and Design Using Java.
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EECS)

University of Ottawa