

# Lecture 10: Stacks

## CS 0445: Data Structures

**Constantinos Costa**

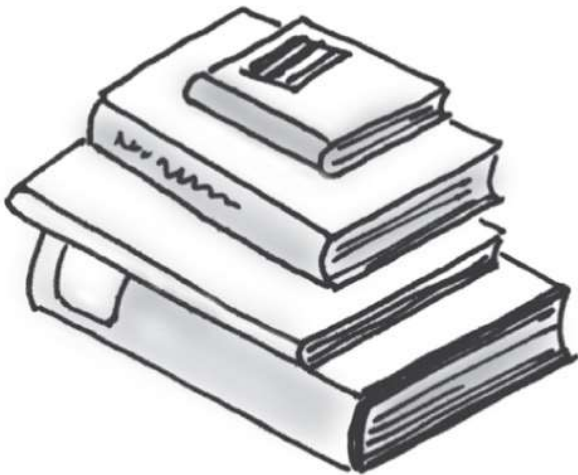
<http://db.cs.pitt.edu/courses/cs0445/current.term/>

Sep 23, 2019, 8:00-9:15  
University of Pittsburgh, Pittsburgh, PA



# Stacks

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



© 2019 Pearson Education, Inc.



# Specifications of the ADT Stack

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

Pseudocode	UML	Description
<b>push(newEntry)</b>	<b>+push(newEntry: T): void</b>	Task: Adds a new entry to the top of the stack. Input: newEntry is the new entry. Output: None.
<b>pop()</b>	<b>+pop(): T</b>	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.
<b>peek()</b>	<b>+peek(): T</b>	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.
<b>isEmpty()</b>	<b>+isEmpty(): boolean</b>	Task: Detects whether the stack is empty. Input: None. Output: Returns true if the stack is empty.
<b>clear()</b>	<b>+clear(): void</b>	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();

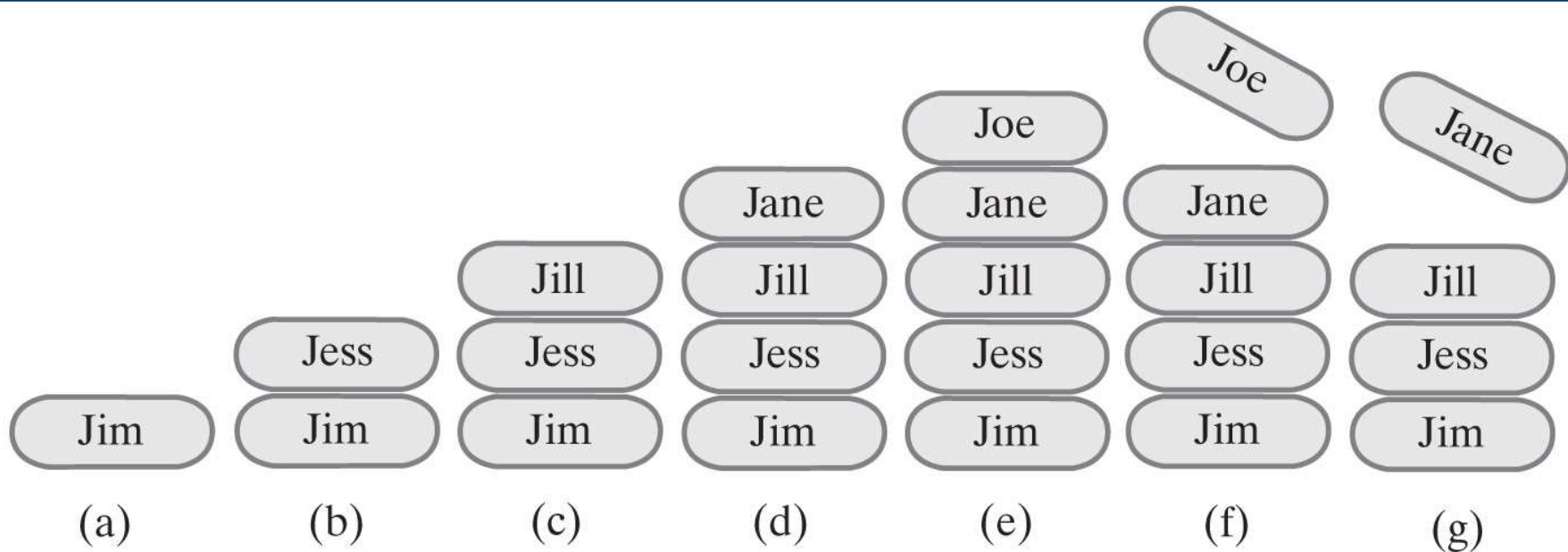
    /** Retrieves this stack's top entry.
     * @return The object at the top of the stack.
     * @throws EmptyStackException if the stack is empty. */
    public T peek();

    /** 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
```



# Example of a Stack



© 2019 Pearson Education, Inc.

```
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();
```



# 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

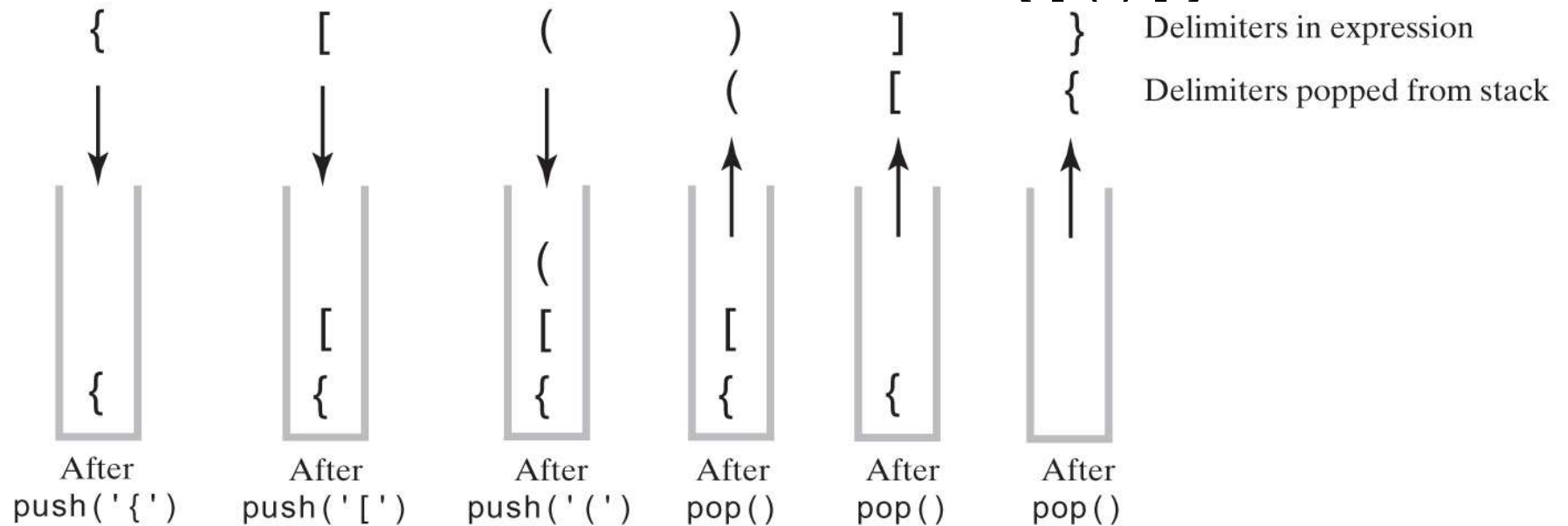
- Infix:
  - each binary operator appears between its operands  
 $a + b$
- Prefix:
  - each binary operator appears before its operands  
 $+ a b$
- Postfix:
  - each binary operator appears after its operands  
 $a b +$
- Balanced expressions: delimiters paired correctly





# Processing Algebraic Expressions

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

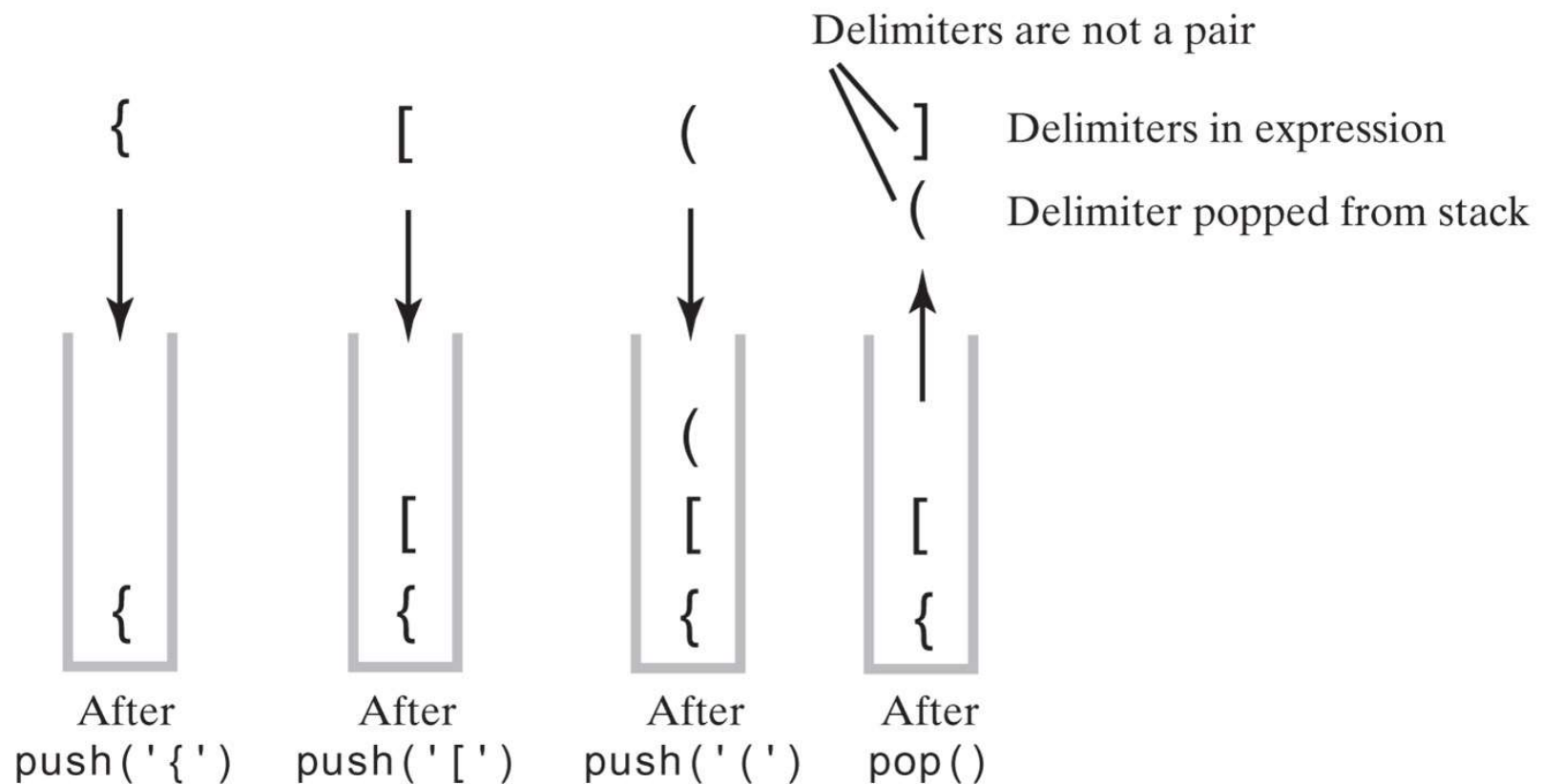


© 2019 Pearson Education, Inc.



# Processing Algebraic Expressions

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

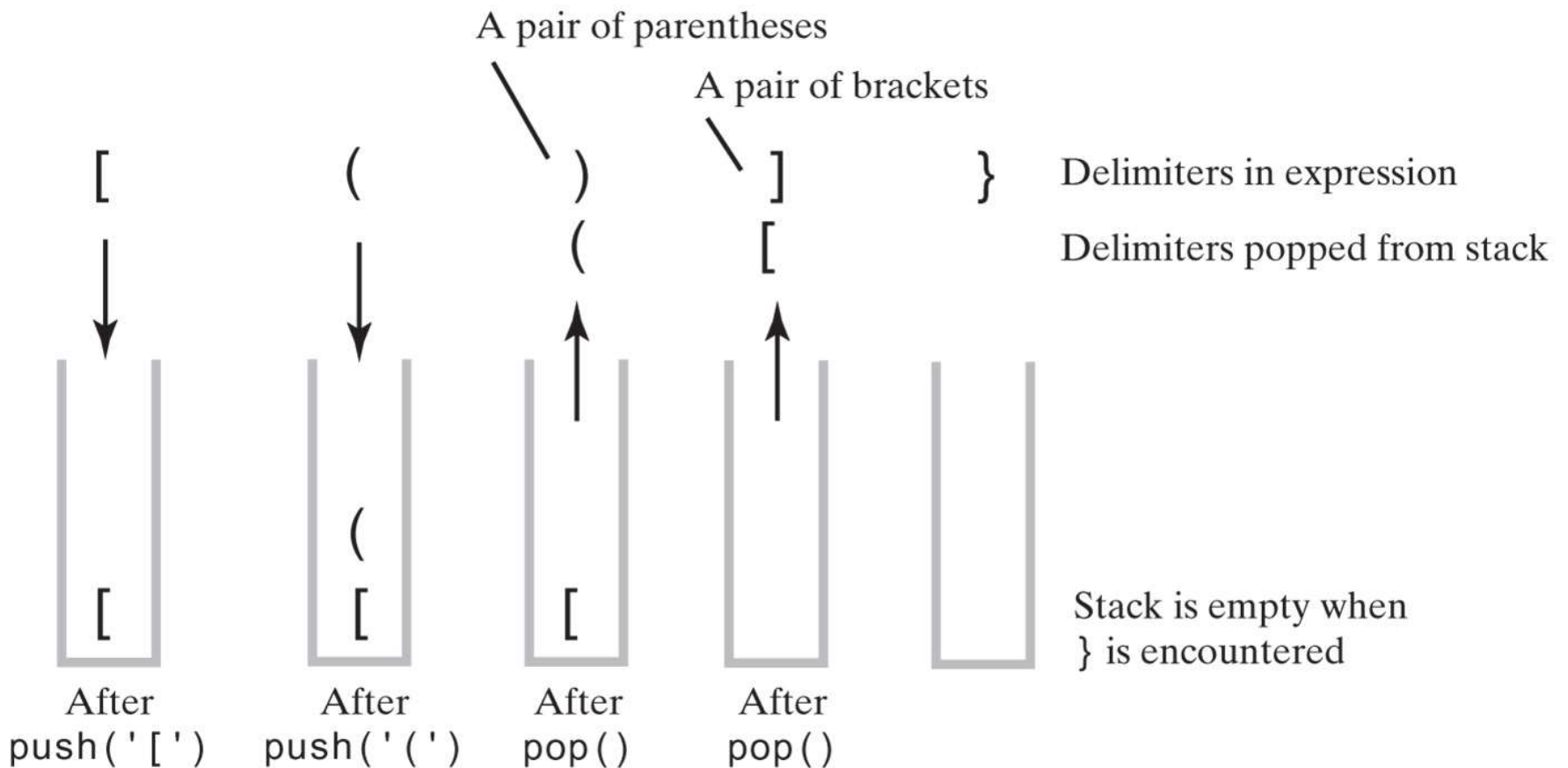


© 2019 Pearson Education, Inc.



# Processing Algebraic Expressions

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

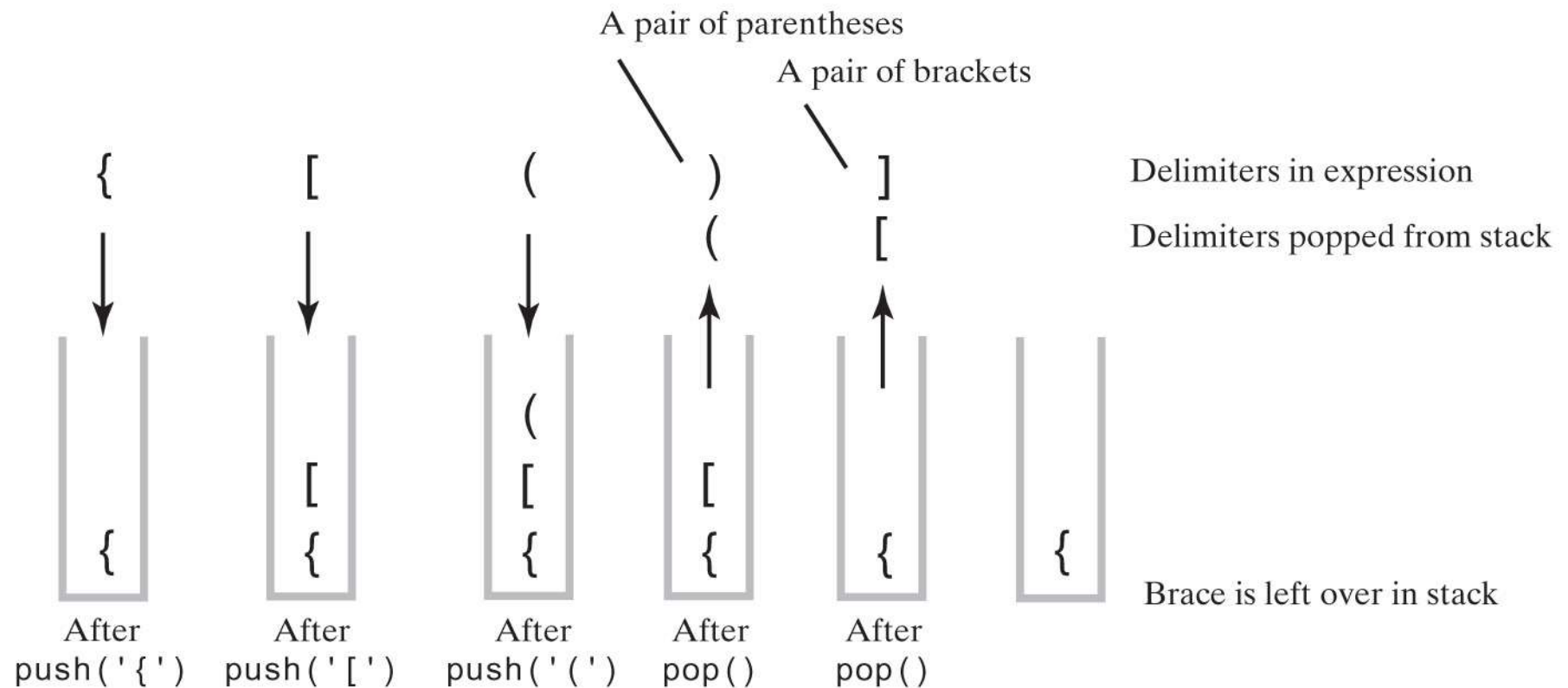


© 2019 Pearson Education, Inc.



# Processing Algebraic Expressions

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



© 2019 Pearson Education, Inc.



# 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**



# 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)  
    {
```

```
    } // end checkBalance
```

```
} // end 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)
$a$	$a$	
$-$	$a$	$-$
$b$	$a b$	$-$
$+$	$a b -$	
	$a b -$	$+$
$c$	$a b - c$	$+$
	$a b - c +$	





# Converting Infix to Postfix

Steps in converting an infix expression to postfix form

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

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



# 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
```



# 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
```



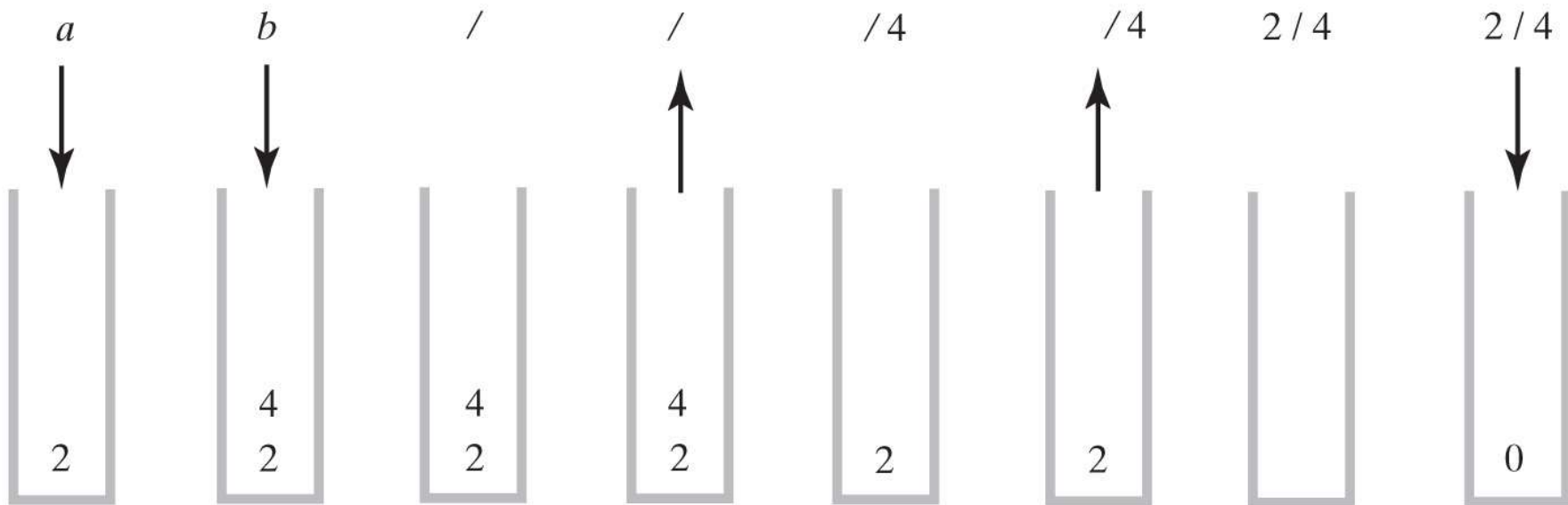
# Evaluating Postfix Expressions

```
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()
```



# Evaluating Infix Expressions

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

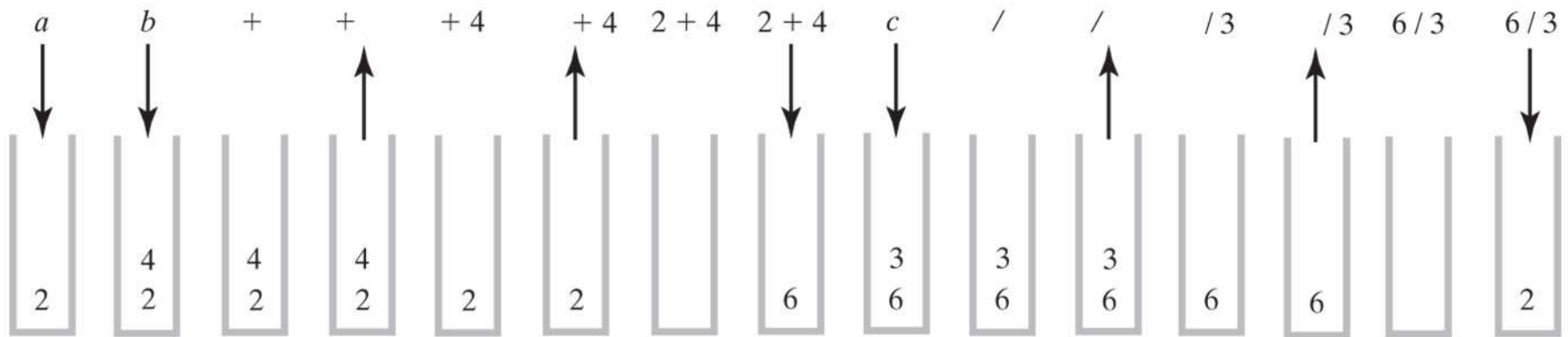


© 2019 Pearson Education, Inc.



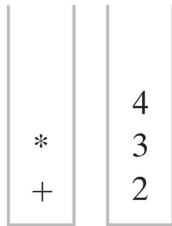
# Evaluating Infix Expressions

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



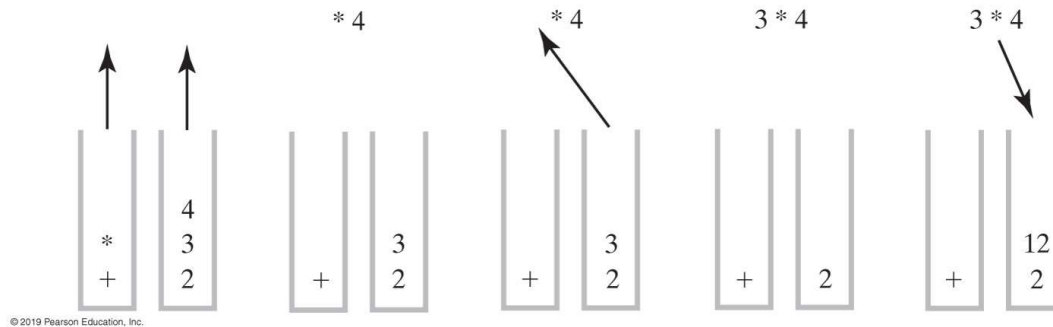
# Evaluating Infix Expressions

(a) After reaching the end of the expression

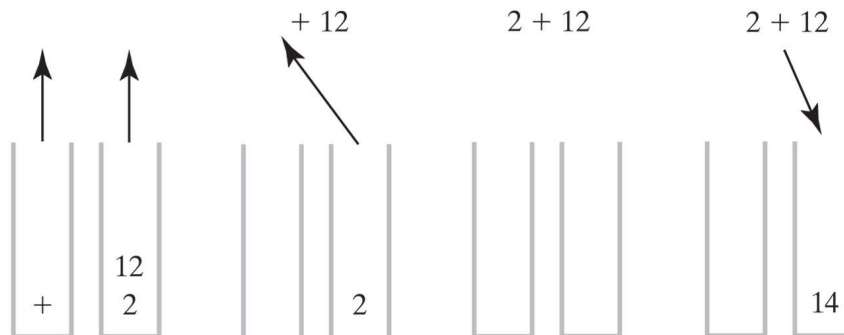


- Two stacks during the evaluation of
- $a + b * c$  when  $a$  is 2,  $b$  is 3, and  $c$  is 4

(b) While performing the multiplication



(c) While performing the addition





# 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*



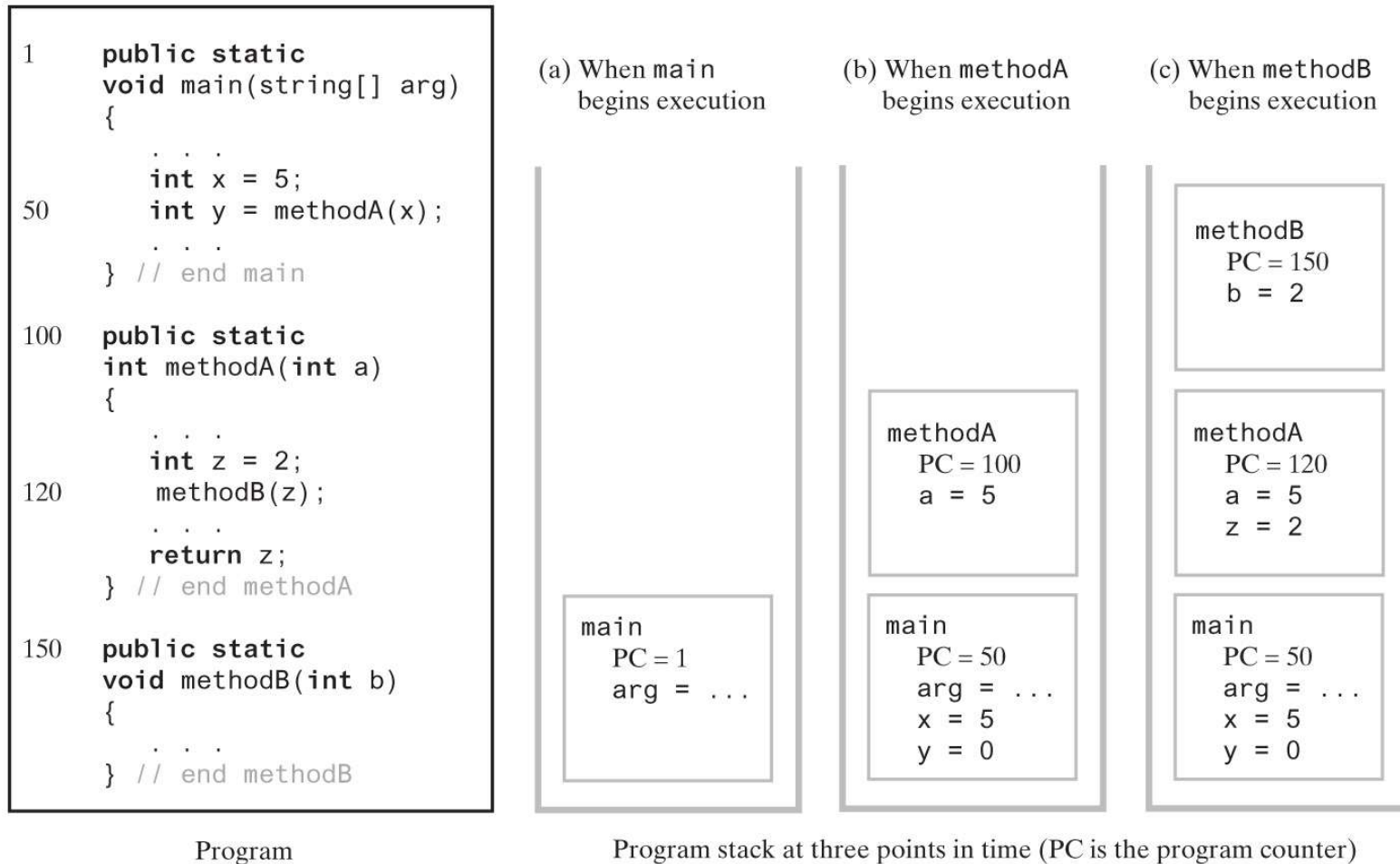
# Evaluating Infix Expressions (Part 2)

```
    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

- 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() ;`

