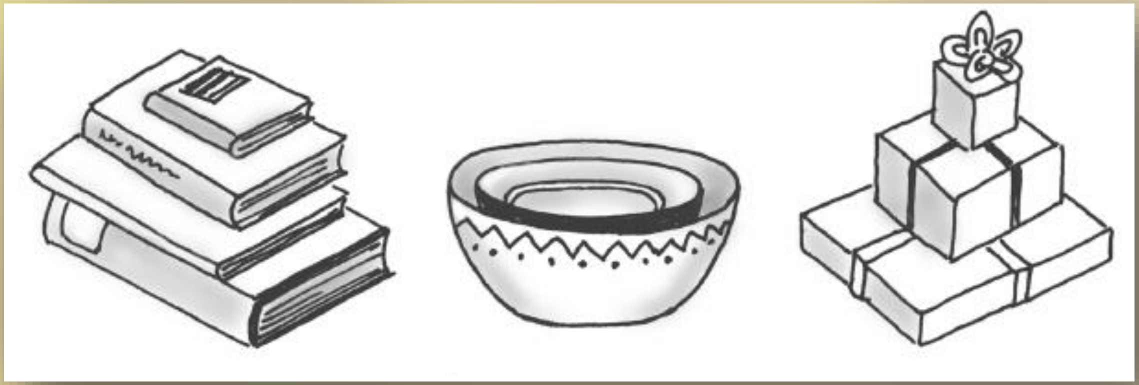
Stacks

Chapter 5

Data Structures and Abstractions with Java, 4e, Global Edition
Frank Carrano

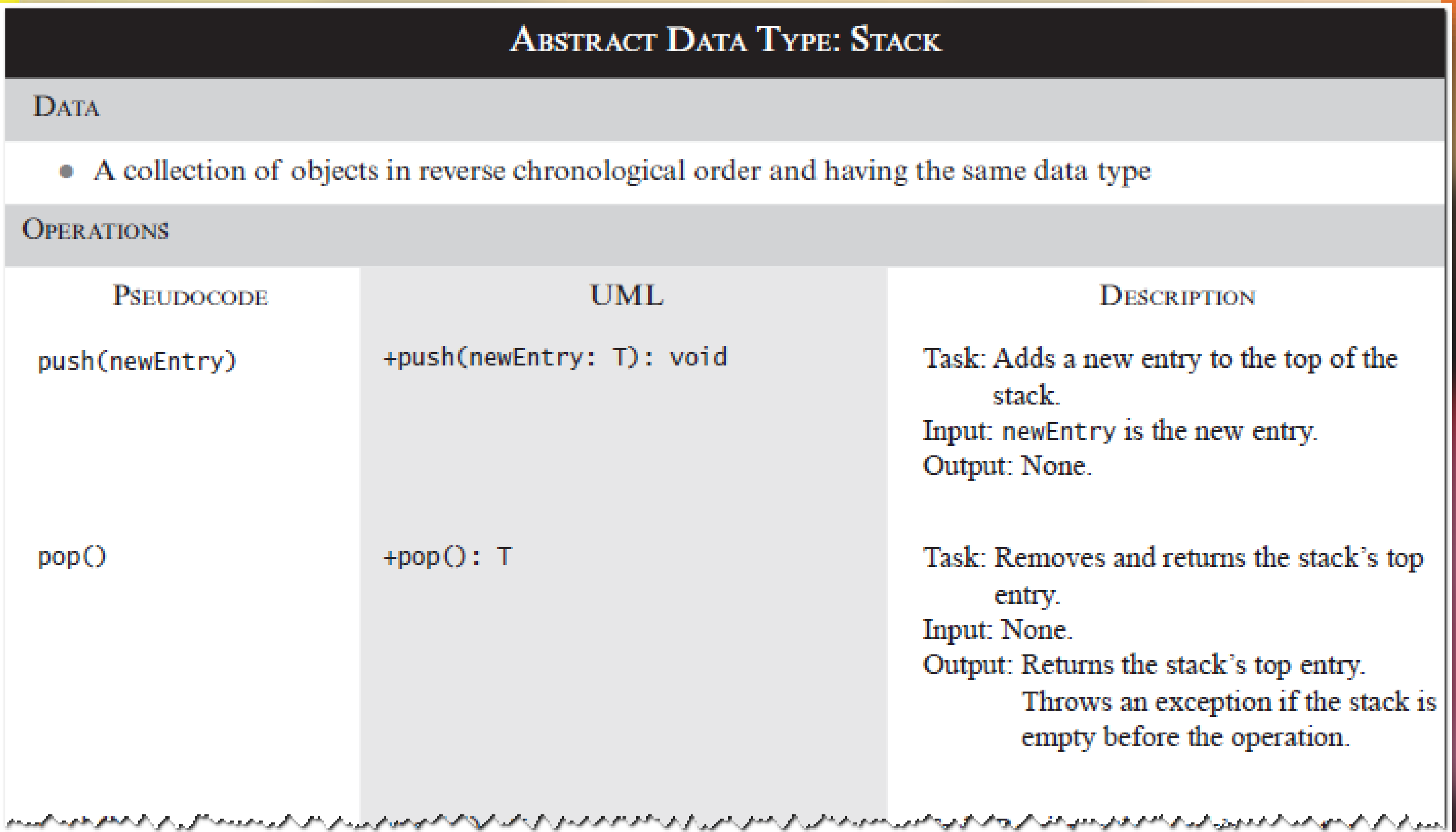
Stacks

- FIGURE 5-1 Some familiar stacks



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

Specifications of the ADT Stack



ABSTRACT DATA TYPE: STACK

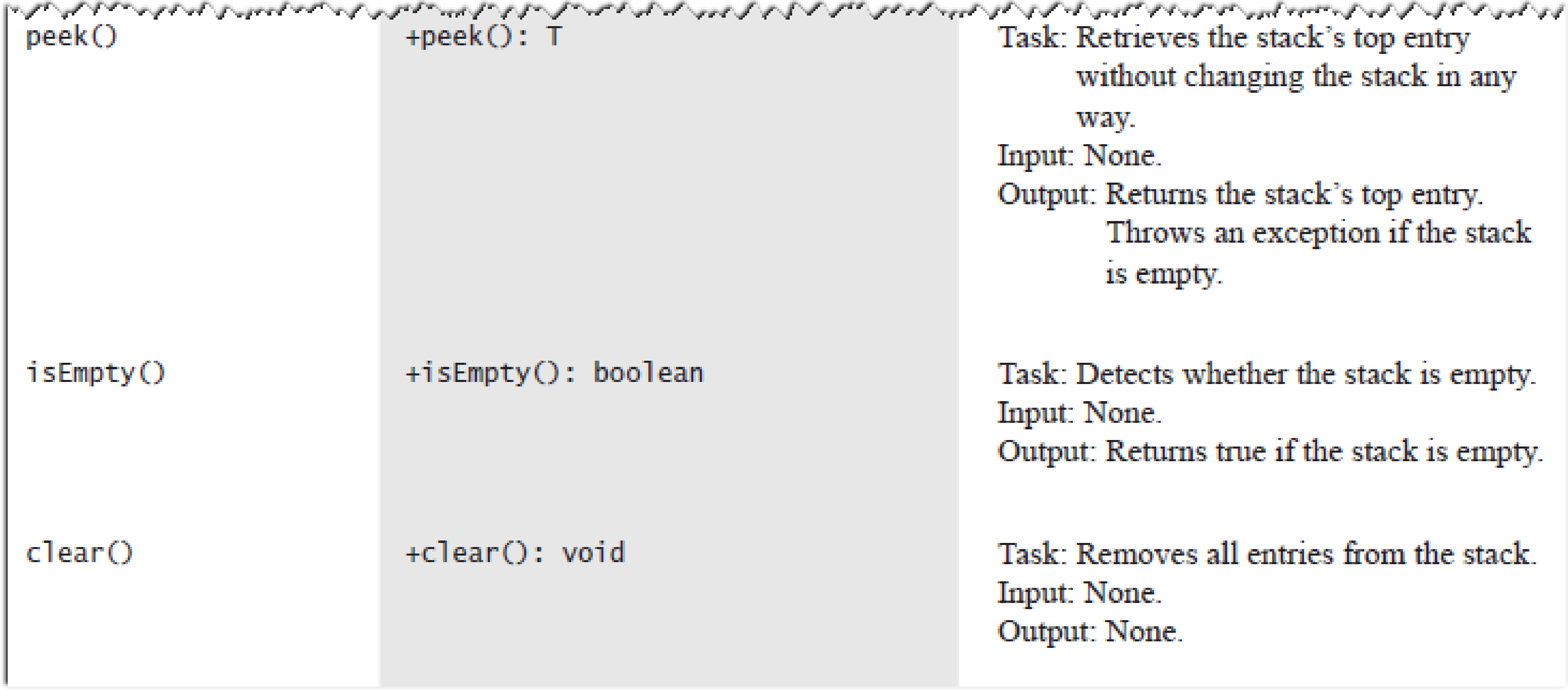
DATA

- 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 the stack's top entry. Throws an exception if the stack is empty before the operation.

Specifications of the ADT Stack

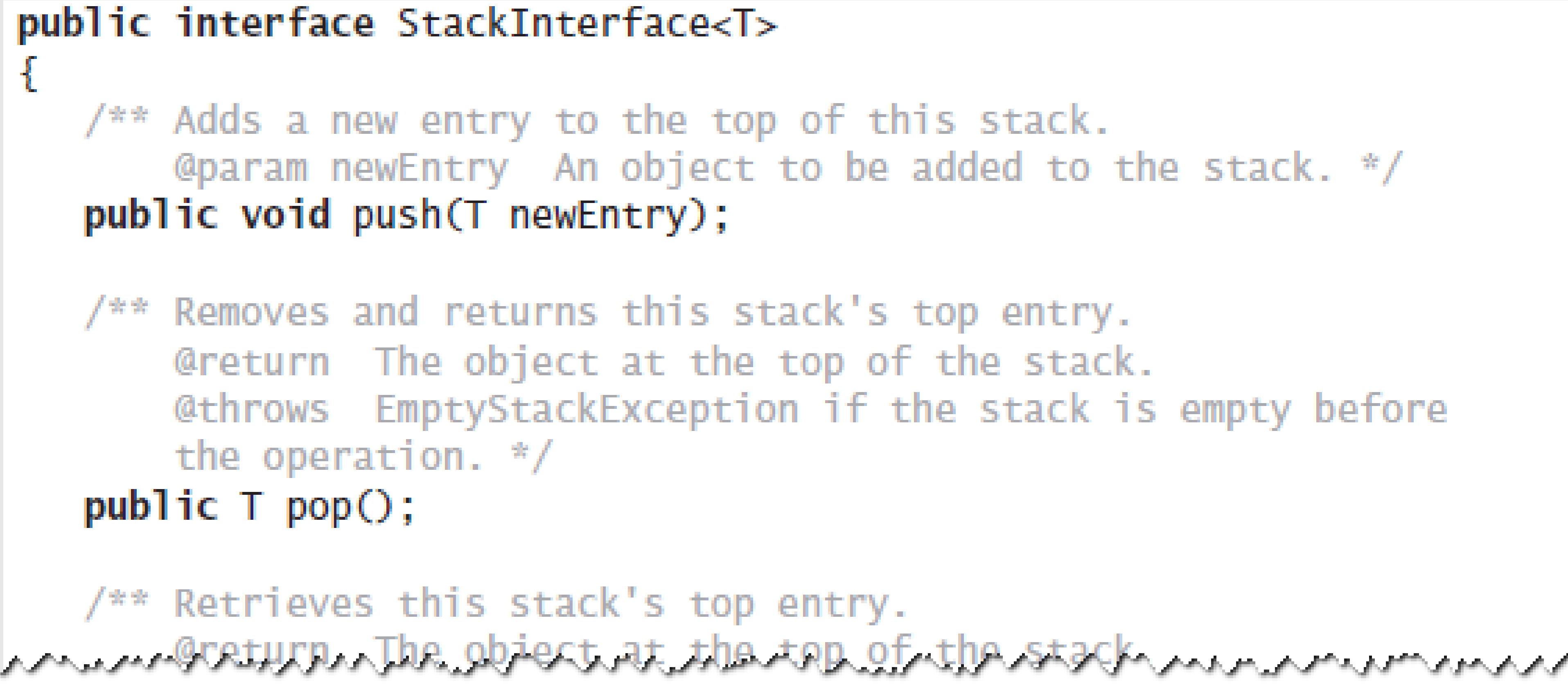


peek()	+peek(): T	Task: Retrieves the stack's top entry without changing the stack in any way. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty.
isEmpty()	+isEmpty(): boolean	Task: Detects whether the stack is empty. Input: None. Output: Returns true if the 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



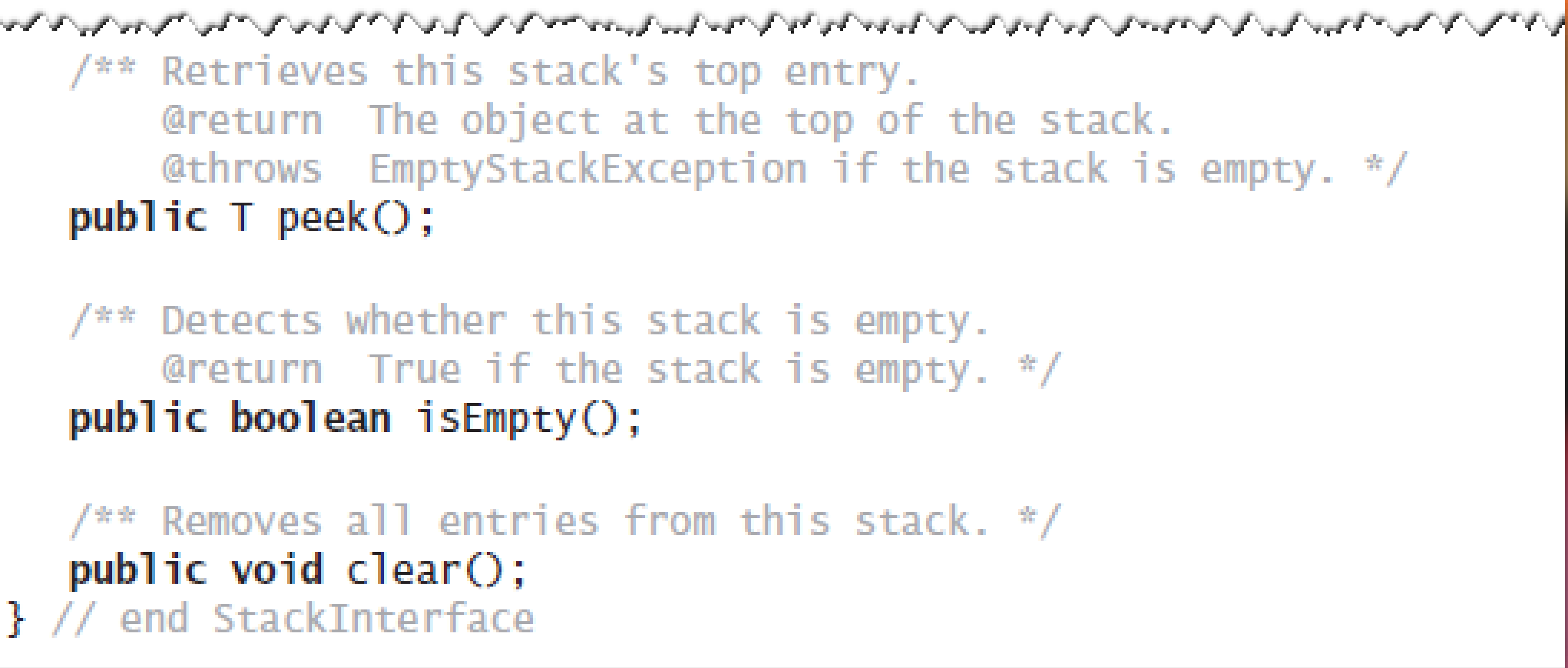
```
public interface StackInterface<T>
{
   /** Adds a new entry to the top of this stack.
       @param newEntry  An object to be added to the stack. */
   public void push(T newEntry);

   /** 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
```

LISTING 5-1 An interface for the ADT stack

Interface



```
   /** 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

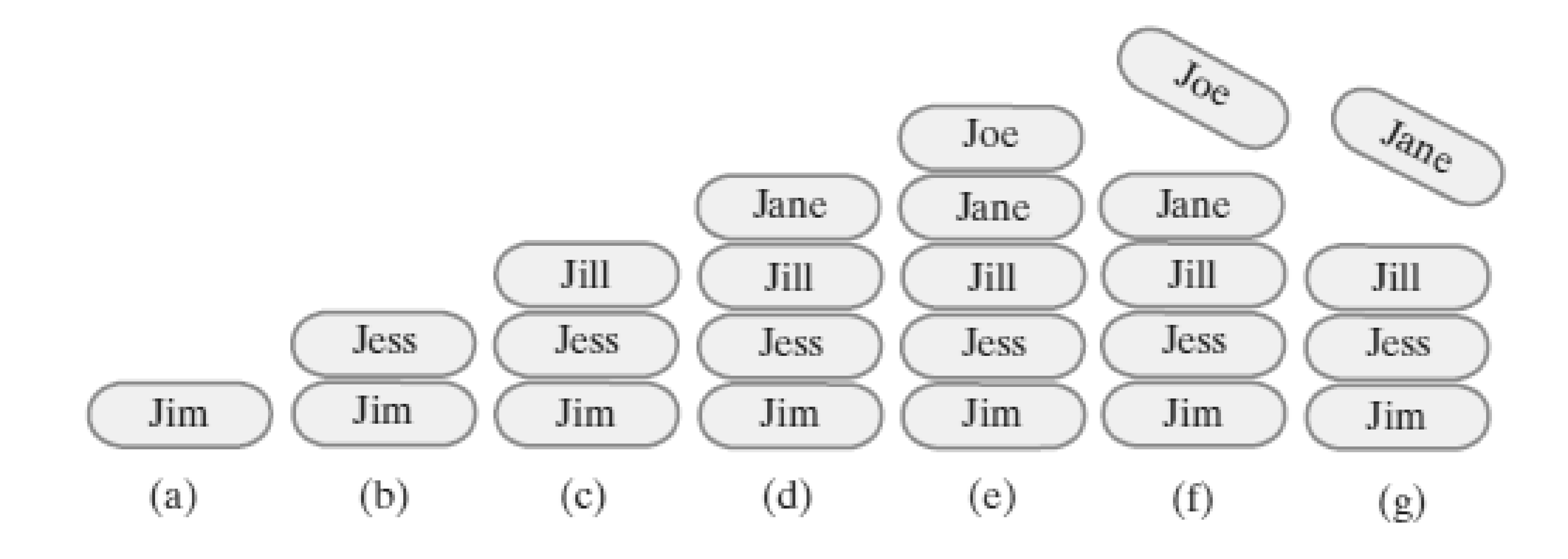
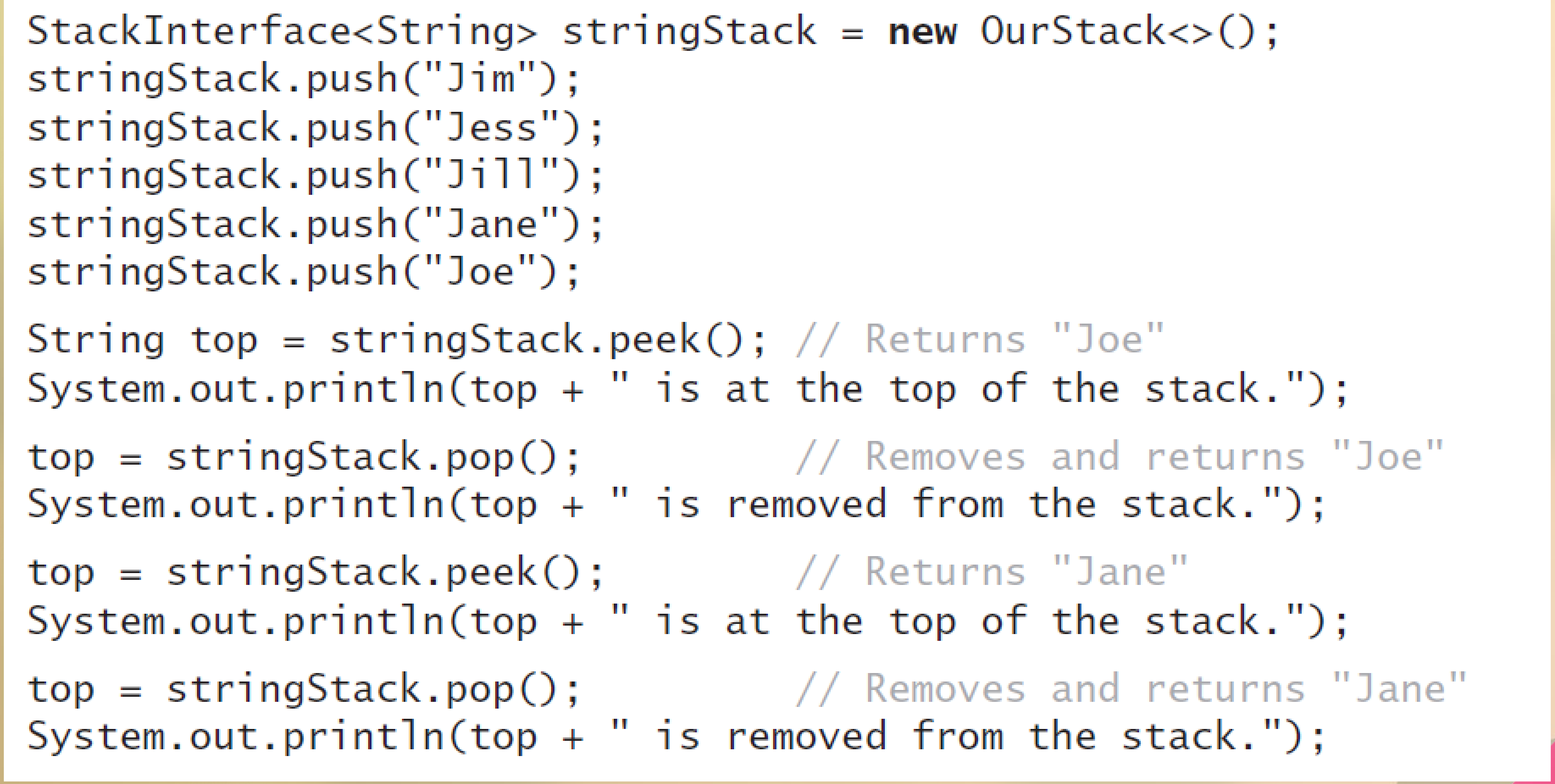


FIGURE 5-2 A stack of strings after (a) push adds Jim; (b) push adds Jess; (c) push adds Jill; (d) push adds Jane; (e) push adds Joe; (f) pop retrieves and removes Joe; (g) pop retrieves and removes Jane

Demo of a stack



```java
StackInterface<String> stringStack = new OurStack<>();
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.");
```

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

- Programmers use parentheses when writing arithmetic expressions in Java.

- Mathematicians use
  - parentheses (‘(‘, ‘)’),
  - square brackets (‘[‘, ‘]’), and
  - braces (‘{‘, ‘}’)

  for the same purpose.

- These delimiters must be paired correctly.
- An open parenthesis must correspond to a close parenthesis.
- Pairs of delimiters must not intersect.

- Thus, an expression can contain a sequence of delimiters such as { [( ) ( )] ( ) } but not [( ])

- We will say that a **balanced expression** contains delimiters that are paired correctly, or are **balanced**.

- We want an algorithm that detects whether an infix expression is balanced.

Processing Algebraic Expressions

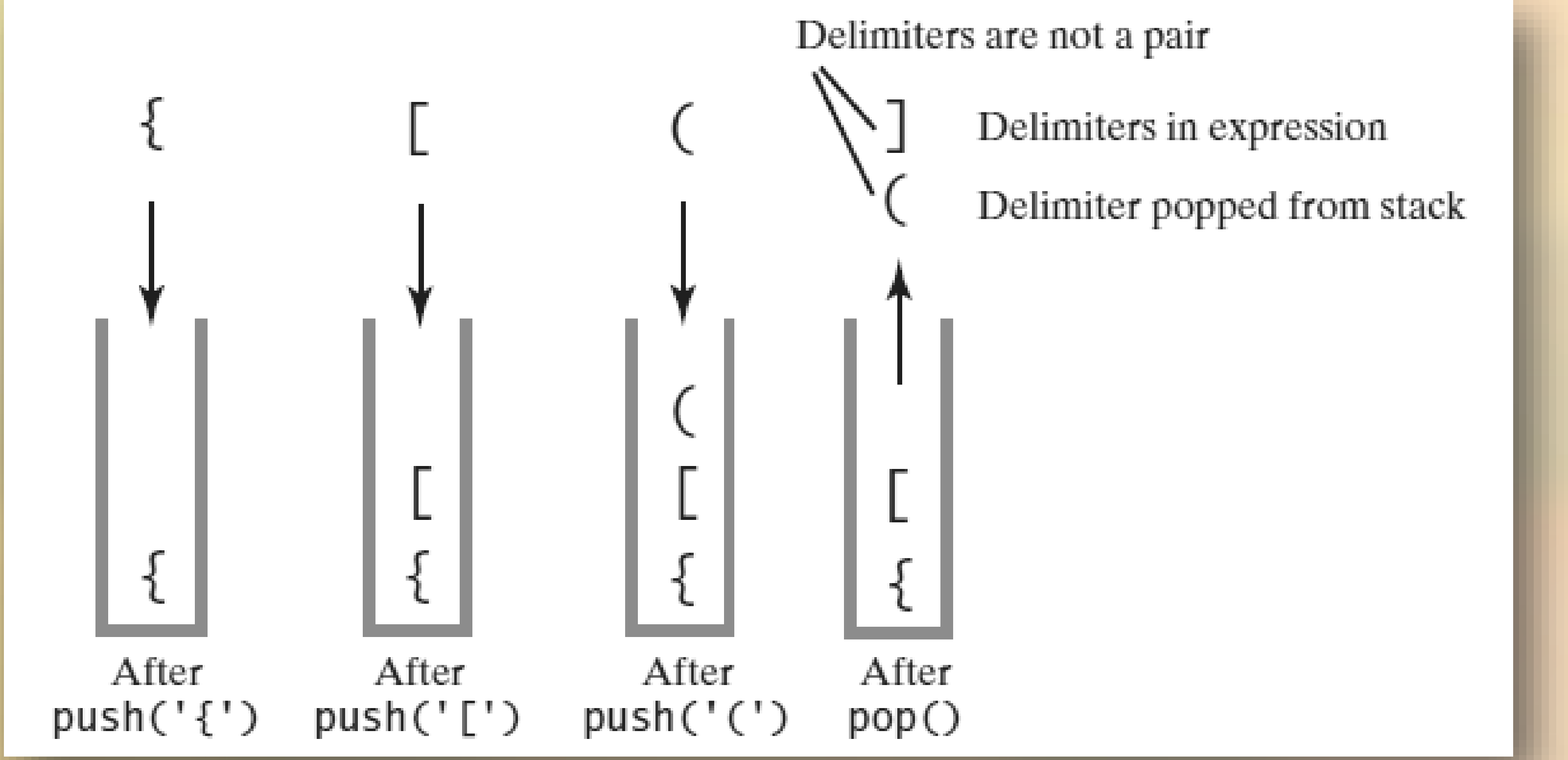


FIGURE 5-4 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( ]) }

Processing Algebraic Expressions

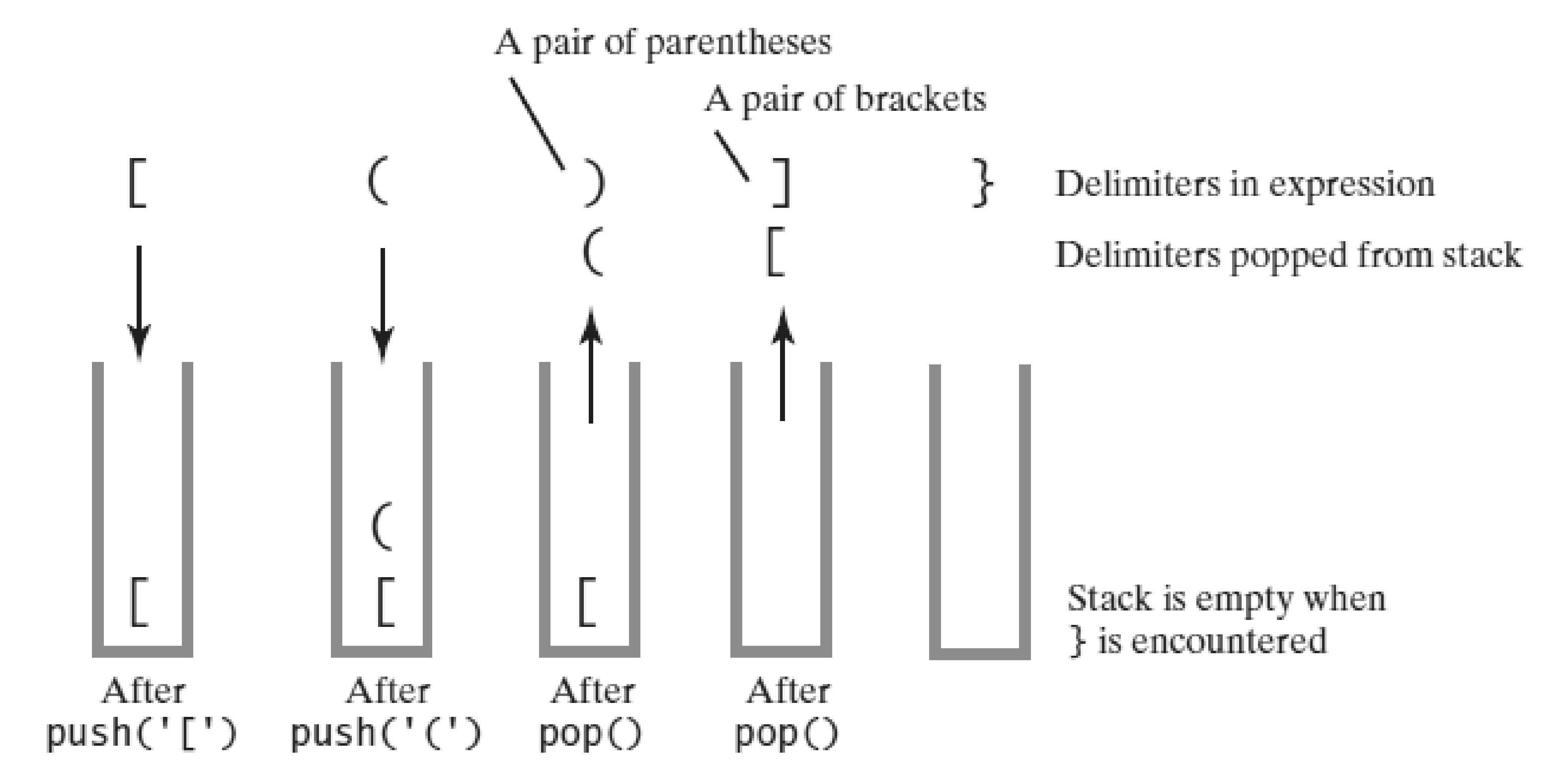


FIGURE 5-5 The contents of a stack during the scan of an expression that contains the unbalanced delimiters [( )] }

Processing Algebraic Expressions

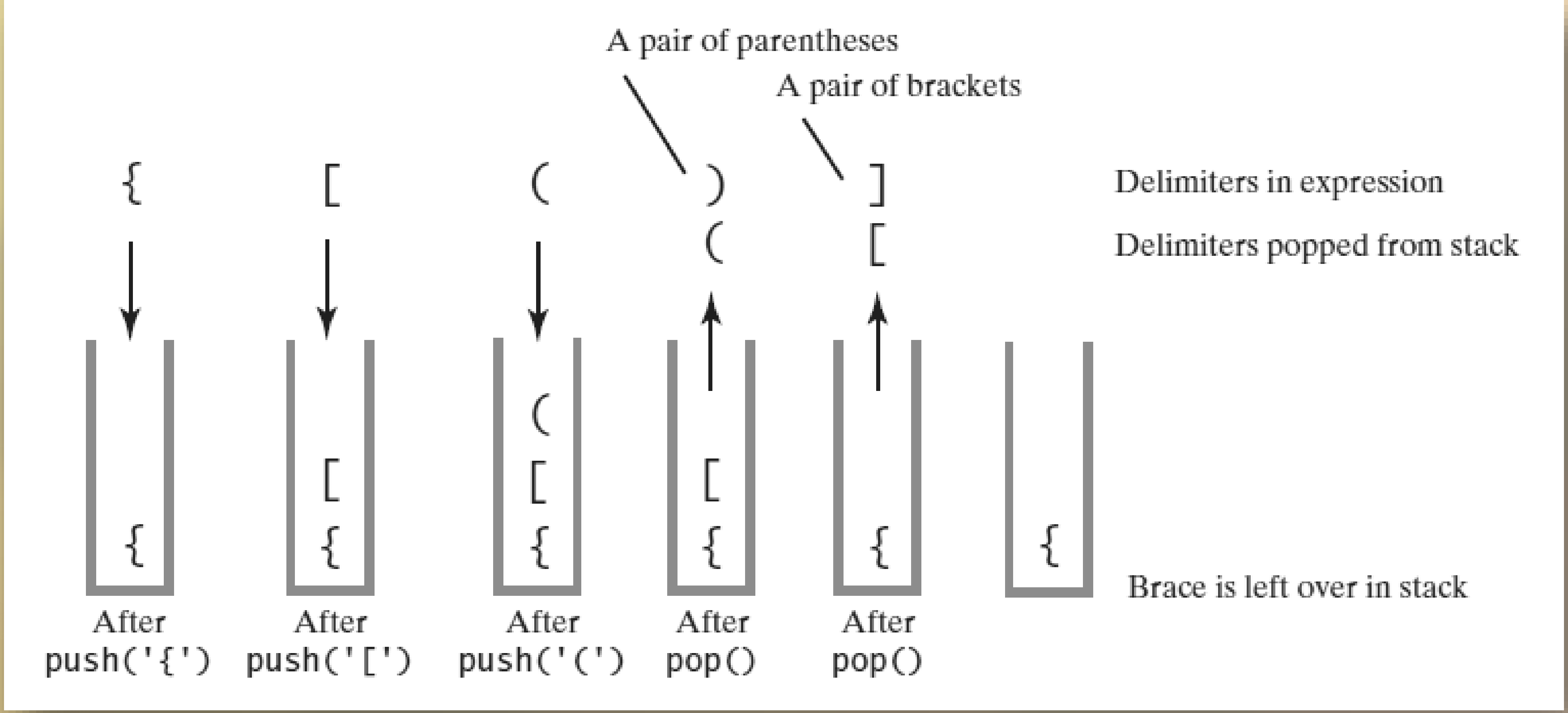
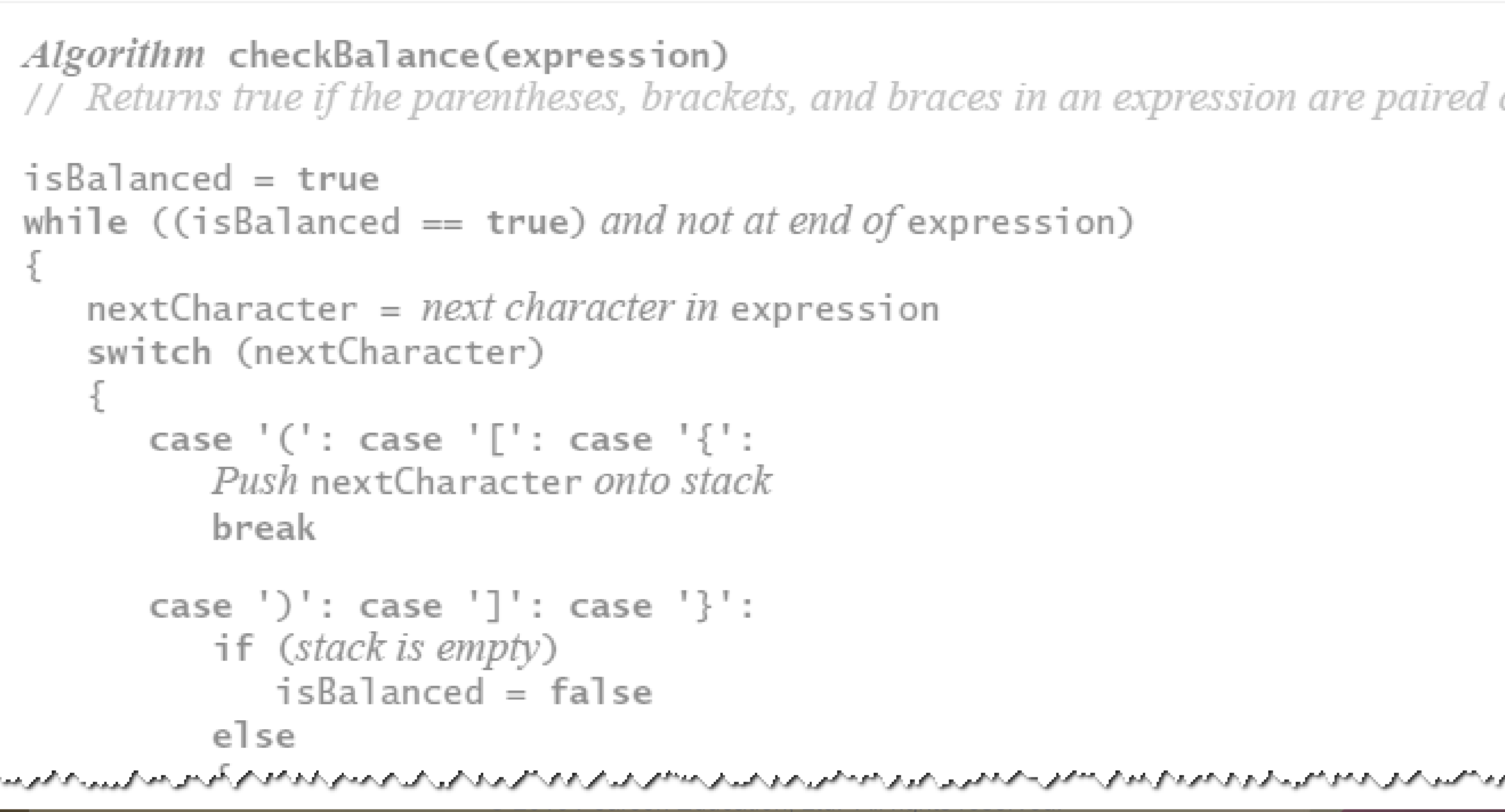


FIGURE 5-6 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( )]

Processing Algebraic Expressions

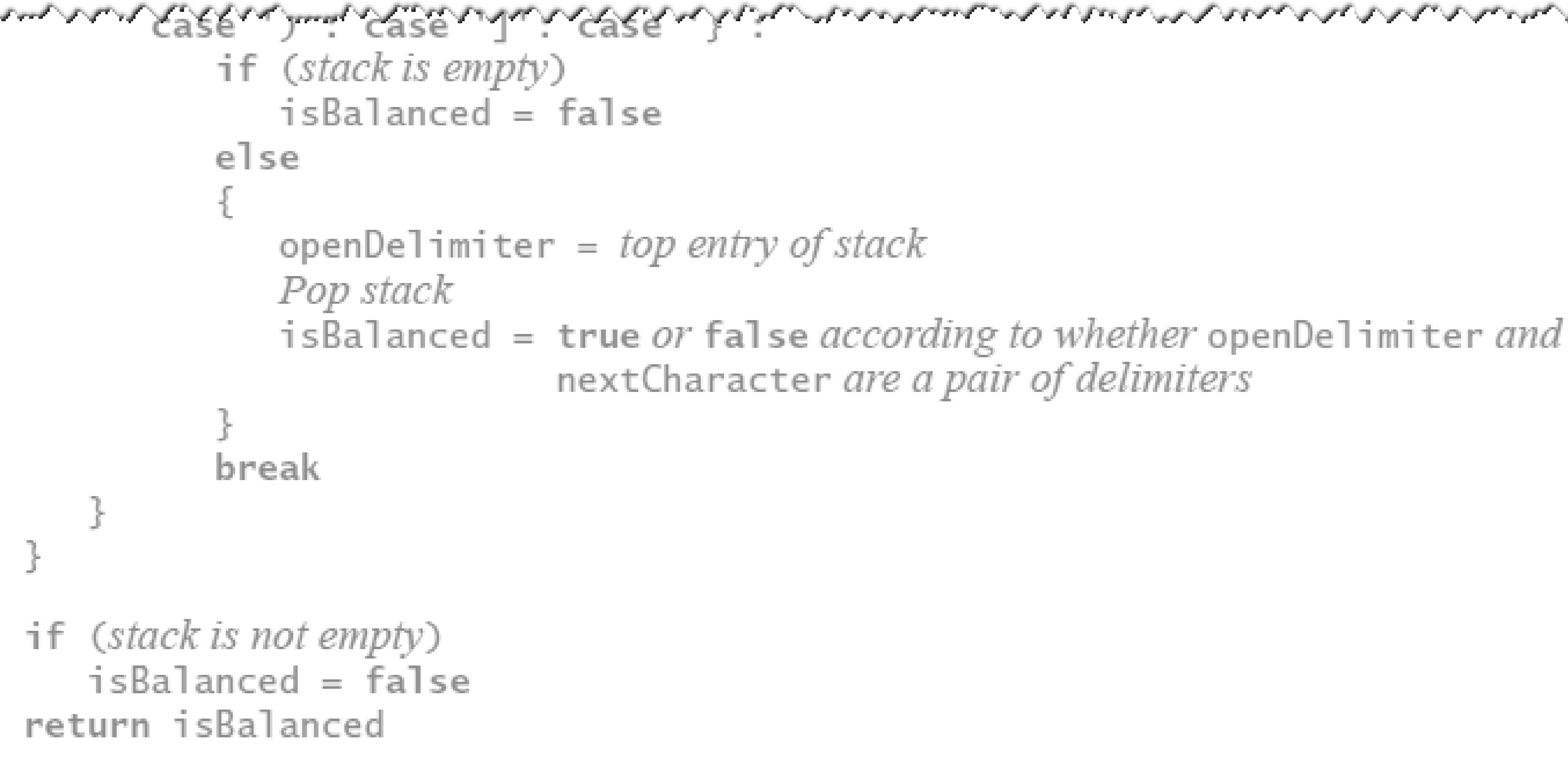


```
Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets, and braces in an expression are paired

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
         {
```

Processing Algebraic Expressions

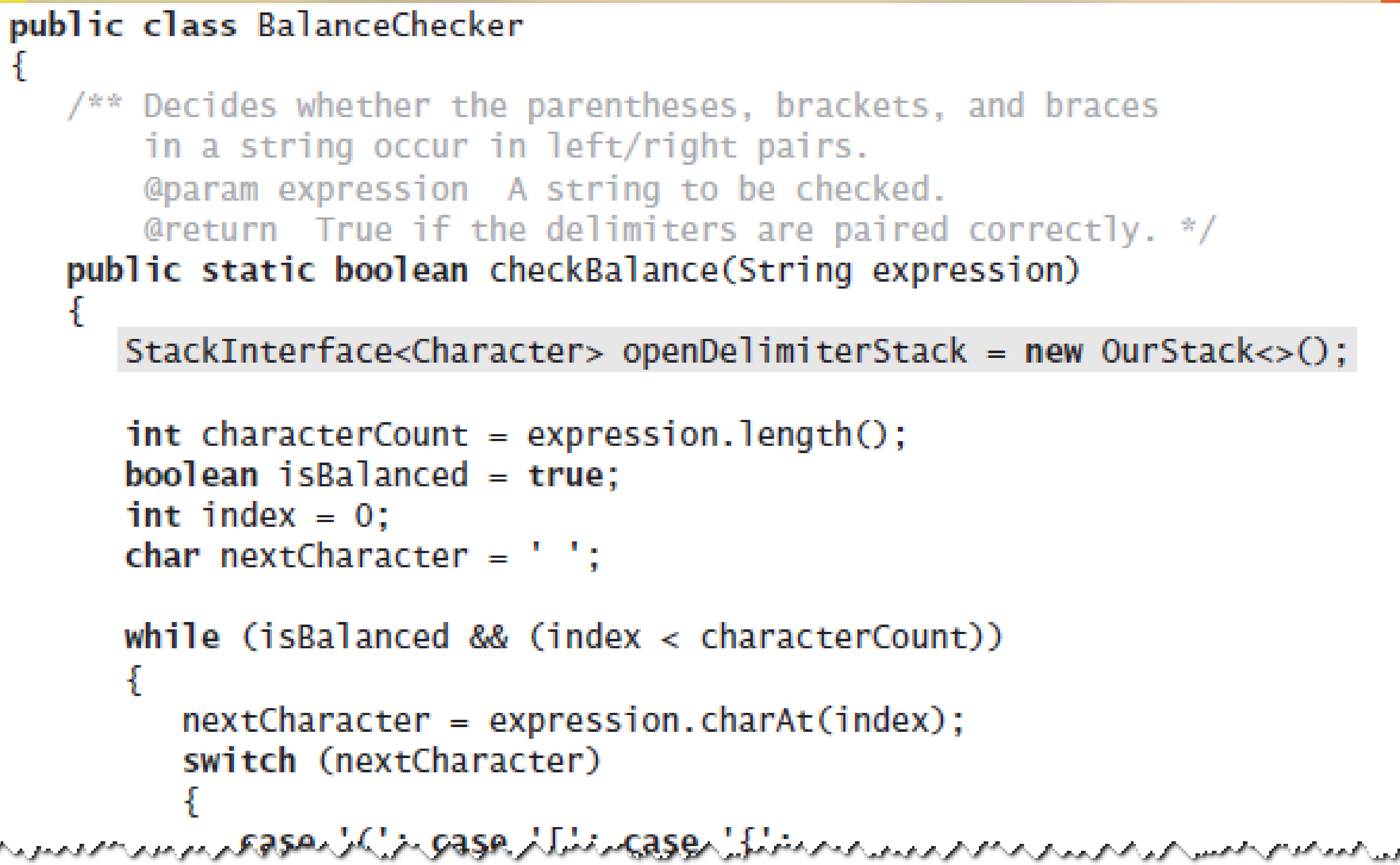


```
      case ')' : case ']' : case '}' :
         if (stack is empty)
            isBalanced = false
         else
         {
            openDelimiter = top entry of stack
            Pop stack
            isBalanced = true or false according to whether openDelimiter and
                         nextCharacter are a pair of delimiters
         }
         break
   }
}

if (stack is not empty)
   isBalanced = false
return isBalanced
```

Algorithm to process for balanced expression.

Java Implementation

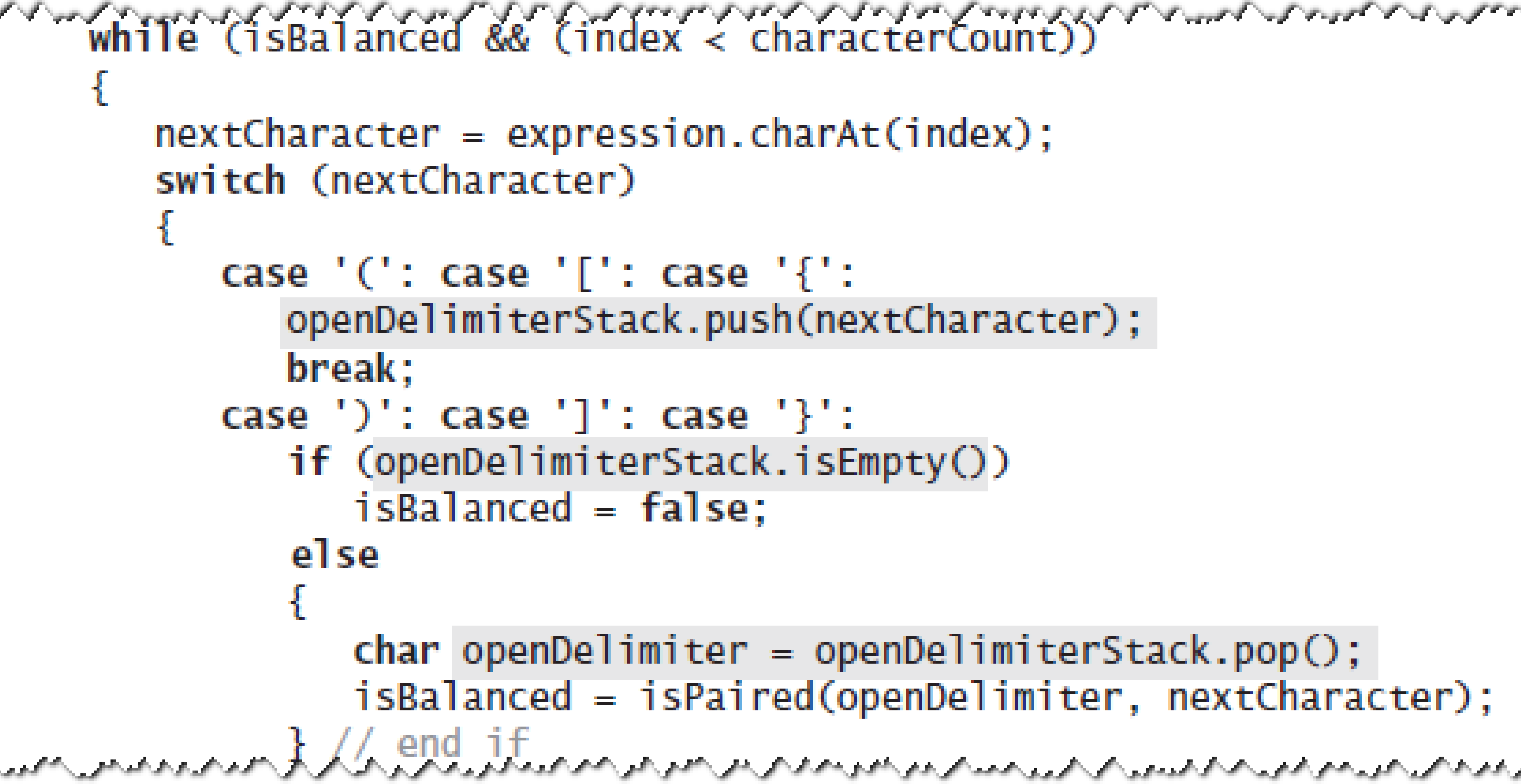


```
public class BalanceChecker
{
   /** Decides whether the parentheses, brackets, and braces
       in a string occur in left/right pairs.
       @param expression  A string to be checked.
       @return  True if the delimiters are paired correctly. */
   public static boolean checkBalance(String expression)
   {
      StackInterface<Character> openDelimiterStack = new OurStack<>();

      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 '{':
```

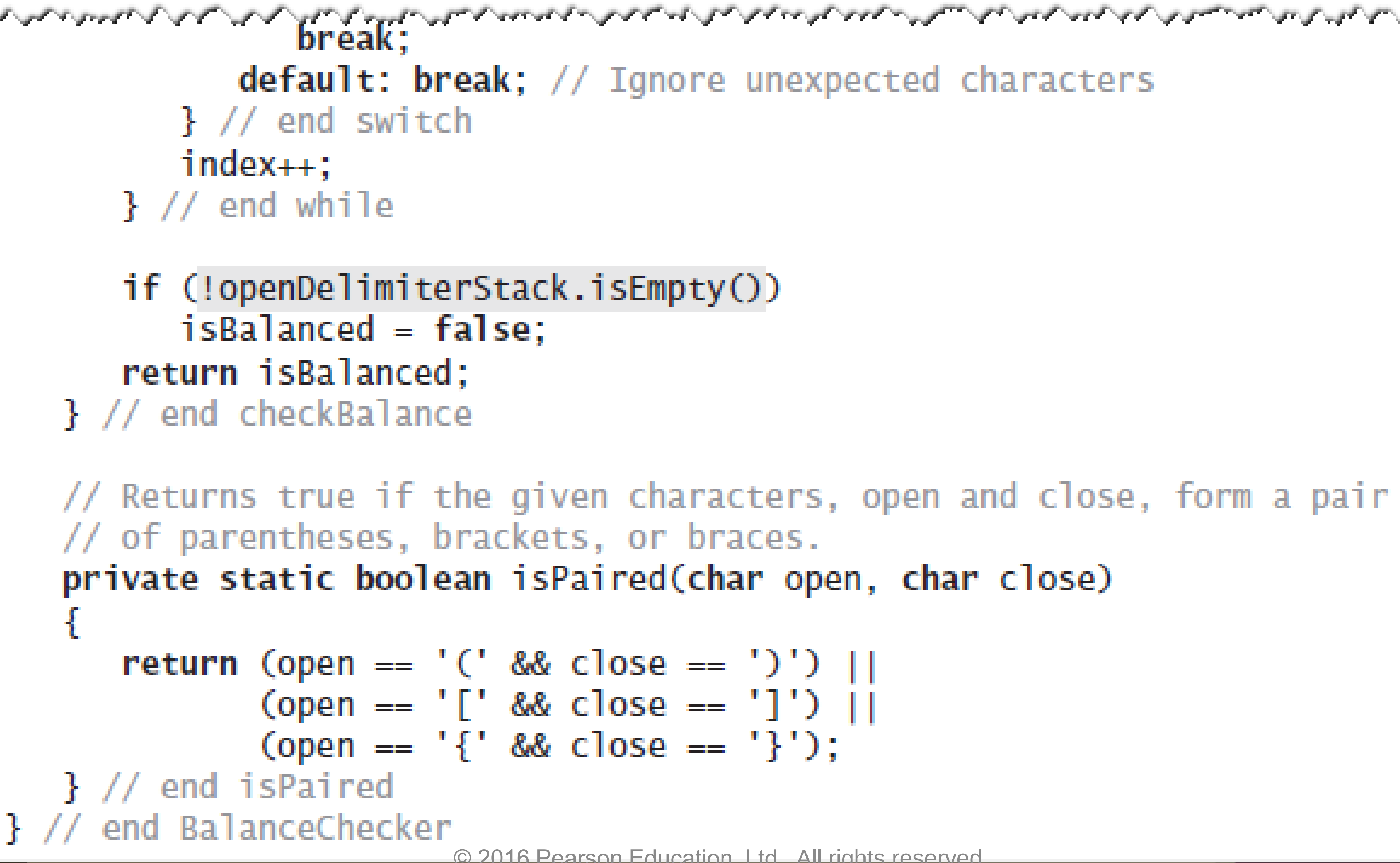
Java Implementation



```
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
```

LISTING 5-2 The class **BalanceChecker**

Java Implementation

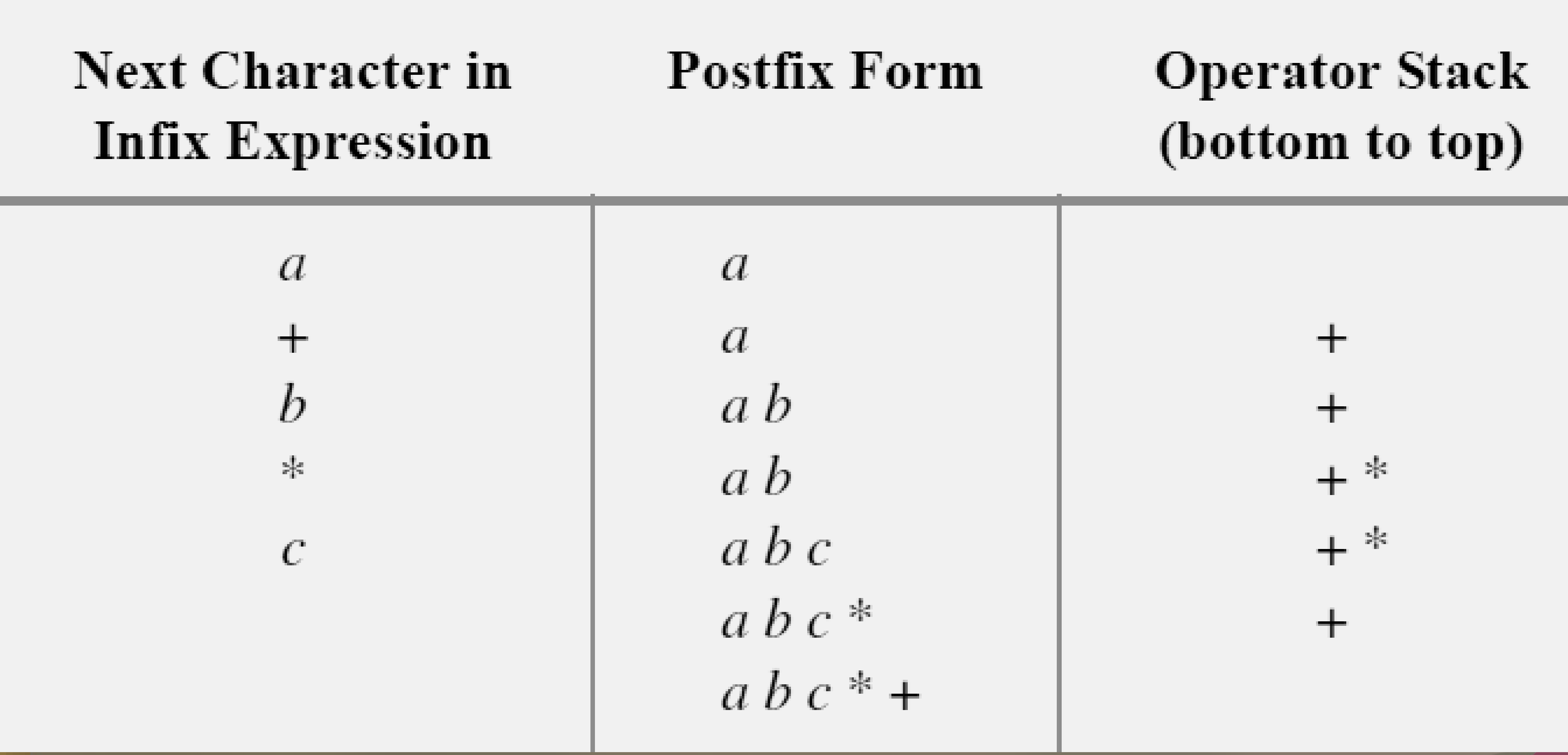


```
               break;
            default: break; // Ignore unexpected characters
         } // end switch
         index++;
      } // end while

      if (!openDelimiterStack.isEmpty())
         isBalanced = false;
      return isBalanced;
   } // end checkBalance

   // 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
} // end BalanceChecker
```

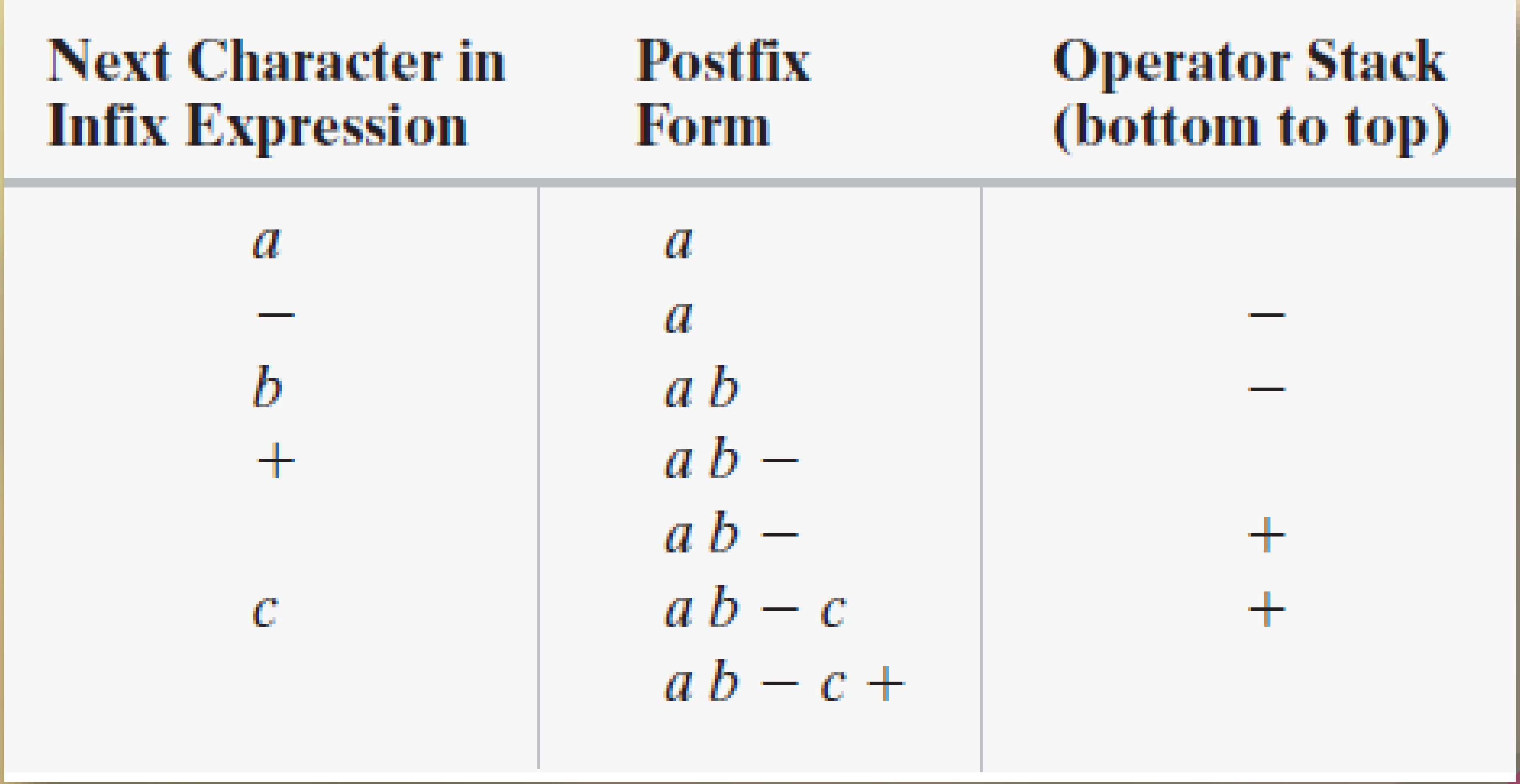
Infix to Postfix



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

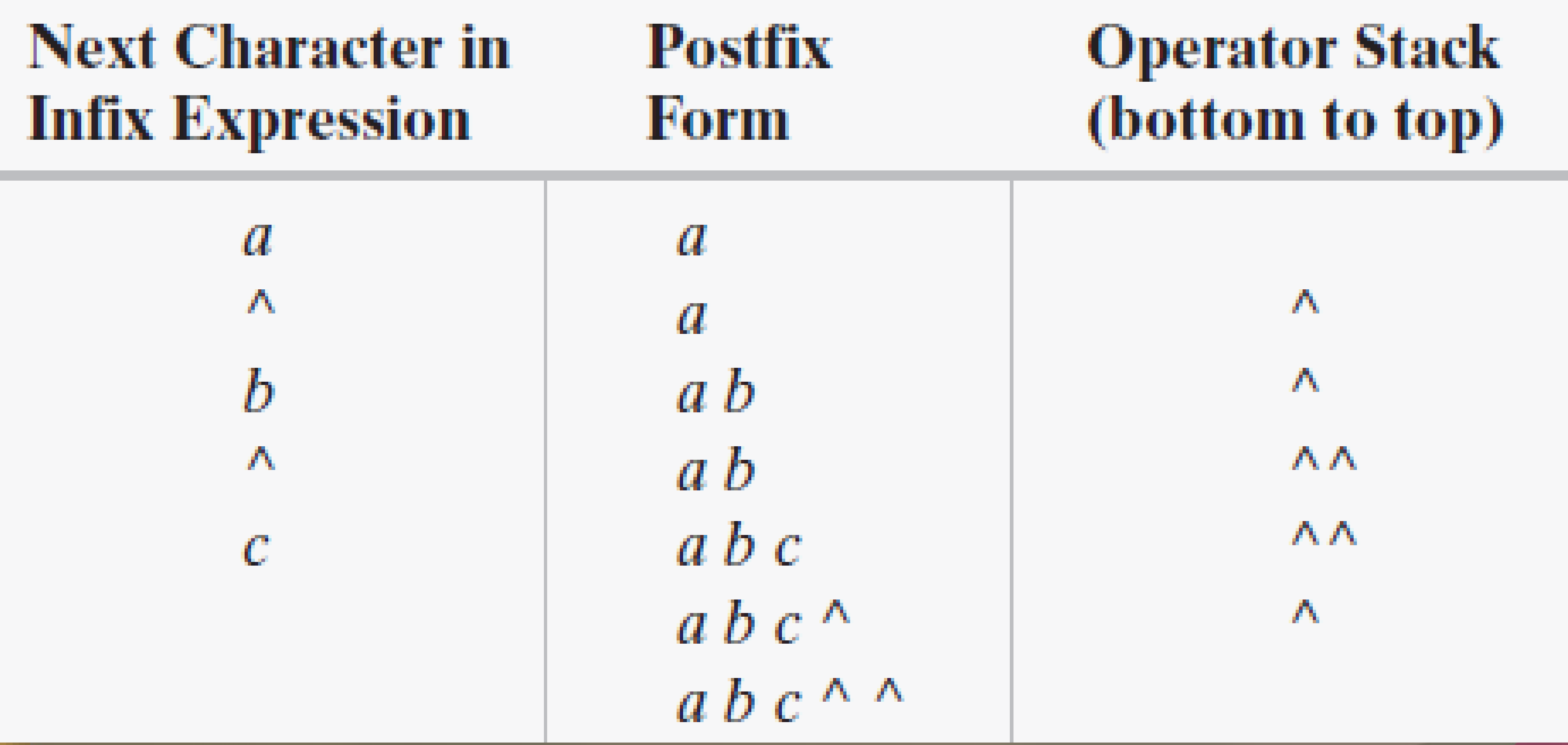
Successive Operators with Same Precedence



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) *a* - *b* + *c*;

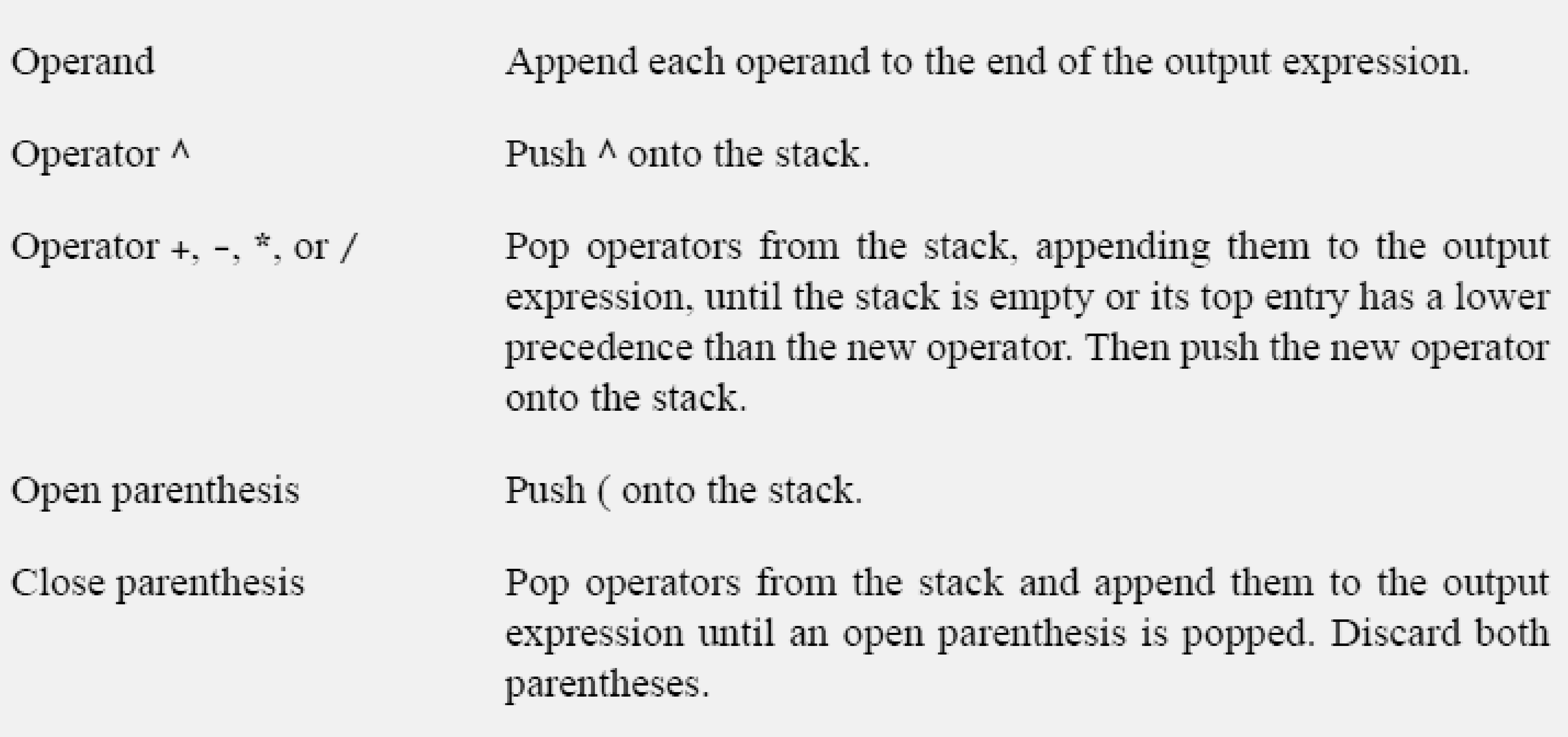
Successive Operators with Same Precedence



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 ^ ^	

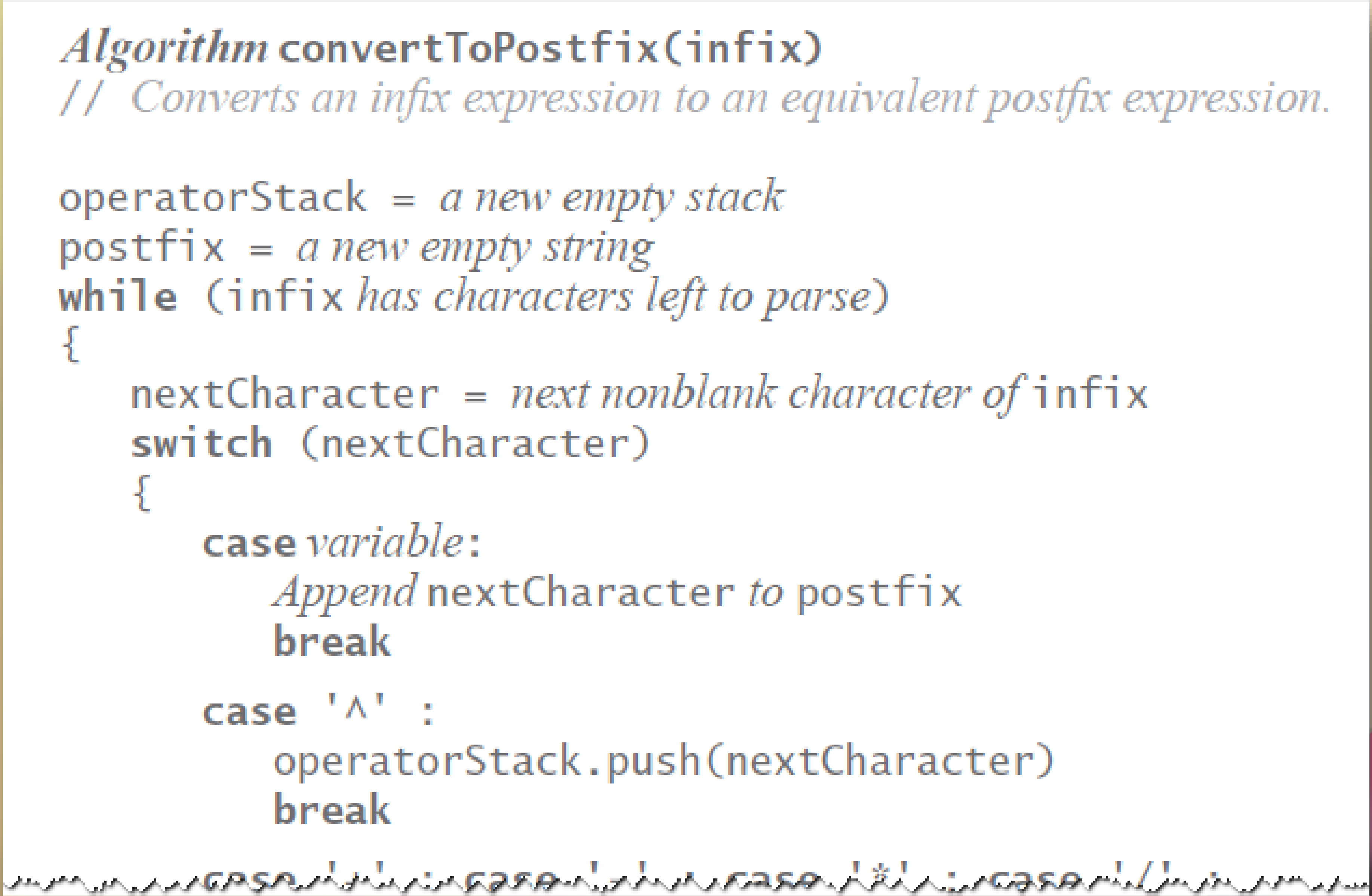
FIGURE 5-8 Converting an infix expression to postfix form: *a* ^ *b* ^ *c*

Infix-to-postfix Conversion



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 the stack is empty or its top entry has a lower precedence than the new 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



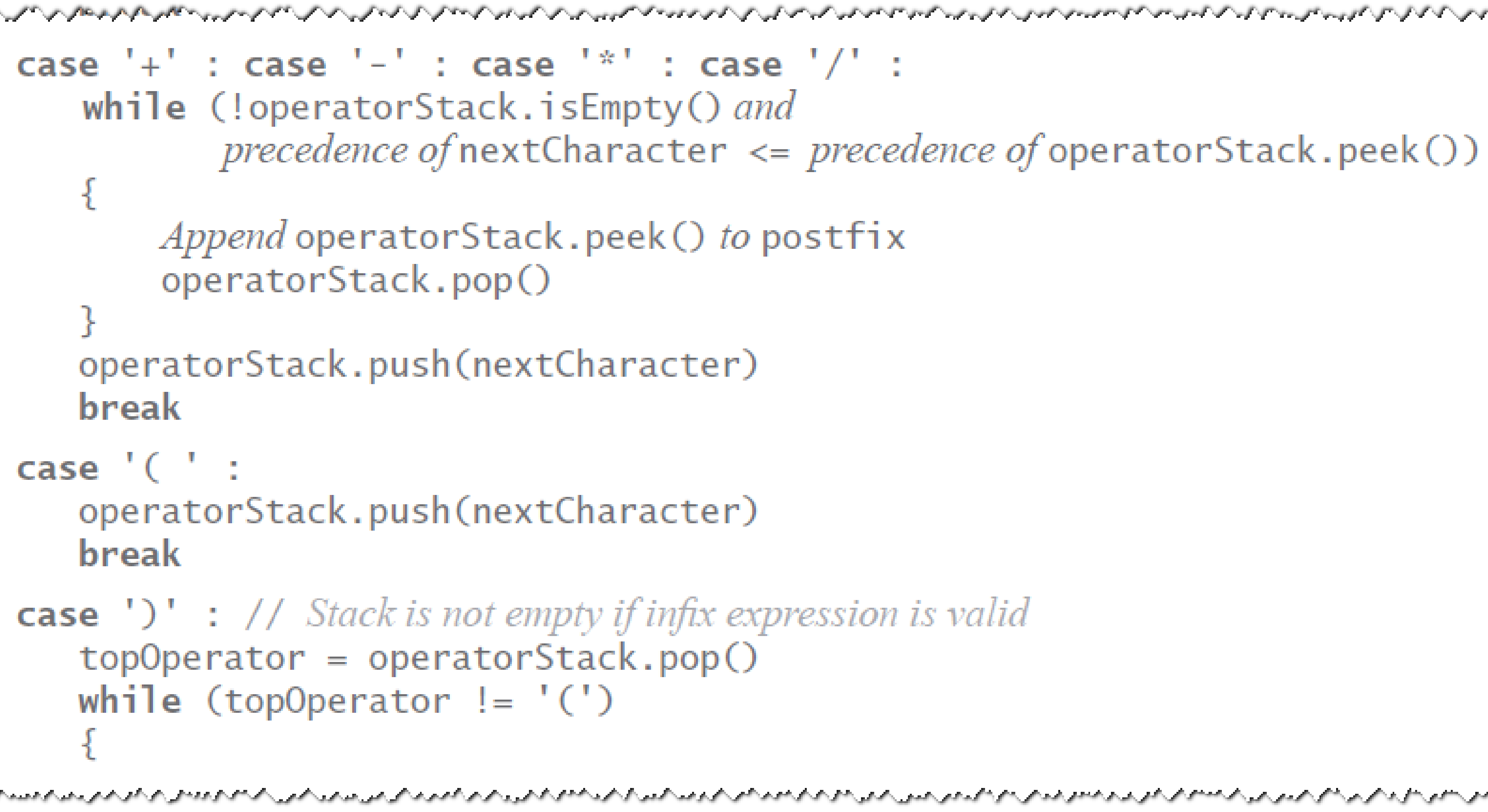
```
Algorithm convertToPostfix(infix)
// Converts an infix expression to an equivalent postfix expression.

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 '/' :
```

Infix-to-postfix Algorithm

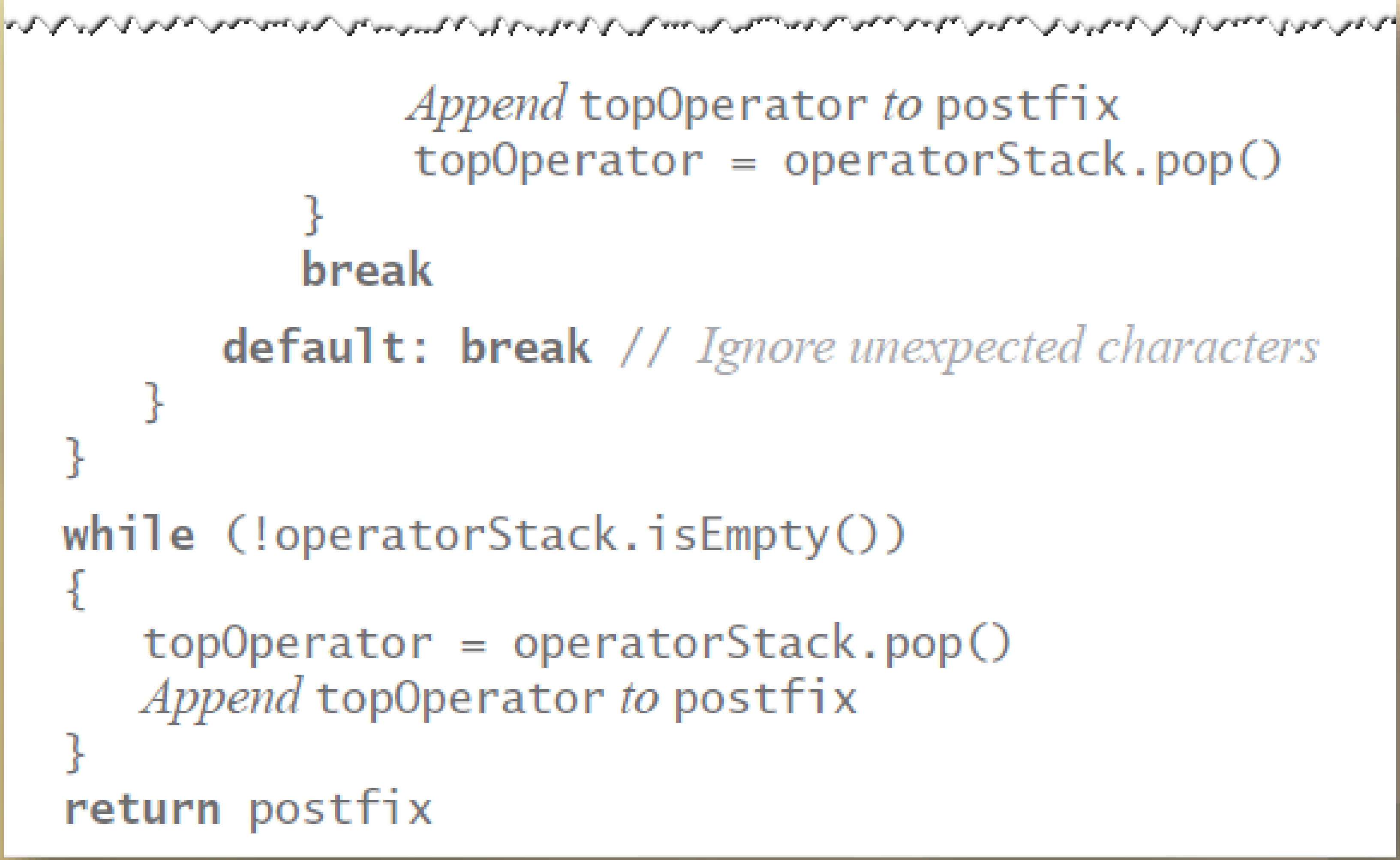


```
case '+' : case '-' : case '*' : case '/' :
    while (!operatorStack.isEmpty() and
            precedence of nextCharacter <= precedence of operatorStack.peek())
    {
        Append operatorStack.peek() to postfix
        operatorStack.pop()
    }
    operatorStack.push(nextCharacter)
    break

case '( ' :
    operatorStack.push(nextCharacter)
    break

case ')' : // Stack is not empty if infix expression is valid
    topOperator = operatorStack.pop()
    while (topOperator != '(')
    {
```

Infix-to-postfix Algorithm

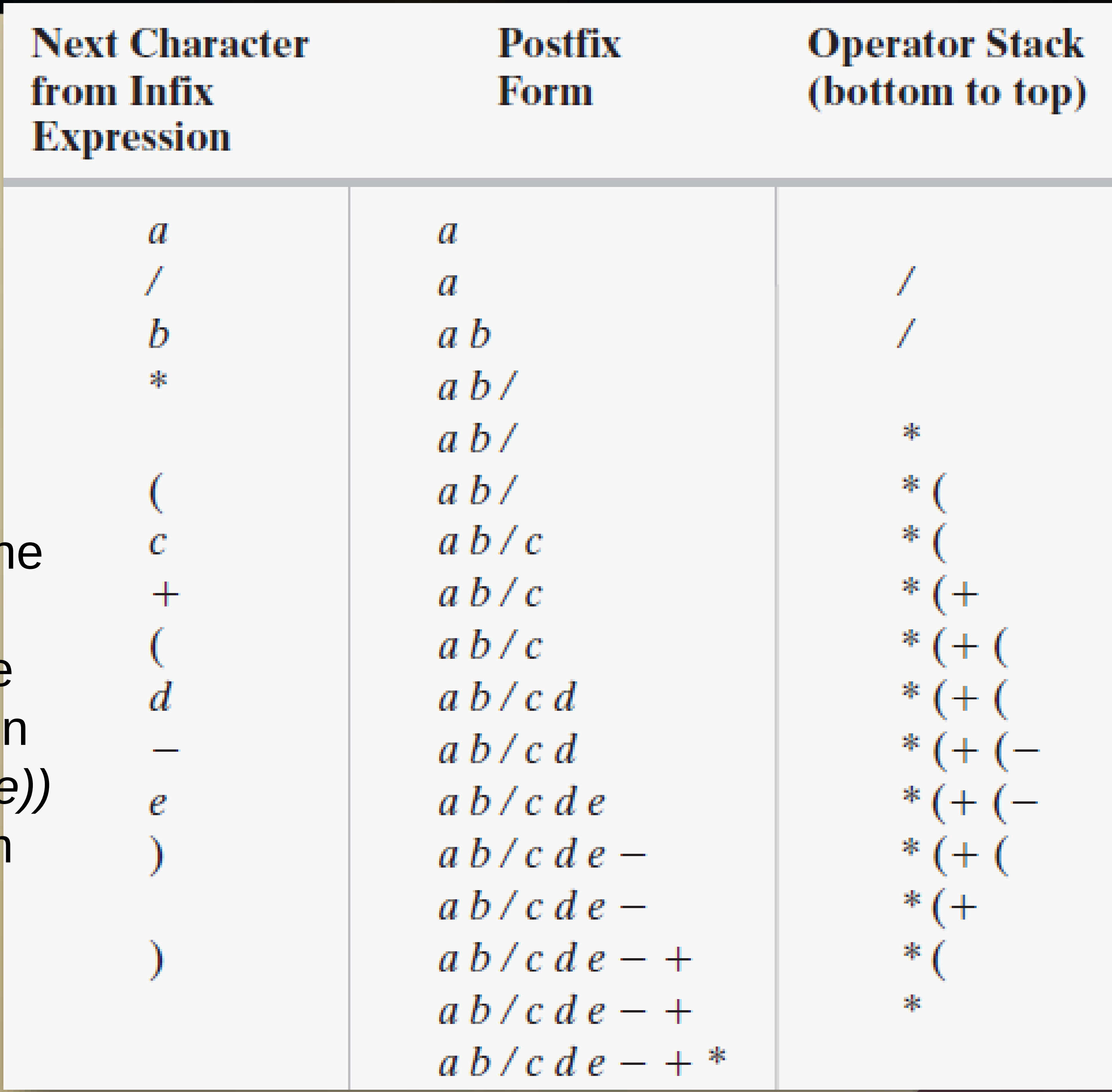


```
                Append topOperator to postfix
                topOperator = operatorStack.pop()
            }
            break

        default: break // Ignore unexpected characters
    }
}

while (!operatorStack.isEmpty())
{
    topOperator = operatorStack.pop()
    Append topOperator to postfix
}
return postfix
```

Infix to Postfix



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

Evaluating Postfix Expressions

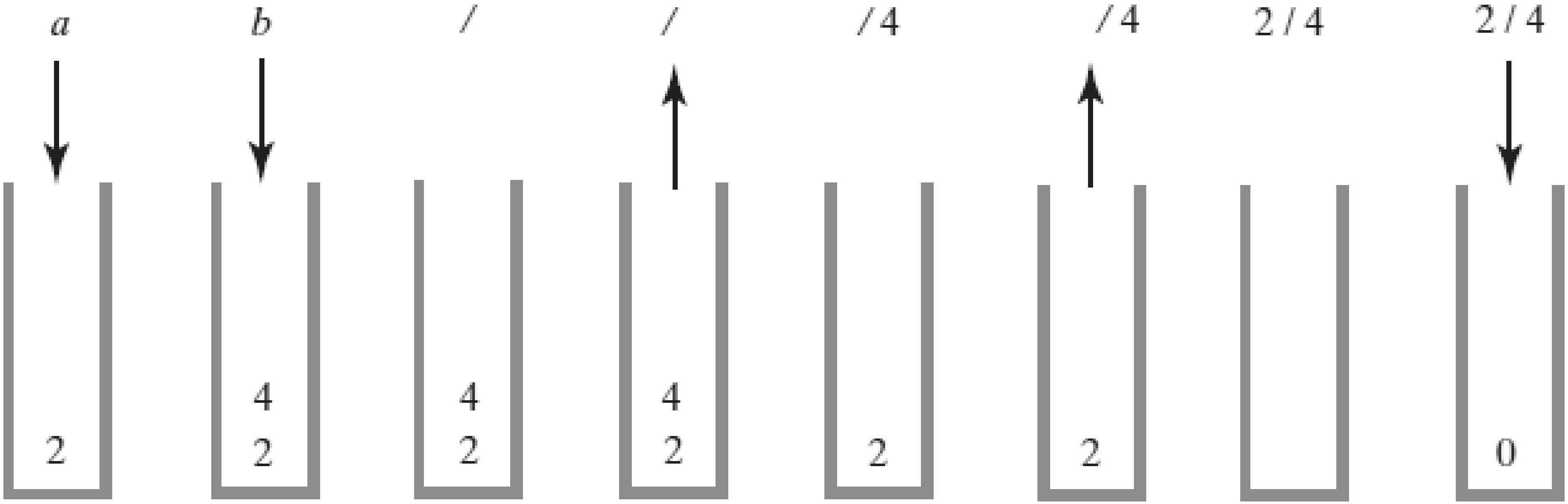


FIGURE 5-10 The stack during the evaluation of the postfix expression *a b /* when *a* is 2 and b is 4

Evaluating Postfix Expressions

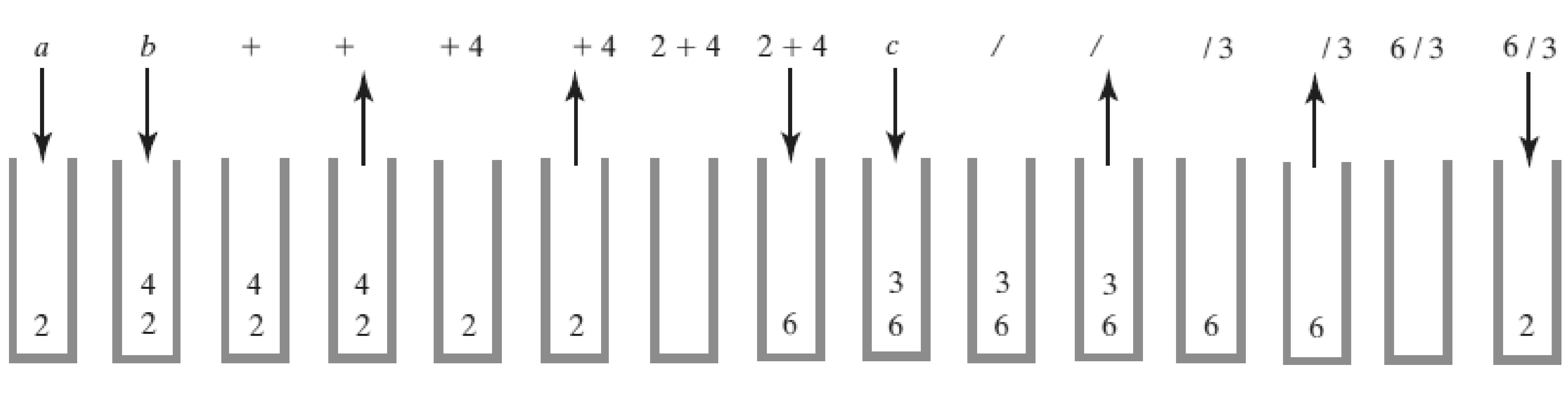
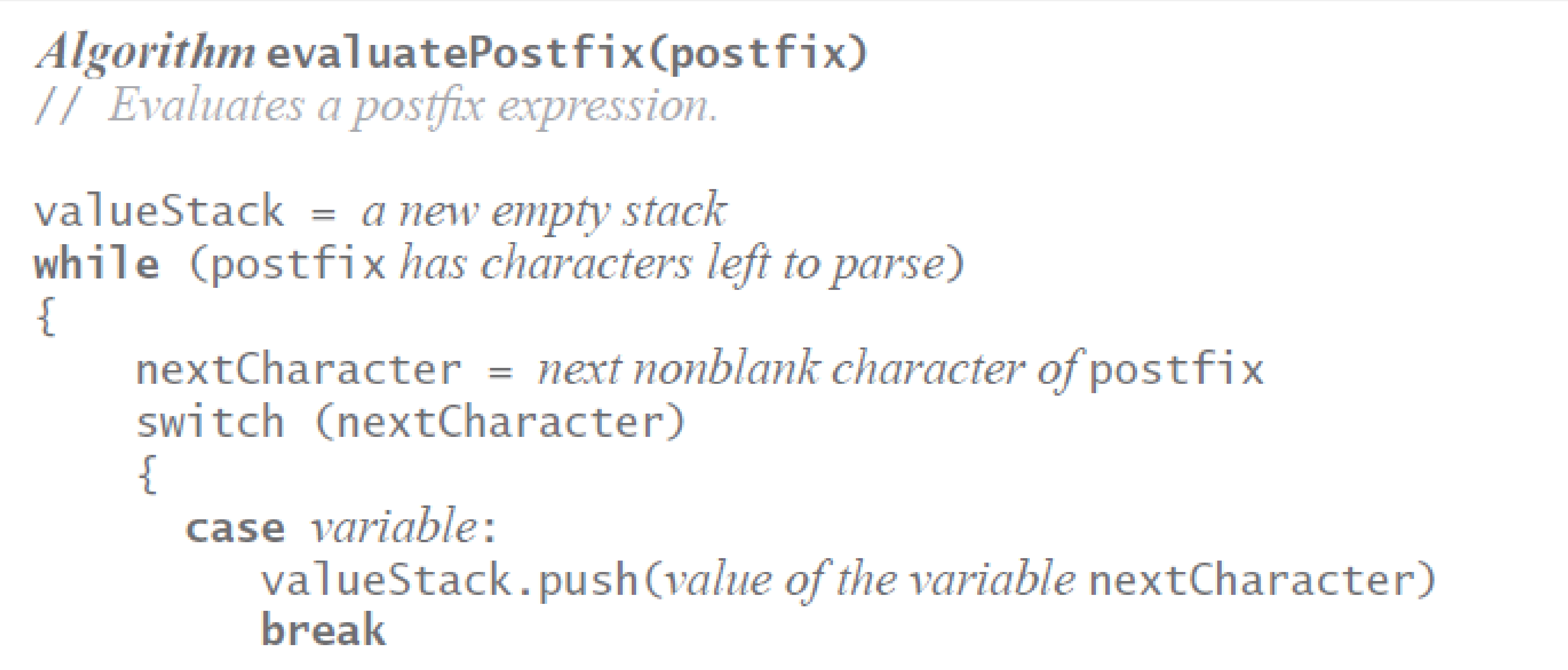


FIGURE 5-11 The stack during the evaluation of the postfix expression a b + c / when *a* is 2, *b* is 4, and *c* is 3

Evaluating Postfix Expressions

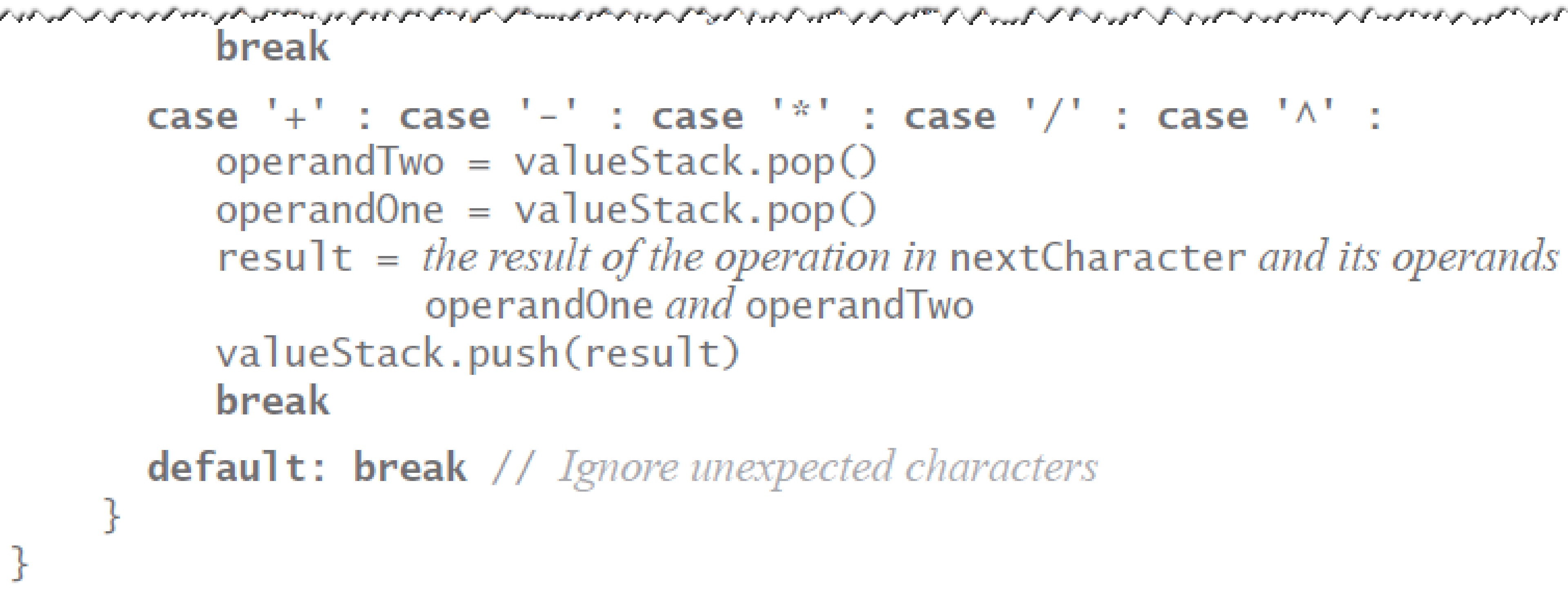


```
Algorithm evaluatePostfix(postfix)
// Evaluates a postfix expression.

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
```

Algorithm for evaluating postfix expressions.

Evaluating Postfix Expressions



```
            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
    }
}
```

Algorithm for evaluating postfix expressions.

Evaluating Infix Expressions

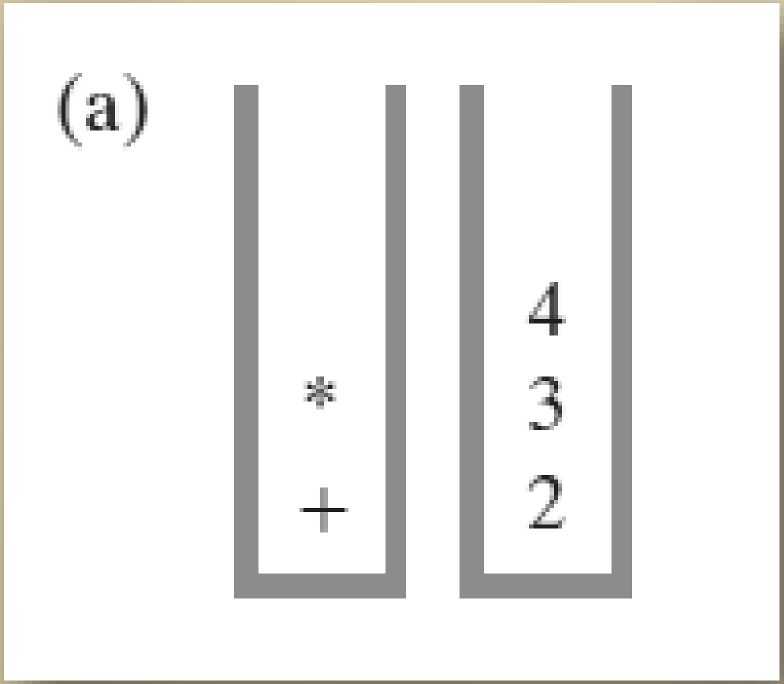


FIGURE 5-12 Two stacks during the evaluation of a + b * c when a is 2, b is 3, and c is 4:

(a) after reaching the end of the expression;

Evaluating Infix Expressions

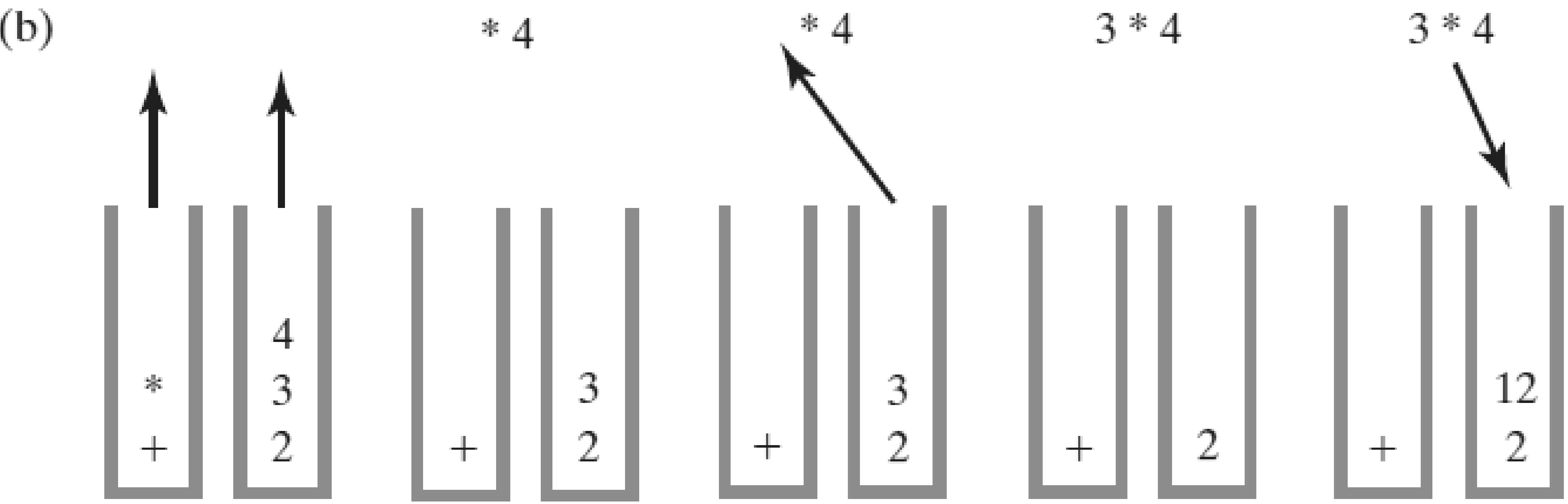


FIGURE 5-12 Two stacks during the evaluation of a + b * c when a is 2, b is 3, and c is 4: (b) while performing the multiplication;

Evaluating Infix Expressions

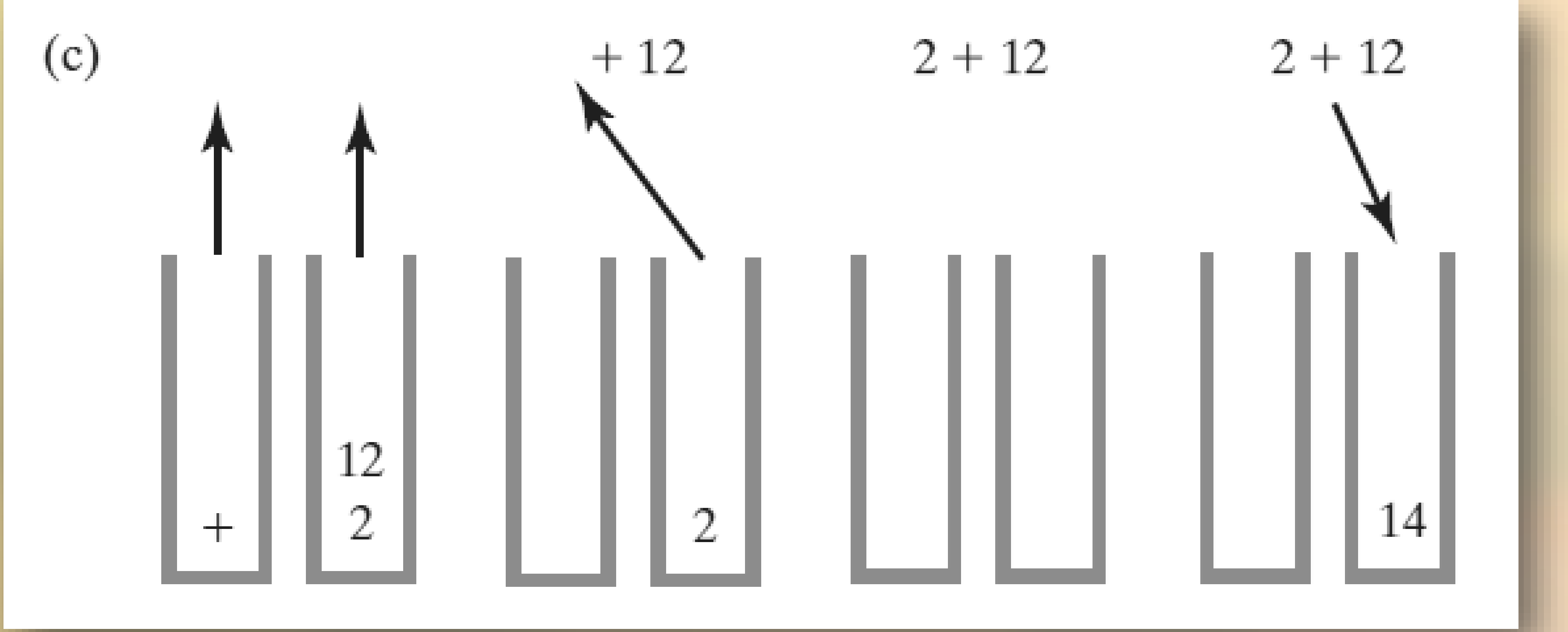
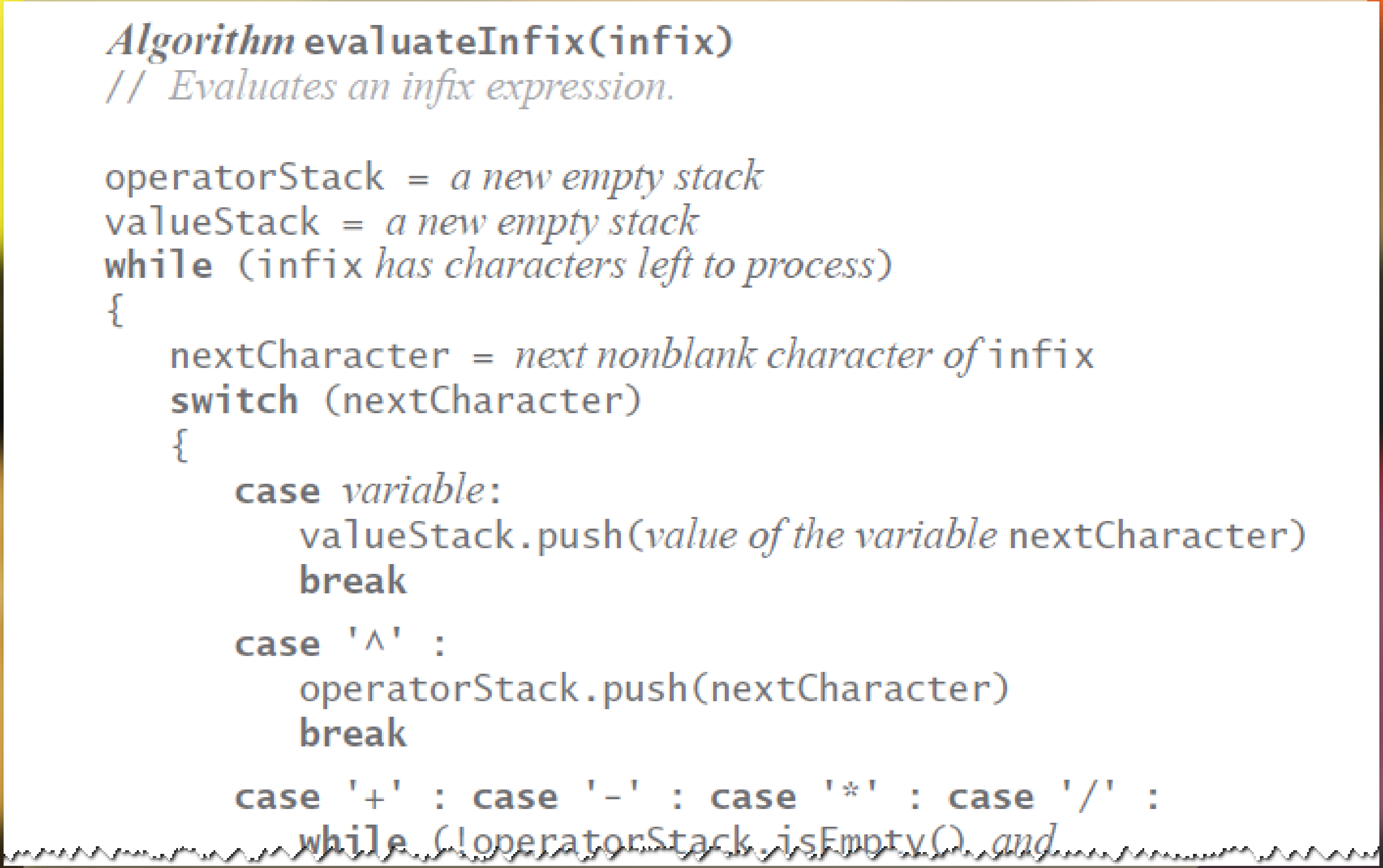


FIGURE 5-12 Two stacks during the evaluation of a + b * c when a is 2, b is 3, and c is 4: (c) while performing the addition

Evaluating Infix Expressions



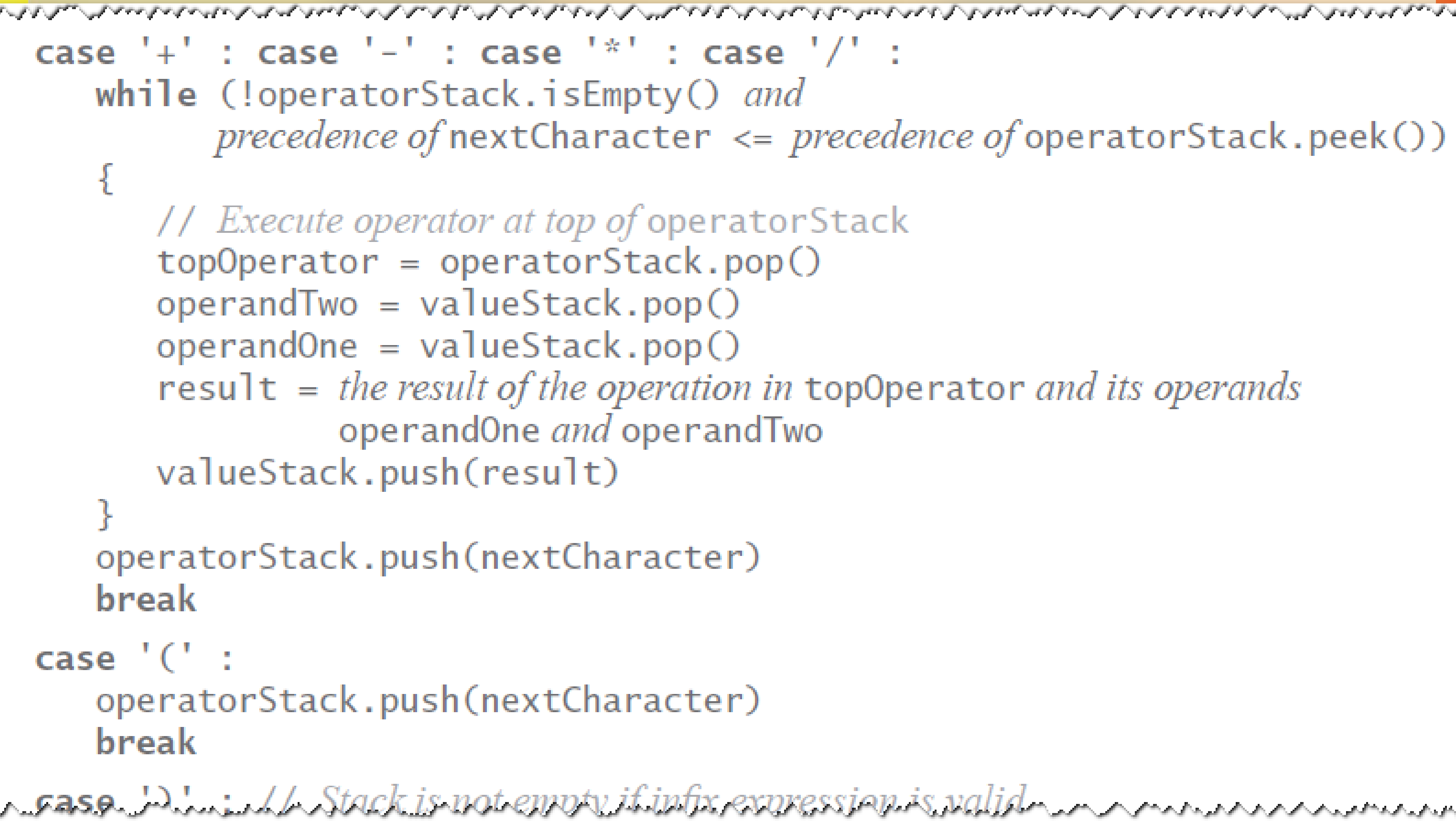
```
Algorithm evaluateInfix(infix)
// Evaluates an infix expression.

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
```

Evaluating Infix Expressions

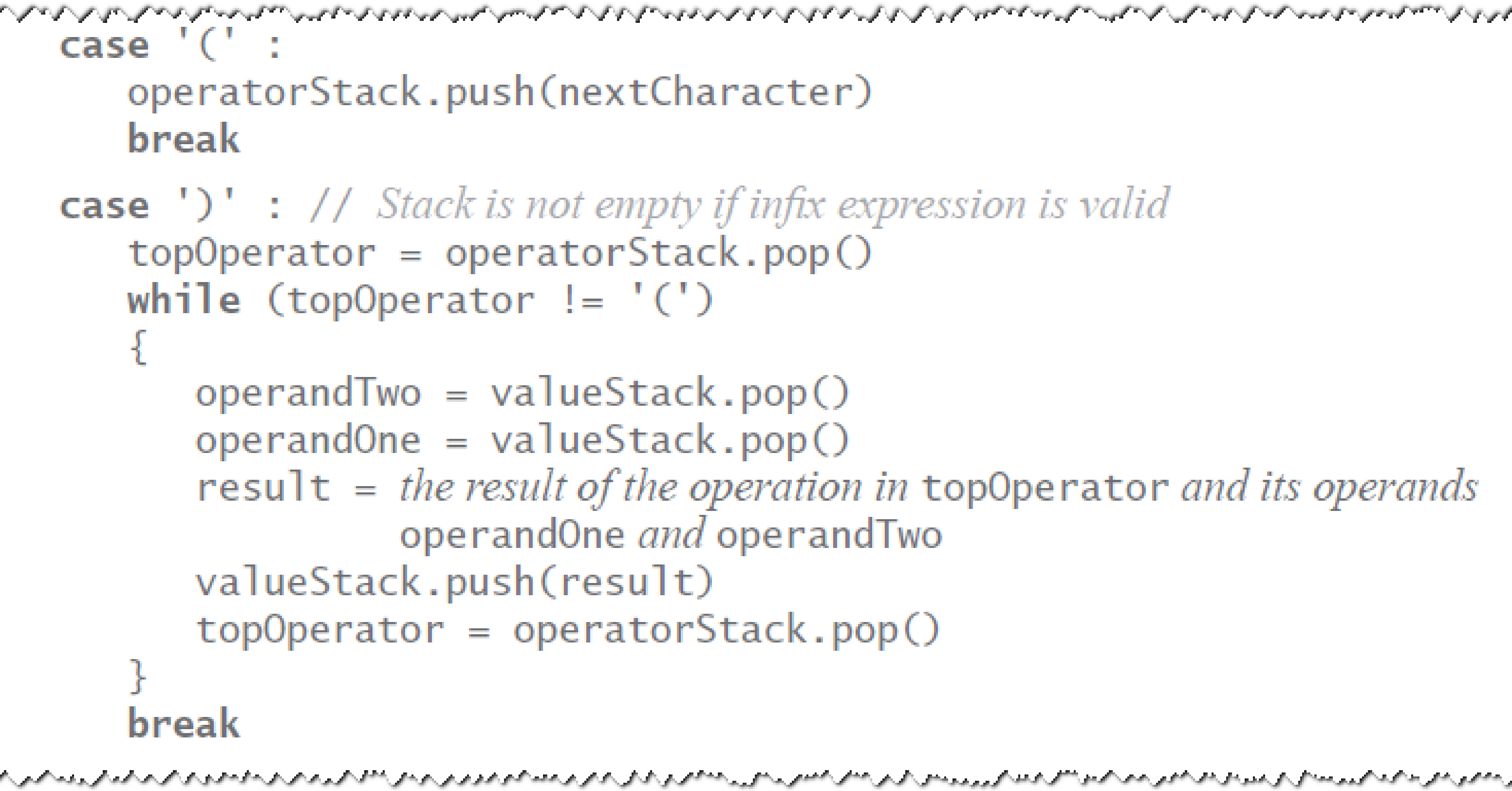


```
case '+' : case '-' : case '*' : case '/' :
   while (!operatorStack.isEmpty() and
         precedence of nextCharacter <= precedence of operatorStack.peek())
   {
      // Execute operator at top of operatorStack
      topOperator = operatorStack.pop()
      operandTwo = valueStack.pop()
      operandOne = valueStack.pop()
      result = the result of the operation in topOperator and its operands
               operandOne and operandTwo
      valueStack.push(result)
   }
   operatorStack.push(nextCharacter)
   break

case '(' :
   operatorStack.push(nextCharacter)
   break

case ')' : // Stack is not empty if infix expression is valid
```

Evaluating Infix Expressions

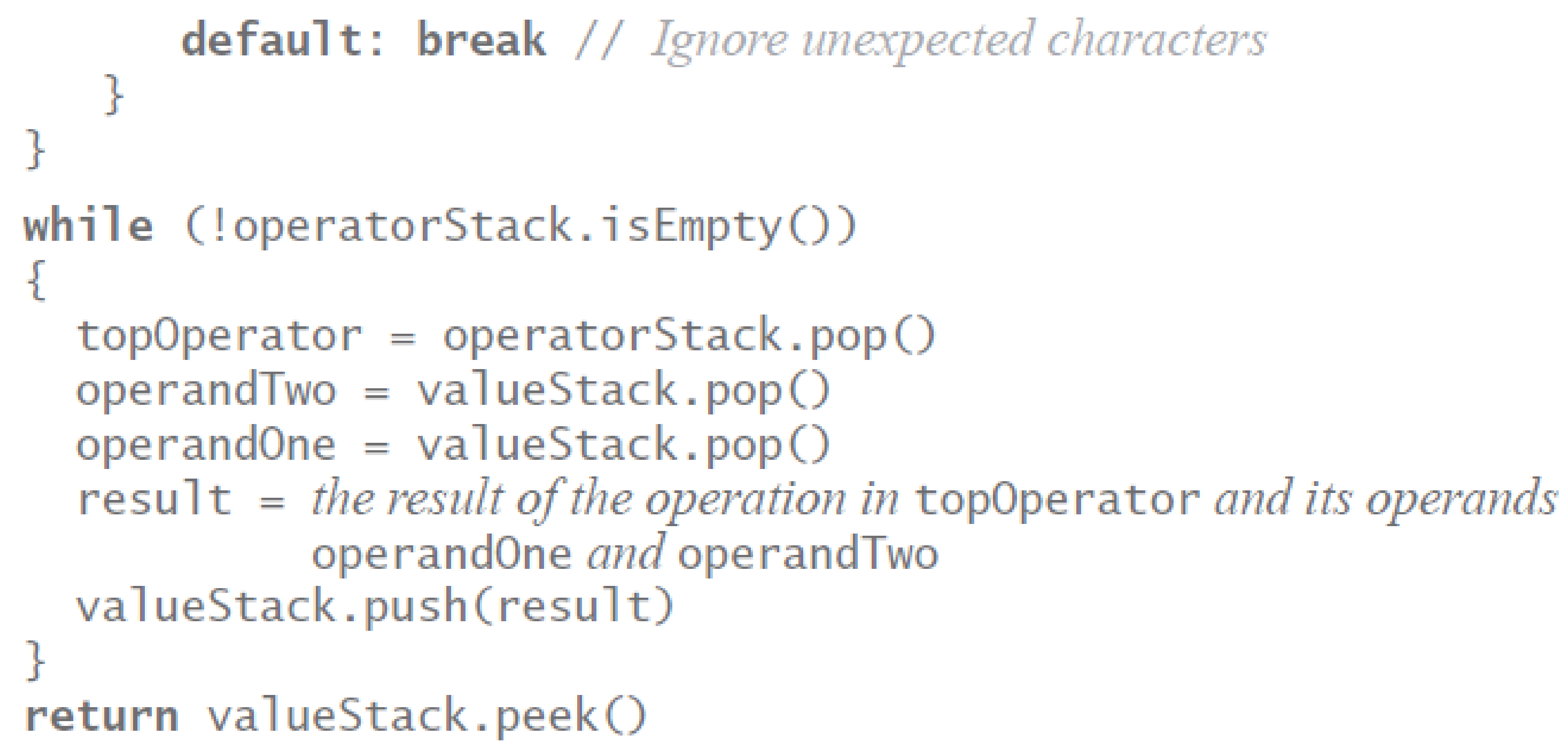


```
case '(' :
   operatorStack.push(nextCharacter)
   break

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
```

Algorithm to evaluate infix expression.

Evaluating Infix Expressions



```
         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()
```

Algorithm to evaluate infix expression.

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