COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

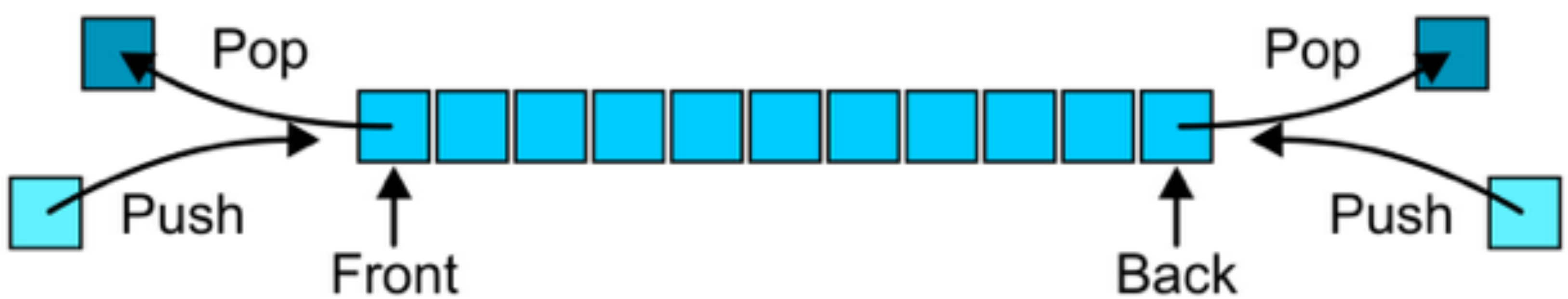
Computer Information System
University of Fraser Valley

* Some slides from “Algorithms and Data Structures” by Douglas Wilhelm Harder

Deque

Abstract Deque

- An Abstract Deque (Deque ADT) is an abstract data structure which emphasizes specific operations:
  - Uses a explicit linear ordering
  - Insertions and removals are performed individually
  - Allows insertions at both the front and back of the deque



- The name deque is short for "double ended queue" and is usually pronounced "deck".

Abstract Deque

The operations will be called

Front	Back(rear)
push	inject
pop	eject

There are four errors associated with this abstract data type:

- It is an undefined operation to access or pop from an empty deque

Implementations

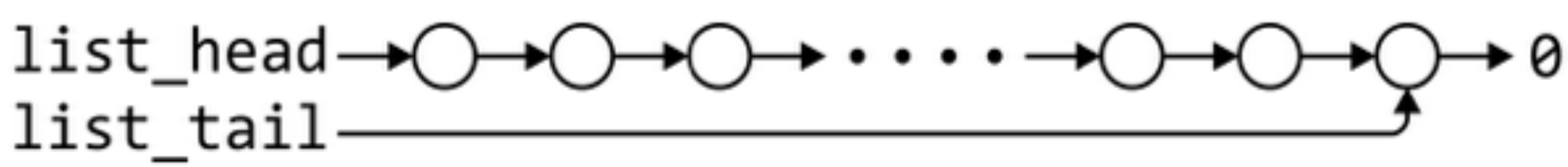
We will look at two implementations of deque:

Like the stack and queue, we will require that all the operations of a deque implementation are $\Theta(1)$

- The run time of the algorithm is independent of the number of objects being stored in the container
- We will always attempt to achieve this lower bound

Linked-List Implementation

Operations at the front of a singly linked list are all $\Theta(1)$

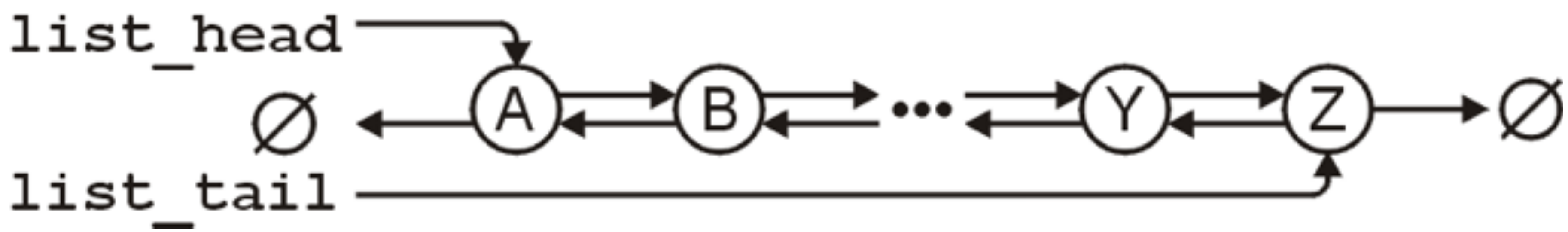


	Front/1st	**End/nth**
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

But not at the back!

Linked-List Implementation

Only a doubly linked list allows both insertions and erases at both the front and back in Θ(1) time.



	Front/1st	End/nth
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)$

We will therefore use a doubly linked list instead of a singly linked list.

Deque Class (linked list)

The deque class using a doubly linked list has a single private member variable:

we use an object of the DList

```
public class LinkListDeque {
  // *** fields ***
   private DList list;  // an object of DList

   public LinkListStack() { ... }
   public void push(Object obj) { ... }
   public void inject(Object obj) { ... }
   public Object pop() throws NoSuchElementException { ... }
   public Object eject() throws NoSuchElementException { ... }
   public Object front() throws NoSuchElementException { ... }
   public Object back() throws NoSuchElementException { ... }
   public int length() { ... }
   public boolean isEmpty() { ... }
}
```

Deque Class (linked list)

- The implementation is reasonably straight-forward
- Like our implementations of stacks and queues using the SList class, each of the member functions will call the corresponding member functions in the DList

Circular Array

- The implementation of a deque using a circular queue is reasonably straight-forward.
- The code is submitted to blackboard!
- Please read it and test it with different examples.

Built in Collections in Java

- Useful as a general-purpose tool:
  - Can be used as either a queue or a stack
- `java.util.Stack` class from the Java Collection API is extended from `Vector` class, supporting 5 methods
  - push(item)
  - pop()
  - peek()   // like top() in our implementation
- A more complete and consistent set of LIFO stack operations is provided by the `Deque` interface and its implementations
  - `Deque<Integer> stack = new ArrayDeque<Integer>();`

Applications of Stack

Stacks

- List of the same kind of elements
    - Addition and deletion of elements occur only at one end, called the top of the stack

- Applications:
    - Many problems
        - Postfix expression calculator, reversing a list
    - Computers use stacks to implement method calls
    - Stacks are also used to convert recursive algorithms into non-recursive algorithms

Postfix
Expression Calculator

- Infix notation: The operator is written between the operands
- Prefix or Polish notation: Operators are written before the operands
  - Does not require parentheses
- Reverse-Polish or postfix notation: Operators follow the operands
  - Has the advantage that the operators appear in the order required for computation

Postfix
Expression Calculator

- Reverse Polish or postfix notation: Operators follow the operands

Infix-Expressions	Equivalent Prefix-Expressions	Equivalent Postfix-Expressions
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
a * b + c	+ * a b c	a b * c +
(a + b) * c	* + a b c	a b + c *
(a − b) * (c + d)	*− a b + c d	a b − c d + *

Infix Expressions

Normally, mathematics is written using what we call *in-fix* expression (notation):

$$(3 + 4) \times 5 - 6$$

The operator is placed between to operands

One weakness: parentheses are required

$$(3 + 4) \times 5 - 6 = 29$$

$$3 + 4 \times 5 - 6 = 17$$

$$3 + 4 \times (5 - 6) = -1$$

$$(3 + 4) \times (5 - 6) = -7$$

Reverse-Polish or Postfix Expressions

Alternatively, we can place the operands first, followed by the operator:

$$(3 + 4) \times 5 - 6$$

$$3 \ 4 \ + \ 5 \ \times \ 6 \ -$$

Calculator reads left-to-right and performs any operation on the last two operands:

$$3 \ 4 \ + \ 5 \ \times \ 6 \ -$$

$$7 \quad 5 \ \times \ 6 \ -$$

$$35 \quad 6 \ -$$

$$29$$

Reverse-Polish Notation

This is called *reverse-Polish* notation
after the mathematician Jan Łukasiewicz

http://www.audiovis.nac.gov.pl/

Reverse-Polish or Postfix

Reverse-Polish notation is used with some programming languages

– *e.g.*, postscript, pdf, and HP calculators

Similar to the thought process required for writing assembly language code

– You cannot perform an operation until you have all of the operands loaded into registers

```
MOVE.L #$2A, D1        ; Load  42 into Register D1
MOVE.L #$100, D2       ; Load 256 into Register D2
ADD D2, D1             ; Add D2 into D1
```

Reverse-Polish or Postfix

Other examples:

$$3 \quad 4 \quad 5 \quad \times \quad + \quad 6 \quad -$$

$$3 \quad 20 \quad + \quad 6 \quad -$$

$$23 \quad 6 \quad -$$

$$17$$

$$3 \quad 4 \quad 5 \quad 6 \quad - \quad \times \quad +$$

$$3 \quad 4 \quad -1 \quad \times \quad +$$

$$3 \quad -4 \quad +$$

$$-1$$

Reverse-Polish or Postfix

Benefits:

- No ambiguity and no brackets are required
- It is the same process used by a computer to perform computations:
  - Operands must be loaded into registers before operations can be performed on them
- Reverse-Polish can be processed using stacks

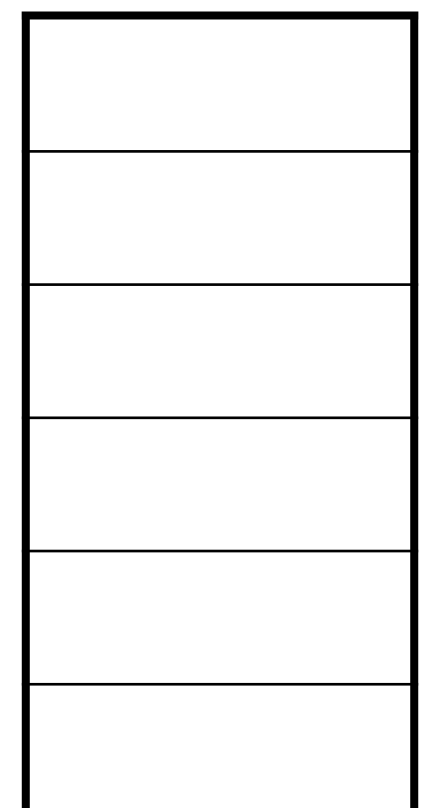
Reverse-Polish or Postfix

- The easiest way to parse reverse-Polish notation is to use an operand stack
- Algorithm to evaluate postfix expressions:
  - Scan the expression from left to right
  - When an operator is found, back up to get the required number of operands
    - pop the last two items off the operand stack (e.g. if it is a unary operator, pop just one item)
    - Perform the operation
    - push the result back onto the stack
  - Continue processing the expression

Reverse-Polish Notation

Evaluate the following reverse-Polish expression using a stack:

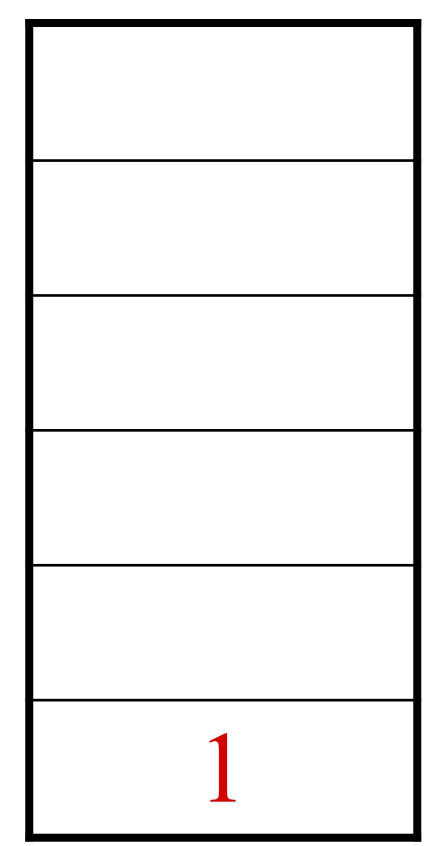
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$



Reverse-Polish Notation

Push 1 onto the stack

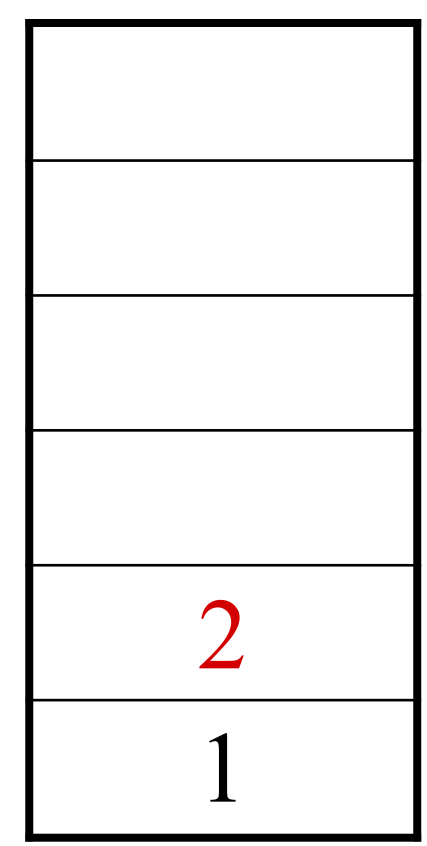
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 1 onto the stack

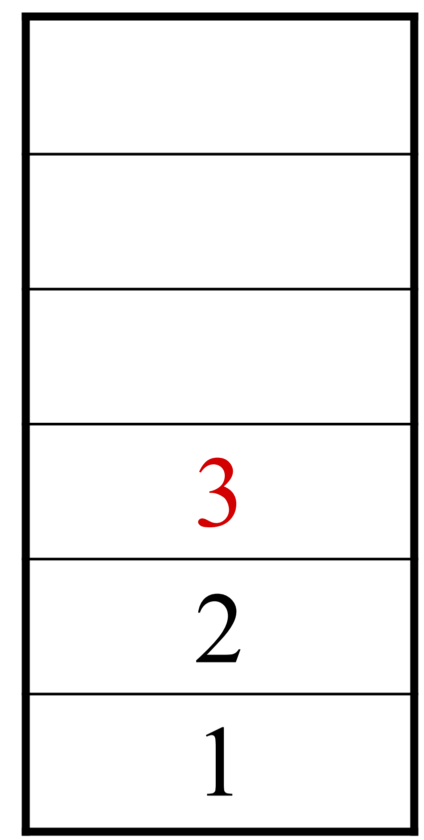
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 3 onto the stack

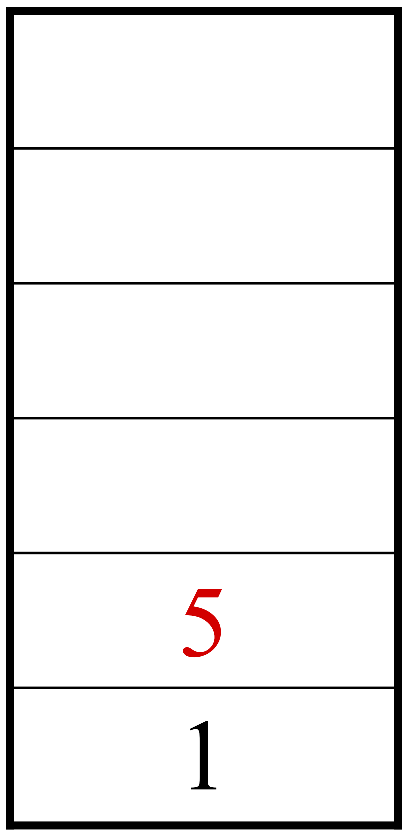
$$1 \quad 2 \quad \color{red}{3} \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Pop 3 and 2 and push $2 + 3 = 5$

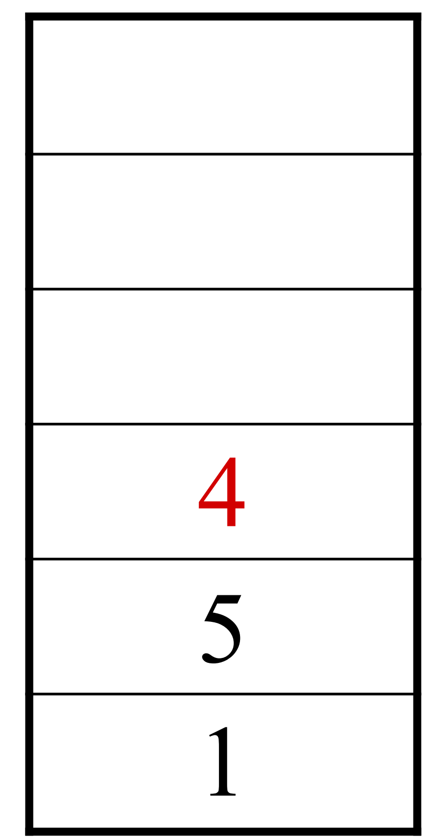
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 4 onto the stack

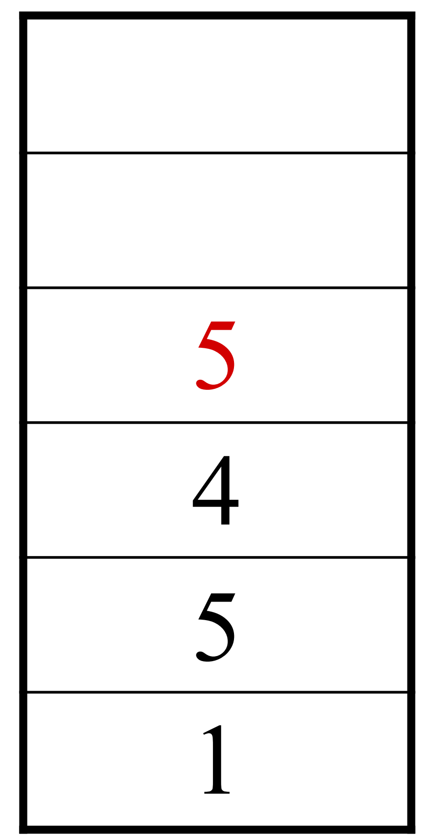
$$1\ \ 2\ \ 3\ \ +\ \ {\color{red}4}\ \ 5\ \ 6\ \ \times\ \ -\ \ 7\ \ \times\ \ +\ \ -\ \ 8\ \ 9\ \ \times\ \ +$$



Reverse-Polish Notation

Push 5 onto the stack

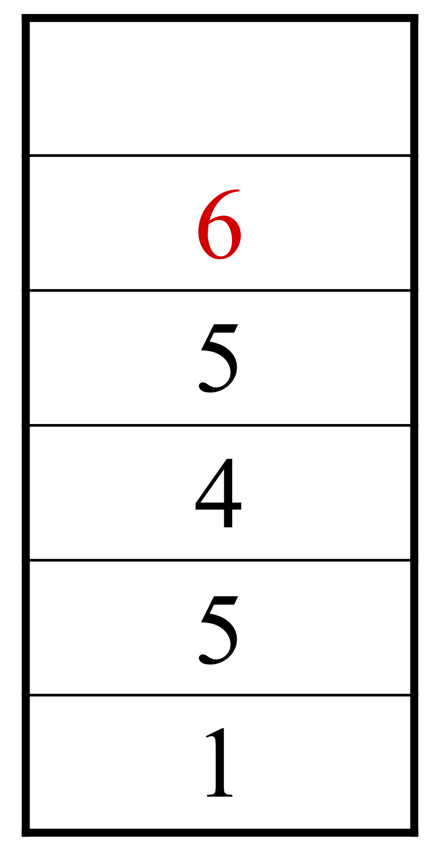
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 6 onto the stack

$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad \color{red}{6} \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Pop 6 and 5 and push $5 \times 6 = 30$

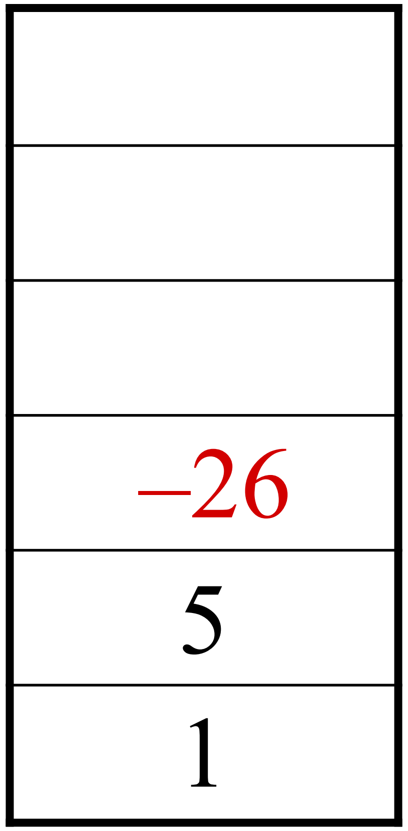
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$

30
4
5
1

Reverse-Polish Notation

Pop 30 and 4 and push $4 - 30 = -26$

$$1 \; 2 \; 3 \; + \; 4 \; 5 \; 6 \; \times \; - \; 7 \; \times \; + \; - \; 8 \; 9 \; \times \; +$$



Reverse-Polish Notation

Push 7 onto the stack

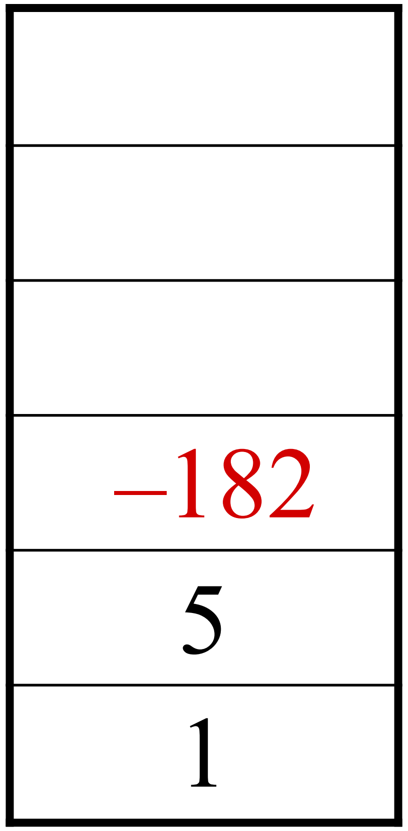
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad \color{red}{7} \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$

7
–26
5
1

Reverse-Polish Notation

Pop 7 and -26 and push $-26 \times 7 = -182$

$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Pop -182 and 5 and push $-182 + 5 = -177$

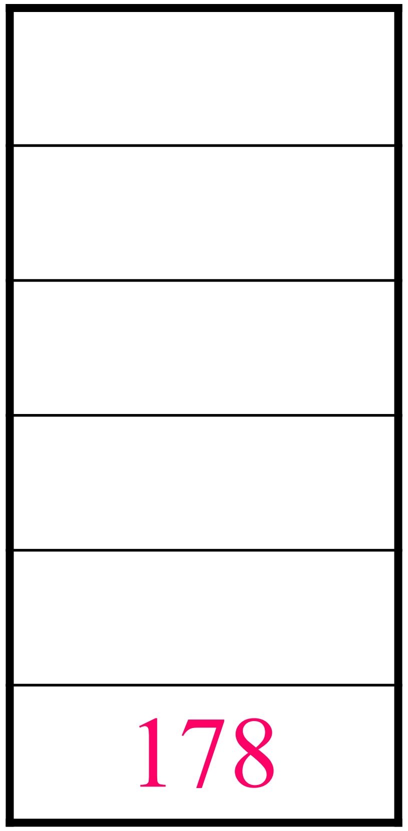
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$

-177
1

Reverse-Polish Notation

Pop -177 and 1 and push $1 - (-177) = 178$

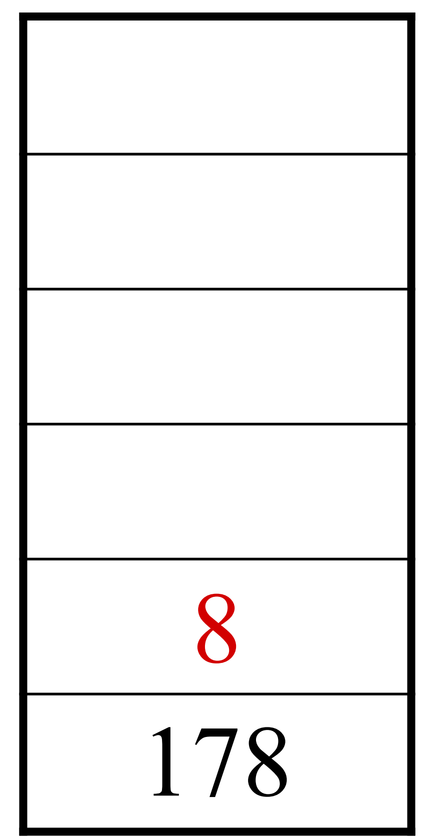
$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$



Reverse-Polish Notation

Push 8 onto the stack

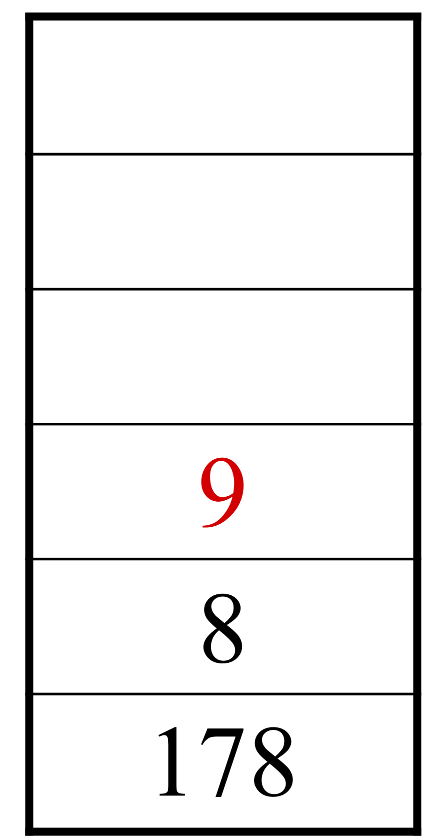
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad {\color{red}8} \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 1 onto the stack

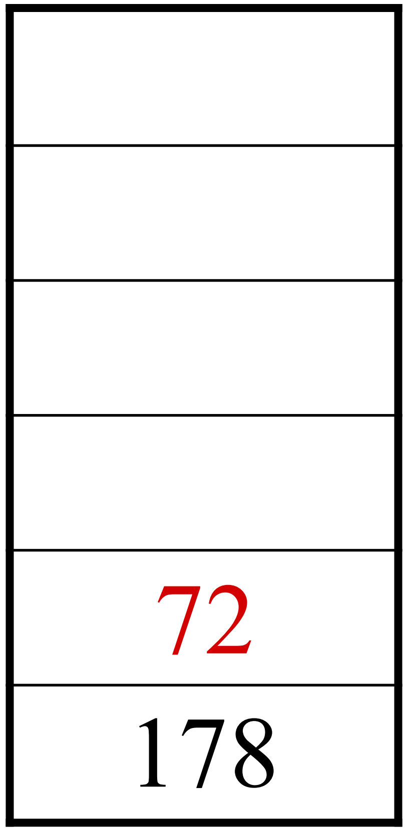
$$1\ \ 2\ \ 3\ +\ 4\ \ 5\ \ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ \ 9\ \times\ +$$



Reverse-Polish Notation

Pop 9 and 8 and push $8 \times 9 = 72$

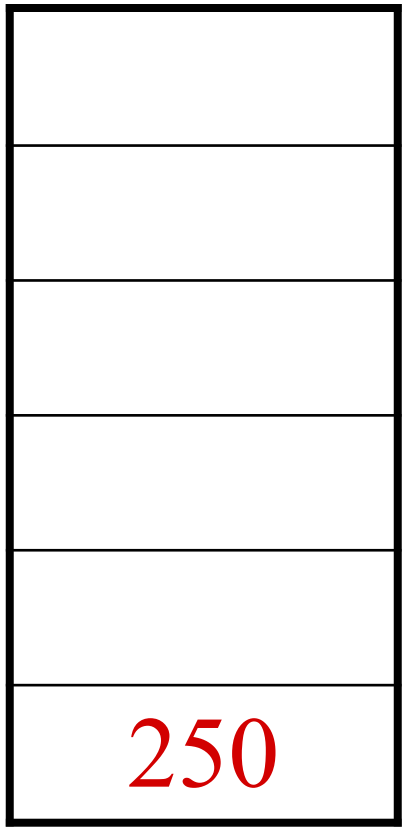
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$



Reverse-Polish Notation

Pop 72 and 178 and push $178 + 72 = 250$

$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Thus

$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$

evaluates to the value on the top: 250

The equivalent in-fix notation is

$$((1 - ((2 + 3) + ((4 - (5 \times 6)) \times 7))) + (8 \times 9))$$

We reduce the parentheses using order-of-operations:

$$1 - (2 + 3 + (4 - 5 \times 6) \times 7) + 8 \times 9$$

Reverse-Polish Notation

Incidentally,

$$1 - 2 + 3 + 4 - 5 \times 6 \times 7 + 8 \times 9 = -132$$

which has the reverse-Polish notation of

$$1 \;\; 2 \;\; - \;\; 3 \;\; + \;\; 4 \;\; + \;\; 5 \;\; 6 \;\; 7 \;\; \times \;\; \times \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$

For comparison, the calculated expression was

$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$

We will get back to Postfix and Prefix expression later in the course!

Reverse a Linked List

- Assume that we have an object of a singly linked list, the task is to reverse the linked list.

- Example:

  Input : 1->2->3->4->NULL
  Output : 4->3->2->1->NULL

  Input : 1->2->3->4->5->NULL
  Output : 5->4->3->2->1->NULL

  Input : NULL
  Output : NULL

  Input : 1->NULL
  Output : 1->NULL

Reverse a Linked List

- Naïve approach:
  - Assume our input `SList` object is called `list`
  - Create a new empty object of `SList` (`new_list`)
  - While `list` is not empty:
    - remove the last item from list
      (`last_item=list.removeEnd()`)
    - add it to the beginning of new_list
      (`new_list.addFront(last_item)`)

Reverse a Linked List

- Naïve approach:
  - Assume our input `SList` object is called `list`
  - Create a new empty object of `SList` (`new_list`)
  - While `list` is not empty:
    - remove the last item from list (`last_item=list.removeEnd()`)
    - add it to the beginning of new_list (`new_list.addFront(last_item)`)
- Very inefficient $\Theta(n^2)$ (why?)

Reverse a Linked List

- We can use a stack!

```
Stack st = new Stack();

while ( !list.isEmpty() )
  st.push(list.removeFront())

SList new_list = new SList();

while ( !st.isEmpty() )
  list.addEnd(st.pop())
```

- time complexity? $\Theta(n)$

Function Calls
Stack Frames

A look into the JVM

Inside the JVM a stack is used to

- create the local variables
- to store the return address from a call
- to pass the method-parameters

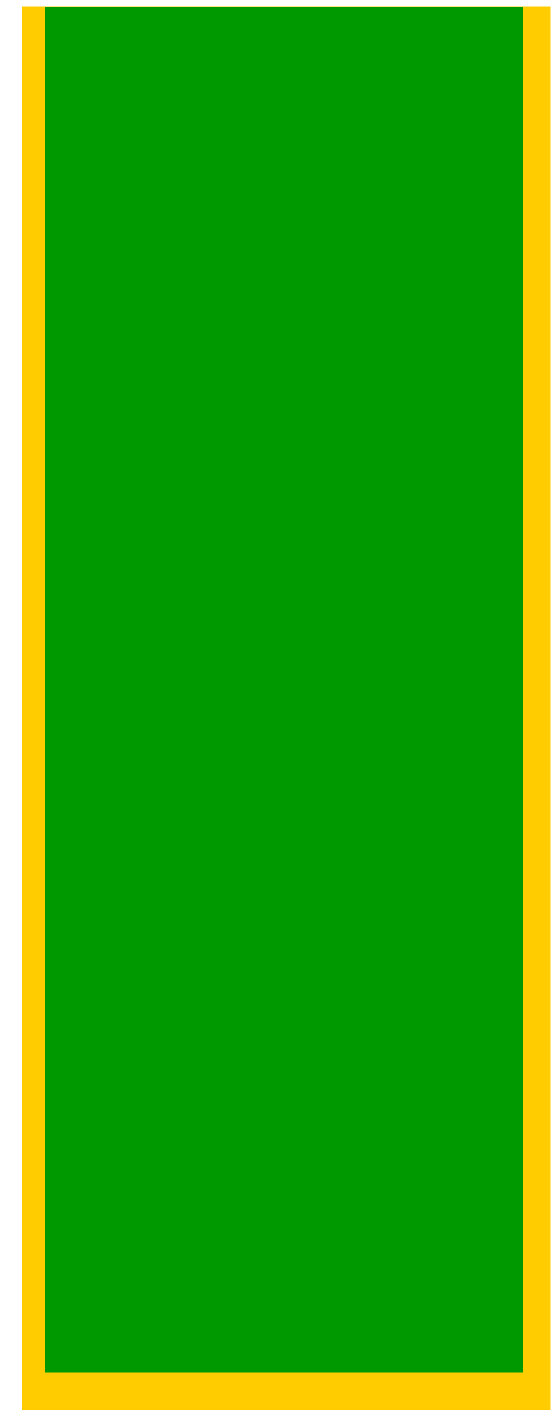
A look into the JVM

Sample Code:

```
public static void main(String args[ ]){
     int a = 3;
     int b = timesFive(a);
     System.out.println(b+““);
}


Public int timesFive(int a){
     int b = 5;
     int c = a * b;
     return (c);
 }
```

A look into the JVM

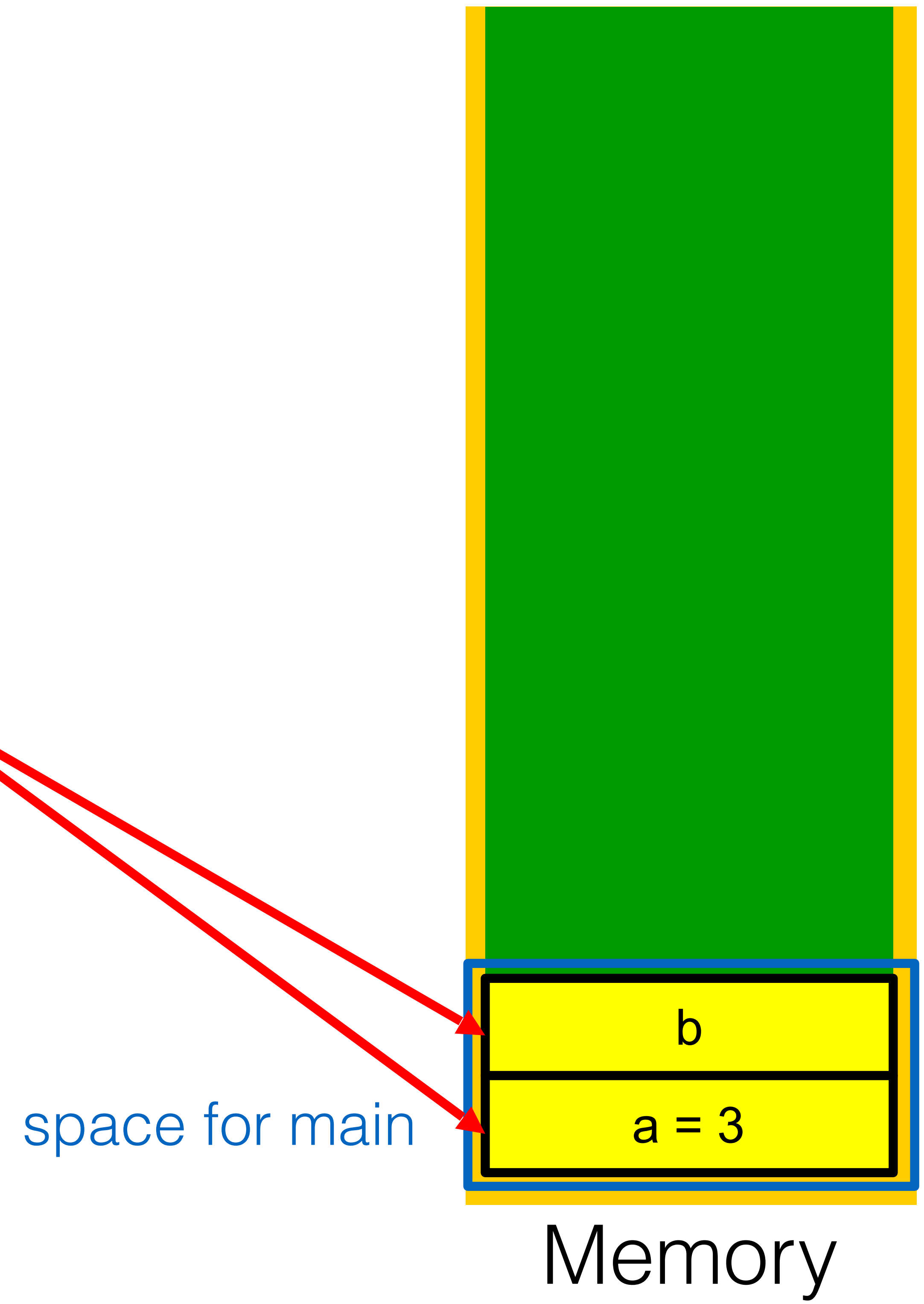
```
public static void main(String
   args[ ]){
     int a = 3;
     int b = timesFive(a);
     System.out.println(b+"");
   }


   Public int timesFive(int a){
     int b = 5;
     int c = a * b;
     return (c);
    }
```



Memory

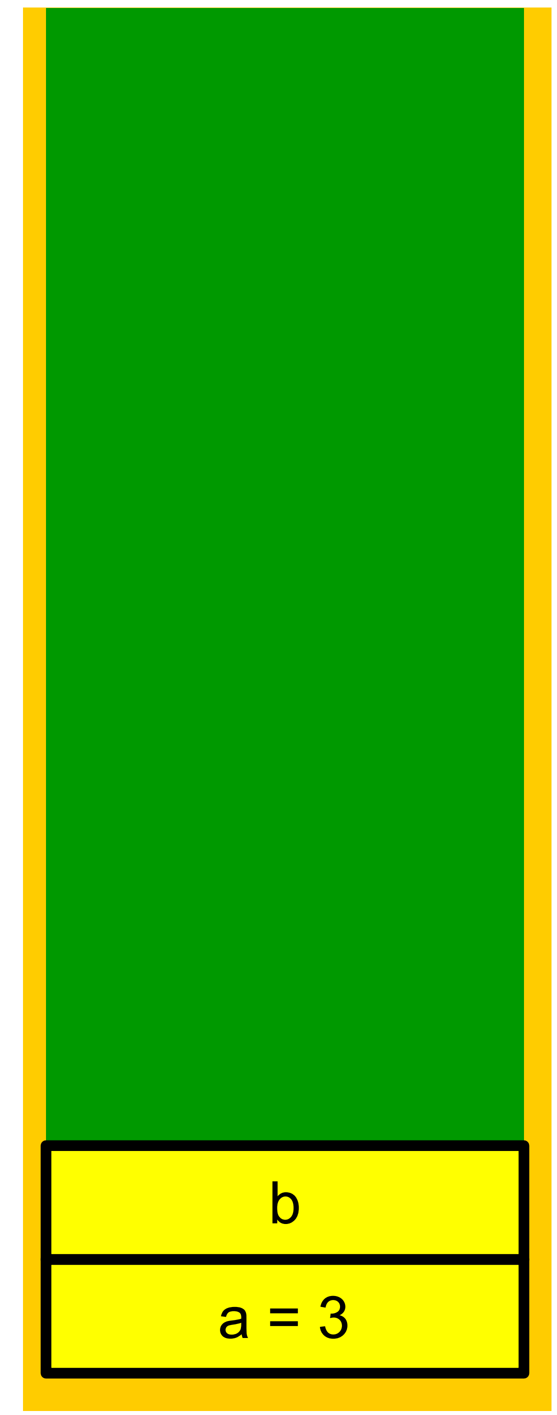
A look into the JVM

```
public static void main(String
   args[ ]){
     int a = 3;
     int b = timesFive(a);
     System.out.println(b+"");
   }


   Public int timesFive(int a){
     int b = 5;
     int c = a * b;
     return (c);
    }
```



A look into the JVM

```
public static void main(String
   args[ ]){
     int a = 3;
     int b = timesFive(a);
     System.out.println(b+"");
   }


   Public int timesFive(int a){
     int b = 5;
     int c = a * b;
     return (c);
    }
```



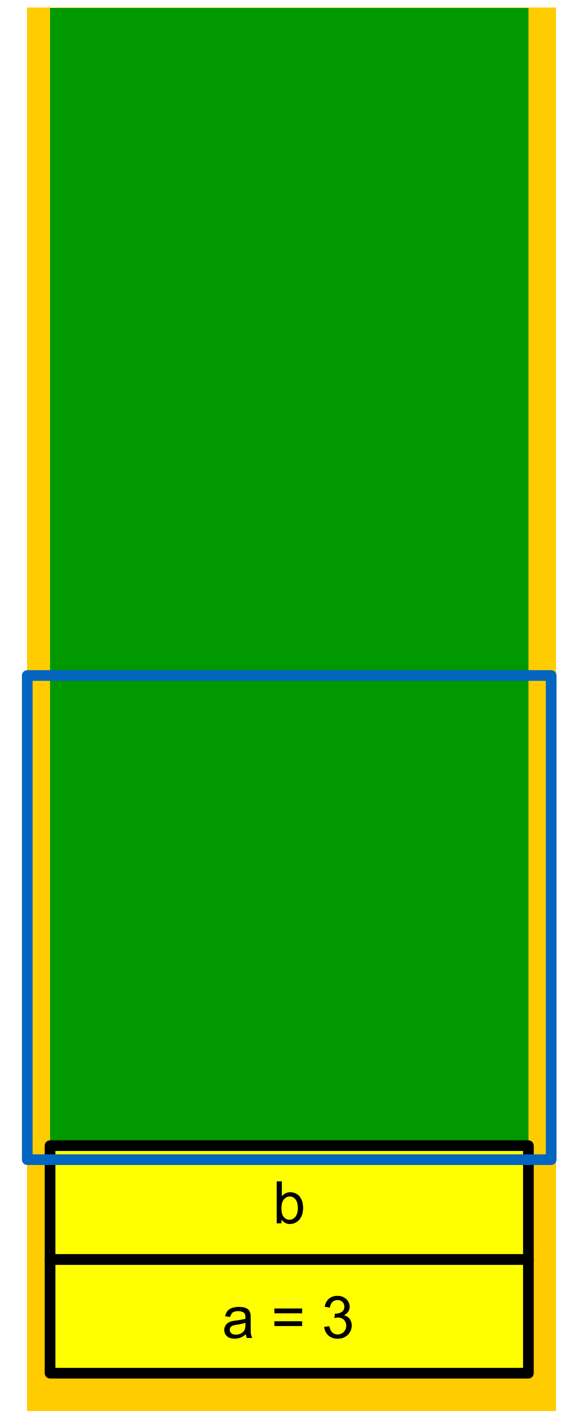
Memory

A look into the JVM

```
public static void main(String
   args[ ]){
     int a = 3;
     int b = timesFive(a);
     System.out.println(b+"");
   }

   Public int timesFive(int a){
     int b = 5;
     int c = a * b;
     return (c);
    }
```

Every call to a method creates a new set of local variables !

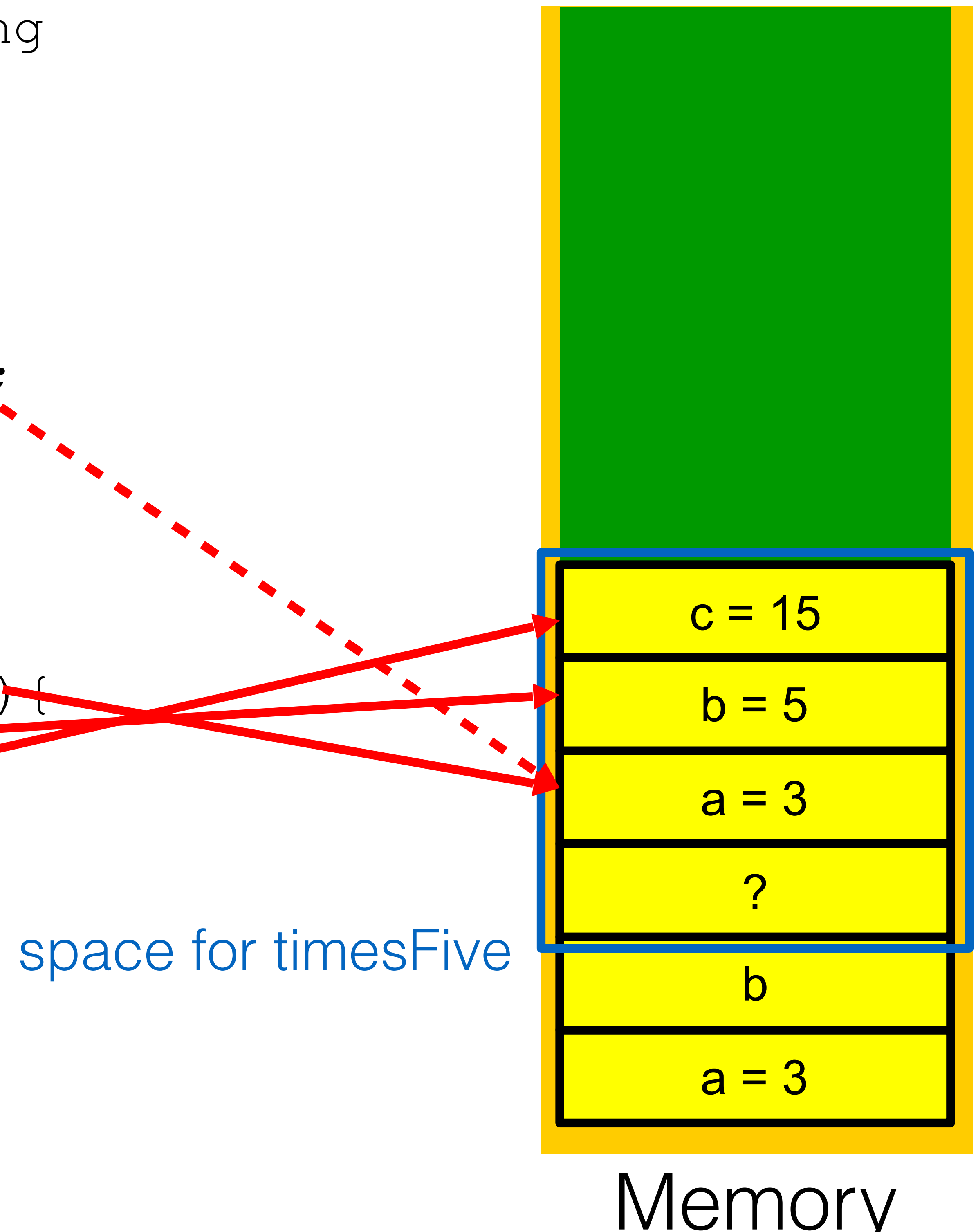
space for timesFive



Memory

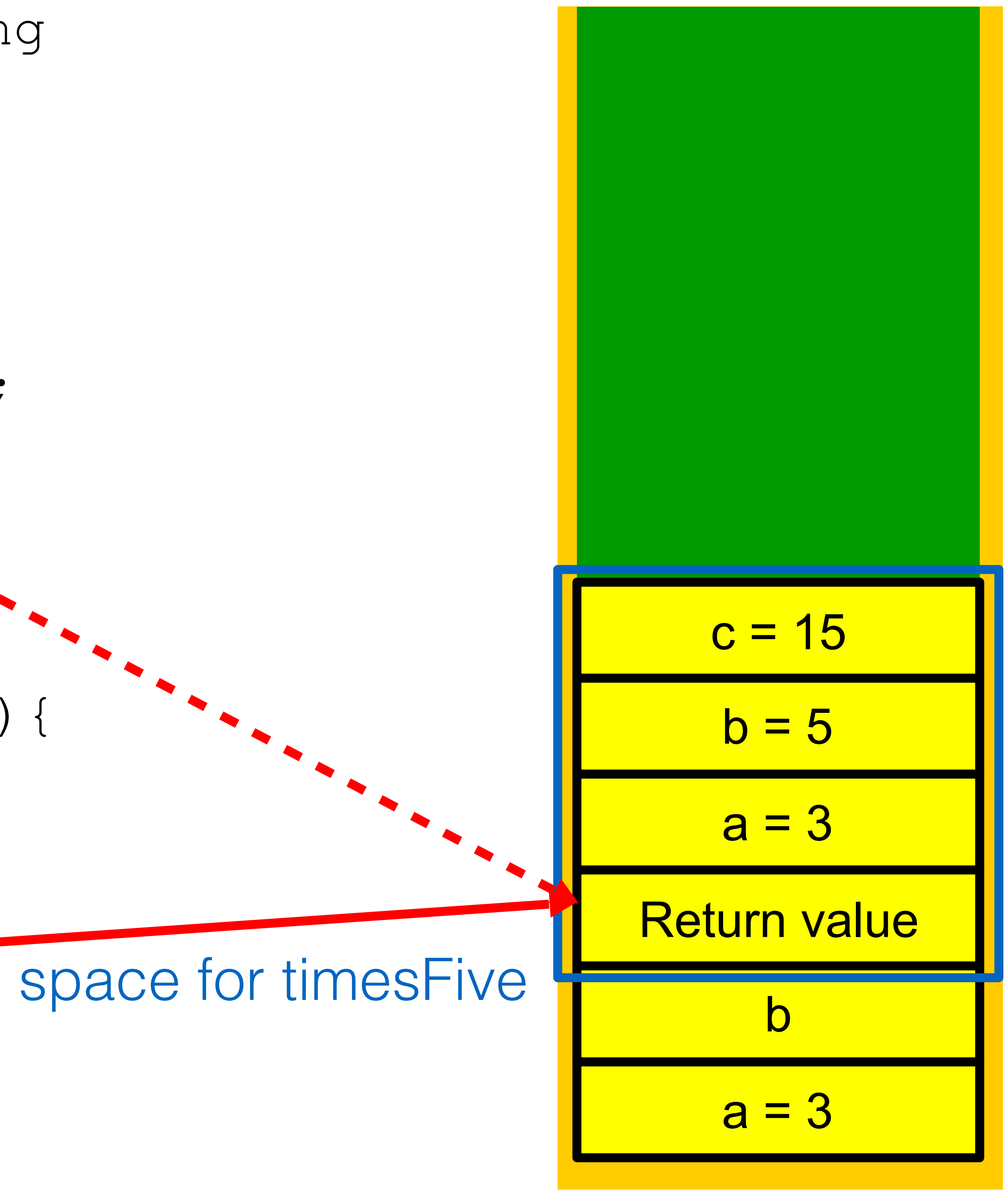
A look into the JVM

```
public static void main(String
   args[ ]){
     int a = 3;
     int b = timesFive(a);
     System.out.println(b+“”);
   }


   Public int timesFive(int a){
     int b = 5;
     int c = a * b;
     return (c);
    }
```



A look into the JVM

```
public static void main(String
   args[ ]){
     int a = 3;
     int b = timesFive(a);
     System.out.println(b+"");
   }


   Public int timesFive(int a){
     int b = 5;
     int c = a * b;
     return (c);
    }
```



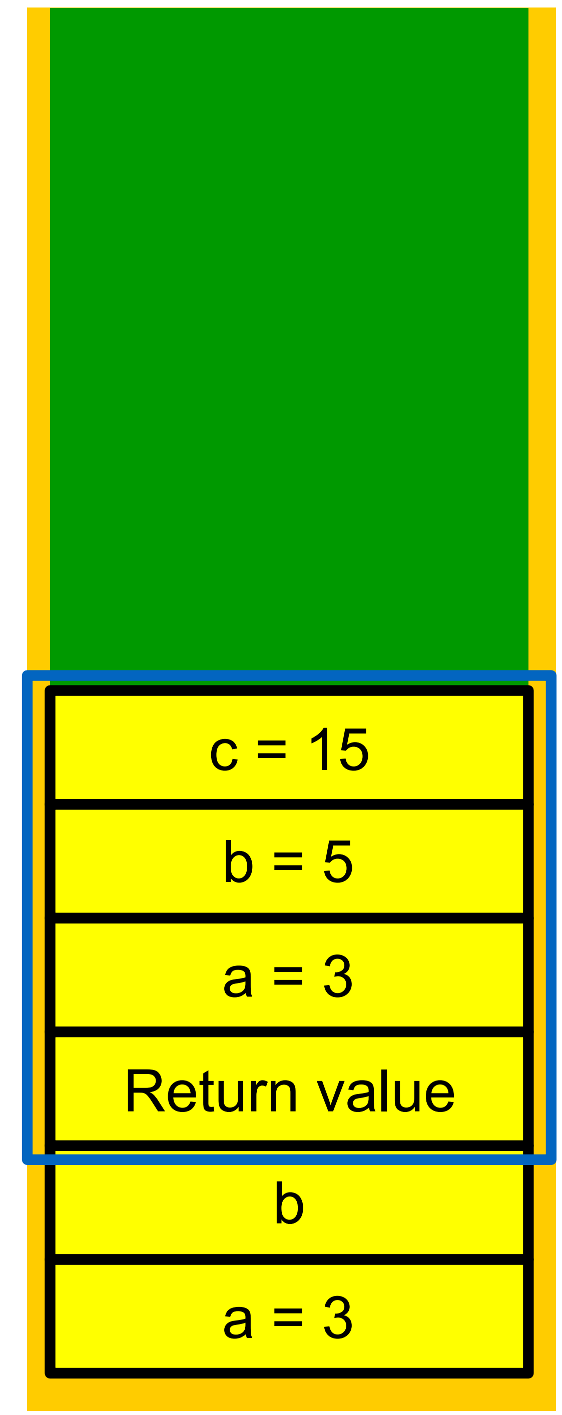
Memory

A look into the JVM

```
public static void main(String
   args[ ]){
     int a = 3;
     int b = timesFive(a);
     System.out.println(b+"");
   }

   Public int timesFive(int a){
     int b = 5;
     int c = a * b;
     return (c);
    }
```

stack frame:

1. local variables
2. arguments of function
3. return value of function

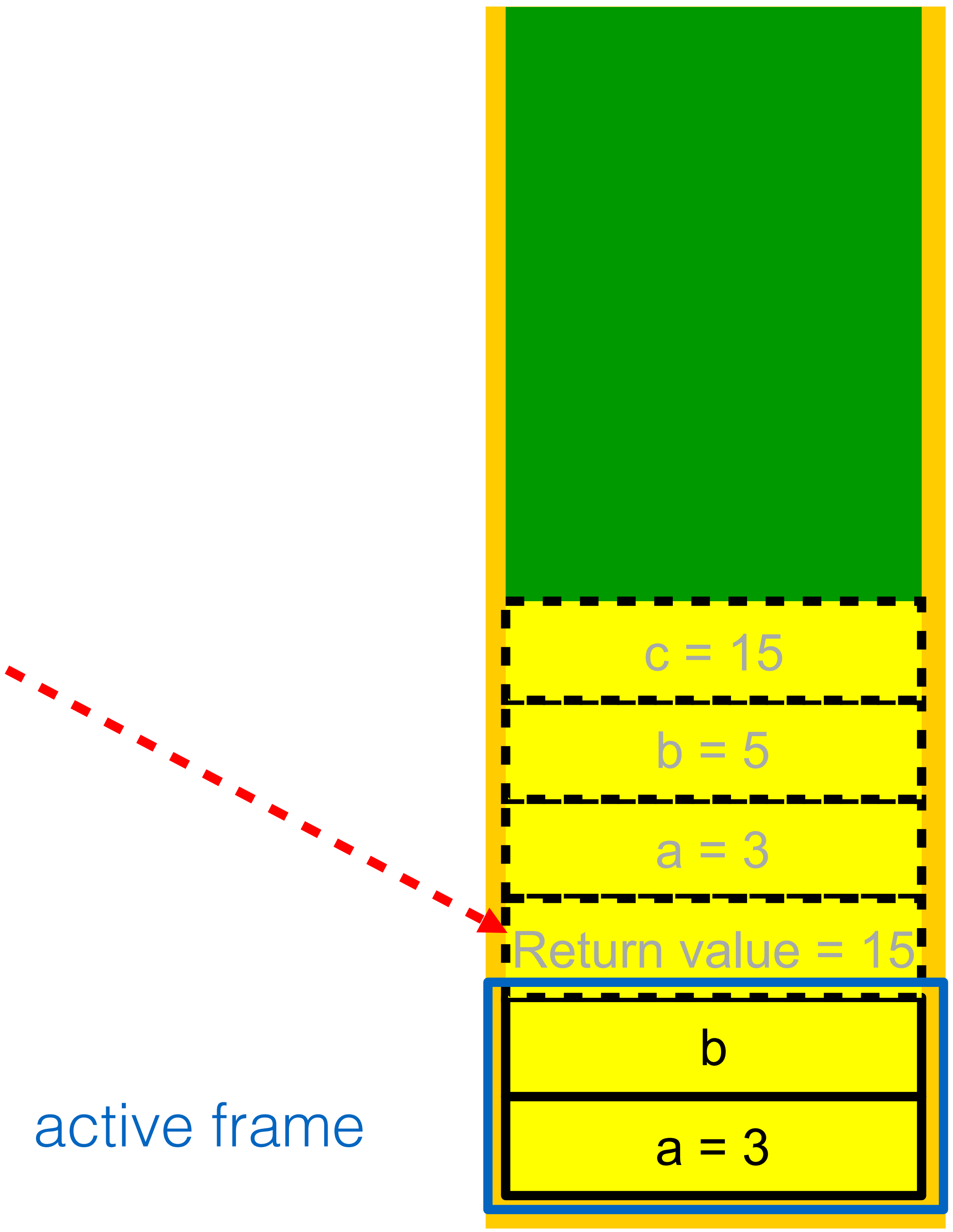


Memory

A look into the JVM

```
public static void main(String
   args[ ]){
     int a = 3;
     int b = timesFive(a);
     System.out.println(b+"");
   }
```

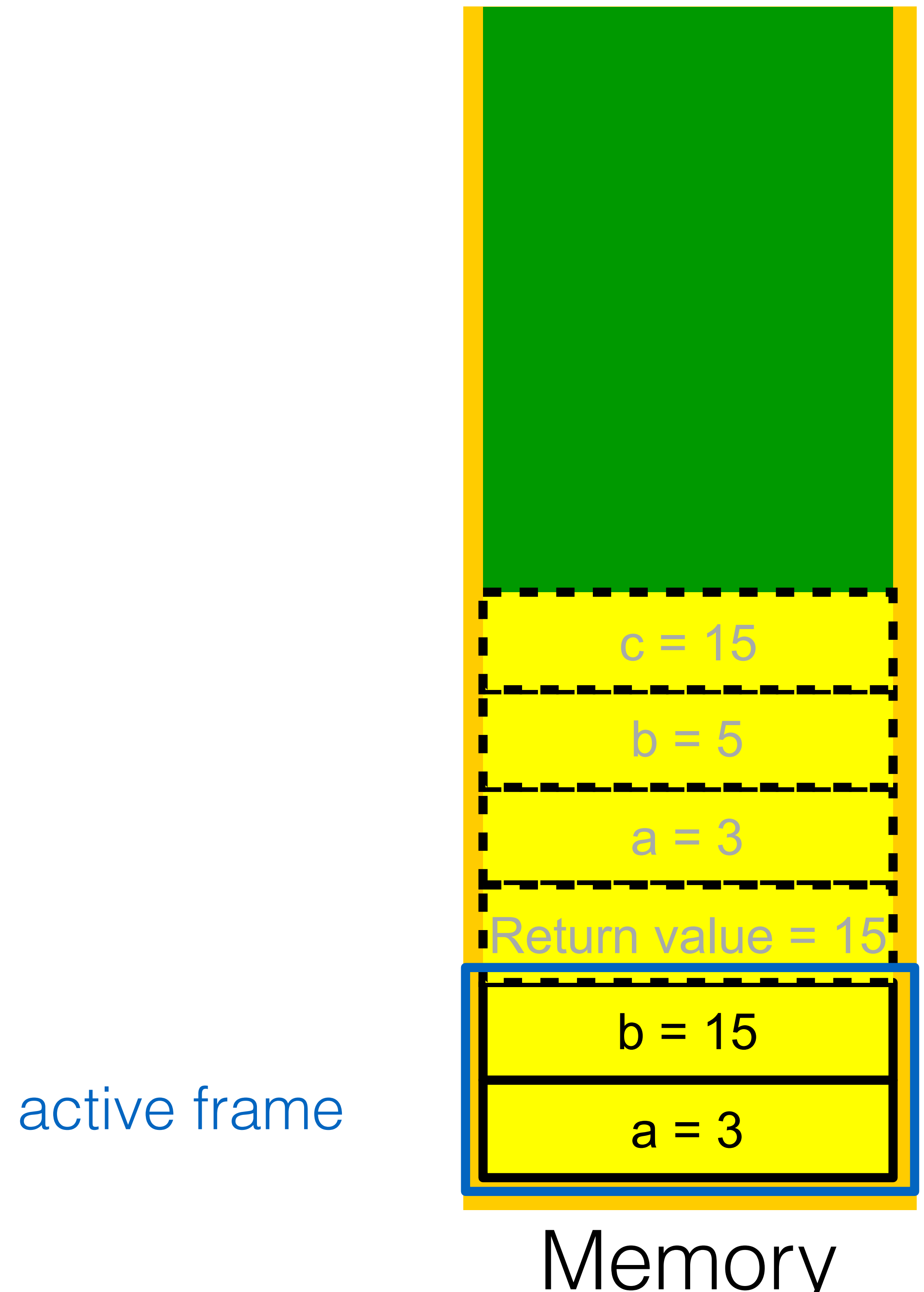
The local variables of each function are created on the stack and deleted when the function returns (the JVM takes the return value immediately after function returns)



Memory

A look into the JVM

```
public static void main(String
   args[ ]){
     int a = 3;
     int b = timesFive(a);
     System.out.println(b+"");
  }
```



A look into the JVM

- Program terminates,
- There is no active frame
- And memory is free

  (memory which is allocated to your program)