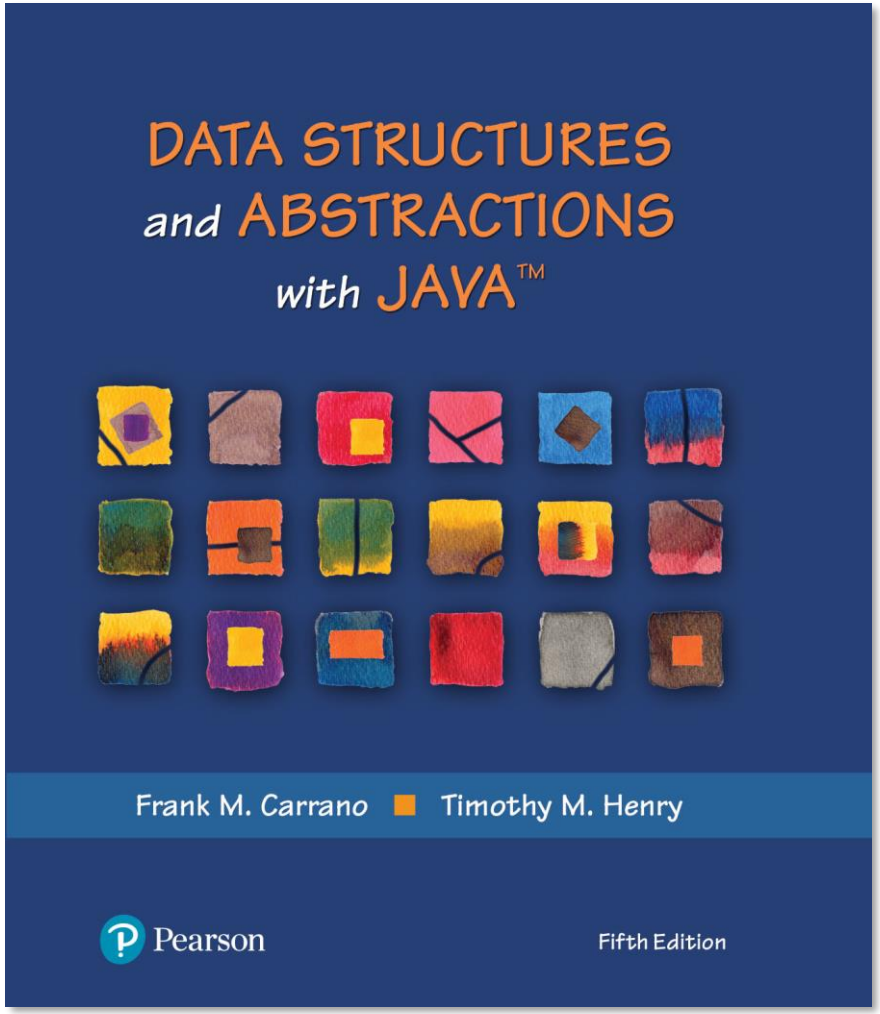
Data Structures and Abstractions with Java™

5th Edition



Chapter 5

Stacks

Stacks

- Add item on top of stack
- Remove item that is topmost
  - Last In, First Out … LIFO

FIGURE 5-1 Some familiar stacks

Specifications of the ADT Stack

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

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 the stack's top entry. Throws an exception if the stack is empty before the operation.
peek()	**+peek(): T**	Task: Retrieves the stack's top entry without changing the stack in any way. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty.
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.

Design Decision

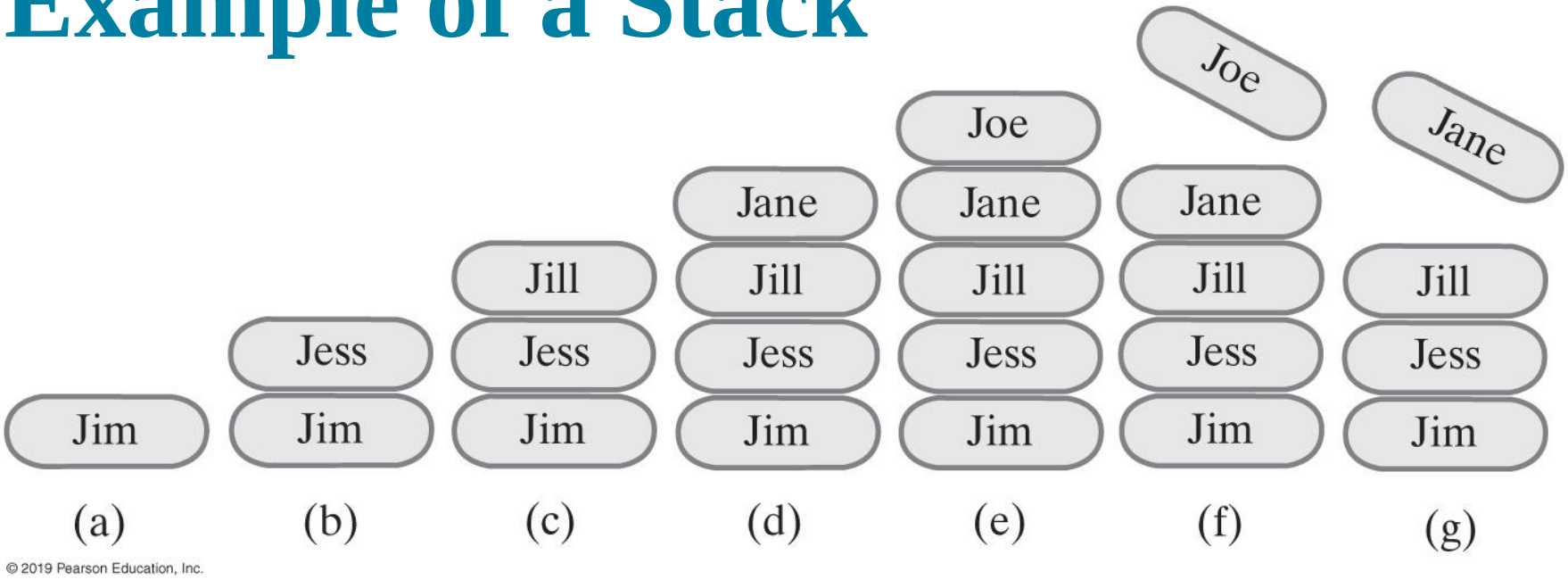
- When stack is empty
  - What to do with **`pop`** and **`peek`**?
- Possible actions
  - Assume that the ADT is not empty;
  - Return null.
  - Throw an exception (which type?).

Interface for the ADT Stack

```
/** An interface for the ADT stack. */
public interface StackInterface<T>
{
  /** Adds a new entry to the top of this stack.
      @param newEntry  An object to be added to the stack. */
  public void push(T newEntry);

  /** Removes and returns this stack's top entry.
      @return  The object at the top of the stack.
      @throws  EmptyStackException if the stack is empty before the operation. */
  public T pop();

  /** Retrieves this stack's top entry.
      @return  The object at the top of the stack.
      @throws  EmptyStackException if the stack is empty. */
  public T peek();

  /** Detects whether this stack is empty.
      @return  True if the stack is empty. */
  public boolean isEmpty();

  /** Removes all entries from this stack. */
  public void clear();
} // end StackInterface
```

LISTING 5-1 An interface for the ADT stack

Example of a Stack




```
     StackInterface<String> stringStack = new OurStack<>();
(a)  stringStack.push("Jim");
(b)  stringStack.push("Jess");
(c)  stringStack.push("Jill");
(d)  stringStack.push("Jane");
(e)  stringStack.push("Joe");
(f)  stringStack.pop();
(g)  stringStack.pop();
```

FIGURE 5-2 A stack of strings

Security Note

- Design guidelines
  - Use preconditions and postconditions to document assumptions.
  - Do not trust client to use public methods correctly.
  - Avoid ambiguous return values.
  - Prefer throwing exceptions instead of returning values to signal problem.

Processing Algebraic Expressions

- Infix:
  - each binary operator appears between its operands

```
a + b
```

- Prefix:
  - each binary operator appears before its operands

```
+ a b
```

- Postfix:
  - each binary operator appears after its operands

```
a b +
```

- Balanced expressions: delimiters paired correctly

Processing Algebraic Expressions

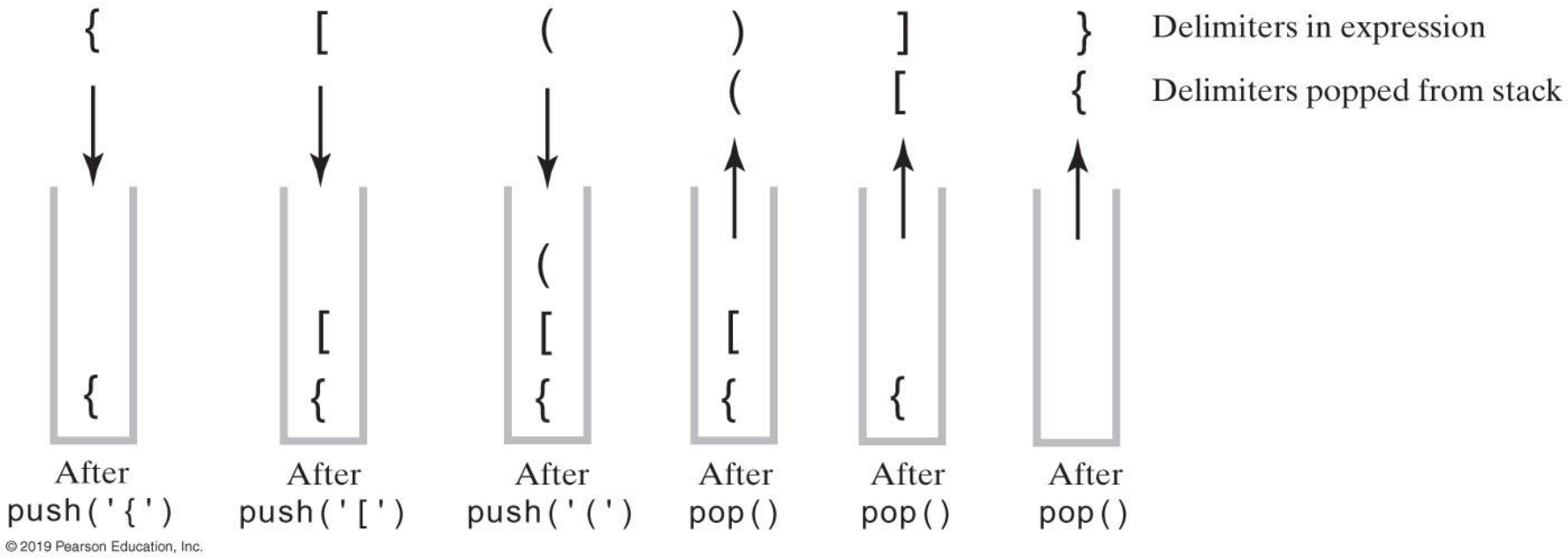


FIGURE 5-3 The contents of a stack during the scan of an expression that contains the balanced delimiters{ [( )] }

Processing Algebraic Expressions

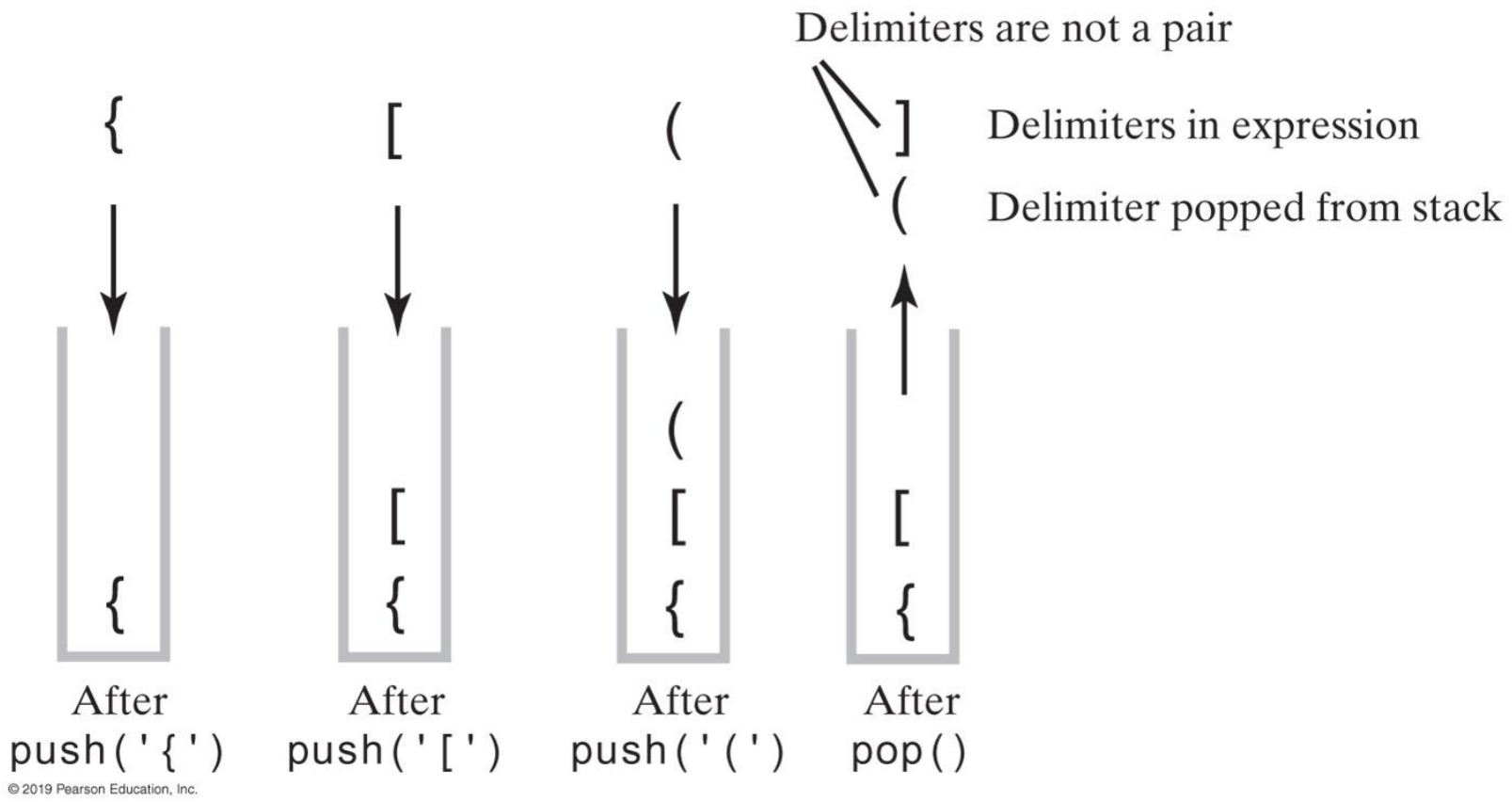


FIGURE 5-4 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( ]) }

Processing Algebraic Expressions

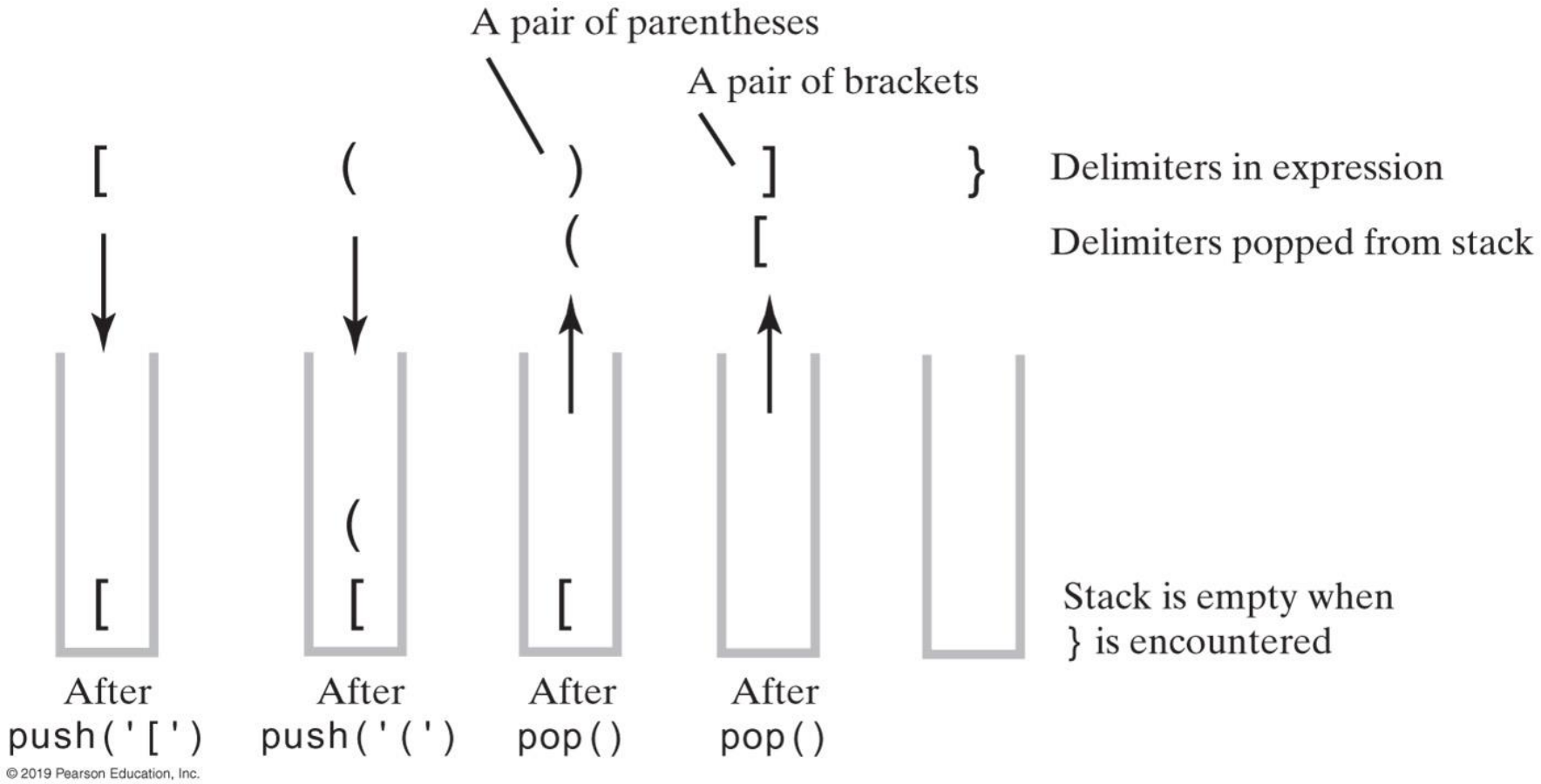


FIGURE 5-5 The contents of a stack during the scan of an expression that contains the unbalanced delimiters [( )] }

Processing Algebraic Expressions

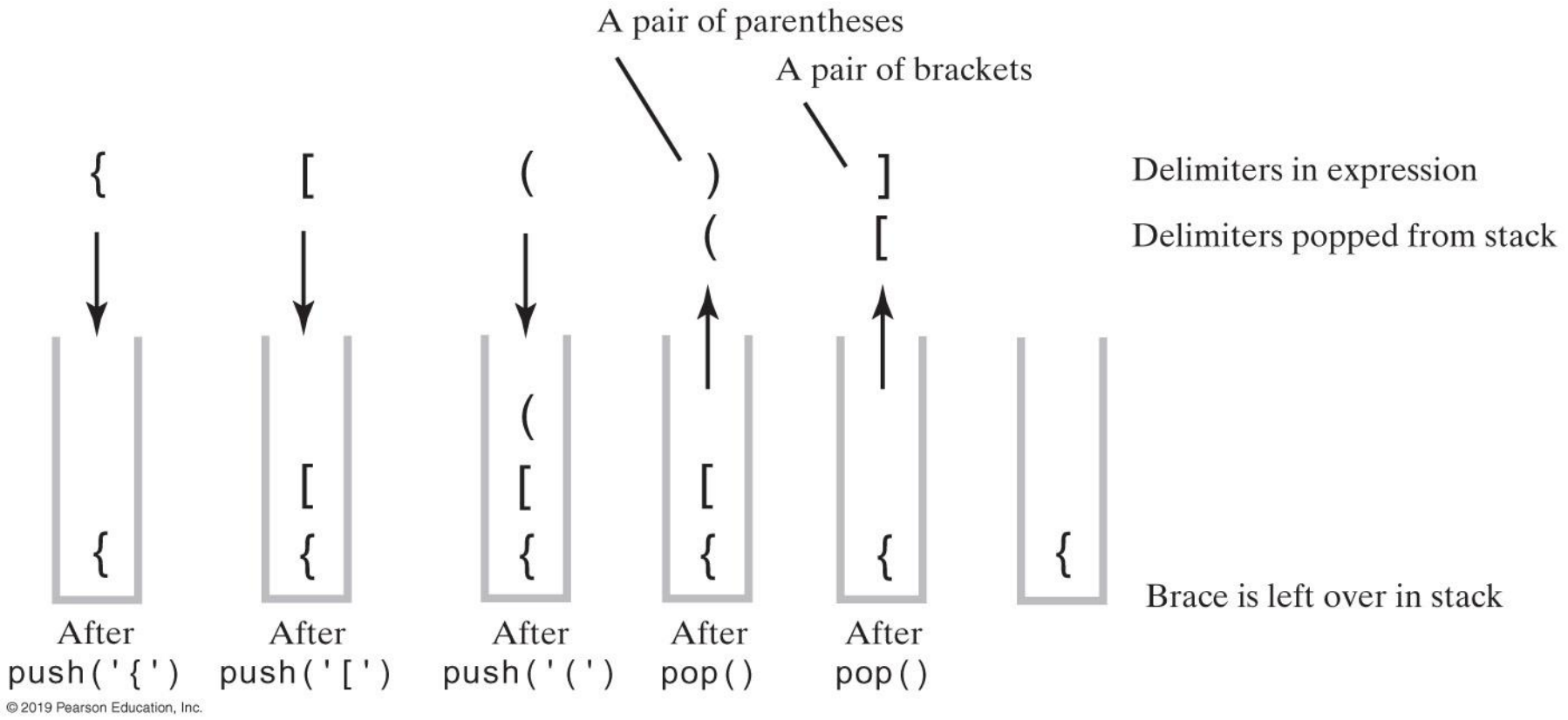


FIGURE 5-6 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( )]

Processing Algebraic Expressions

```
Algorithm checkBalance(expression)
//Returns true if the parentheses, brackets, and braces in an expression are paired correctly.

isBalanced = true // The absence of delimiters is balanced
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 entry 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
```

Algorithm to process for balanced expression.

Implementation of Algorithm (Part 1)

```
/** A class that checks whether the parentheses, brackets, and braces
  in a string occur in left/right pairs. */
public class BalanceChecker
{

  // Returns true if the given characters, open and close, form a pair
  // of parentheses, brackets, or braces.
  private static boolean isPaired(char open, char close)
  {
    return (open == '(' && close == ')') ||
           (open == '[' && close == ']') ||
           (open == '{' && close == '}');
  } // end isPaired
 /** Decides whether the parentheses, brackets, and braces
     in a string occur in left/right pairs.
     @param expression  A string to be checked.
     @return  True if the delimiters are paired correctly. */
  public static boolean checkBalance(String expression)
  {
                    [ SEE NEXT SLIDE FOR IMPLEMENTATION]
  } // end checkBalance

} // end BalanceChecker
```

LISTING 5-2 The class `BalanceChecker`

Implementation of Algorithm (Part 2)

```
StackInterface<Character> openDelimiterStack = new LinkedStack<>();
int characterCount = expression.length();
boolean isBalanced = true;
int index = 0;
char nextCharacter = ' ';

while (isBalanced && (index < characterCount)) {
  nextCharacter = expression.charAt(index);
  switch (nextCharacter) {
    case '(': case '[': case '{':
      openDelimiterStack.push(nextCharacter);
      break;
    case ')': case ']': case '}':
      if (openDelimiterStack.isEmpty())
        isBalanced = false;
      else {
        char openDelimiter = openDelimiterStack.pop();
        isBalanced = isPaired(openDelimiter, nextCharacter);
      } // end if
      break;
    default: break; // Ignore unexpected characters
  } // end switch
  index++;
} // end while

if (!openDelimiterStack.isEmpty())
  isBalanced = false;

return isBalanced;
} // end checkBalance
```

Converting Infix to Postfix

`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 * +	

`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 ^ ^	

`a - b + c`

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* +	

FIGURES 5-7 & 5-8 Converting the infix expressions to postfix form

Converting Infix to Postfix

```
a / b * (c + (d - e))
```

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 Steps in converting an infix expression to postfix form

Infix-to-postfix Conversion

To convert an infix expression to postfix form, you take the following actions, according to the symbols you encounter, as you process the infix expression from left to right:

Operand	Append each operand to the end of the output expression.
Operator ^	Push ^ onto the stack.
Operator +, -, *, or /	Pop operators from the stack, appending them to the output expression, until either the stack is empty or its top entry has a lower precedence than the newly encountered operator. Then push the new operator onto the stack.
Open parenthesis	Push (onto the stack.
Close parenthesis	Pop operators from the stack and append them to the output expression until an open parenthesis is popped. Discard both parentheses.

Infix-to-postfix Algorithm (Part 1)

```
Algorithm convertToPostfix(infix)
// Converts an infix expression to an equivalent postfix expression.
operatorStack = a new empty stack
postfix = a new empty string
while (infix has characters left to parse)
{
   nextCharacter = next nonblank character of infix
   switch (nextCharacter)
   {
      case variable:
         Append nextCharacter to postfix
         break
      case '^' :
         operatorStack.push(nextCharacter)
         break
      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
            precedence of nextCharacter <= precedence of operatorStack.peek())
         {
            Append operatorStack.peek() to postfix
            operatorStack.pop()
         }
        operatorStack.push(nextCharacter)
        break
```

Algorithm for converting infix to postfix expressions.

Infix-to-postfix Algorithm (Part 2)

```
        case '( ' :
            operatorStack.push(nextCharacter)
            break
        case ')' : // Stack is not empty if infix expression is valid
            topOperator = operatorStack.pop()
            while (topOperator != '(')
            {
                Append topOperator to postfix
                topOperator = operatorStack.pop()
            }
            break
        default:
            break // Ignore unexpected characters
    }
}
while (!operatorStack.isEmpty())
{
    topOperator = operatorStack.pop()
    Append topOperator to postfix
}
return postfix
```

Algorithm for converting infix to postfix expressions.

Evaluating Postfix Expressions

```
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 // Ignore unexpected characters
   }
}
return valueStack.peek()
```

Algorithm for evaluating postfix expressions.

Evaluating Infix Expressions

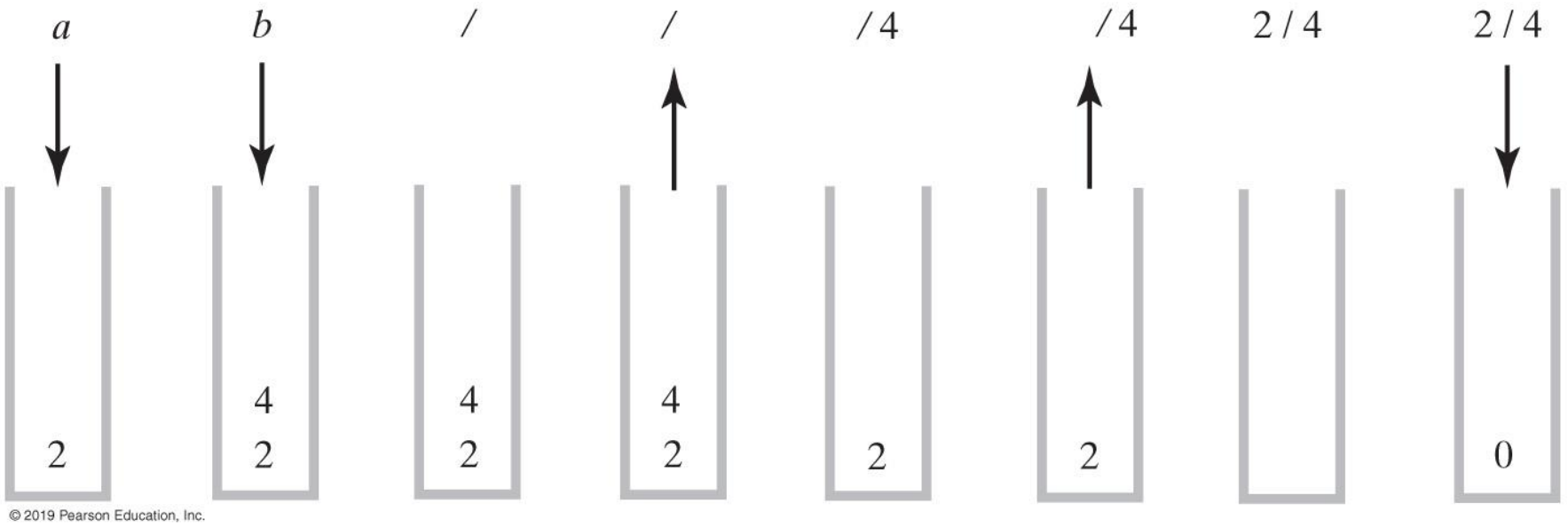


FIGURE 5-10 The stack during the evaluation of the postfix expression `a b /` when `a` is 2 and `b` is 4

Evaluating Infix Expressions

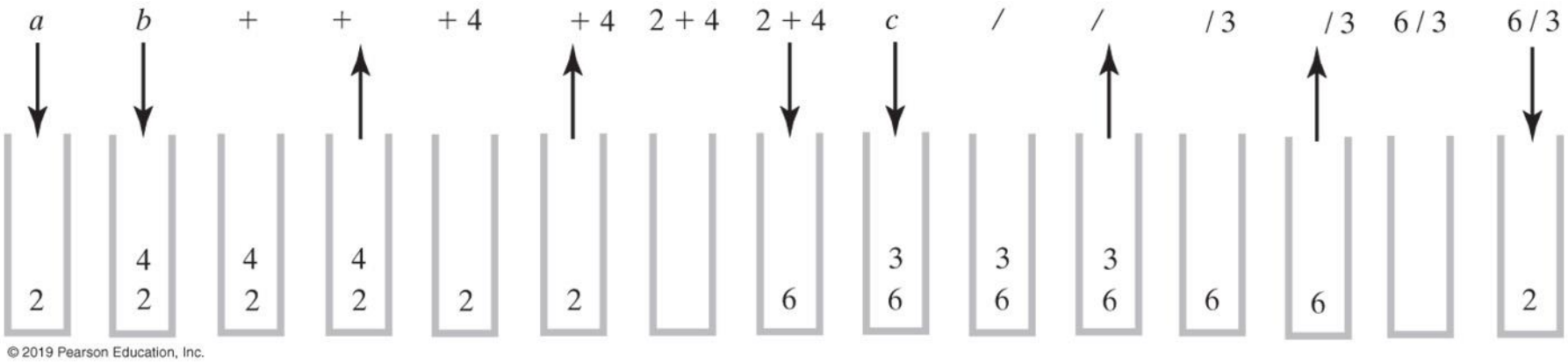


FIGURE 5-11 The stack during the evaluation of the postfix expression `a b + c /` when `a` is 2, `b` is 4, and `c` is 3

Evaluating Infix Expressions

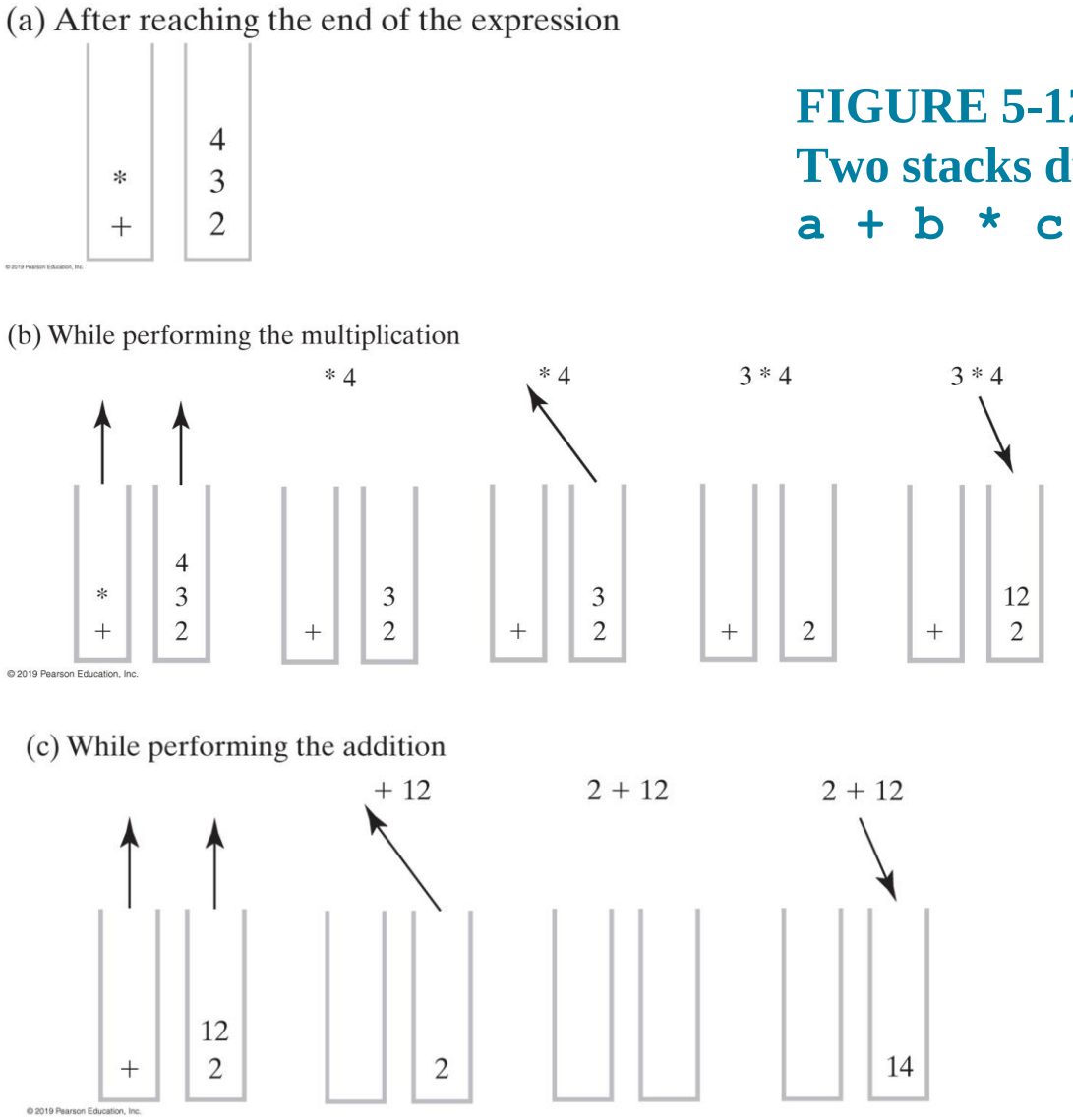


FIGURE 5-12
Two stacks during the evaluation of `a + b * c` when `a` is 2, `b` is 3, and `c` is 4

Evaluating Infix Expressions (Part 1)

```
Algorithm evaluateInfix(infix)    // Evaluates an infix expression.
operatorStack = a new empty stack
valueStack = a new empty stack
while (infix has characters left to process)
{
   nextCharacter = next nonblank character of infix
   switch (nextCharacter)
   {
      case variable:
         valueStack.push(value of the variable nextCharacter)
         break
      case '^' :
         operatorStack.push(nextCharacter)
         break
      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
                  precedence of nextCharacter <= precedence of operatorStack.peek())
         {
            // Execute operator at top of operatorStack
            topOperator = operatorStack.pop()
            operandTwo = valueStack.pop()
            operandOne = valueStack.pop()
            result = the result of the operation in
                           topOperator and its operands operandOne and operandTwo
            valueStack.push(result)
         }
         operatorStack.push(nextCharacter)
         break
```

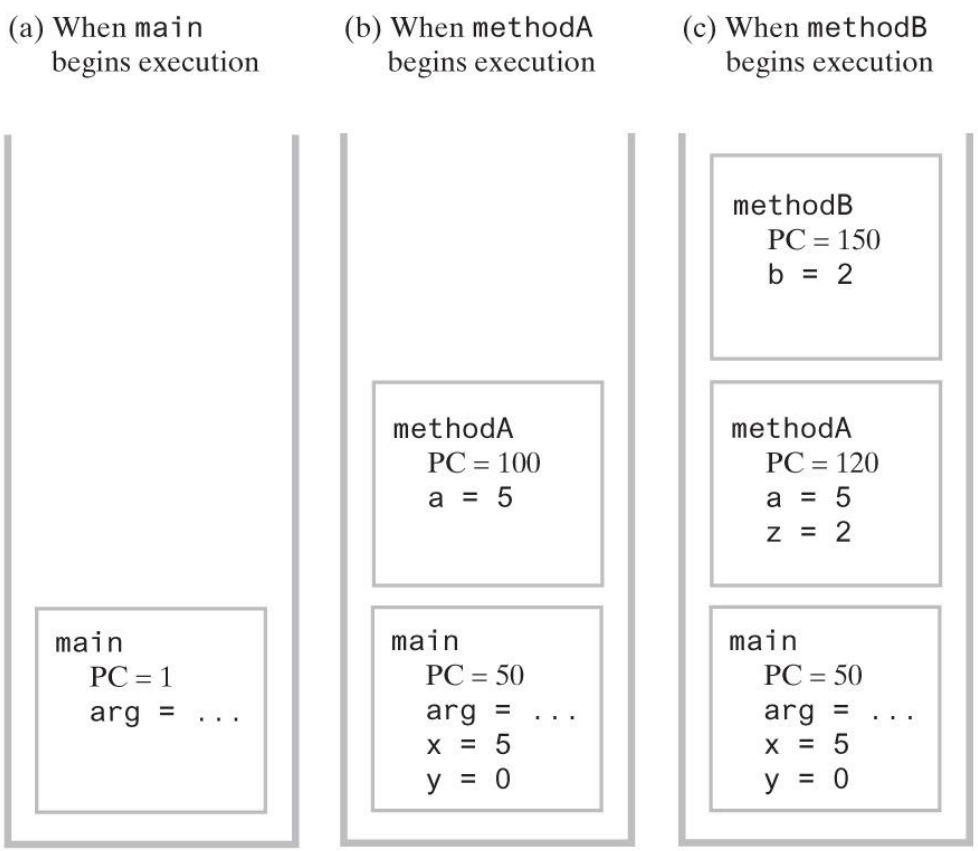
Evaluating Infix Expressions (Part 2)

```
        case '(' :
            operatorStack.push(nextCharacter)
            break
        case ')' : // Stack is not empty if infix expression is valid
            topOperator = operatorStack.pop()
            while (topOperator != '(')
            {
                operandTwo = valueStack.pop()
                operandOne = valueStack.pop()
                result = the result of the operation in
                                topOperator and its operands operandOne and operandTwo
                valueStack.push(result)
                topOperator = operatorStack.pop()
            }
            break
        default: break // Ignore unexpected characters
    }
}
while (!operatorStack.isEmpty())
{
    topOperator = operatorStack.pop()
    operandTwo = valueStack.pop()
    operandOne = valueStack.pop()
    result = the result of the operation in
                    topOperator and its operands operandOne and operandTwo
    valueStack.push(result)
}
return valueStack.peek()
```

The Application 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
```

Program



Program stack at three points in time (PC is the program counter)

FIGURE 5-13 The program stack as a program executes

Java Class Library: The Class `Stack`

- Found in **`java.util`**
- Methods
  - A constructor – creates an empty stack
  - **`public T push(T item);`**
  - **`public T pop();`**
  - **`public T peek();`**
  - **`public boolean empty();`**

End

Chapter 5