



## 4.3 Stacks and Queues



Dr. Dmitry Debugalov  
Finds the bug on top  
of the Stack.

### Data Structures and Data Types

#### Data types

- Set of values.
- Set of operations on those values.
- Some are built in to Java: `int`, `double`, `char`, `String`, ...
- Most are not: `Complex`, `Picture`, `Charge`, `Stack`, `Queue`, `Graph`, ...

#### Data structures.

- Represent data.
- Represent relationships among data.
- Some are built in to Java: `arrays`, ...
- Most are not: `linked list`, `circular list`, `tree`, `sparse array`, `graph`, ...



↑  
this lecture

#### Design challenge for every data type: What data structure to use?

- Requirement 1: Space usage.
- Requirement 2: Time usage for data-type methods

### Collections

#### Fundamental data types.

- Set of operations (`add`, `remove`, `test if empty`) on generic data.
- Intent is clear when we insert.
- Which item do we remove?

#### Stack. (this lecture)

- Remove the item **most** recently added.
- Ex: cafeteria trays, Web surfing.

↙ LIFO = "last in first out"

#### Queue. (see text)

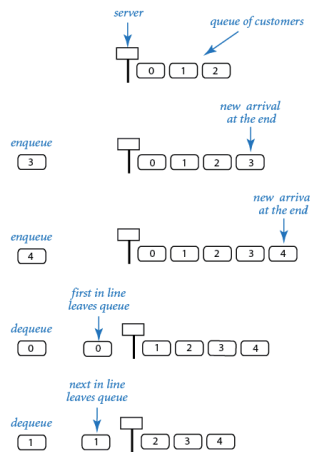
- Remove the item **least** recently added.
- Ex: Registrar's line.

↙ FIFO = "first in first out"

#### Symbol Table. (next lecture)

- Remove item with a given key.
- Ex: Phone book

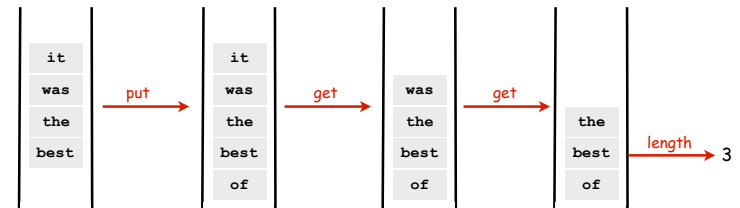
# FIFO Queues



5

# FIFO Queue API

```
public class QueueOfStrings
{
    QueueOfStrings() create an empty queue
    int length() size of the queue
    void put(String item) put a string onto the queue
    String get() get a string from the queue
}
```



6

## Queue Client Code Example: Read from input stream into an array

```
from previous lecture
public class WhiteFilter
{
    public static void main(String[] args)
    {
        In in = new In(args[0]);
        String[] words;
        // Fill words[] with strings from In (stay tuned).
        while (!StdIn.isEmpty())
        {
            String key = StdIn.readString();
            if (search(key, words) != -1)
                StdOut.println(key);
        }
    }
}

QueueOfStrings q = new QueueOfStrings();
while (!in.isEmpty())
    q.put(in.readString());
int N = q.length();
words = new String[N];
for (int i = 0; i < N; i++)
    words[i] = q.get();
```

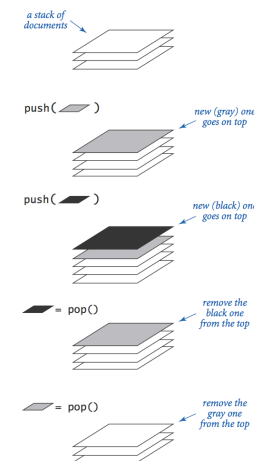
### Solves basic problem

- Can't store strings in array until it is created.
- Can't create array without knowing how many strings in input stream.
- Can't know how many strings in input stream without reading them all.
- Solution: keep them in a Queue

See text for implementation/applications (after learning about Stacks).

7

# Pushdown Stacks

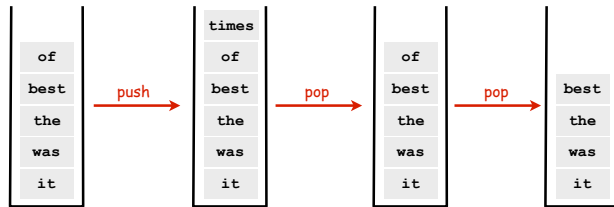


8

## Stack API

```
public class *StackOfStrings
{
    *StackOfStrings() create an empty stack
    boolean isEmpty() is the stack empty?
    void push(String item) push a string onto the stack
    String pop() pop the stack
}
```

*\*: we will consider more than one implementation*



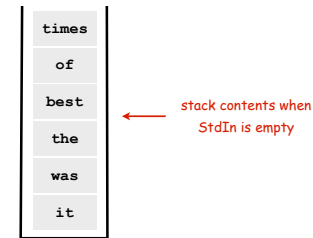
9

## Stack Client Example 1: Reverse

```
public class Reverse
{
    public static void main(String[] args)
    {
        StackOfStrings stack = new StackOfStrings();
        while (!StdIn.isEmpty())
            stack.push(StdIn.readString());
        while (!stack.isEmpty())
            StdOut.print(stack.pop());
        StdOut.println();
    }
}
```

```
% more tiny.txt
it was the best of times

% java Reverse tiny.txt
times of best the was it
```



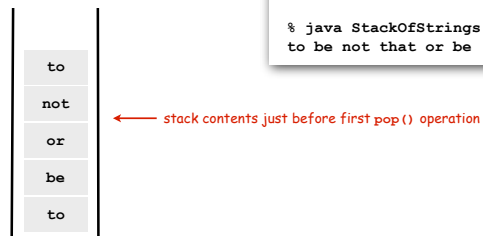
10

## Stack Client Example 2: Test Client

```
public static void main(String[] args)
{
    StackOfStrings stack = new StackOfStrings();
    while (!StdIn.isEmpty())
    {
        String item = StdIn.readString();
        if (item.compareTo("-") != 0)
            stack.push(item);
        else
            System.out.print(stack.pop());
    }
    System.out.println();
}
```

```
% more test.txt
to be or not to - be - - that - - - is

% java StackOfStrings < test.txt
to be not that or be
```



11

## Stack Client Example 3: Balanced Parentheses

```
( ( ( a + b ) * d ) + ( e * f ) )
(
↑ push
(
↑ push
(
↑ push
)
↑ pop
)
↑ pop
)
↑ pop
)
↑ pop
```

12

### Stack Client Example 3: Balanced Parentheses

```
public class Balanced
{
    public static void main(String[] args)
    {
        StackOfStrings stack = new StackOfStrings();
        while (!StdIn.isEmpty())
        {
            String item = StdIn.readString();
            if (item.compareTo("(") == 0)
                stack.push(item);
            if (item.compareTo(")") == 0)
            {
                if (stack.isEmpty())
                { StdOut.println("Not balanced"); return; }
                stack.pop();
            }
        }
        if (!stack.isEmpty()) StdOut.println("Not balanced");
        else StdOut.println("Balanced");
    }
}
```

```
% java Balanced
( ( ( a + b ) * d ) + ( e * f ) )
Balanced

% java Balanced
( ( a + b ) * d ) + ( e * f )
Not balanced
```

### Stack: Array Implementation

#### Array implementation of a stack.

- Use array `a[]` to store `N` items on stack. ← PROBLEM: How big to make array? (Stay tuned.)
- `push()` add new item at `a[N]`.
- `pop()` remove item from `a[N-1]`.



```
public class ArrayStackOfStrings
{
    private String[] a;
    private int N = 0;

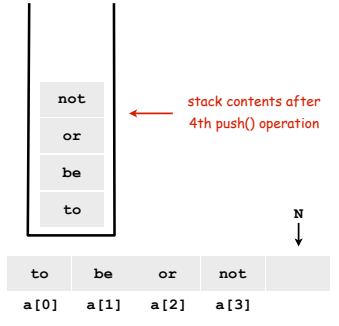
    public ArrayStackOfStrings(int max)
    { a = new String[max]; }

    public boolean isEmpty()
    { return (N == 0); }

    public void push(String item)
    { a[N++] = item; }

    public String pop()
    { return a[--N]; }
}
```

Strawman solution: Make client provide capacity.  
NOTE: This 'solution' violates the API!



### Array Stack: Trace

	StdIn	StdOut	N	a[]				
				0	1	2	3	4
			0					
push	to		1	to				
	be		2	to	be			
	or		3	to	be	or		
	not		4	to	be	or	not	
	to		5	to	be	or	not	to
pop	-	to	4	to	be	or	not	to
	be		5	to	be	or	not	be
	-	be	4	to	be	or	not	be
	-	not	3	to	be	or	not	be
	that		4	to	be	or	that	be
	-	that	3	to	be	or	that	be
	-	or	2	to	be	or	that	be
	-	be	1	to	be	or	that	be
	is		2	to	is	or	not	to

### TEQ on Stacks

Q. Can we always insert pop commands (-) to make strings come out sorted?

Ex 1: 6 5 4 3 2 1 - - - - -

Ex 2: 1 - 2 - 3 - 4 - 5 - 6 -

Ex 3: 4 1 - 3 2 - - - 6 5 - -

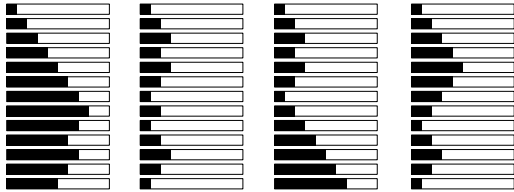
## Array Stack: Performance

**Running time.** Push and pop take constant time. ✓

**Memory.** Proportional to client-supplied capacity, **not** number of items. ✗

### Problem.

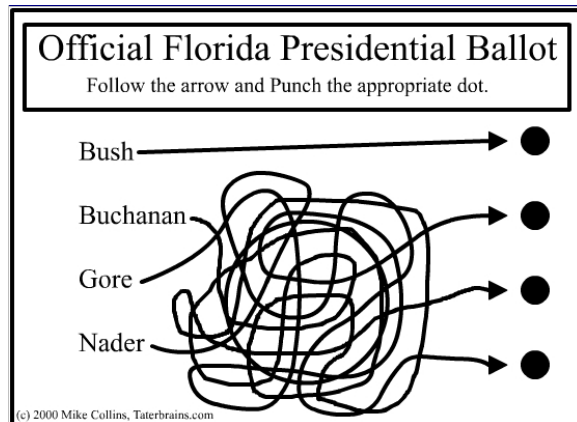
- Original API does not call for capacity (never good to change API)
- Client might have multiple stacks
- Client might not know what capacity to use (depends on *its* client)



**Challenge.** Stack implementation where space use is not fixed ahead of time.

17

## Linked Lists



(c) 2000 Mike Collins, Taterbrains.com

19

## Example: potential stack client

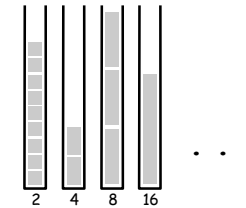
Possible implementation of Java memory management system (sketch)

### Maintain N stacks

- stack  $i$ : blocks of contiguous  $2^i$  byte chunks of memory
- **new**: pop from stack  $t$ , where  $2^t$  is smallest block that will hold new object
- stack  $t$  empty? pop from  $t+1$ , split in half, push 2 blocks on stack  $t$
- **garbage collector**: periodically finds unused memory blocks ← How? See COS 226. and pushes onto appropriate stack.

### Properties

- many stacks
- stack size unpredictable



Stack implementation without capacity restriction (as in API) is a **requirement**

18

## Sequential vs. Linked Data Structures

**Sequential data structure.** Put object one next to another.

- TOY: consecutive memory cells.
- Java: array of objects.

**Linked data structure.** Include in each object a **link** to the another one.

- TOY: link is memory address of next object.
- Java: link is reference to next object.

### Key distinctions.

- Array: arbitrary access, fixed size.
- Linked list: sequential access, variable size.

get  $i^{\text{th}}$  element

get next element

addr	value	addr	value
C0	"Alice"	C0	"Carol"
C1	"Bob"	C1	null
C2	"Carol"	C2	-
C3	-	C3	-
C4	-	C4	"Alice"
C5	-	C5	CA
C6	-	C6	-
C7	-	C7	-
C8	-	C8	-
C9	-	C9	-
CA	-	CA	"Bob"
CB	-	CB	C0

array


linked list

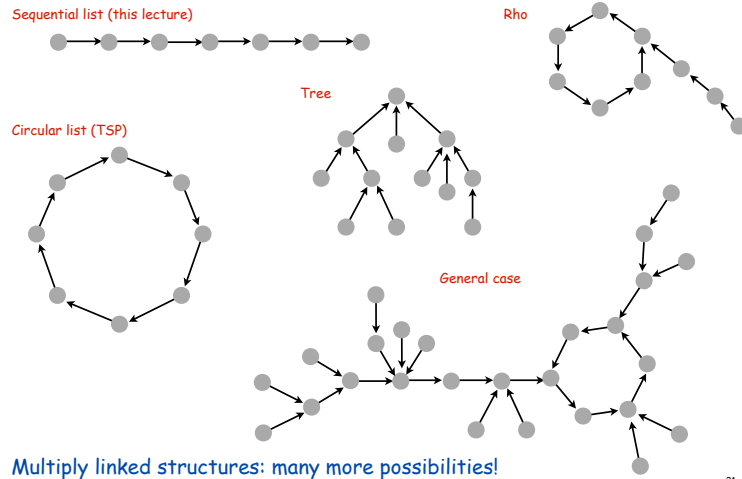
### Linked structures.

- Not intuitive, overlooked by naive programmers
- Flexible, widely used method for organizing data

20

## Singly-linked data structures

From the point of view of a particular object, all of these structures look the same: 



Multiply linked structures: many more possibilities!

21

## Linked Lists

Linked list.

- Simplest linked structure.
- A recursive data structure.
- An item plus a pointer to another linked list (or empty list).
- Unwind recursion: linked list is a sequence of items.

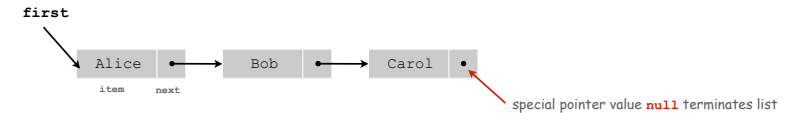
Node data type.

- A reference to a String.
- A reference to another Node.

```
public class Node
{
    private String item;
    private Node next;
}
```

Confusing point:

Purpose of data structure is to represent data in a data type but, we also use data types to implement data structures  
Example: The data type Node acts behind the scenes to implement the linked list data structure. It is not visible to the client.



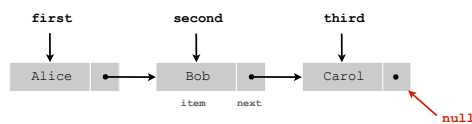
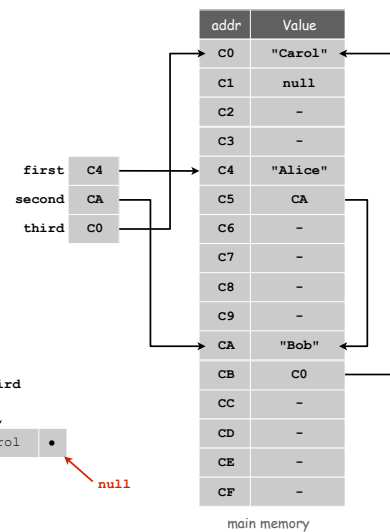
22

## Building a Linked List

```
Node third = new Node();
third.item = "Carol";
third.next = null;

Node second = new Node();
second.item = "Bob";
second.next = third;

Node first = new Node();
first.item = "Alice";
first.next = second;
```



23

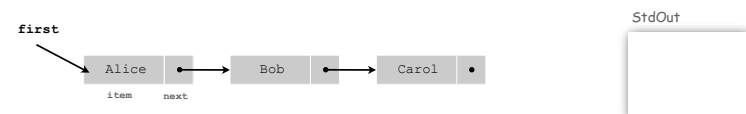
## Traversing a List

Iteration. Idiom for traversing a null-terminated linked list.

```
Node x = first;
while (x != null)
{
    StdOut.println(x.item);
    x = x.next;
}
```

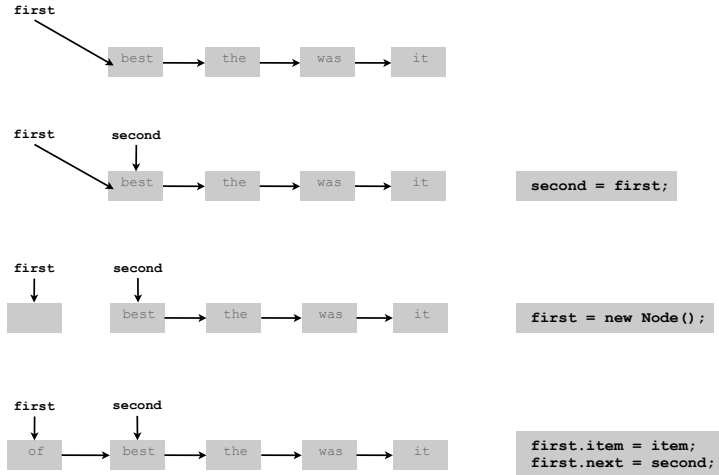
shorthand version

```
for (Node x = first; x != null; x = x.next)
    StdOut.println(x.item);
```



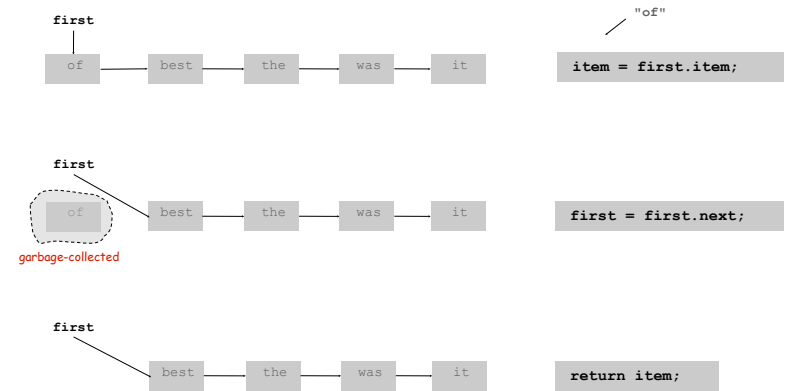
24

## Stack Push: Linked List Implementation



25

## Stack Pop: Linked List Implementation



26

## Stack: Linked List Implementation

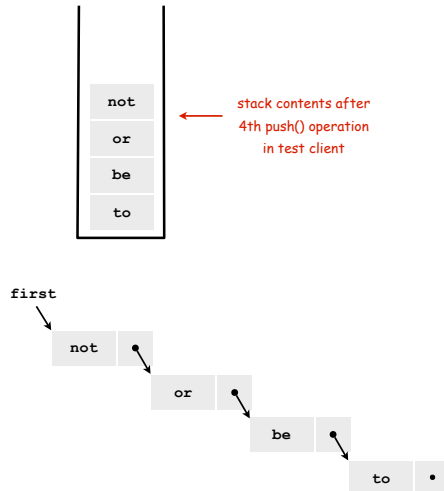
```
public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        private String item;
        private Node next;
    }

    public boolean isEmpty()
    { return first == null; }

    public void push(String item)
    {
        Node second = first;
        first = new Node();
        first.item = item;
        first.next = second;
    }

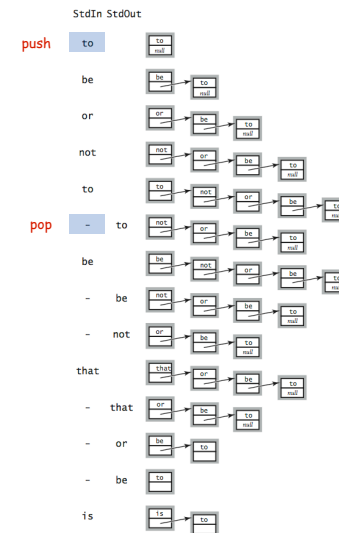
    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```



Note difference between `first` and `second`:  
`first`: an instance variable that retains state  
`second`: a local variable that goes out of scope

27

## Linked List Stack: Trace



28

## Linked-List Stack: Performance

**Running time.** Push and pop take constant time. ✓

**Memory.** Always proportional to number of items in stack. ✓

29

## Stack Data Structures: Tradeoffs

Two data structures to implement the Stack data type.

**Array.**

- Every push/pop operation take constant time.
- **But does not implement API...** (must fix max capacity ahead of time).

**Linked list.**

- Every push/pop operation takes constant time.
- **But...** uses extra space and time to deal with references.

**Client** can evaluate performance tradeoffs to choose among APIs  
(implicitly choosing among underlying data structures)

30

## TEQ on List Processing 1

What does the following code do?

```
...
Node list = null;
while (!StdIn.isEmpty())
{
    Node old = list;
    list = new Node();
    list.item = StdIn.readString();
    list.next = old;
}
for (Node t = list; t != null; t = t.next)
    StdOut.println(t.item);
...
```

31

## TEQ on List Processing 2

What does the following code do?

```
...
Node list = new Node();
list.item = StdIn.readString();
Node last = list;
while (!StdIn.isEmpty())
{
    last.next = new Node();
    last = last.next;
    last.item = StdIn.readString();
}
...
```

32



# Parameterized Data Types

We implemented: StackOfStrings.

We also want: StackOfMemoryBlocks, StackOfURLs, StackOfInts, ...

Strawman. Implement a separate stack class for each type.

- Rewriting code is tedious and **error-prone**.
- Maintaining cut-and-pasted code is tedious and **error-prone**.

33

34

## Generics

Generics. Parameterize stack by a single type.

```

"Stack of Apples"
Stack<Apple> stack = new Stack<Apple>();
Apple a = new Apple();
Orange b = new Orange();
stack.push(a);
stack.push(b); // compile-time error
a = stack.pop();
    
```

parameterized type

sample client

Can't push an "Orange" onto a "Stack of Apples"

35

## Generic Stack: Linked List Implementation

String stack (for reference)

```

public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        private String item;
        private Node next;
    }

    public boolean isEmpty()
    { return first == null; }

    public void push(String item)
    {
        Node second = first;
        first = new Node();
        first.item = item;
        first.next = second;
    }

    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
    
```

```

public class Stack<Item>
{
    private Node first = null;

    private class Node
    {
        private Item item;
        private Node next;
    }

    public boolean isEmpty()
    { return first == null; }

    public void push(Item item)
    {
        Node second = first;
        first = new Node();
        first.item = item;
        first.next = second;
    }

    public Item pop()
    {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
    
```

parameterized type name chosen by programmer

36

## Autoboxing

### Generic stack implementation.

- Cannot use primitives with parameterized data types
- Can only substitute a reference type name for a parameterized name.

### Wrapper type.

- Each primitive type has a **wrapper** reference type.
- Ex: `Integer` is wrapper type for `int`.
- Wrapper type has larger set of operations than primitive type.
- Values of wrapper type are objects.

**Autoboxing.** Automatic cast from primitive type to wrapper type.

**Autounboxing.** Automatic cast from wrapper type to primitive type.

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);      // Autobox (int -> Integer)
int a = stack.pop(); // Auto-unbox (Integer -> int)
```

37

## Stack Applications

### Real world applications.

- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.

38

## Stack Client 4: Arithmetic Expression Evaluation

**Goal.** Evaluate infix expressions.

( 1 + ( ( 2 + 3 ) \* ( 4 \* 5 ) ) )

operand                      operator

value stack  
operator stack

**Two stack algorithm.** [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parens: ignore.
- Right parens: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

39

## Arithmetic Expression Evaluation

```
public class Evaluate
{
    public static void main(String[] args)
    {
        Stack<String> ops = new Stack<String>();
        Stack<Double> vals = new Stack<Double>();
        while (!StdIn.isEmpty())
        {
            String s = StdIn.readString();
            if (s.equals("("))
            else if (s.equals("+")) ops.push(s);
            else if (s.equals("*")) ops.push(s);
            else if (s.equals("("))
            {
                String op = ops.pop();
                if (op.equals("+"))
                    vals.push(vals.pop() + vals.pop());
                else if (op.equals("*"))
                    vals.push(vals.pop() * vals.pop());
            }
            else vals.push(Double.parseDouble(s));
        }
        StdOut.println(vals.pop());
    }
}

% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

40

## Correctness

**Why correct?** When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

So it's as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

Repeating the argument:

```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

**Extensions.** More ops, precedence order, associativity, whitespace.

```
1 + ( 2 - 3 - 4 ) * 5 * sqrt(6*6 + 7*7)
```

41

## Postfix

**Observation 1.** Remarkably, the 2-stack algorithm computes the same value if the operator occurs **after** the two values.

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

**Observation 2.** Now **all** of the parentheses are redundant!

```
1 2 3 + 4 5 * * +
```



Jan Lukasiewicz

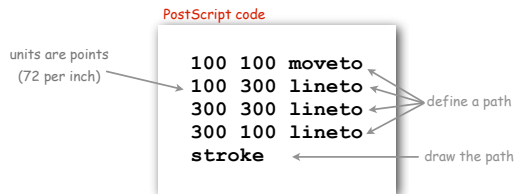
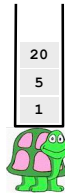
**Bottom line.** Postfix or "reverse Polish" notation.

42

## Real-World Stack Application: PostScript

**PostScript (Warnock-Geschke, 1980s).** A turtle with a stack.

- postfix program code
- add commands to drive virtual graphics machine
- add loops, conditionals, functions, types



**Simple virtual machine, but not a toy.**

- Easy to specify published page.
- Easy to implement on various specific printers
- Revolutionized world of publishing.
- Virtually all printed material is PostScript.

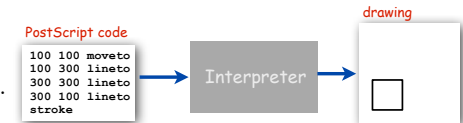


43

## Context/Definitions/Summary

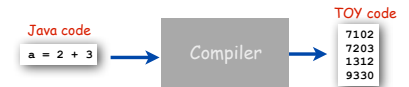
**Interpreter.**

- Takes a program as input
- Does what that program would do.
- Simulates a **virtual machine**.



**Compiler.**

- Takes a program as input
- Produces a program as output.
- Produces code for a (real) machine.



TOY is our proxy for a real machine

**Data Type and Virtual Machine are the same thing!**

- Set of values = machine state.
- Operations on values = machine operations.

- Virtual machines you have used
- LFSR
  - Stack
  - TOY
  - PostScript
  - Java Virtual Machine (another stack machine)

**Data Structure.**

- Represent data and relationships among data in a data type.
- array, linked list, compound, multiple links per node

44