Stacks

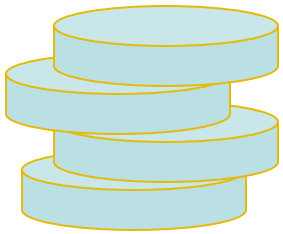
Java Class Library: The Class `Stack`

- Methods in class `Stack` in `java.util`

```
public Object push(Object item);
public Object pop();
public Object peek();
public boolean empty();
public int search(Object desiredItem);
public Iterator iterator();
public ListIterator listIterator();
```

Specifications of the ADT Stack

- Organizes entries according to order in which added
- Additions are made to one end, the top
- The item most recently added is always on the top



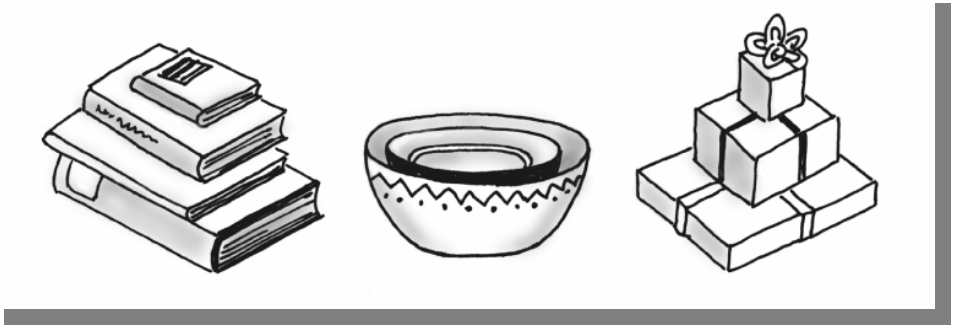


Fig. Some familiar stacks.

Specifications of the ADT Stack

- Specification of a stack of objects

```
public interface StackInterface
{       /** Task: Adds a new entry to the top of the stack.
        * @param newEntry an object to be added to the stack */
        public void push(Object newEntry);

        /** Task: Removes and returns the top of the stack.
        * @return either the object at the top of the stack or null if  the stack was empty */
        public Object pop();

        /** Task: Retrieves the top of the stack.
        * @return either the object at the top of the stack or null if the stack is empty */
        public Object peek();

        /** Task: Determines whether the stack is empty.
        * @return true if the stack is empty */
        public boolean isEmpty();

        /** Task: Removes all entries from the stack */
        public void clear();
} // end StackInterface
```

Specifications of the ADT Stack

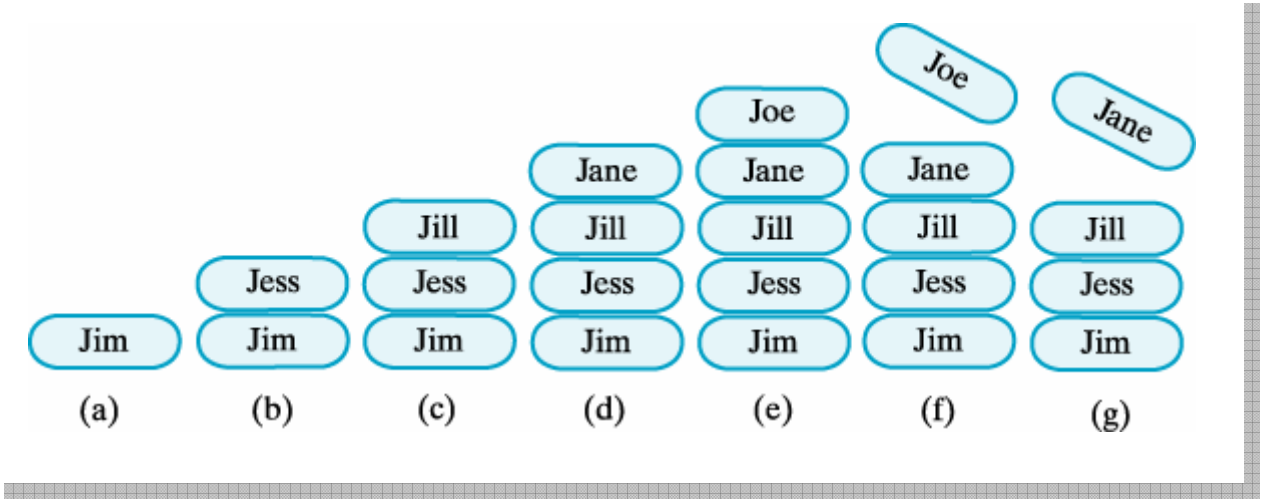


Fig. A stack of strings after (a) push adds *Jim*; (b) push adds *Jess*; (c) push adds *Jill*; (d) push adds *Jane*; (e) push adds *Joe*; (f) pop retrieves and removes *Joe*; (g) pop retrieves and removes *Jane*

Using a Stack to Process Algebraic Expressions

- Infix expressions
  - Binary operators appear between operands
  - `a + b`
- Prefix expressions
  - Binary operators appear before operands
  - `+ a b`
- Postfix expressions
  - Binary operators appear after operands
  - `a b +`
  - Easier to process – no need for parentheses nor precedence

Checking for Balanced `()`, `[]`, `{}`

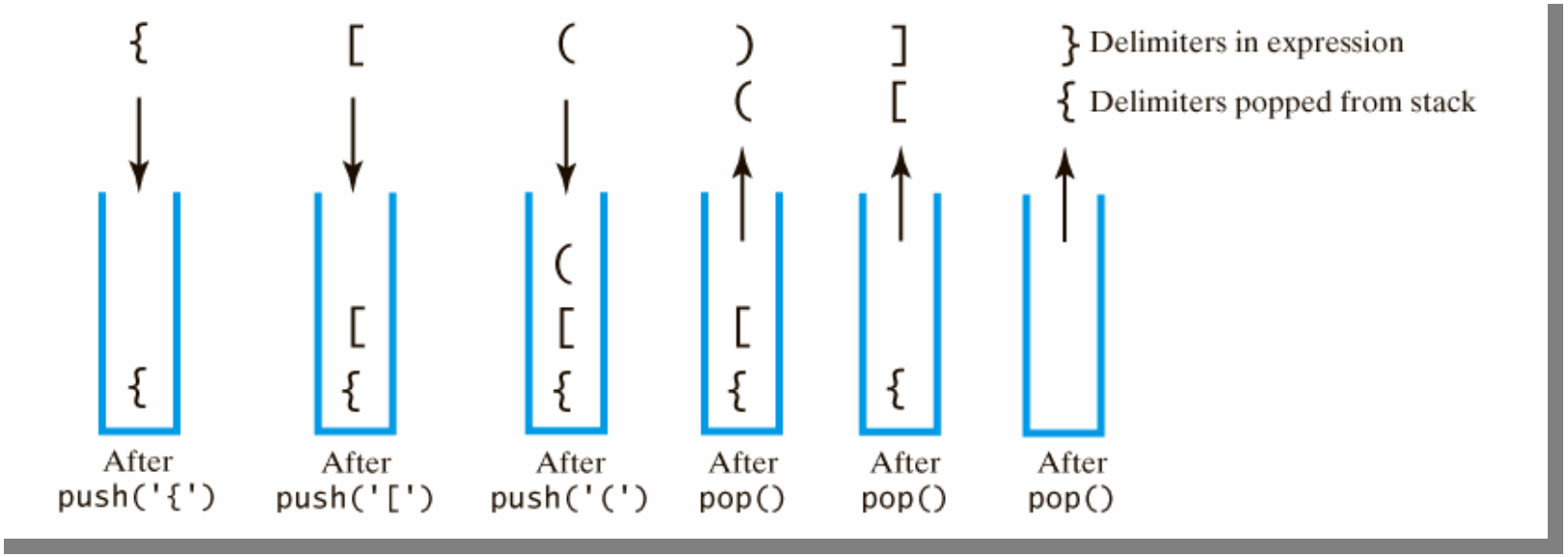
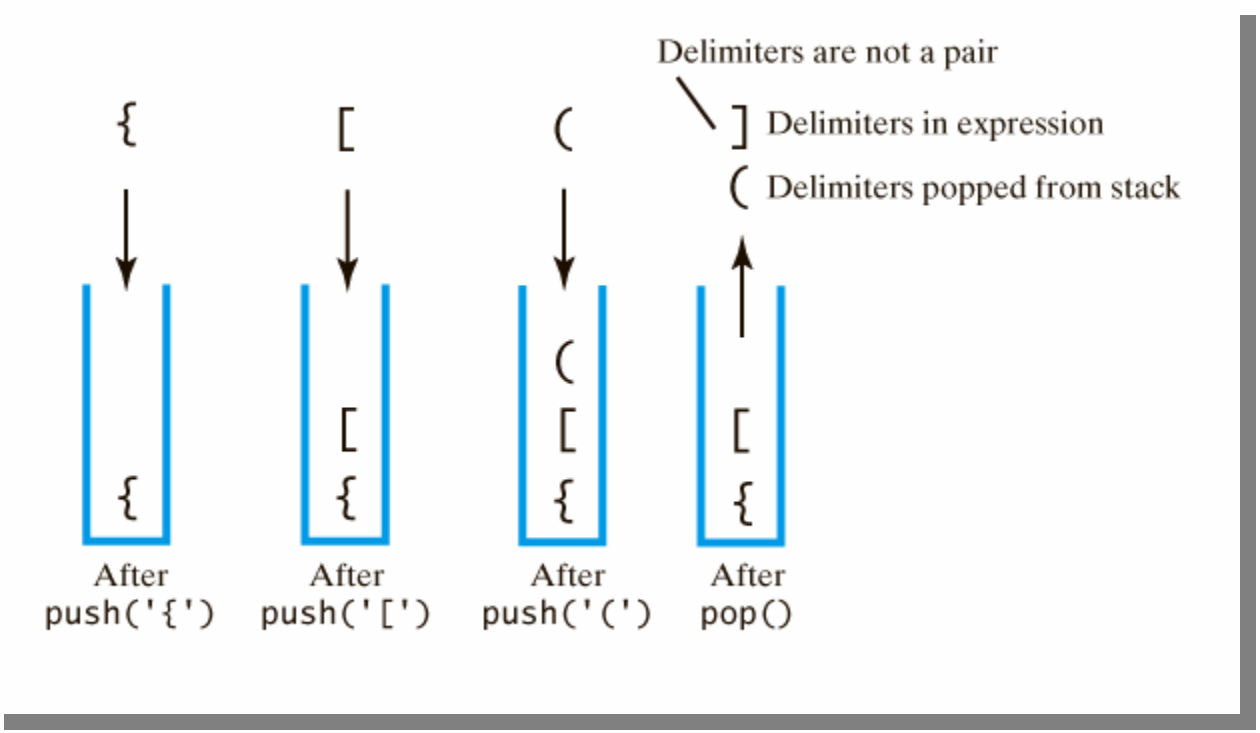


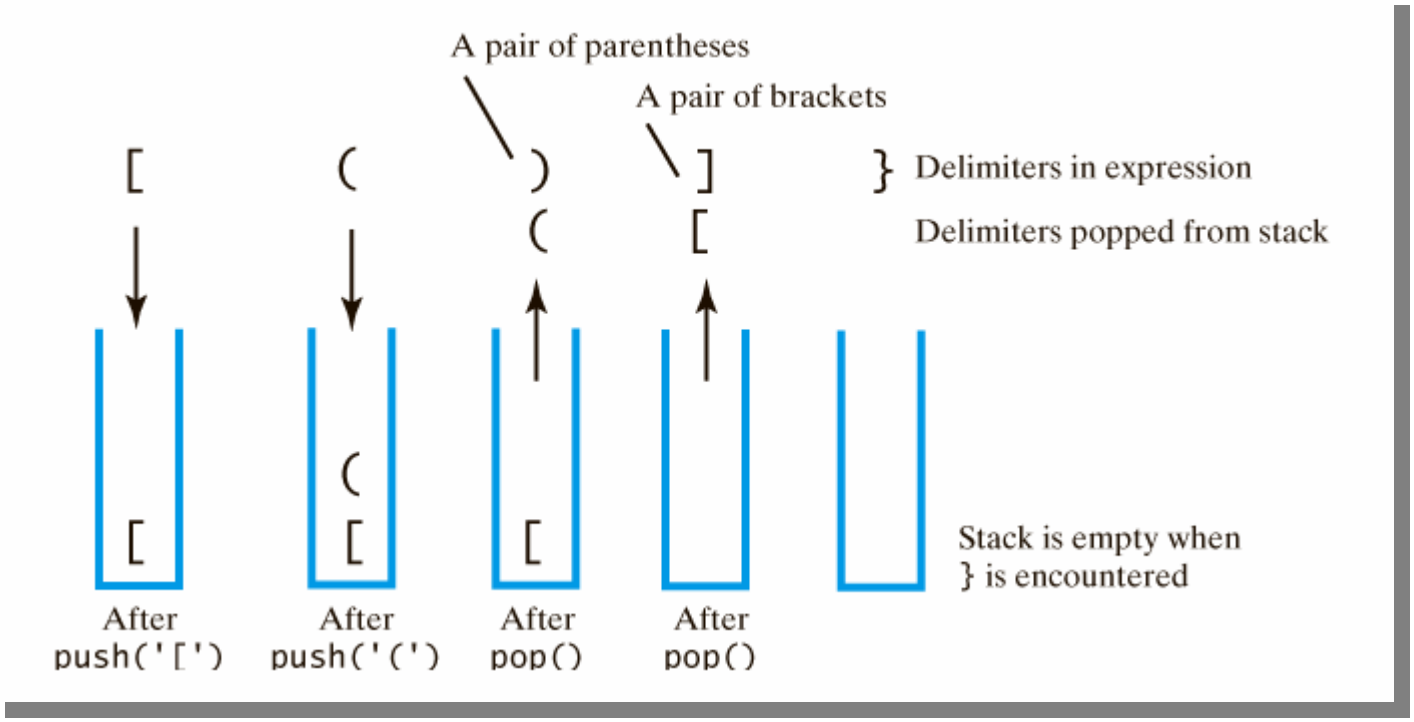
Fig. The contents of a stack during the scan of an expression that contains the balanced delimiters `{[()]}`

Checking for Balanced `()`, `[]`, `{}`



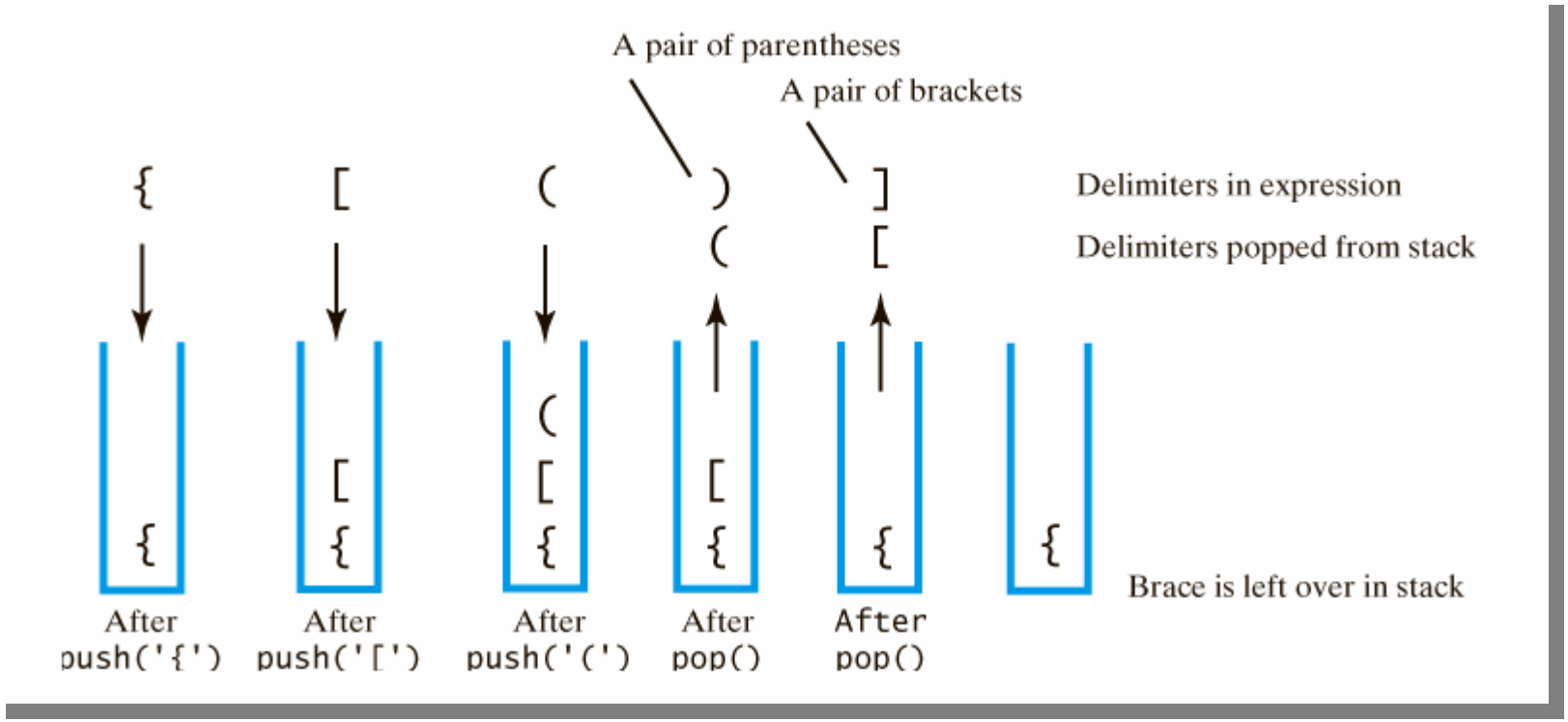
The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[(])}`

Checking for Balanced `()`, `[]`, `{}`



The contents of a stack during the scan of an expression that contains the unbalanced delimiters `[()]}`

Checking for Balanced `()`, `[]`, `{}`



The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[()]`

Checking for Balanced `()`, `[]`, `{}`

```
Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets, and braces in an expression are paired correctly.
isBalanced = true
while ( (isBalanced == true) and not at end of expression)
{       nextCharacter = next character in expression
        switch (nextCharacter)
        {       case '(': case '[': case '{':
                        Push nextCharacter onto stack
                        break
                case ')': case ']': case '}':
                        if (stack is empty)  isBalanced = false
                        else
                        {       openDelimiter = top of stack
                                Pop stack
                                isBalanced = true or false according to whether openDelimiter and
                                        nextCharacter are a pair of delimiters
                        }
                        break
        }
}
if (stack is not empty)  isBalanced = false
return isBalanced
```

Transforming Infix to Postfix

Heuristic method

- *Fully* parenthesize the expression - to represent your desired priorities or the default precedence of the operators
- move each operator to its nearest right parenthesis
- remove parentheses

Transforming Infix to Postfix

Next Character	Postfix	Operator Stack (bottom to top)
a	*a*	
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

Converting the infix expression
`a + b * c` to postfix form

Transforming Infix to Postfix

Next Character	Postfix	Operator Stack (bottom to top)
a	a	
$-$	a	$-$
b	$a\ b$	$-$
$+$	$a\ b\ -$	
	$a\ b\ -$	$+$
c	$a\ b\ -\ c$	$+$
	$a\ b\ -\ c\ +$	

a) Converting infix expression to postfix form:

```
a - b + c
```

Transforming Infix to Postfix

Next Character	Postfix	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^^
c	*a b c*	^^
	a b c ^	^
	a b c ^ ^	

(b) Converting infix expression to postfix form:

```
a ^ b ^ c
```

Infix-to-Postfix Algorithm

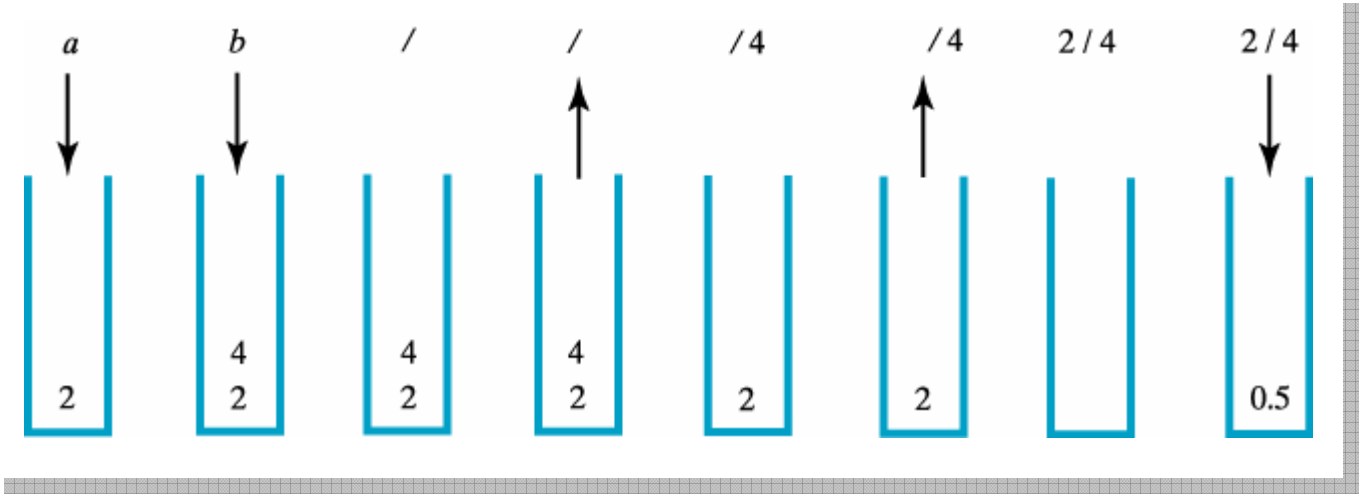
Symbol in Infix	Action
Operand	Append to end of output expression
Operator ^	Push ^ onto stack
Operator +,-, *, or /	Pop operators from stack, append to output expression until stack empty or top has lower precedence than new operator. Then push new operator onto stack
Open parenthesis	Push (onto stack)
Close parenthesis	Pop operators from stack, append to output expression until we pop an open parenthesis. Discard both parentheses.

Transforming Infix to Postfix

Next Character	Postfix	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b /*	
	a b /	*
(	*a b /*	* (
c	*a b / c*	* (
+	*a b / c*	* (+
(	*a b / c*	* (+ (
d	*a b / c d*	* (+ (
−	*a b / c d*	* (+ (−
e	*a b / c d e*	* (+ (−
)	*a b / c d e* −	* (+ (
	a b / c d e −	* (+
)	*a b / c d e* − +	* (
	a b / c d e − +	*
	a b / c d e − + *	

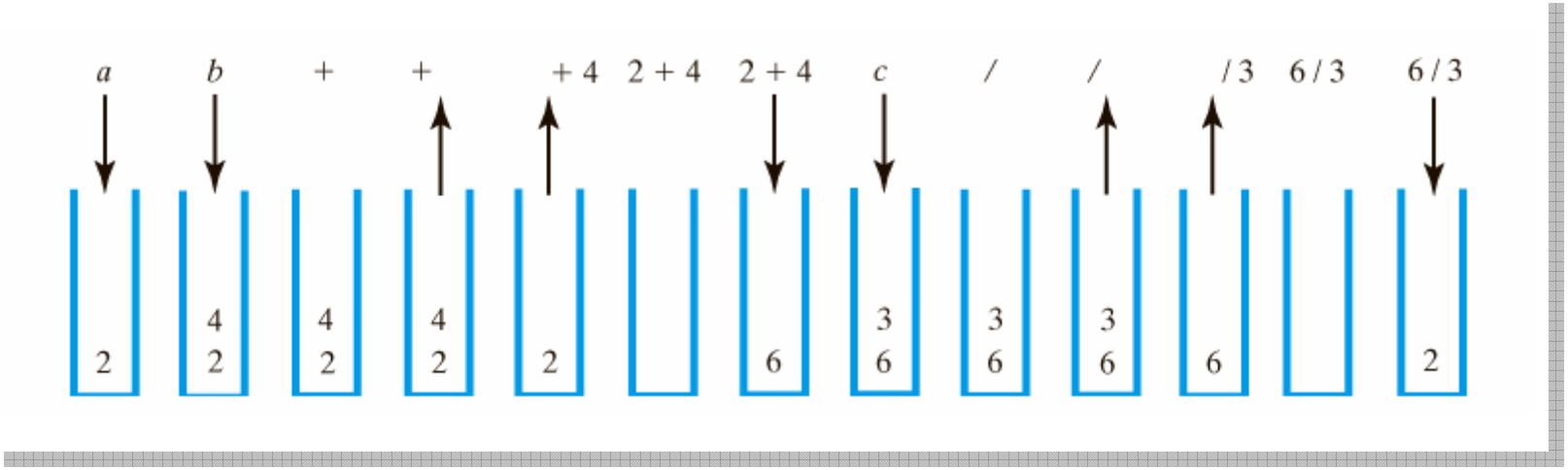
Steps to convert the infix expression
`a / b * ( c + ( d - e ) )` to postfix form.

Evaluating Postfix Expression



The stack during the evaluation of the postfix expression `a b /` when `a` is 2 and `b` is 4

Transforming Infix to Postfix

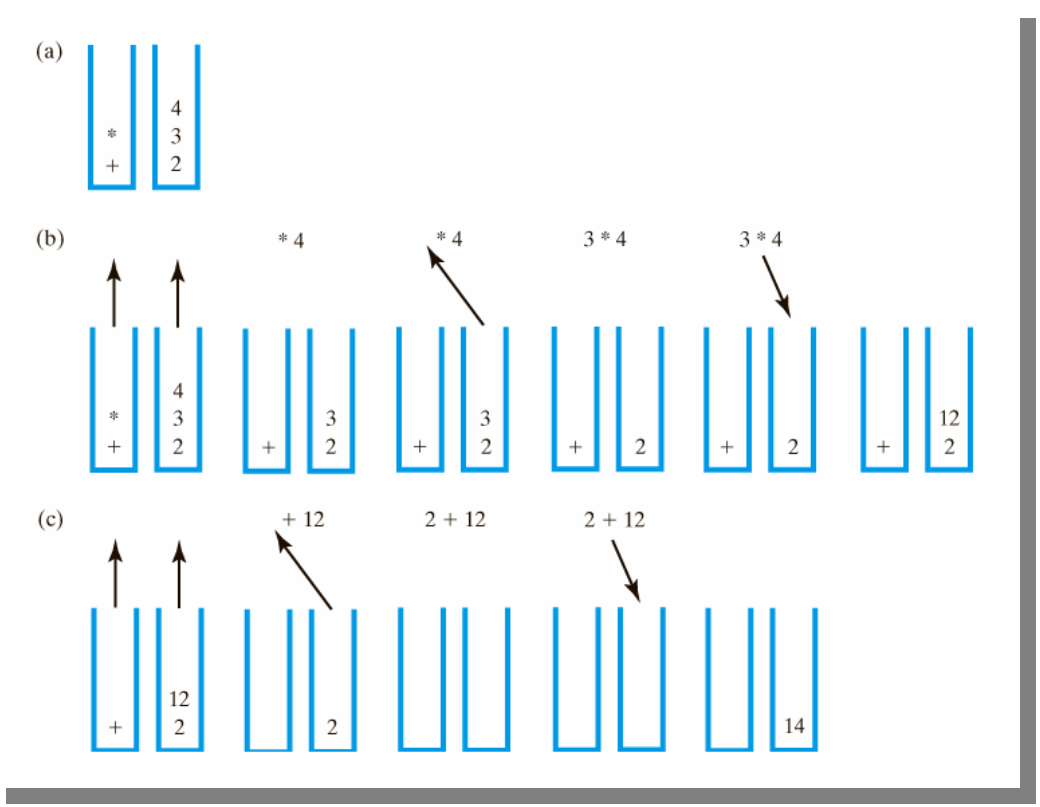


The stack during the evaluation of the postfix expression `a b + c /` when `a` is 2, `b` is 4 and `c` is 3

Transforming Infix to Postfix

```
Algorithm evaluatePostfix(postfix)    // Evaluates a postfix expression.
valueStack = a new empty stack
while (postfix has characters left to parse)
{       nextCharacter = next nonblank character of postfix
        switch (nextCharacter)
        {       case variable:
                        valueStack.push(value of the variable nextCharacter)
                        break
                case '+': case '-': case '*': case '/': case '^':
                        operandTwo = valueStack.pop()
                        operandOne = valueStack.pop()
                        result = the result of the operation in nextCharacter and its operands
                                        operandOne and operandTwo
                        valueStack.push(result)
                        break
                default: break
        }
}
return valueStack.peek()
```

Evaluating Infix Expressions

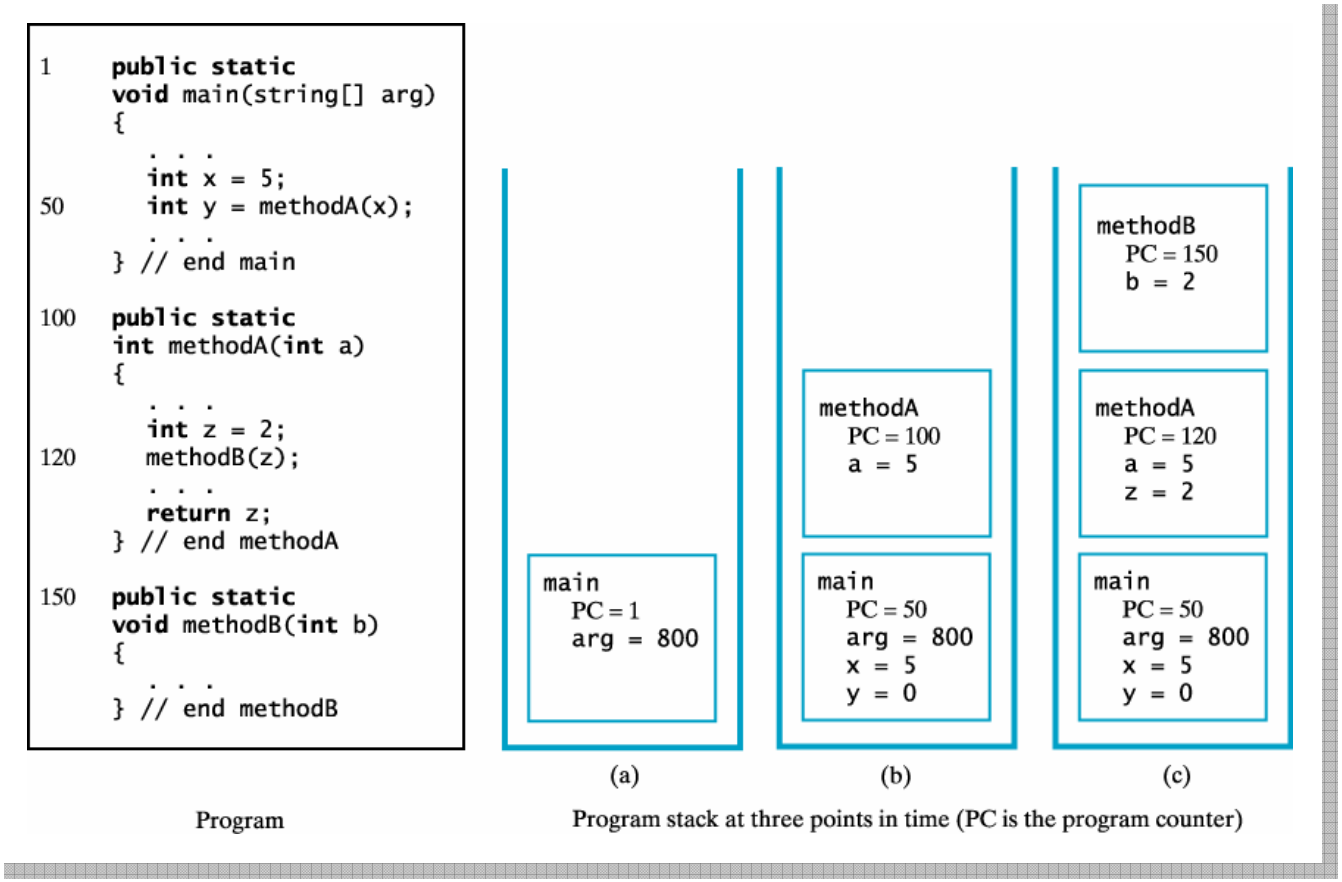


Two stacks during evaluation of a + b * c when
a = 2, b = 3, c = 4; (a) after reaching end of expression;
(b) while performing multiplication;
(c) while performing the addition

The Program Stack

- When a method is called
  - Runtime environment creates activation record
  - Shows method's state during execution
- Activation record pushed onto the program stack (Java stack)
  - Top of stack belongs to currently executing method
  - Next method down is the one that called current method

The Program Stack



The program stack at 3 points in time; (a) when **`main`** begins execution; (b) when **`methodA`** begins execution, (c) when **`methodB`** begins execution.

Recursive Methods

- A recursive method making many recursive calls
  - Places many activation records in the program stack
  - Thus the reason recursive methods can use much memory
- Possible to replace recursion with iteration by using a stack

Using a Stack Instead of Recursion

```
public boolean contains(Object desiredItem)
{       return binarySearch(0, length-1, (Comparable)desiredItem);      } // end contains

/** Task: Searches entry[first] through entry[last] for
      desiredItem, where the array entry is a data field.
* @param first an integer index >= 0 and < length of list
* @param last an integer index >= 0 and < length of list
* @param desiredItem the object to be found in the array
* @return true if desiredItem is found */
private boolean binarySearch(int first, int last, Comparable desiredItem)
{       boolean found;
        int mid = (first + last)/2;
        if (first > last)                                       found = false;
        else if (desiredItem.equals(entry[mid]))        found = true;
        else if (desiredItem.compareTo(entry[mid]) < 0)
                found = binarySearch(first, mid-1, desiredItem);
        else
                found = binarySearch(mid+1, last, desiredItem);
        return found;
} // end binarySearch
```