Stacks

Chapter 5

Contents

- Specifications of the ADT Stack
- Using a Stack to Process Algebraic Expressions
  - A Problem Solved: Checking for Balanced Delimiters in an Infix Algebraic Expression
  - A Problem Solved: Transforming an Infix Expression to a Postfix Expression
  - A Problem Solved: Evaluating Postfix Expressions
  - A Problem Solved: Evaluating Infix Expressions

Contents

- The Program Stack
- Java Class Library: The Class **`Stack`**

Objectives

- Describe operations of ADT stack
- Use stack to decide whether delimiters in an algebraic expression are paired correctly
- Use stack to convert infix expression to postfix expression

Objectives

- Use stack to evaluate postfix expression
- Use stack to evaluate infix expression
- Use a stack in a program
- Describe how Java run-time environment uses stack to track execution of methods

Specifications of a Stack

- Organizes entries according to order added
- All additions added to one end of stack
  - Added to “top”
  - Called a “push”
- Access to stack restricted
  - Access only top entry
  - Remove called a “pop”

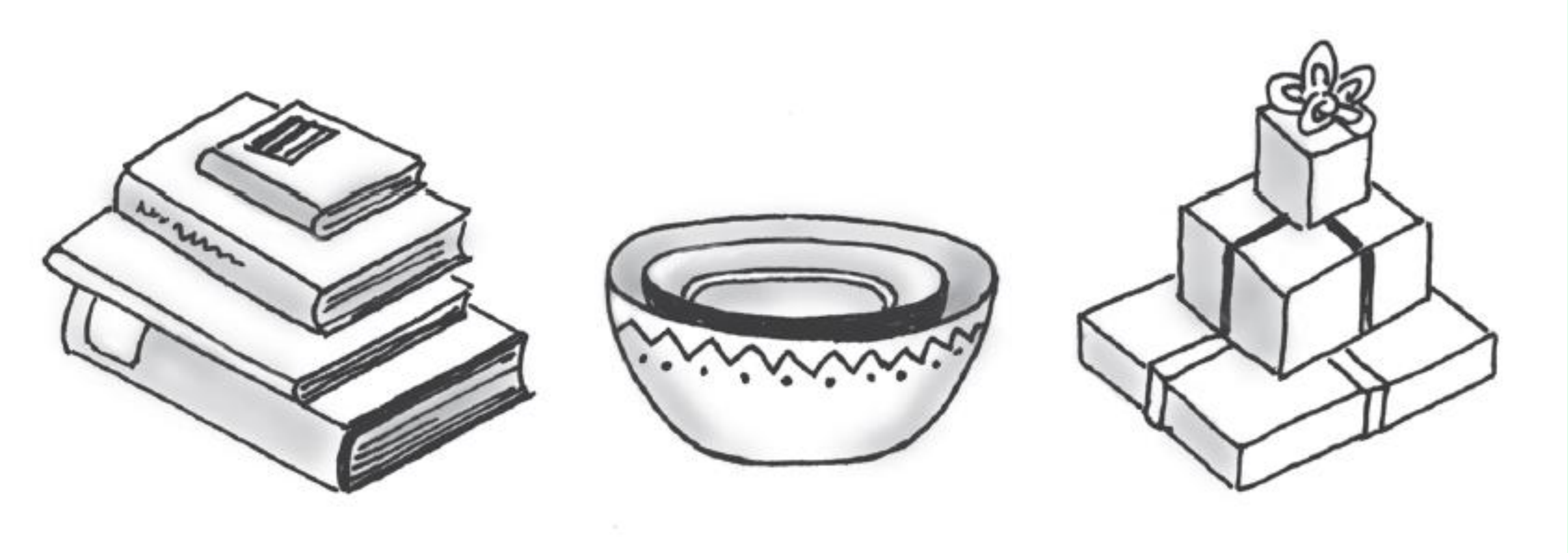
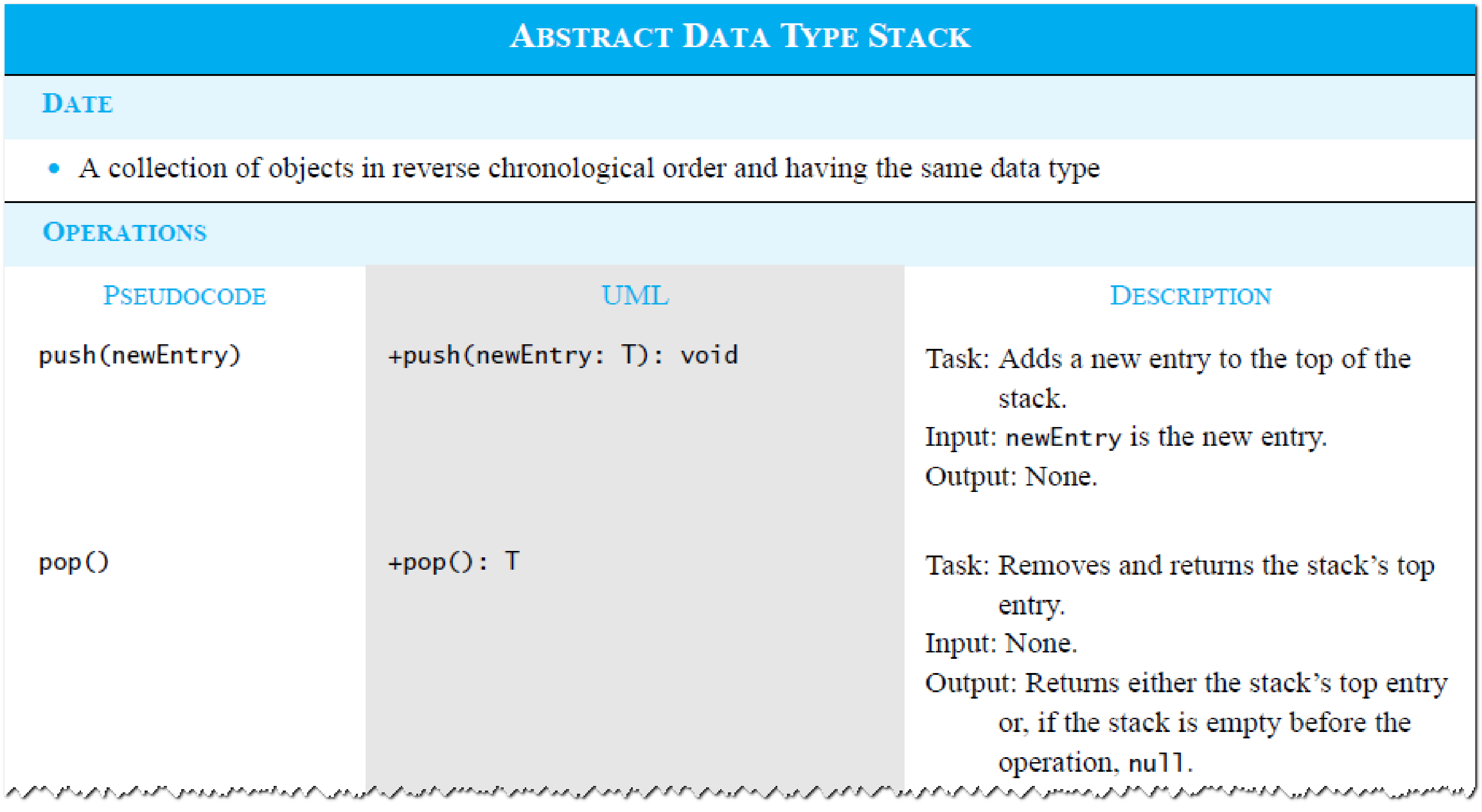


Figure 5-1 Some familiar stacks



Abstract Data Type Stack

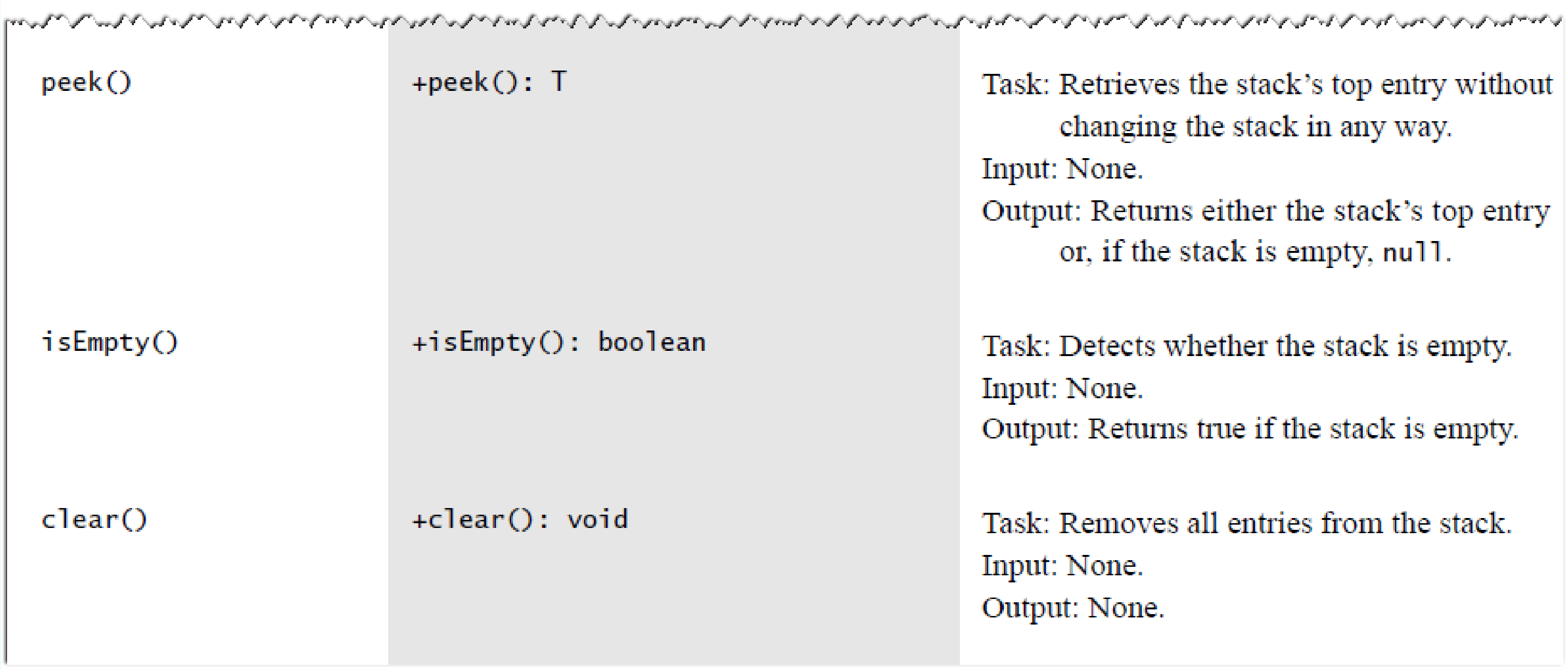
Date

- A collection of objects in reverse chronological order and having the same data type

Operations

Pseudocode	UML	Description
`push(newEntry)`	`+push(newEntry: T): void`	Task: Adds a new entry to the top of the stack. Input: `newEntry` is the new entry. Output: None.
`pop()`	`+pop(): T`	Task: Removes and returns the stack's top entry. Input: None. Output: Returns either the stack's top entry or, if the stack is empty before the operation, `null`.

ADT Stack



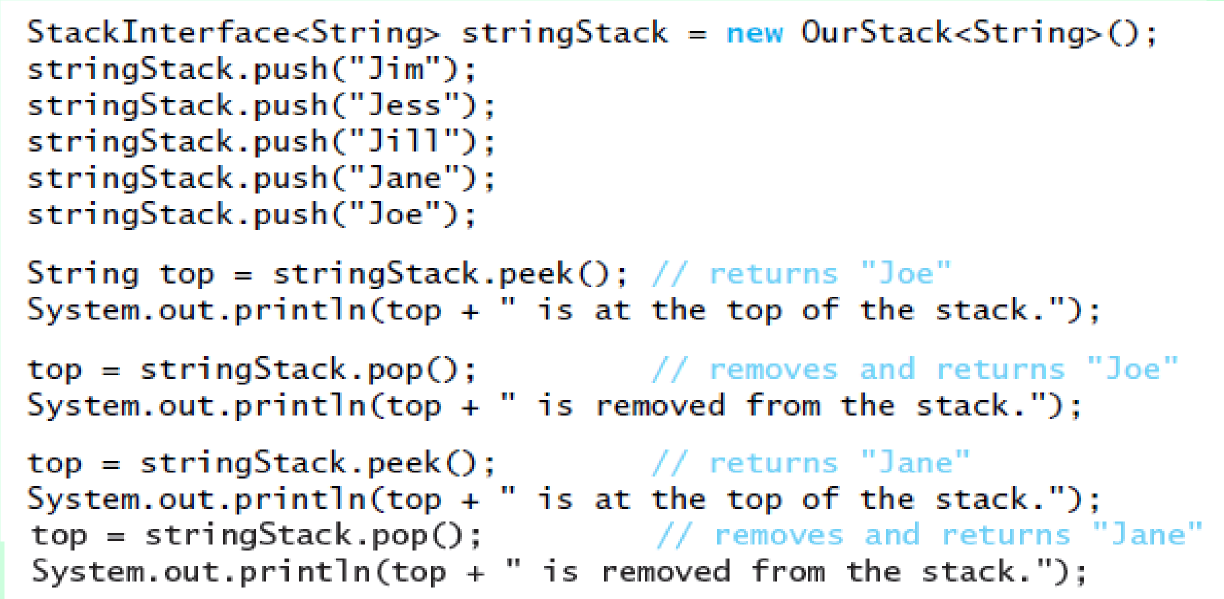
peek()	+peek(): T	Task: Retrieves the stack's top entry without changing the stack in any way. Input: None. Output: Returns either the stack's top entry or, if the stack is empty, null.
isEmpty()	+isEmpty(): boolean	Task: Detects whether the stack is empty. Input: None. Output: Returns true if the stack is empty.
clear()	+clear(): void	Task: Removes all entries from the stack. Input: None. Output: None.

ADT Stack

Specify Class Stack

- Interface
  - Note source code, Listing 5-1
- Example usage

Note: Code listing files must be in same folder as PowerPoint files for links to work



```
StackInterface<String> stringStack = new OurStack<String>();
stringStack.push("Jim");
stringStack.push("Jess");
stringStack.push("Jill");
stringStack.push("Jane");
stringStack.push("Joe");

String top = stringStack.peek(); // returns "Joe"
System.out.println(top + " is at the top of the stack.");

top = stringStack.pop();         // removes and returns "Joe"
System.out.println(top + " is removed from the stack.");

top = stringStack.peek();        // returns "Jane"
System.out.println(top + " is at the top of the stack.");
top = stringStack.pop();         // removes and returns "Jane"
System.out.println(top + " is removed from the stack.");
```

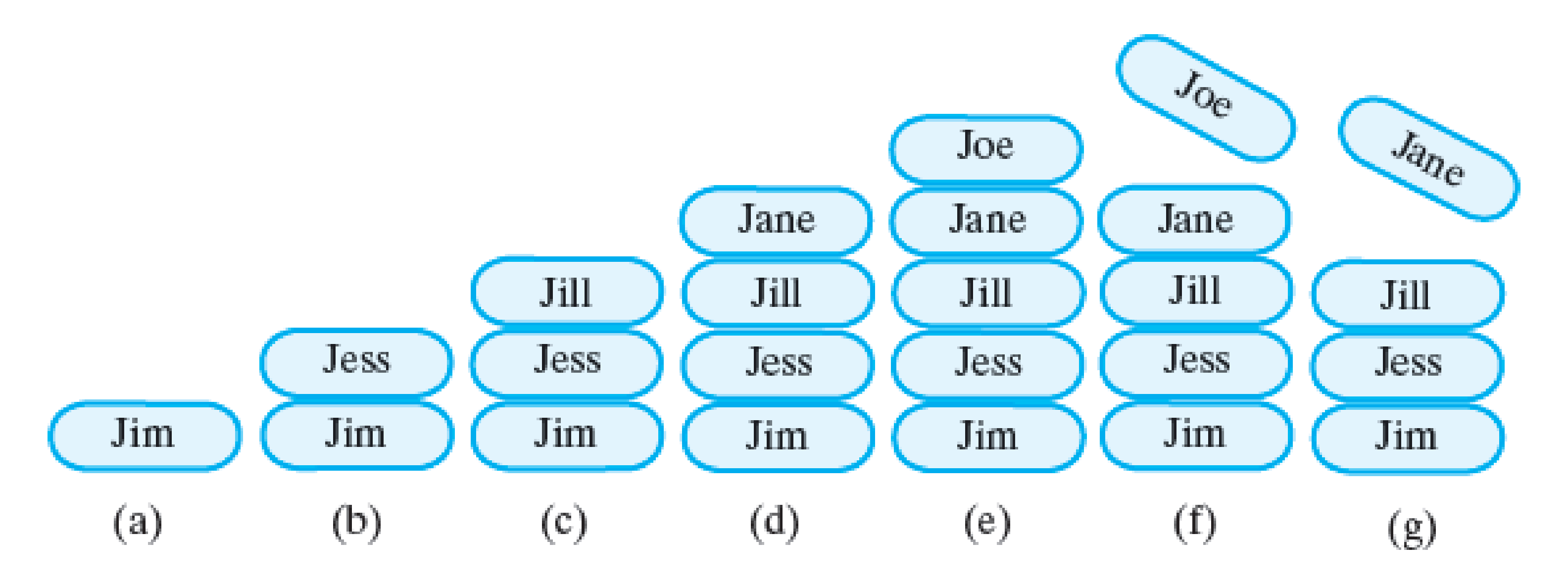
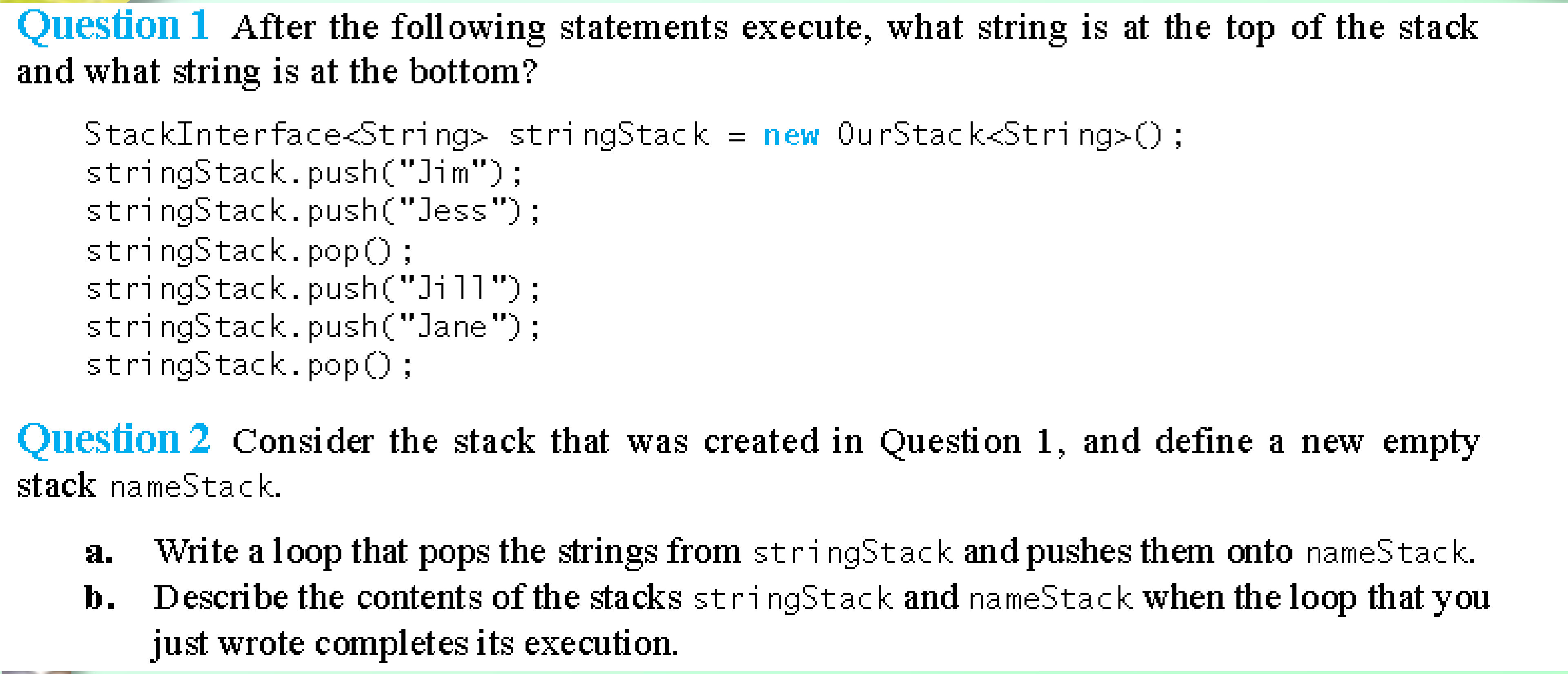


Figure 5-2 A stack of strings after (a) **`push`** adds Jim; (b) **`push`** adds Jess; (c) **`push`** adds Jill; (d) **`push`** adds Jane; (e) **`push`** adds Joe; (f) **`pop`** retrieves and removes Joe; (g) **`pop`** retrieves and removes Jane



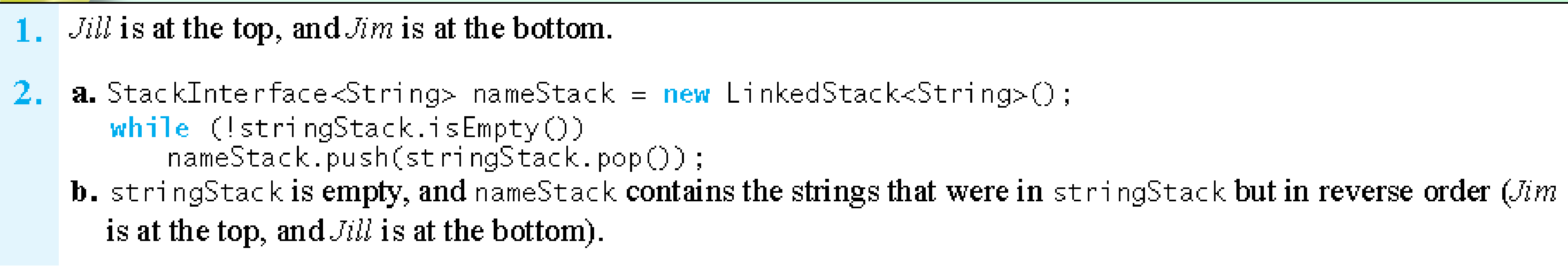
Question 1 After the following statements execute, what string is at the top of the stack and what string is at the bottom?

```
StackInterface<String> stringStack = new OurStack<String>();
stringStack.push("Jim");
stringStack.push("Jess");
stringStack.pop();
stringStack.push("Jill");
stringStack.push("Jane");
stringStack.pop();
```

Question 2 Consider the stack that was created in Question 1, and define a new empty stack `nameStack`.

a. Write a loop that pops the strings from `stringStack` and pushes them onto `nameStack`.

b. Describe the contents of the stacks `stringStack` and `nameStack` when the loop that you just wrote completes its execution.



1. *Jill* is at the top, and *Jim* is at the bottom.

2. **a.**

```
StackInterface<String> nameStack = new LinkedStack<String>();
while (!stringStack.isEmpty())
   nameStack.push(stringStack.pop());
```

b. `stringStack` is empty, and `nameStack` contains the strings that were in `stringStack` but in reverse order (*Jim* is at the top, and *Jill* is at the bottom).

Using a Stack to Process Algebraic Expressions

- Algebraic expressions composed of
  - Operands (variables, constants)
  - Operators (+, -, /, *, ^)
- Operators can be unary or binary
- Different precedence notations
  - Infix a + b
  - Prefix + a b
  - Postfix a b +

Using a Stack to Process Algebraic Expressions

- Precedence must be maintained
  - Order of operators
  - Use of parentheses (must be balanced)
- Use stacks to evaluate parentheses usage
  - Scan expression
  - Push symbols
  - Pop symbols

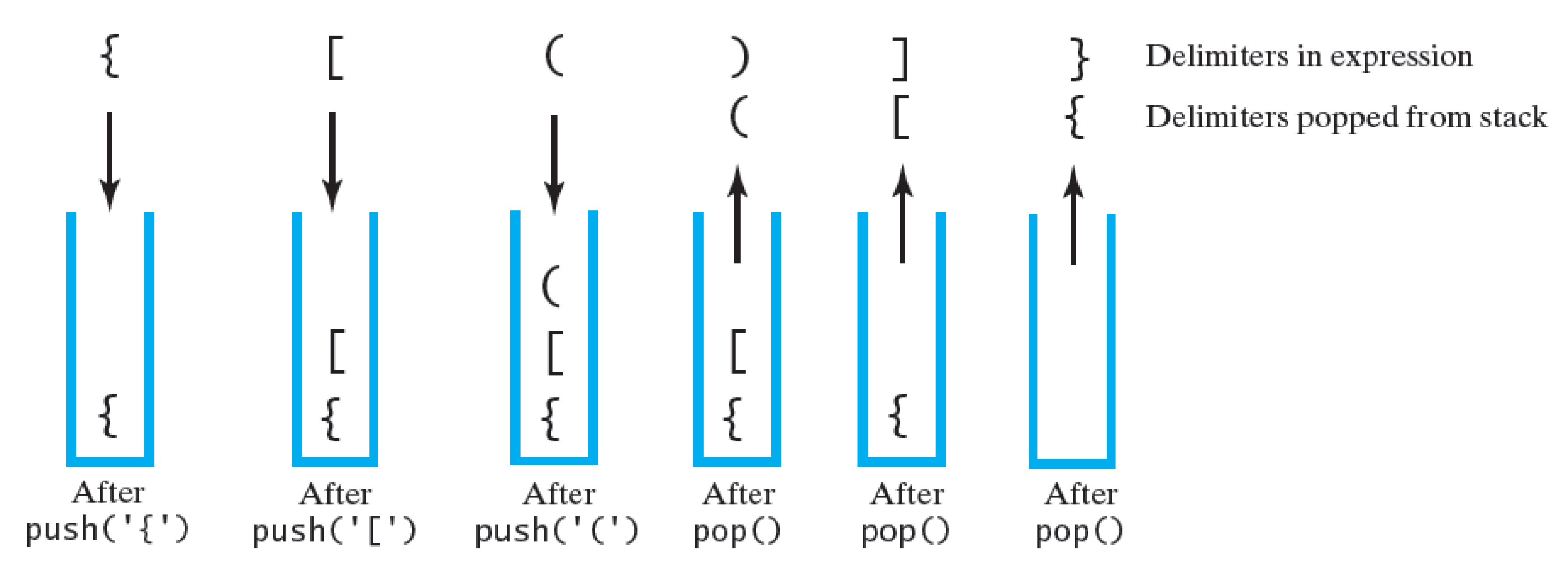


Figure 5-3 The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }

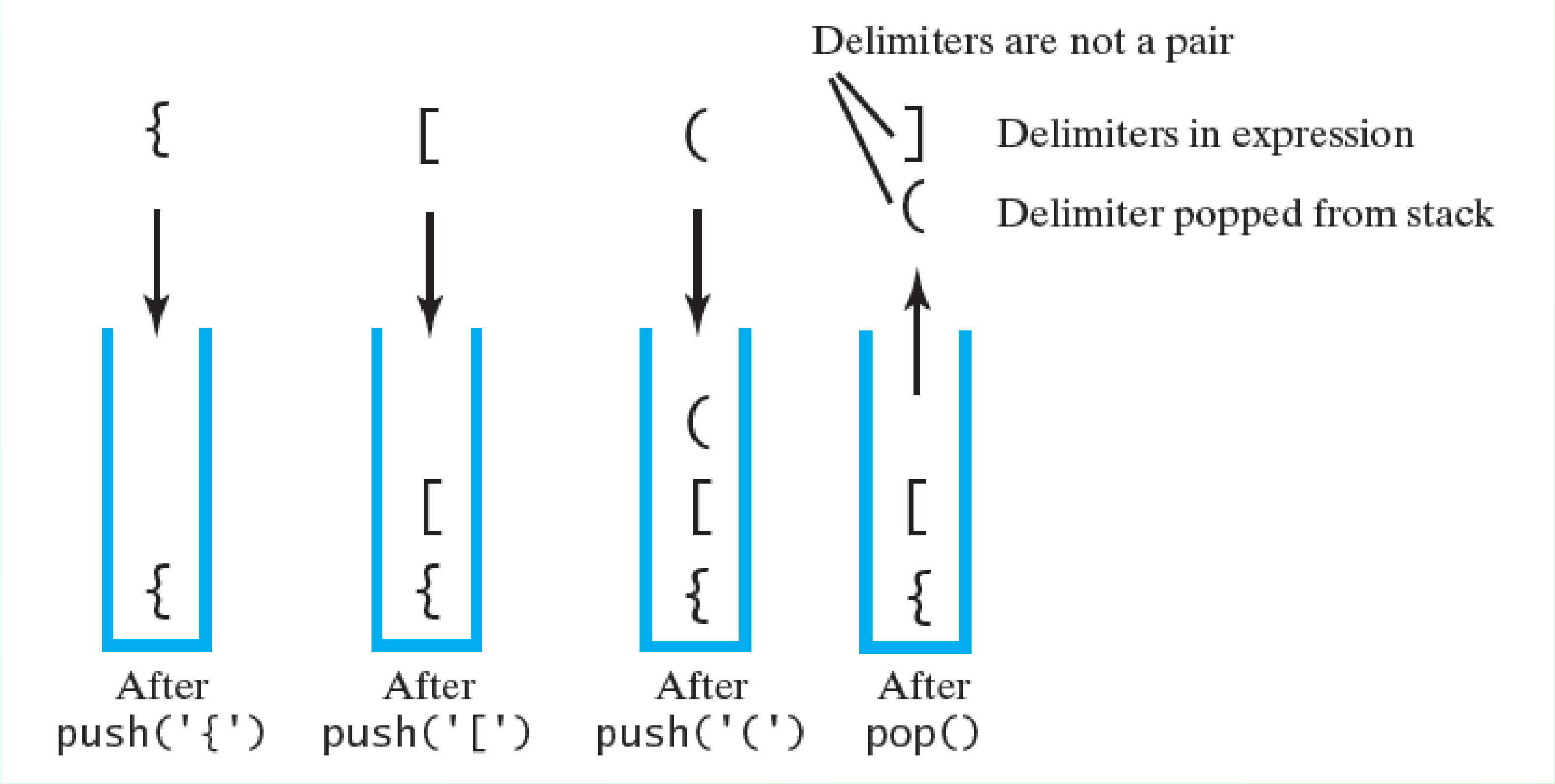


Figure 5-4 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( ]) }

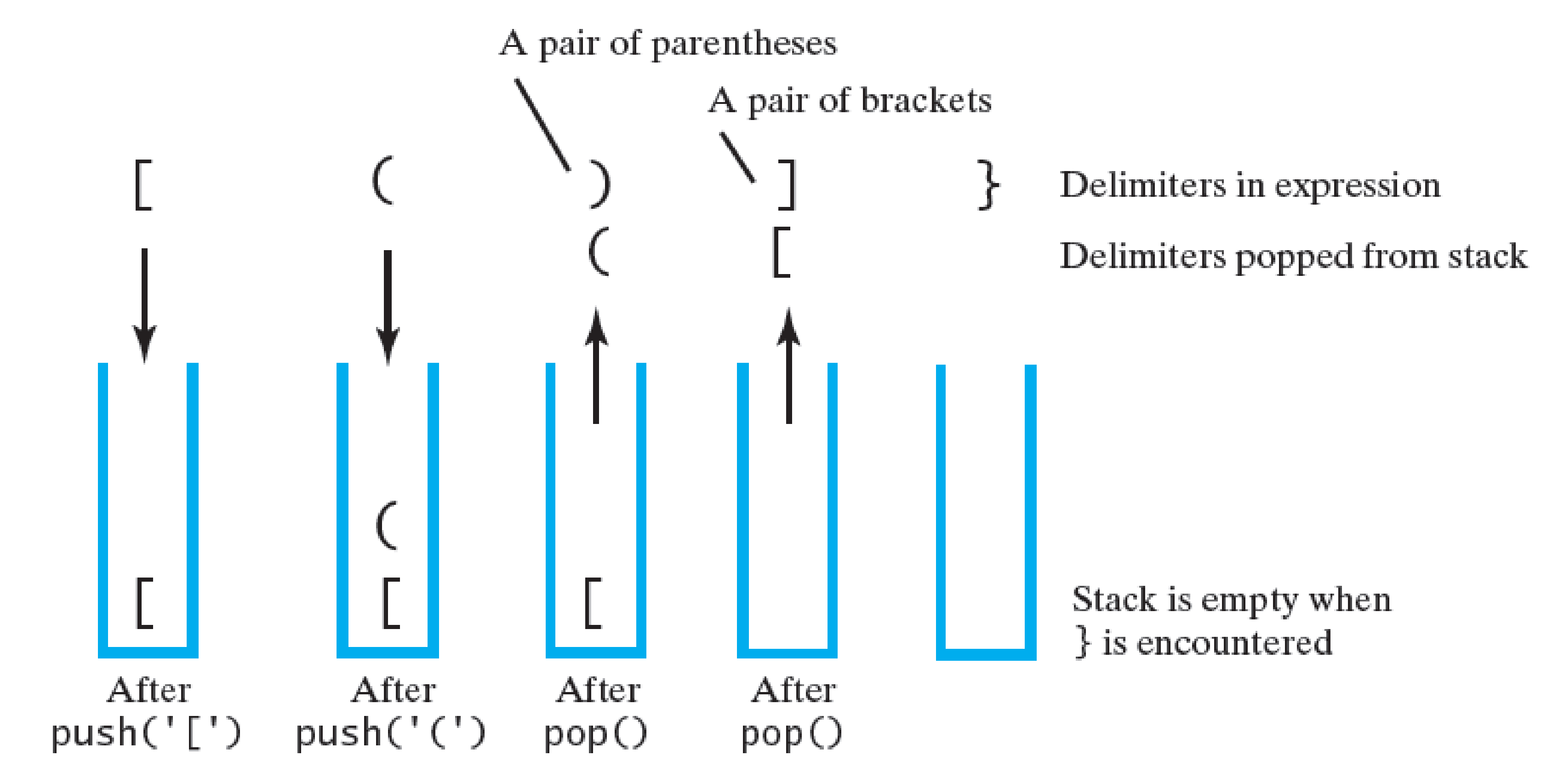


Figure 5-5 The contents of a stack during the scan of an expression that contains the unbalanced delimiters [( )] }

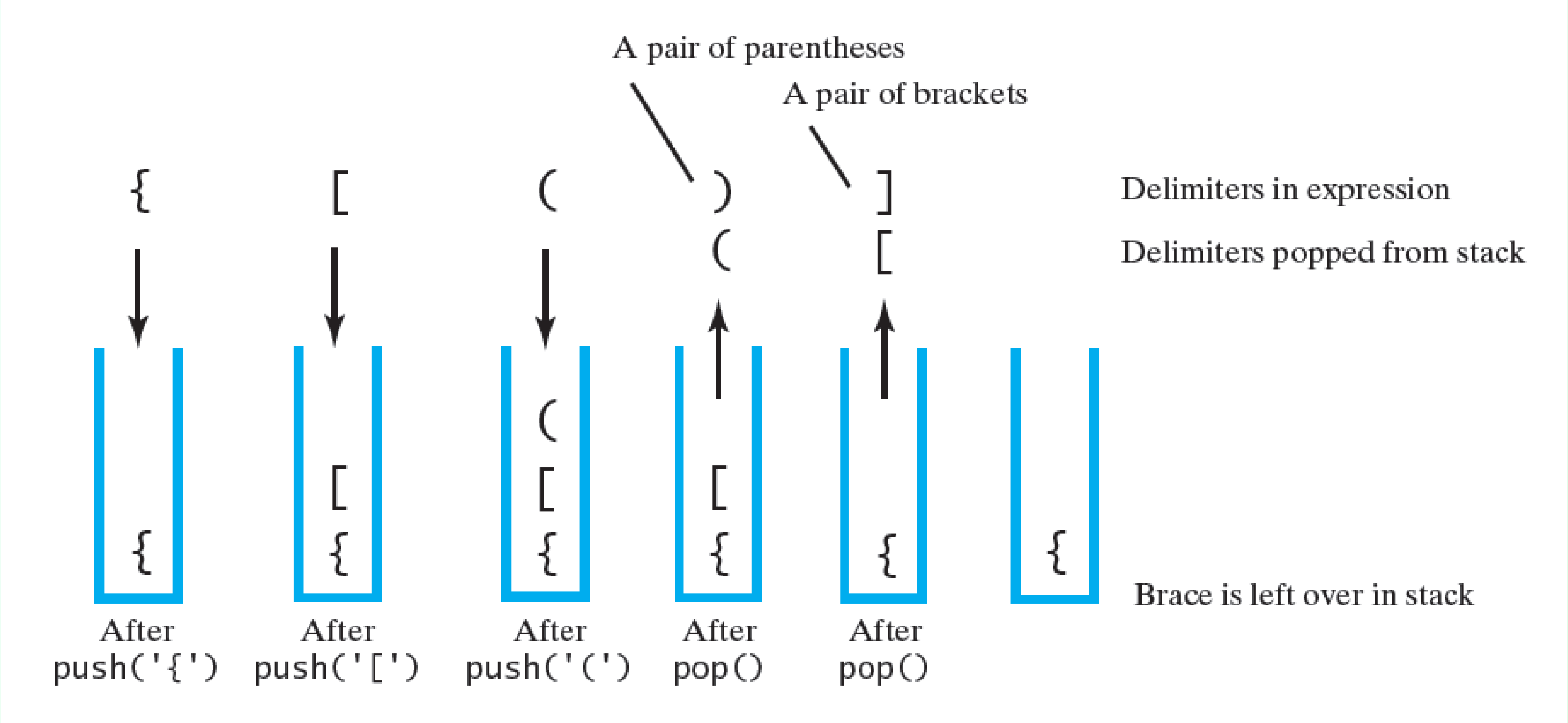
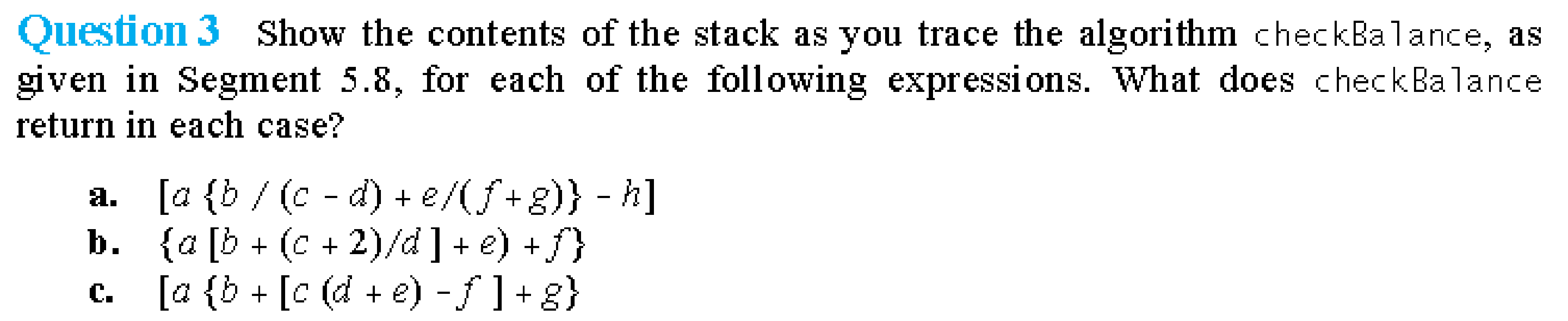


Figure 5-6 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( )]

- Implementation of algorithm to check for balanced parentheses, Listing 5-2



Question 3 Show the contents of the stack as you trace the algorithm `checkBalance`, as given in Segment 5.8, for each of the following expressions. What does `checkBalance` return in each case?

- **a.** $[a \{b / (c - d) + e/(f + g)\} - h]$
- **b.** $\{a [b + (c + 2)/d] + e) + f\}$
- **c.** $[a \{b + [c (d + e) - f] + g\}$

3. The following stacks are shown bottom to top when read from left to right:

a. [

[{

[{ (

[{

{ { (

[{

[

empty

b. {

{ [

{ [(

{ [

{

c. [

[{

[{ [

[{ [(

[{ [

[{

[

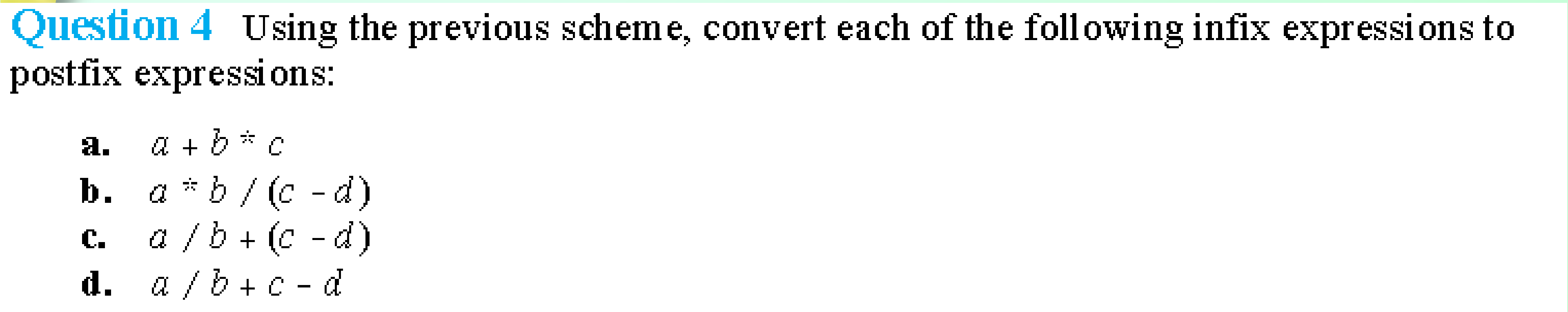
The algorithm `checkBalance` returns true for the expression in Part *a* and false for the other two.

Infix to Postfix

- Manual algorithm for converting infix to postfix (a + b) * c
  - Write with parentheses to force correct operator precedence ((a + b) * c)
  - Move operator to right inside parentheses

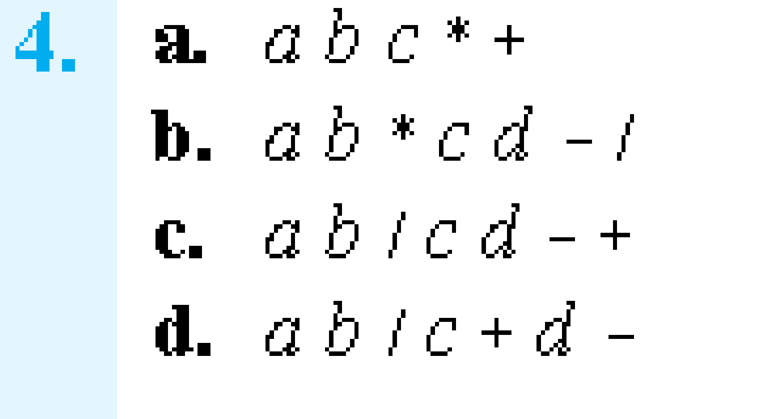
    ((a b +) c *)
  - Remove parentheses

    a b + c *



Question 4 Using the previous scheme, convert each of the following infix expressions to postfix expressions:

- **a.** $a + b * c$
- **b.** $a * b / (c - d)$
- **c.** $a / b + (c - d)$
- **d.** $a / b + c - d$



4.

a. $a\ b\ c\ *\ +$

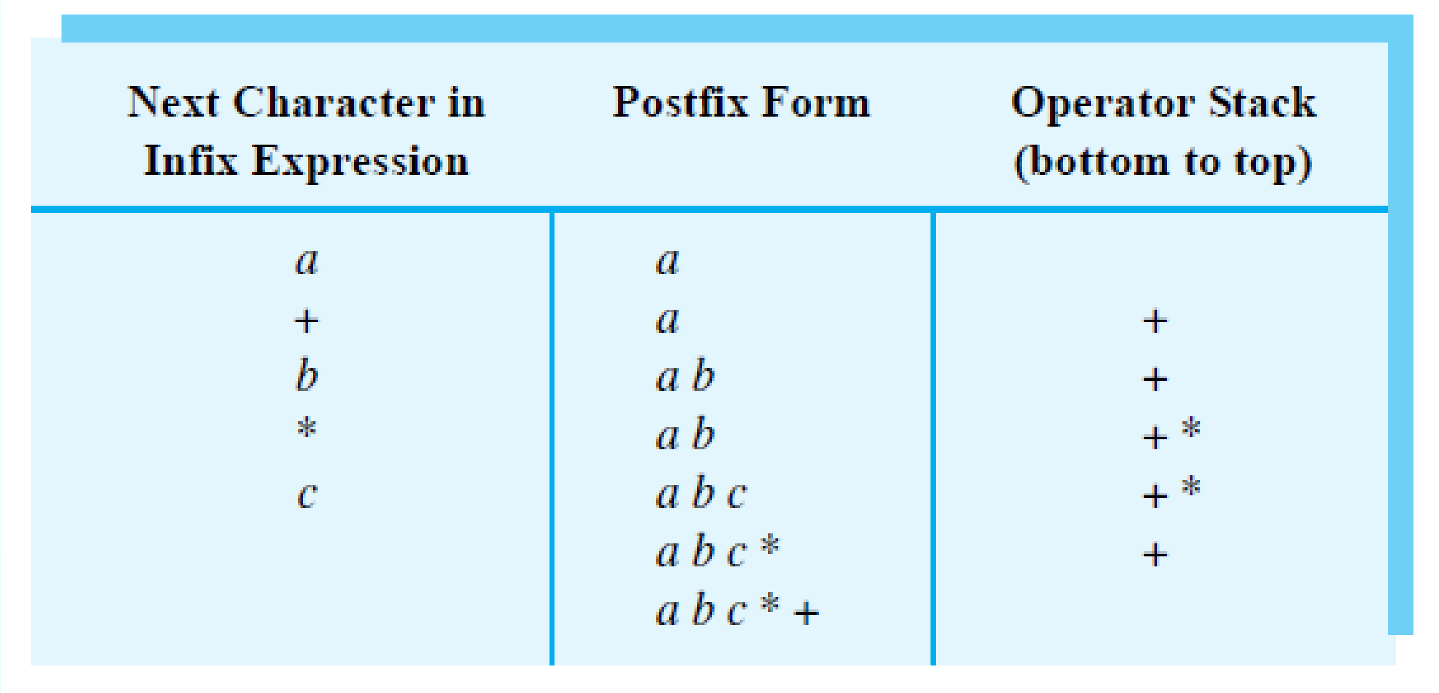
b. $a\ b\ *\ c\ d\ -\ /$

c. $a\ b\ /\ c\ d\ -\ +$

d. $a\ b\ /\ c\ +\ d\ -$

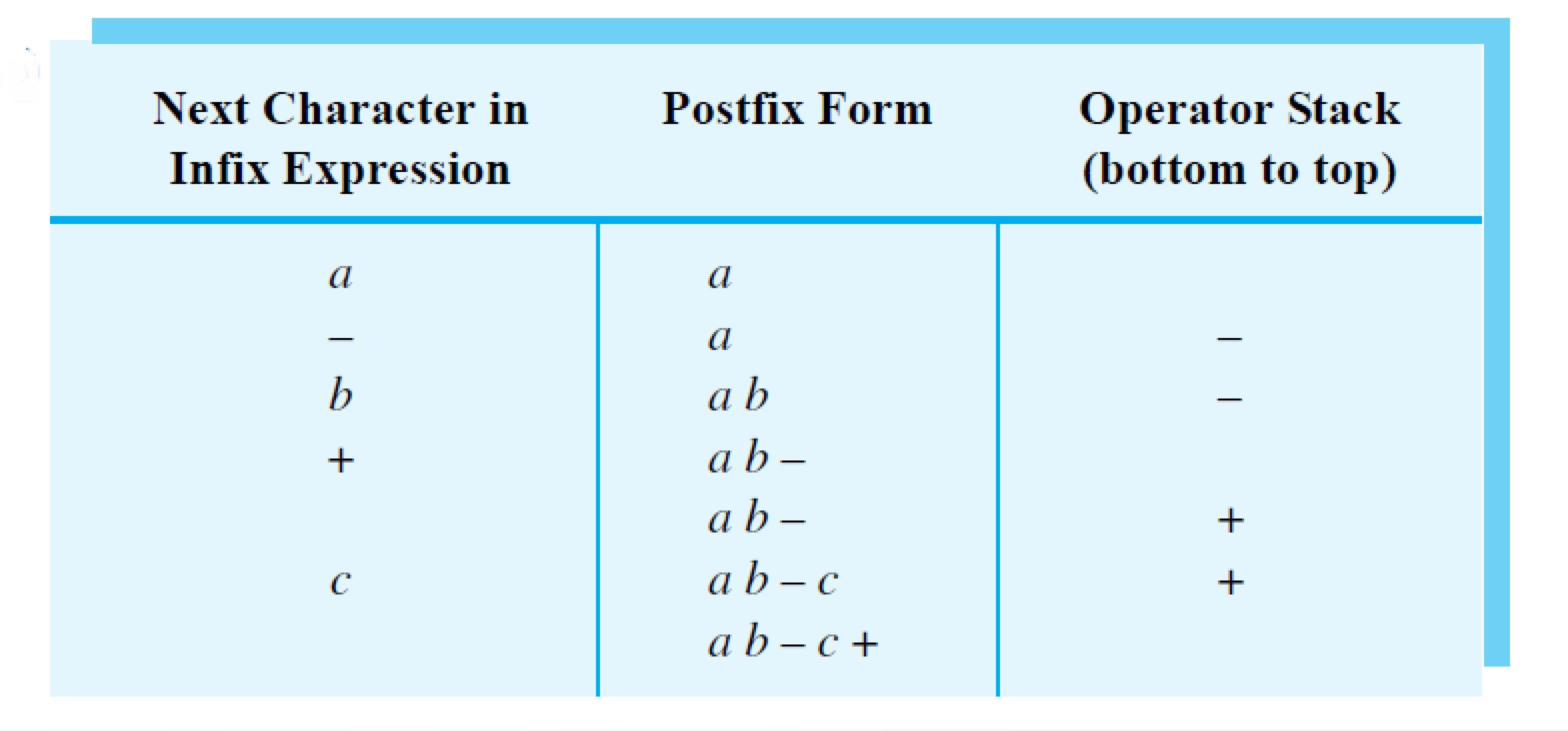
Infix to Postfix

- Algorithm basics
  - Scan expression left to right
  - When operand found, place at end of new expression
  - When operator found, save to determine new position



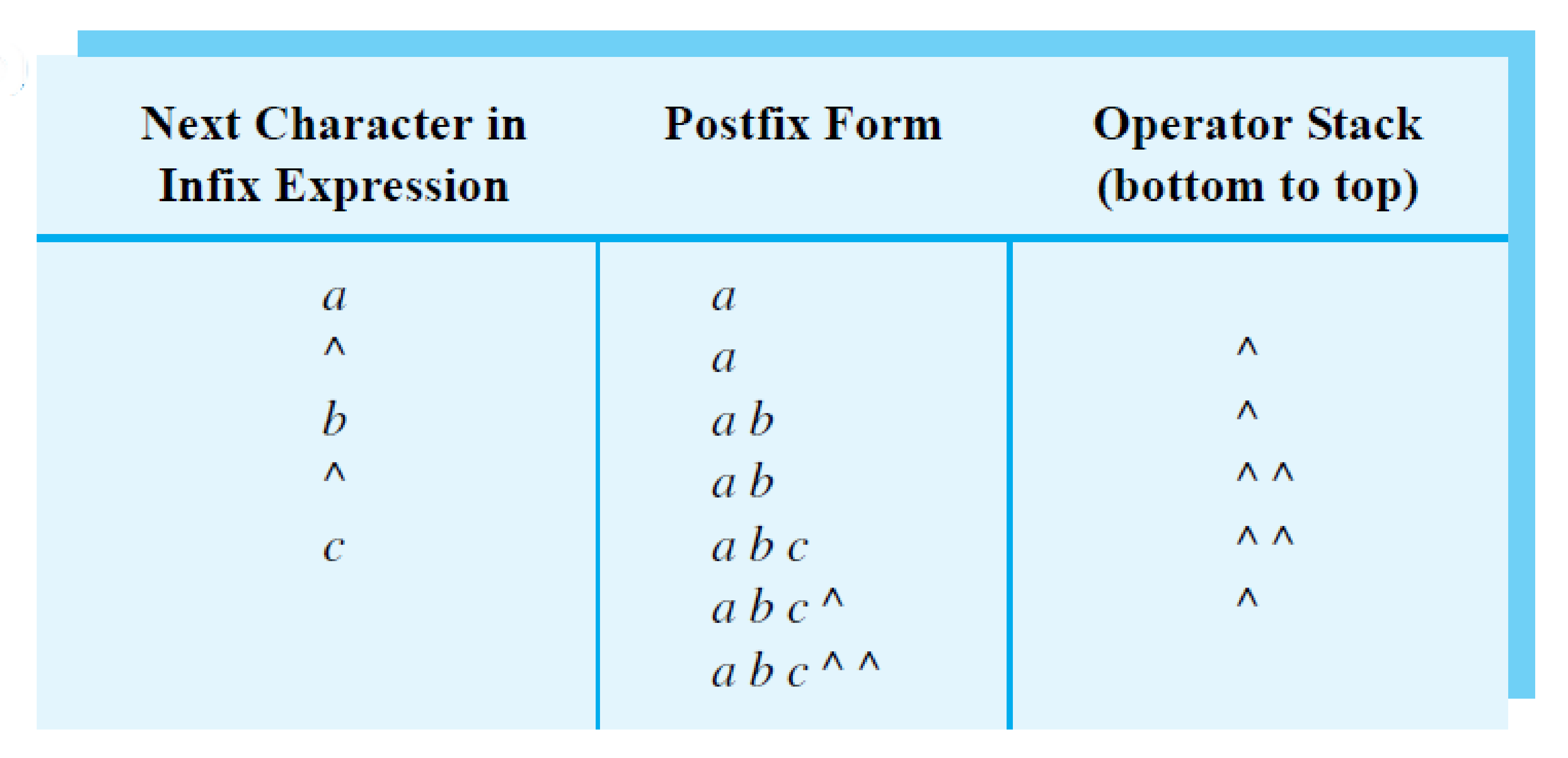
Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

Figure 5-7 Converting the infix expression a + b * c to postfix form



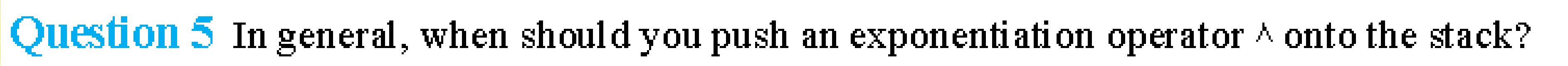
Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
–	*a*	–
b	*a b*	–
+	*a b* –	
	a b –	+
c	*a b* – *c*	+
	a b – *c* +	

Figure 5-8 Converting an infix expression to postfix form: a - b + c

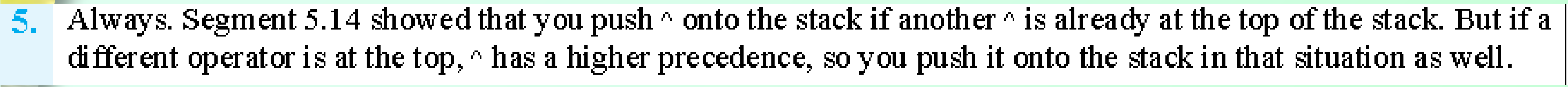


Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^ ^
c	*a b c*	^ ^
	a b c ^	^
	a b c ^ ^	

Figure 5-8 Converting an infix expression to postfix form: a ^ b ^ c



Question 5 In general, when should you push an exponentiation operator ^ onto the stack?



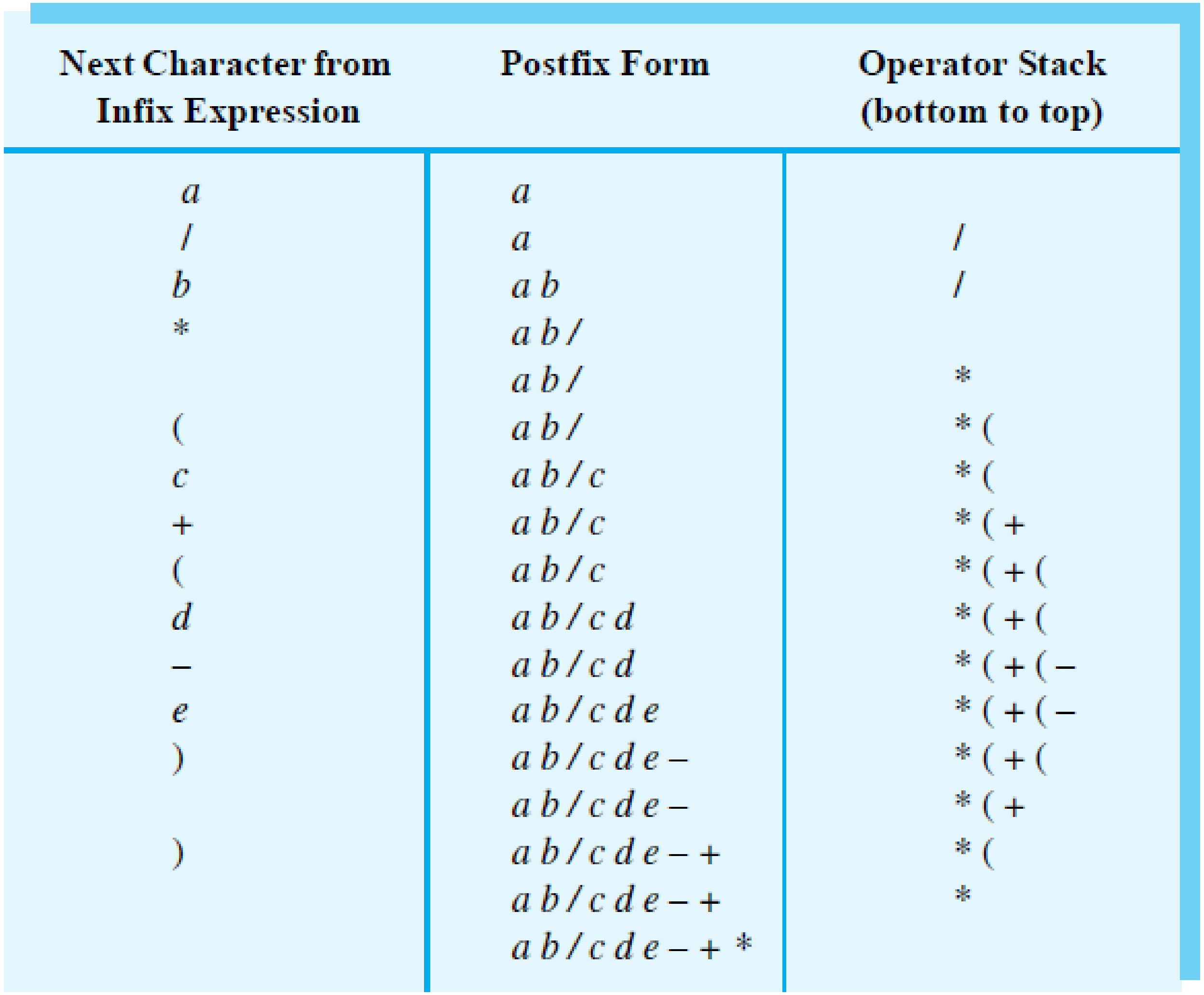
5. Always. Segment 5.14 showed that you push ^ onto the stack if another ^ is already at the top of the stack. But if a different operator is at the top, ^ has a higher precedence, so you push it onto the stack in that situation as well.

Infix to Postfix Conversion

1. Operand
   - Append to end of output expression
2. Operator ^
   - Push ^ onto stack
3. Operators +, -, *, /
   - Pop from stack, append to output expression
   - Until stack empty or top operator has lower precedence than new operator
   - Then push new operator onto stack

Infix to Postfix Conversion

4. Open parenthesis
   - Push (onto stack
5. Close parenthesis
   - Pop operators from stack and append to output
   - Until open parenthesis is popped.
   - Discard both parentheses

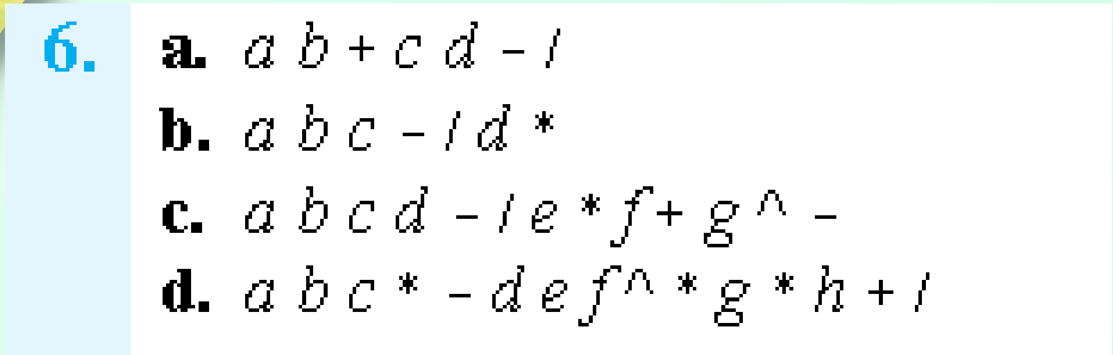


Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
/	a	/
b	a b	/
*	a b /	
	a b /	*
(	a b /	* (
c	a b / c	* (
+	a b / c	* (+
(	a b / c	* (+ (
d	a b / c d	* (+ (
–	a b / c d	* (+ (–
e	a b / c d e	* (+ (–
)	a b / c d e –	* (+ (
	a b / c d e –	* (+
)	a b / c d e – +	* (
	a b / c d e – +	*
	a b / c d e – + *	

FIGURE 5-9 The steps in converting the infix expression a / b * (c + (d - e)) to postfix form

Question 6 Using the previous algorithm, represent each of the following infix expressions as a postfix expression:

- **a.** $(a + b) / (c - d)$
- **b.** $a / (b - c) * d$
- **c.** $a - (b / (c - d) * e + f) \wedge g$
- **d.** $(a - b * c) / (d * e \wedge f * g + h)$



6.

a. *a b* + *c d* − /

b. *a b c* − / *d* *

c. *a b c d* − / *e* * *f* + *g* ^ −

d. *a b c* * − *d e f* ^ * *g* * *h* + /

Evaluating Postfix Expressions

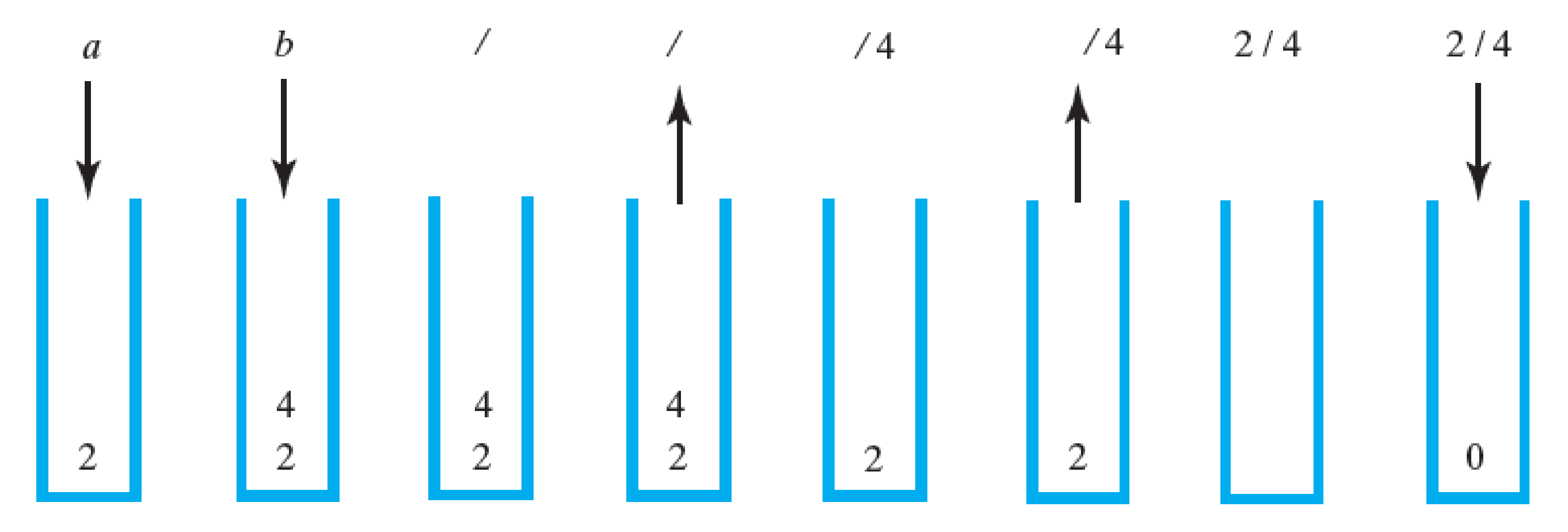


FIGURE 5-10 The stack during the evaluation of the postfix expression a b / when a is 2 and b is 4

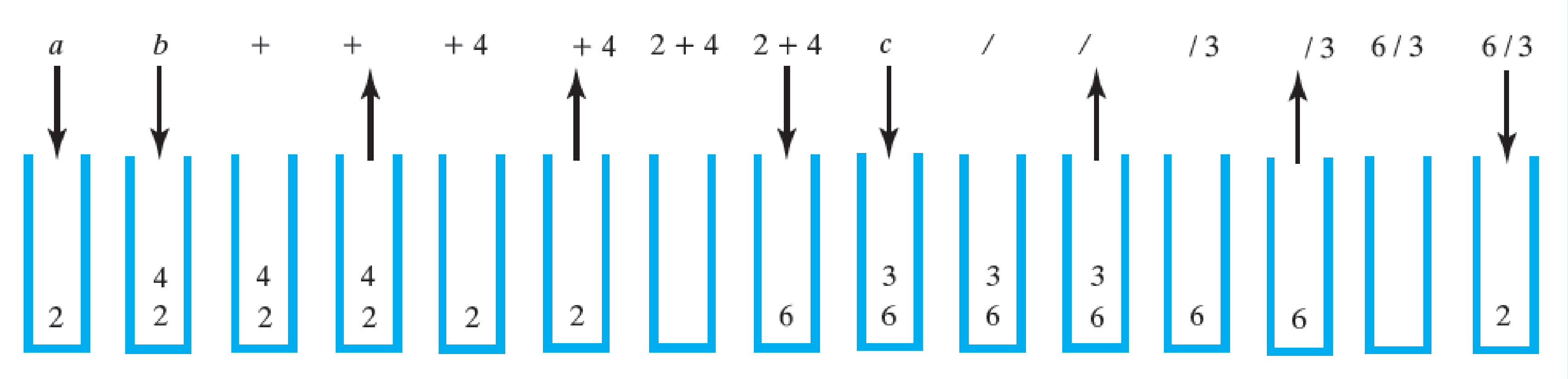
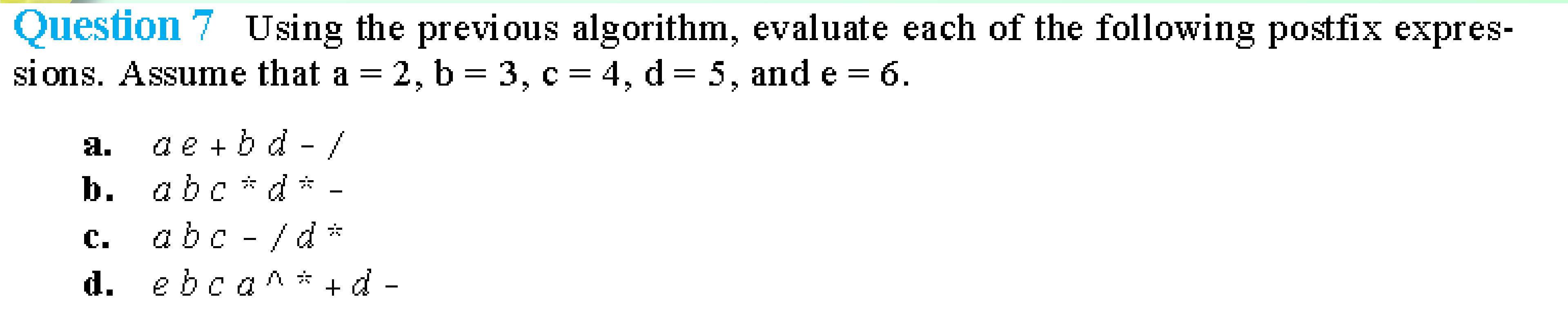
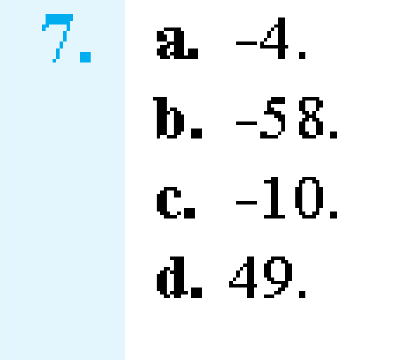


FIGURE 5-11 The stack during the evaluation of the postfix expression a b + c / when a is 2, b is 4, and c is 3



Question 7 Using the previous algorithm, evaluate each of the following postfix expressions. Assume that a = 2, b = 3, c = 4, d = 5, and e = 6.

- **a.** $a\ e + b\ d - /$
- **b.** $a\ b\ c * d * -$
- **c.** $a\ b\ c - / d *$
- **d.** $e\ b\ c\ a\ \hat{}\ * + d -$



7. a. -4.
b. -58.
c. -10.
d. 49.

Evaluating Infix Expressions

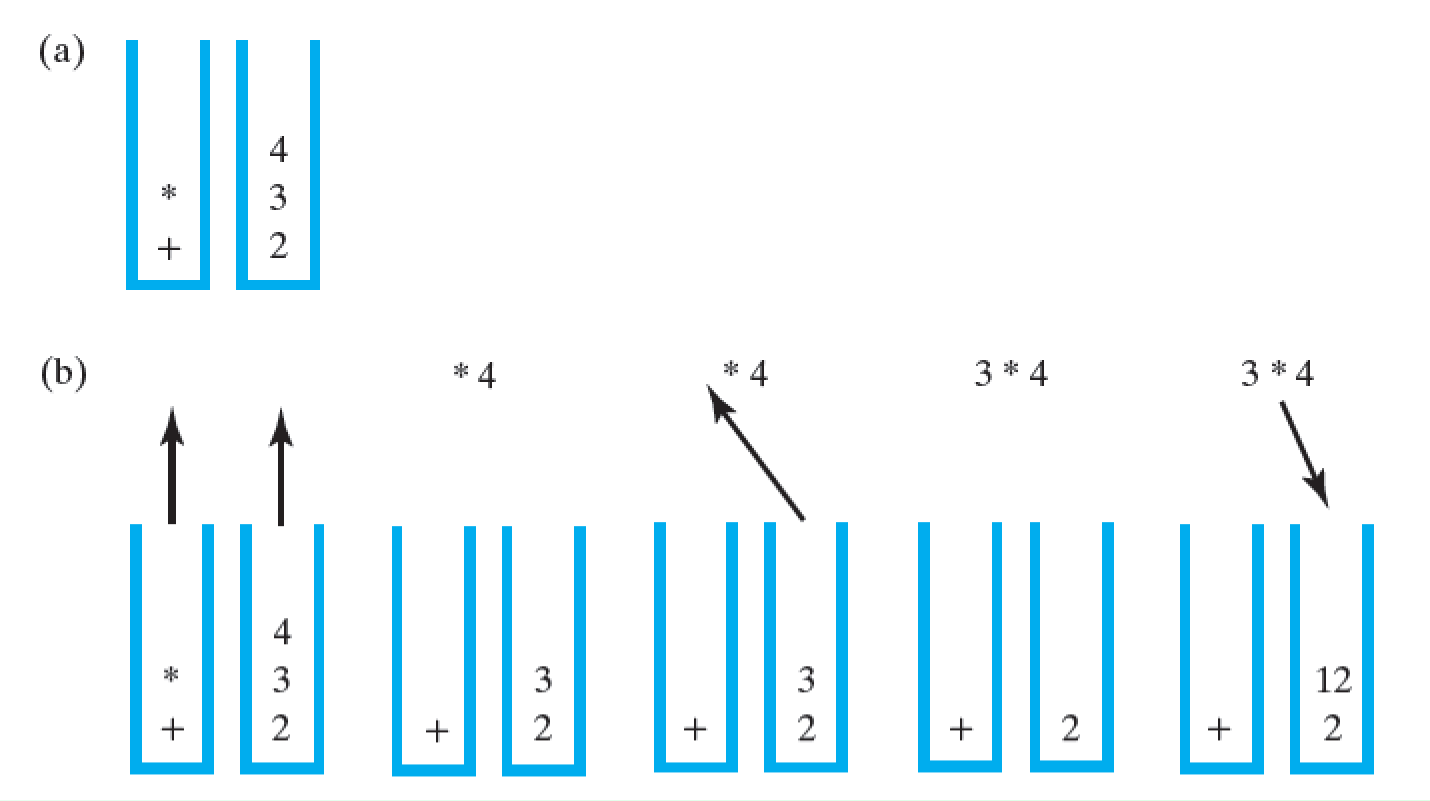


FIGURE 5-12 Two stacks during the evaluation of a + b * c when a is 2, b is 3, and c is 4: (a) after reaching the end of the expression; (b) while performing the multiplication;

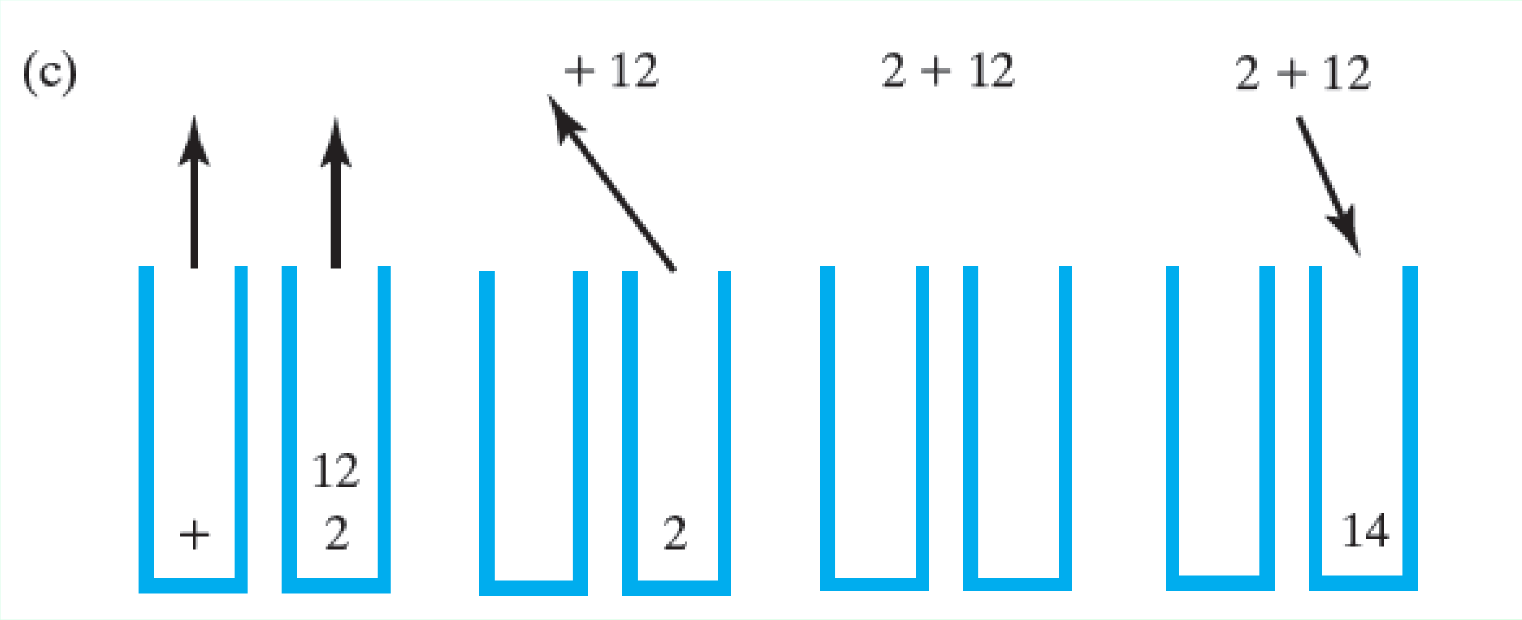
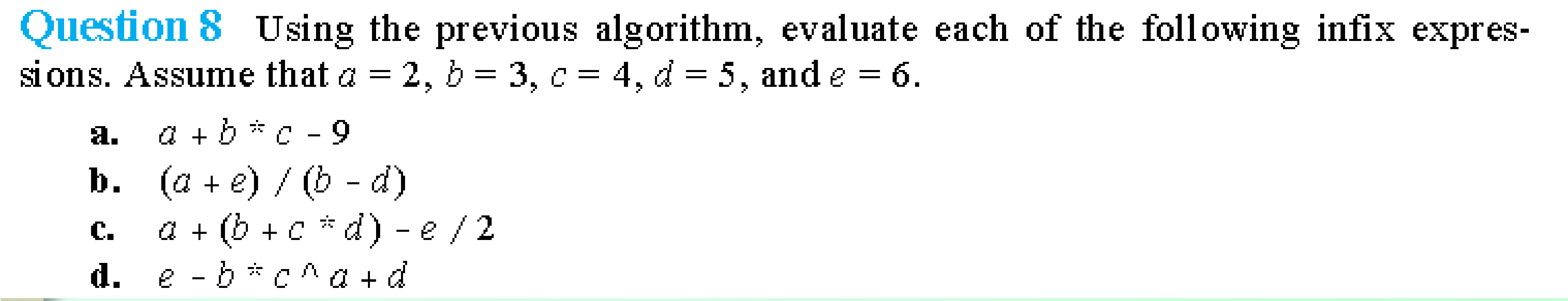


FIGURE 5-12 Two stacks during the evaluation of a + b * c when a is 2, b is 3, and c is 4: (c) while performing the addition



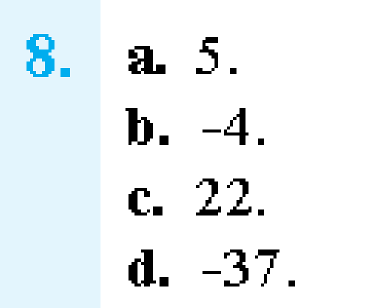
Question 8 Using the previous algorithm, evaluate each of the following infix expressions. Assume that $a = 2$, $b = 3$, $c = 4$, $d = 5$, and $e = 6$.

a. $a + b * c - 9$

b. $(a + e) / (b - d)$

c. $a + (b + c * d) - e / 2$

d. $e - b * c \wedge a + d$



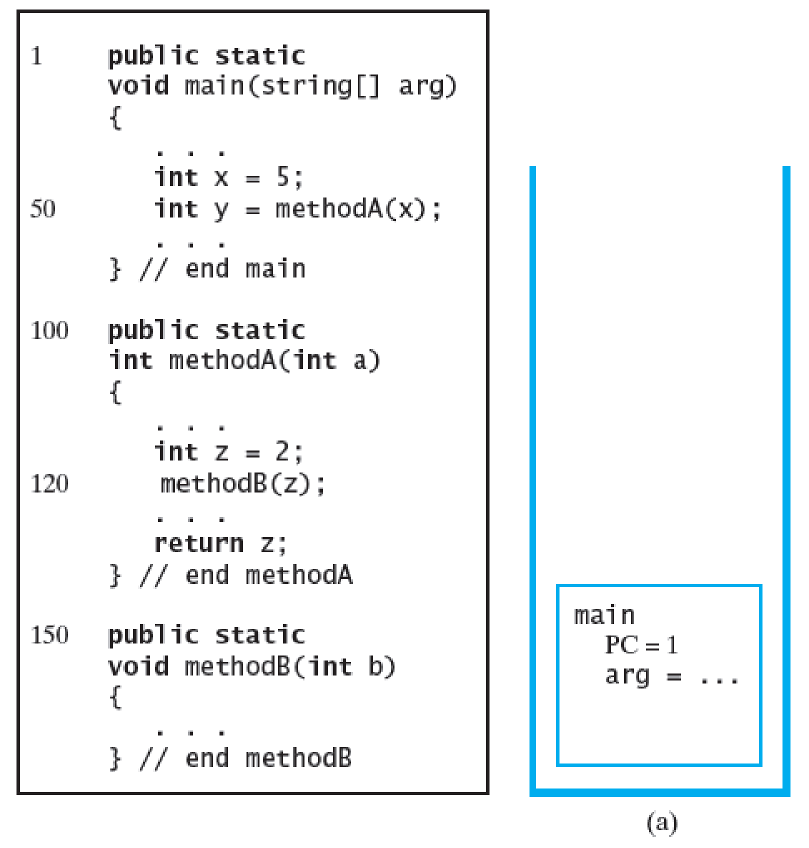
8. a. 5.

b. -4.

c. 22.

d. -37.

The Program Stack



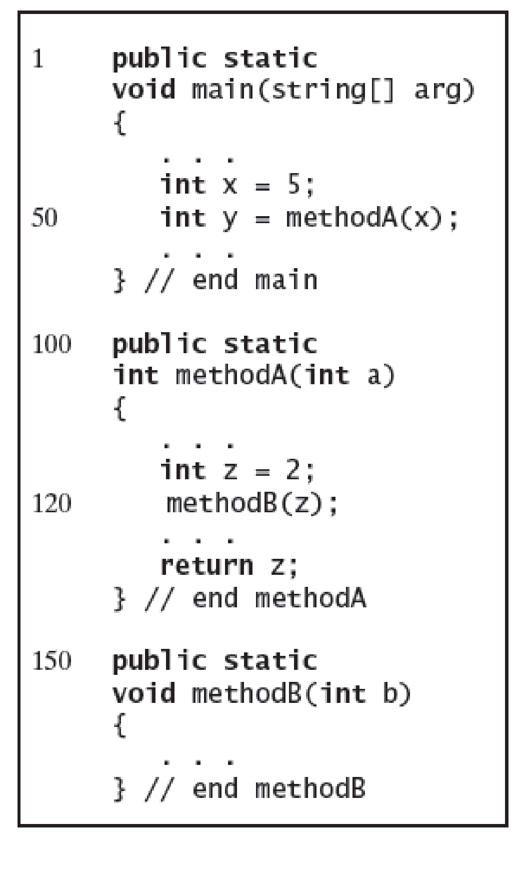
```
1     public static
      void main(string[] arg)
      {
         . . .
         int x = 5;
50       int y = methodA(x);
         . . .
      } // end main

100   public static
      int methodA(int a)
      {
         . . .
         int z = 2;
120      methodB(z);
         . . .
         return z;
      } // end methodA

150   public static
      void methodB(int b)
      {
         . . .
      } // end methodB
```

FIGURE 5-13 The program stack at three points in time:
(a) when **main** begins execution; (PC is the program counter)

The Program Stack



```
1      public static
       void main(string[] arg)
       {
          . . .
          int x = 5;
50        int y = methodA(x);
          . . .
       } // end main

100    public static
       int methodA(int a)
       {
          . . .
          int z = 2;
120        methodB(z);
          . . .
          return z;
       } // end methodA

150    public static
       void methodB(int b)
       {
          . . .
       } // end methodB
```

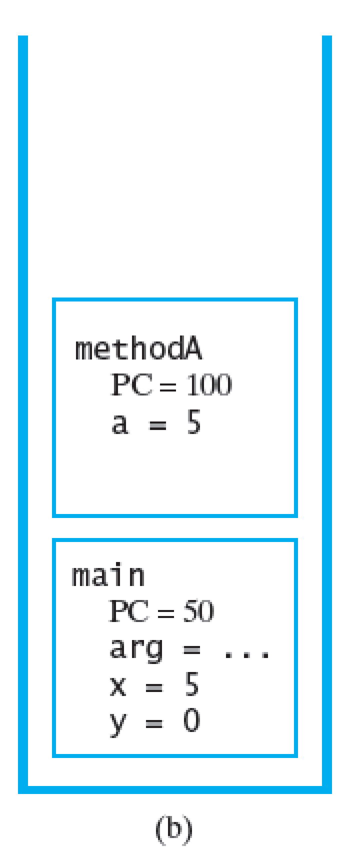
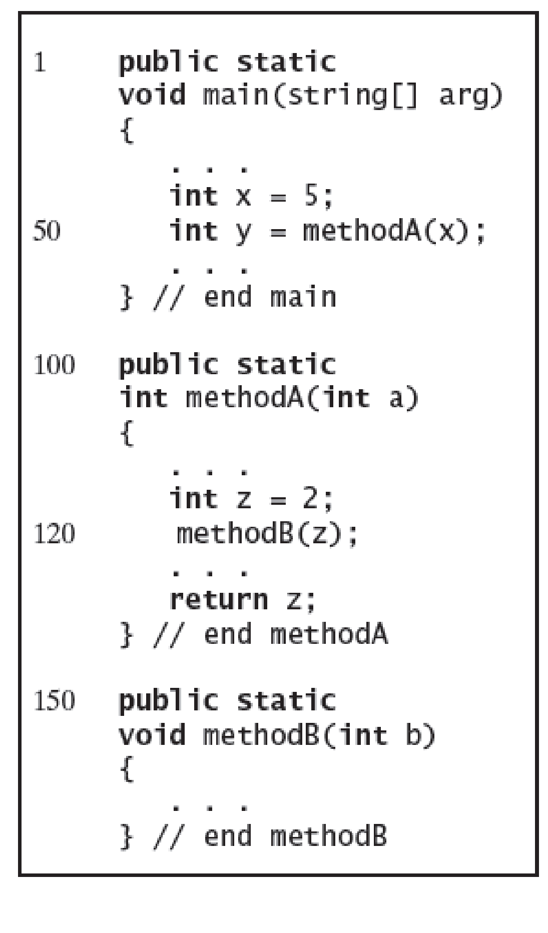


FIGURE 5-13 The program stack at three points in time:
(b) when **methodA** begins execution; (PC is the program counter)

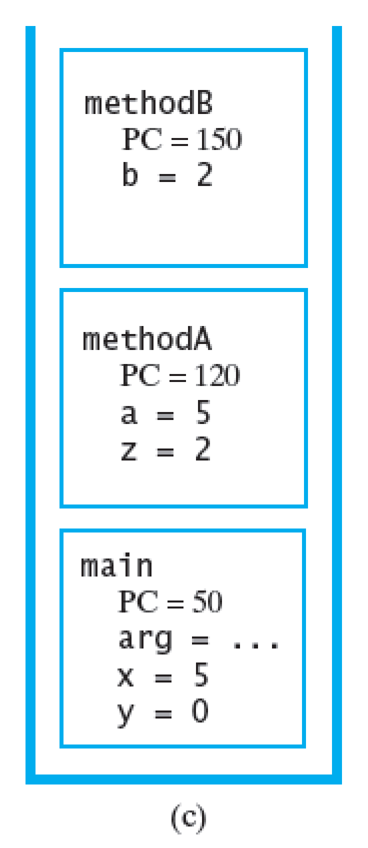
The Program Stack



```
1      public static
       void main(string[] arg)
       {
          . . .
          int x = 5;
50        int y = methodA(x);
          . . .
       } // end main

100    public static
       int methodA(int a)
       {
          . . .
          int z = 2;
120        methodB(z);
          . . .
          return z;
       } // end methodA

150    public static
       void methodB(int b)
       {
          . . .
       } // end methodB
```



methodB
PC = 150
b = 2

methodA
PC = 120
a = 5
z = 2

main
PC = 50
arg = ...
x = 5
y = 0

(c)

FIGURE 5-13 The program stack at three points in time:
(c) when **methodB** begins execution; (PC is the program counter)

Java Class Library: The Class **`Stack`**

- Has a single constructor
  - Creates an empty stack
- Remaining methods – differences from our **`StackInterface`** are highlighted
  - **`public T push(T item);`**
  - **`public T pop();`**
  - **`public T peek();`**
  - **`public boolean empty();`**

End

Chapter 5