

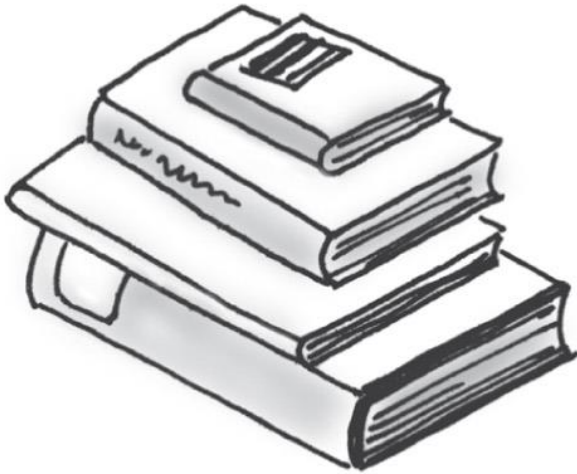
Class 03 – Stacks

CSIS 3475 Data Structures and Algorithms

©Michael Hrybyk and others
NOT TO BE REDISTRIBUTED

Stacks

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




© 2019 Pearson Education, Inc.

• FIGURE 5-1 Some familiar stacks

Specifications of the ADT Stack

- 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.
size()	+size(): int	Task: Returns the size of the stack Input: None Output: size of the stack
toArray()	+toArray(): Object[]	Task: Returns a copy of the stack as an array Input: None Output: array of objects in the stack

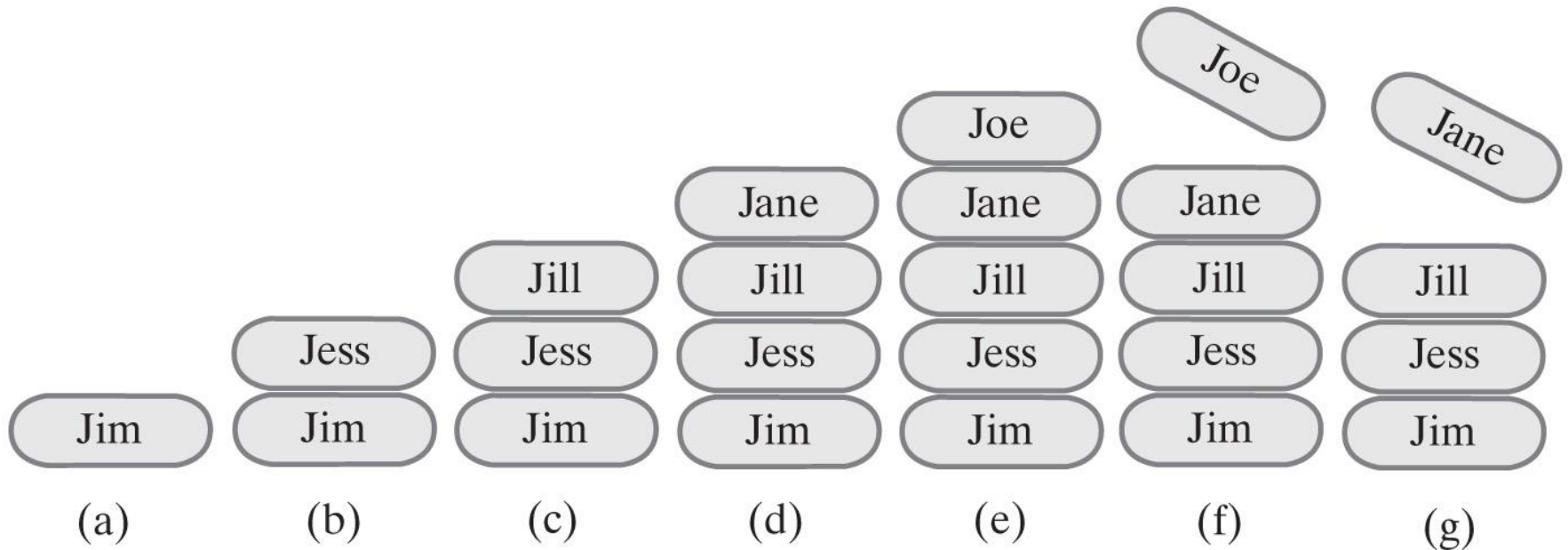
Design Decision

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

Stack Interface (modified from textbook)

```
public interface StackInterface<T> {  
    /**  
     * Adds a new entry to the top of this stack.  
     *  
     * @param newEntry An object added to the stack.  
     */  
    public void push(T newEntry);  
  
    /**  
     * Removes and returns this stack's top entry.  
     *  
     * @return the top of the stack or null if empty  
     */  
    public T pop();  
  
    /**  
     * Retrieves this stack's top entry.  
     *  
     * @return The top of the stack or null if 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();  
  
    /**  
     * Gets the number of elements in the stack  
     * @return stack size  
     */  
    public int size();  
  
    /**  
     * Gets a copy of the stack as an array.  
     *  
     * Top of stack is the last element in the array.  
     * @return copy of the stack  
     */  
    public T[] toArray();  
}
```

Example of a Stack of Strings



© 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();
```

Processing Algebraic Expressions

- **Infix:**

- each binary operator appears between its operands

a + b

- **Prefix:**

- each binary operator appears before its operands

+ a b

- **Postfix:**

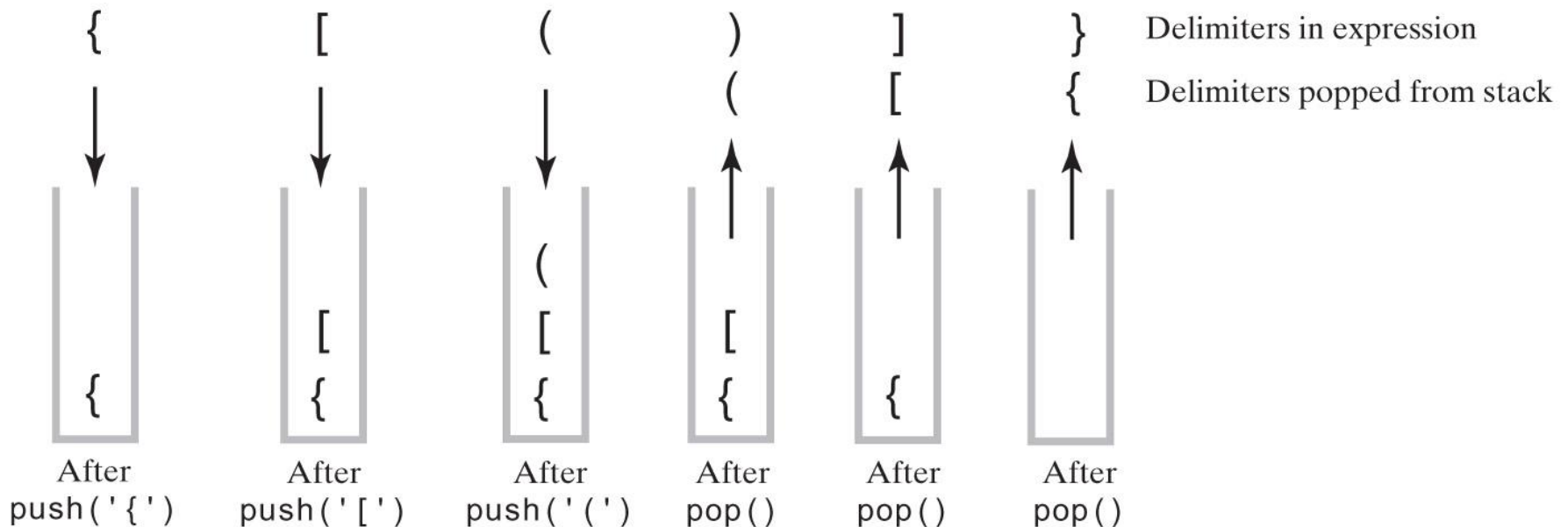
- each binary operator appears after its operands

a b +

- **Balanced expressions: delimiters paired correctly**

Processing Algebraic Expressions

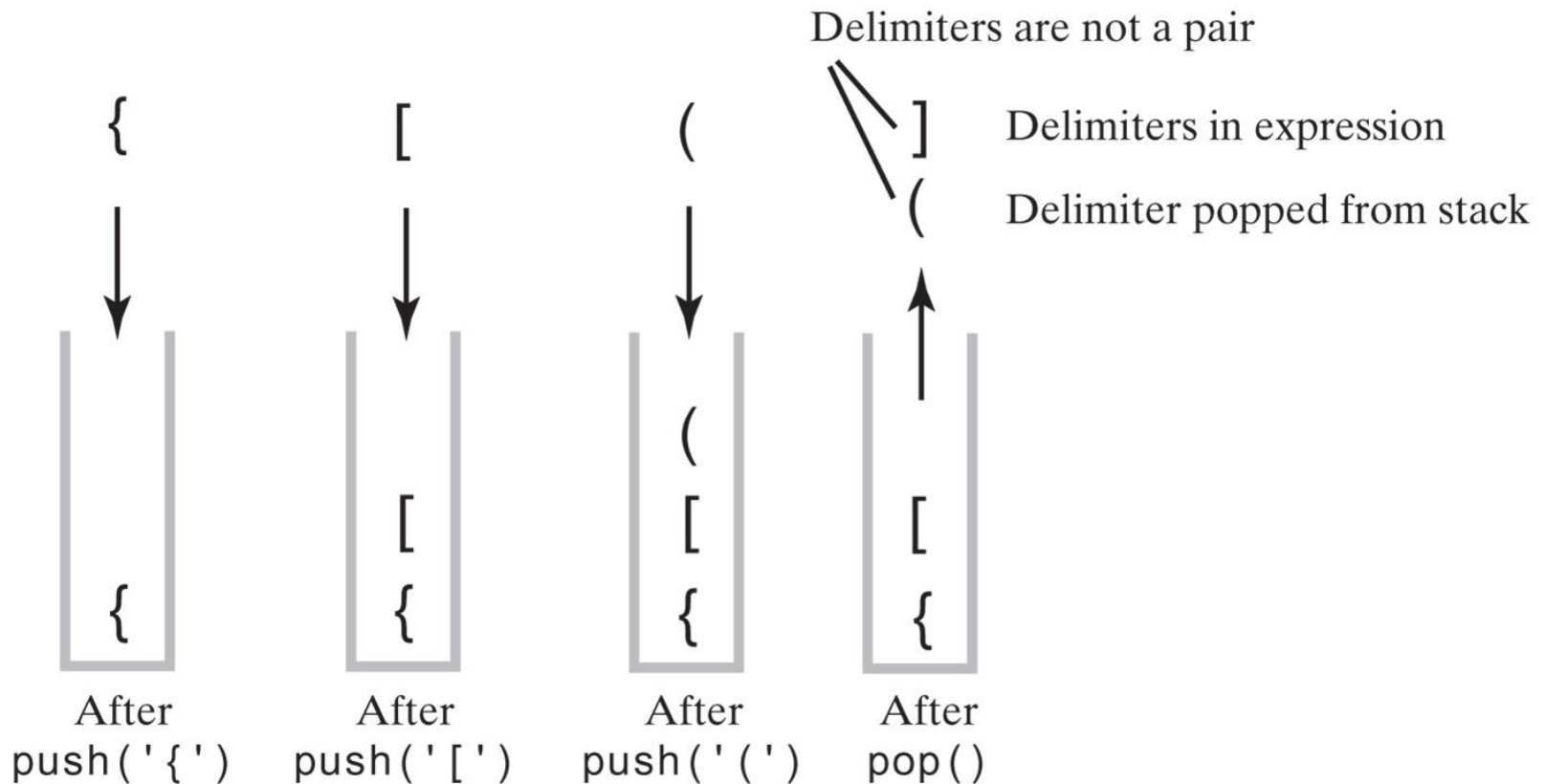
- FIGURE 5-3 The contents of a stack during the scan of an expression that contains the balanced delimiters{ [()] }



© 2019 Pearson Education, Inc.

Processing Algebraic Expressions

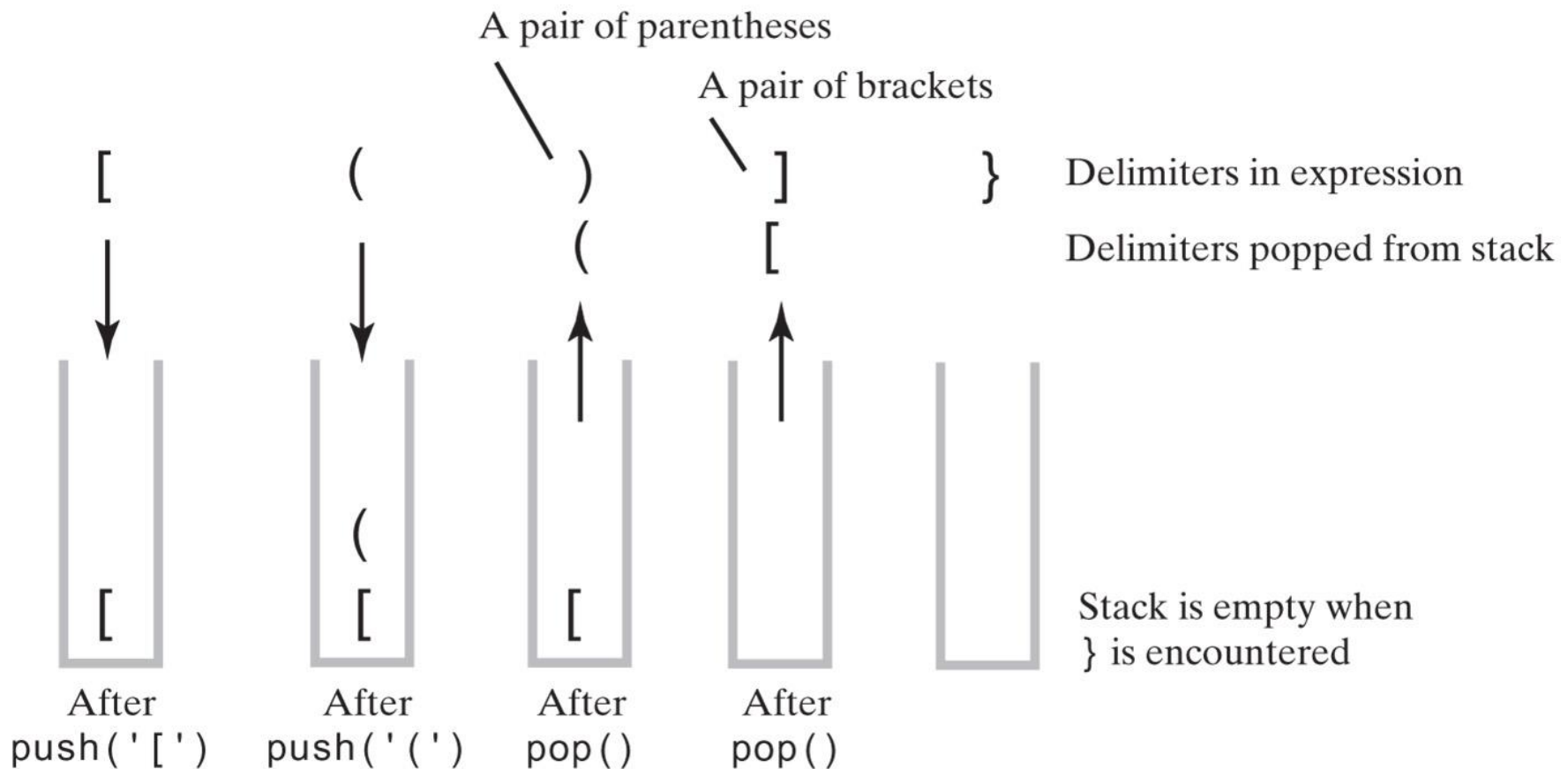
- The contents of a stack during the scan of an expression that contains the **unbalanced** delimiters { [() }



© 2019 Pearson Education, Inc.

Processing Algebraic Expressions

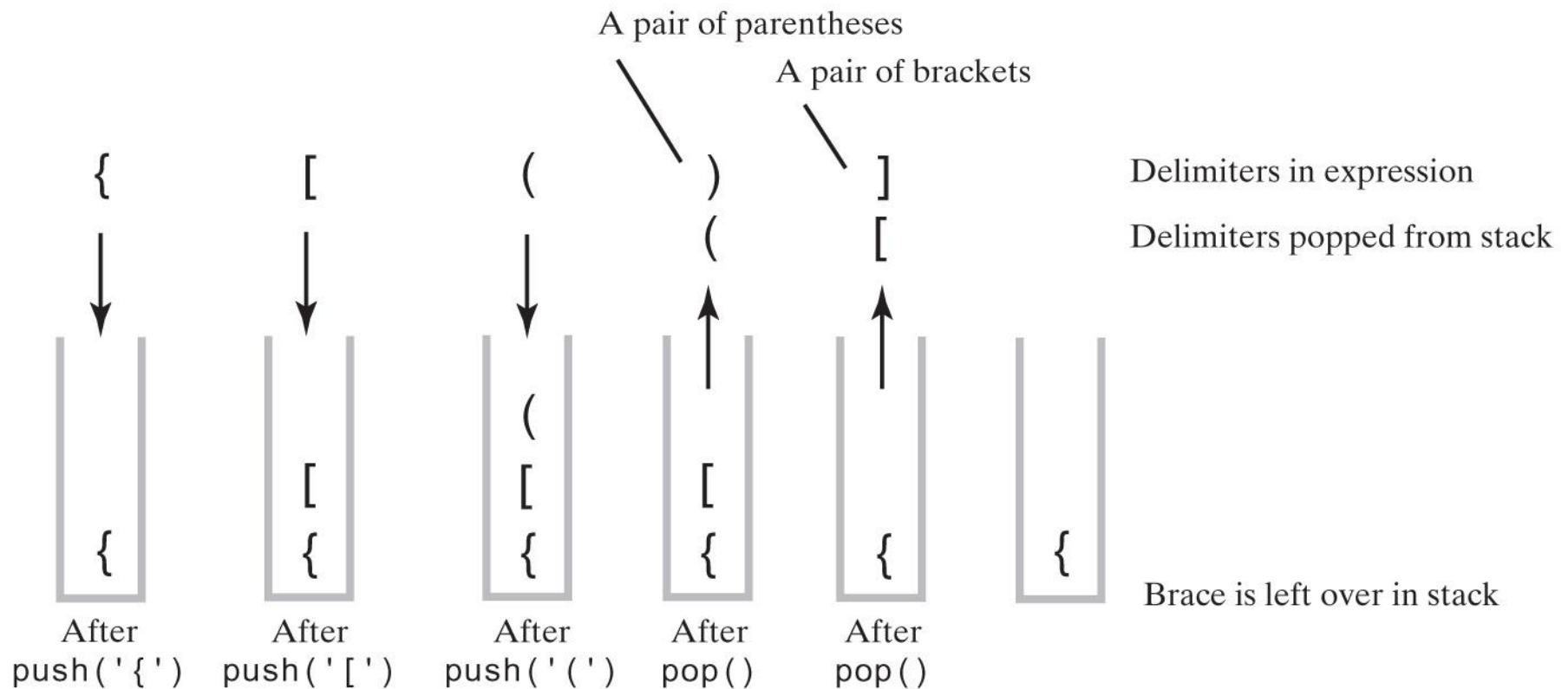
- The contents of a stack during the scan of an expression that contains the **unbalanced** delimiters [()] }



© 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

BalanceChecker Class (see Demo)

```
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) {  
        Stack<Character> openDelimiterStack = new Stack<Character>();  
  
        boolean isBalanced = true; // assume it is balanced to start  
        // continue as long as it is balanced and we have characters in the expression  
  
        for(int i = 0; isBalanced && (i < expression.length()); i++) {  
            char c = expression.charAt(i); // get the character  
            switch (c) {  
                case '(':  
                case '[':  
                case '{':  
                    // it is an opening symbol  
                    openDelimiterStack.push(c);  
                    break;  
                case ')':  
                case ']':  
                case '}':  
                    // it is a closing symbol  
                    // did we already see the opening one?  
                    if (openDelimiterStack.isEmpty())  
                        isBalanced = false;  
                    else {  
                        char openDelimiter = openDelimiterStack.pop();  
                        isBalanced = isPaired(openDelimiter, c);  
                    } // end if  
                    break;  
                default:  
                    break; // Ignore unexpected characters  
            } // end switch  
        }  
  
        // if the stack still has elements left, it is unbalanced!  
  
        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  
}
```

Evaluating algebraic expressions

- How to evaluate $a + b + c$?
- Convert to postfix
- Evaluate postfix
- Steps
 - $a + b + c$
 - $ab+c+$ 
 - Evaluate
 - abc are placed on the stack until an operator is found
 - When operator is found, evaluate, the push result back on stack
 - Repeat

Converting Infix to Postfix

a + b * c

Next Character in 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>c</i>	<i>a b c</i>	+ *
	<i>a b c *</i>	+
	<i>a b c * +</i>	

a ^ b ^ c

Next Character in 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>c</i>	<i>a b c</i>	^ ^
	<i>a b c ^</i>	^
	<i>a b c ^ ^</i>	

a - b + c

Next Character in 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>	

Converting Infix to Postfix

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

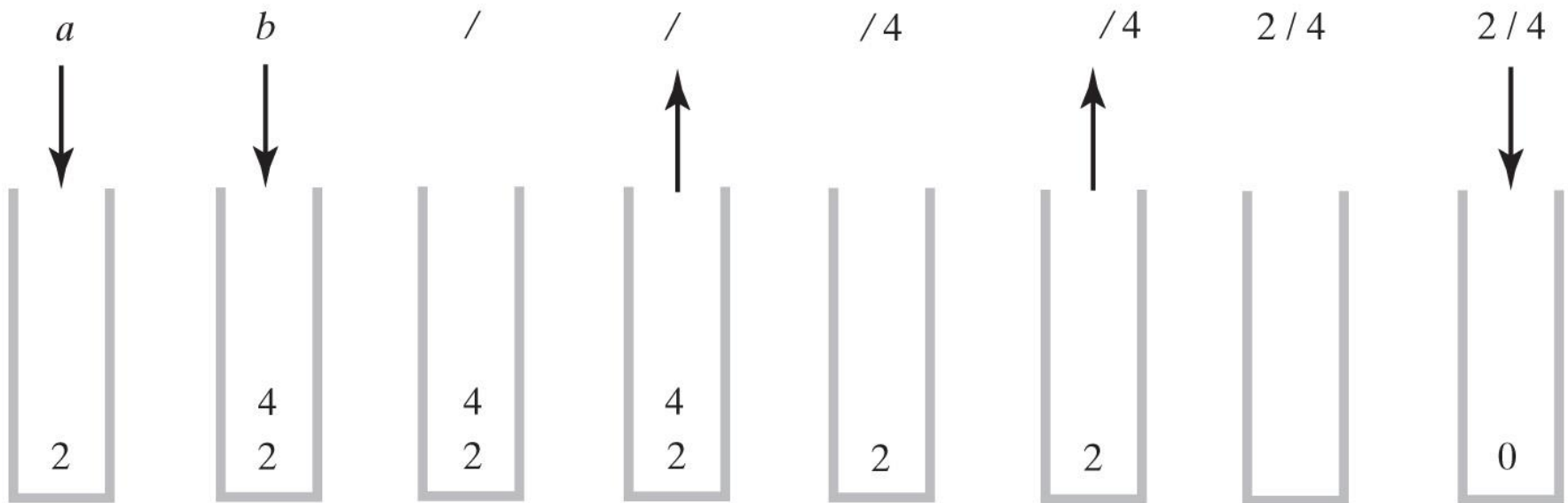
}

}

return valueStack.peek()

Evaluating Postfix Expressions

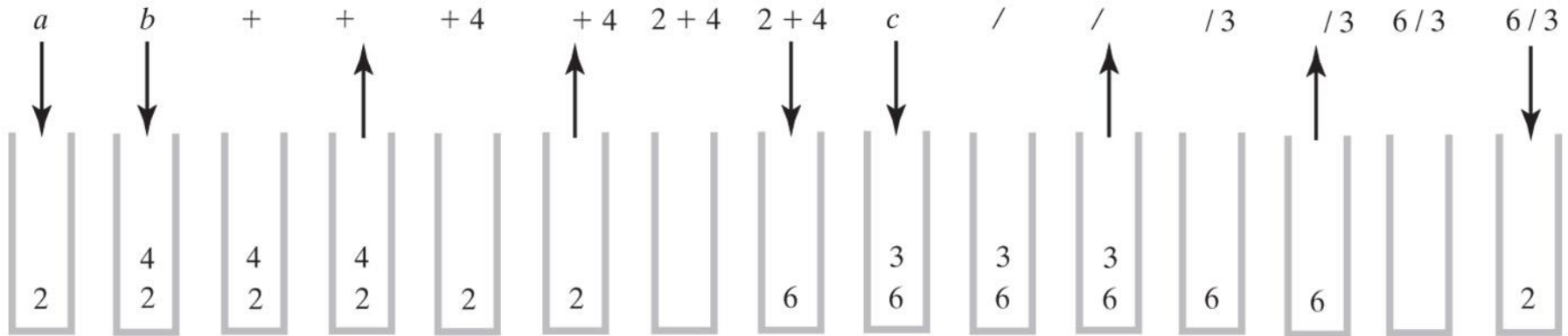
- The stack during the evaluation of the postfix expression $a \ b \ /$ when a is 2 and b is 4



© 2019 Pearson Education, Inc.

Evaluating Postfix Expressions

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

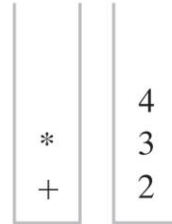


© 2019 Pearson Education, Inc.

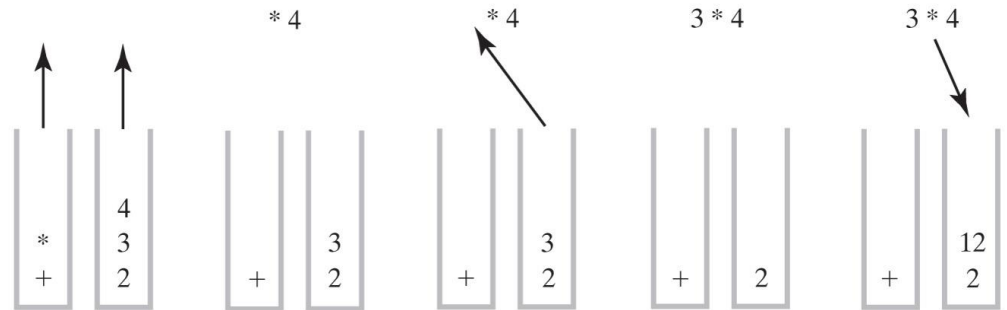
Evaluating Infix Expressions

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

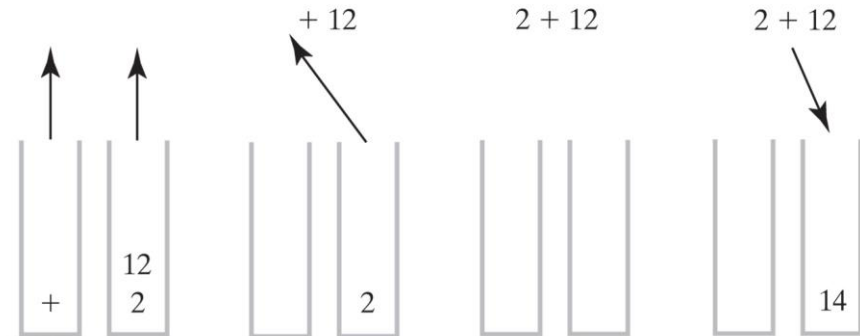
(a) After reaching the end of the expression



(b) While performing the multiplication



(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()
```

Calculator

- See Calculator.java for a complete infix to postfix example

The Application Program Stack

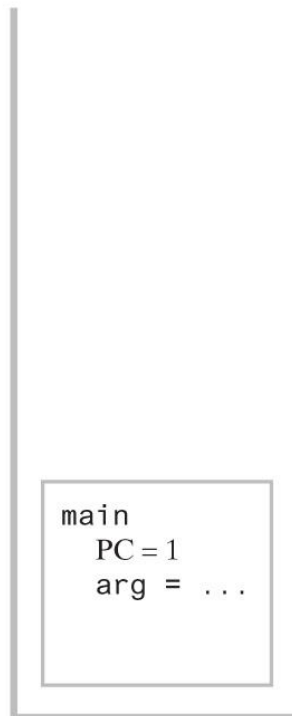
- 1 **public static**
 void main(string[] arg)
 {
 . . .
 int x = 5;
50 **int y** = **methodA**(x);
 . . .
 } // end main

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

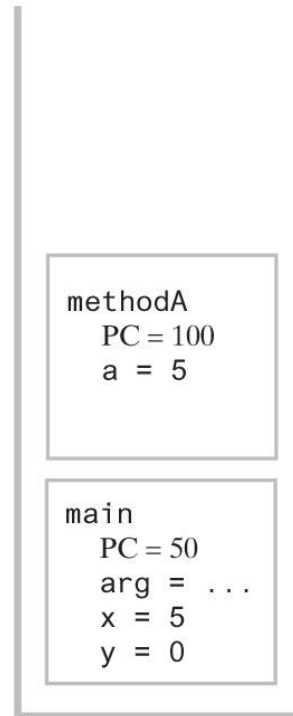
150 **public static**
 void methodB(**int b**)
 {
 . . .
 } // end methodB

Program

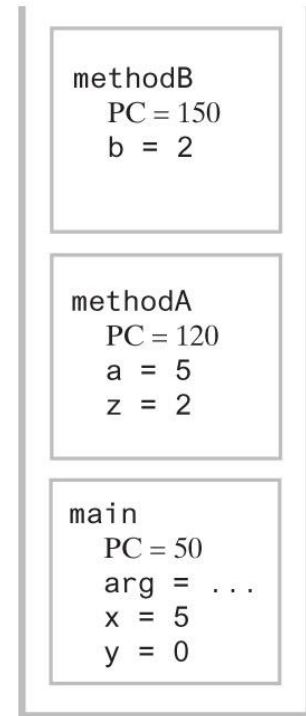
(a) When main
begins execution



(b) When methodA
begins execution



(c) When methodB
begins execution



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

© 2019 Pearson Education, Inc.

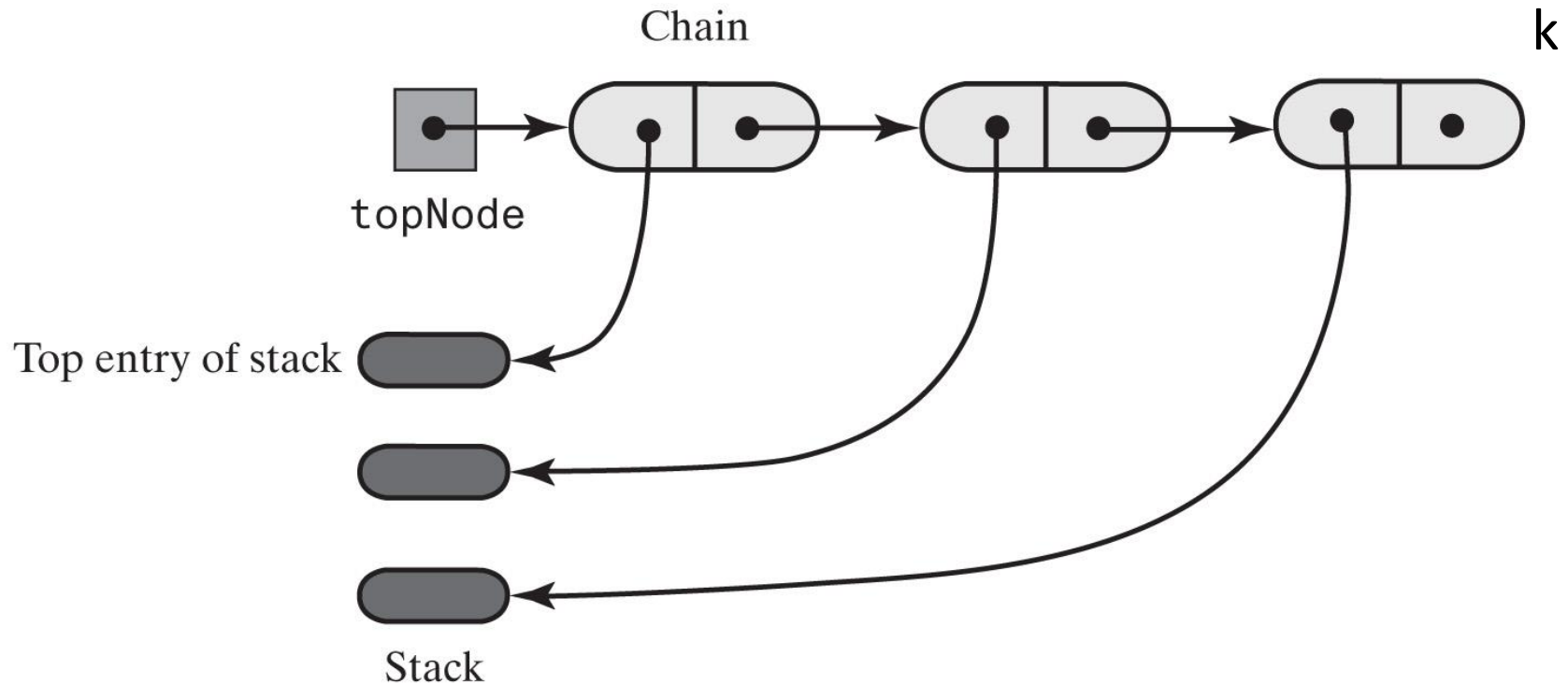
Java Class Library: The Class Stack

- 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() ;`

Linked Stack Implementation

- Each operation involves top of stack
 - **push**
 - **pop**
 - **peek**
- Head of linked list easiest, fastest to access
 - Let this be the top of the stack

Linked Implementation



© 2019 Pearson Education, Inc.

Use of the Node Class in LinkedStack

```
/**
 * Node in a linked list. Each node
 * contains data and a link to the next node in the
 * list.
 *
 * @author mhrybyk
 *
 * @param <T> data type
 */
public class Node<T> {
    private T data; // Entry in bag
    private Node<T> next; // Link to next node

    /**
     * Create a new node containing data
     * @param dataPortion
     */
    public Node(T dataPortion) {
        this(dataPortion, null);
    }

    /**
     * Create a new node containing data
     * and set the next node.
     * @param dataPortion
     * @param nextNode
     */
    public Node(T dataPortion, Node<T> nextNode) {
        data = dataPortion;
        next = nextNode;
    }

    /**
     * Get the data from the node
     * @return
     */
    public T getData() {
        return data;
    }

    /**
     * Set the data in the node
     * @param newData
     */
    public void setData(T newData) {
        data = newData;
    }

    /**
     * Get the next node
     * @return
     */
    public Node<T> getNextNode() {
        return next;
    }

    /**
     * Set the next node
     * @param nextNode
     */
    public void setNextNode(Node<T> nextNode) {
        next = nextNode;
    }
}
```

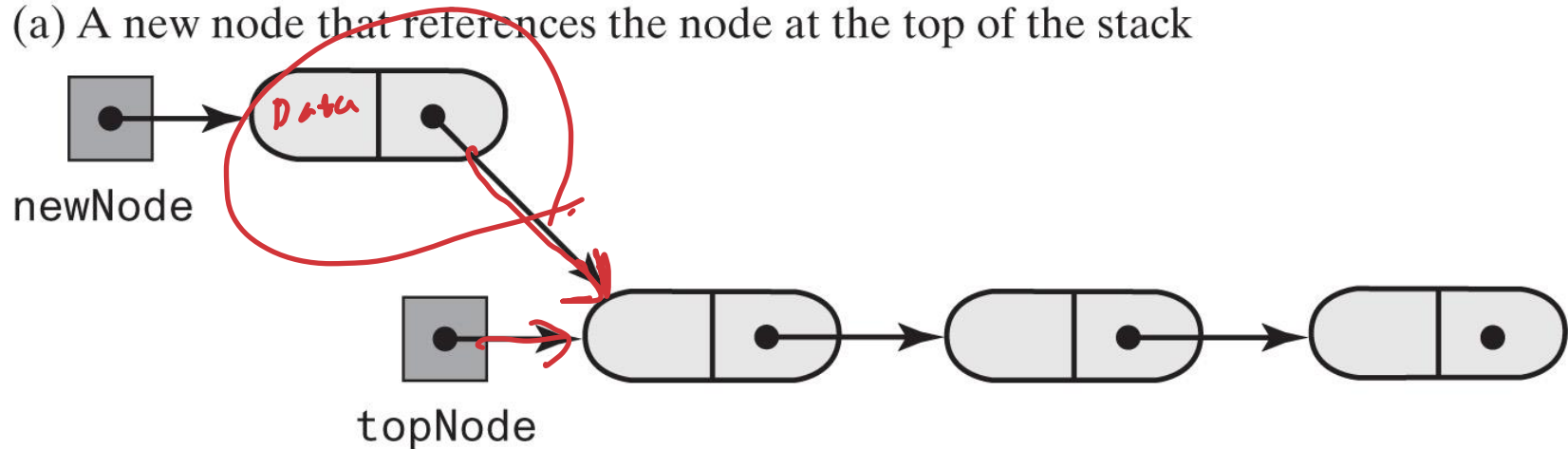
LinkedList implementation

- Keep the top node and a count of the number of entries

```
public final class LinkedList<T> implements StackInterface<T> {  
    private Node<T> topNode; // References the first node in the chain  
    private int numberOfEntries;  
  
    public LinkedList() {  
        topNode = null;  
        numberOfEntries = 0;  
    }  
}
```

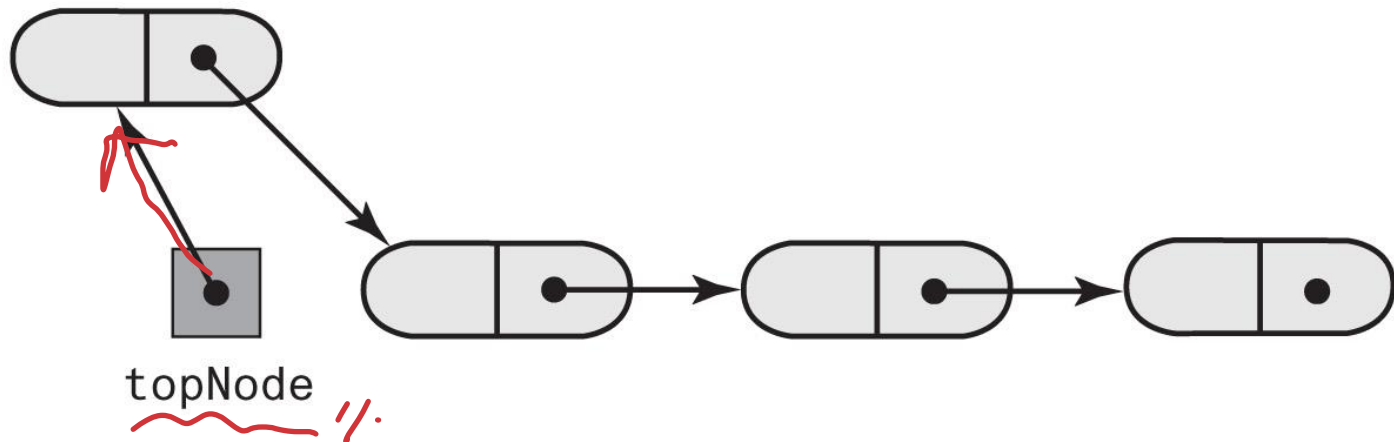

Pushing a new node onto the stack

(a) A new node that references the node at the top of the stack



© 2019 Pearson Education, Inc.

(b) The new node is now at the top of the stack



© 2019 Pearson Education, Inc.

push()

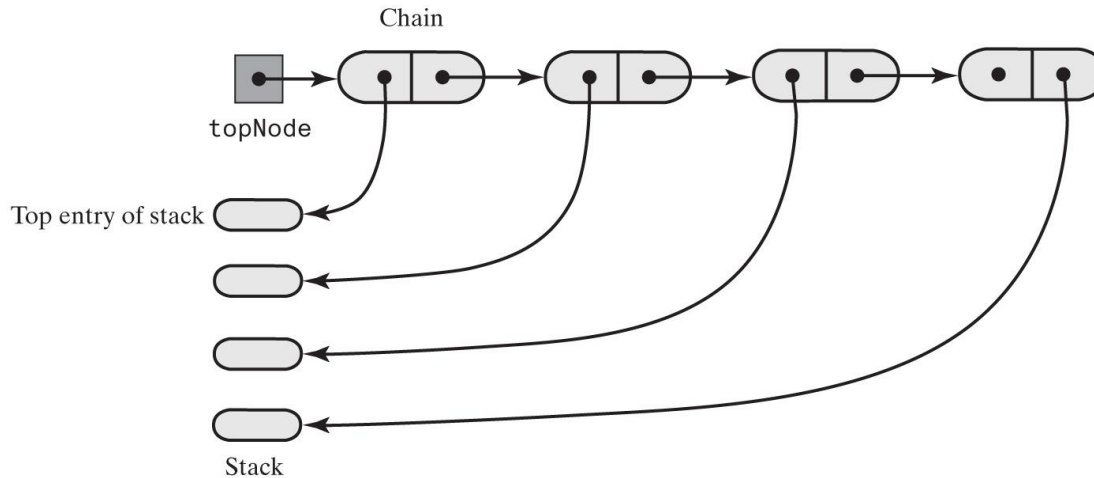
- Create a new node, and make that the top
- The new node's next link is the old top node.
- New items are therefore always at the front
- Make sure we increase the number of entries (used for size())

```
public void push(T newEntry) {  
    // create a new node, set its next node to the top  
    Node<T> newNode = new Node<T>(newEntry, topNode);  
    // set the top to the new node just created  
    topNode = newNode;  
    numberOfEntries++;  
    // topNode = new Node<T>(newEntry, topNode); // Alternate code  
}
```

Popping a node from the stack

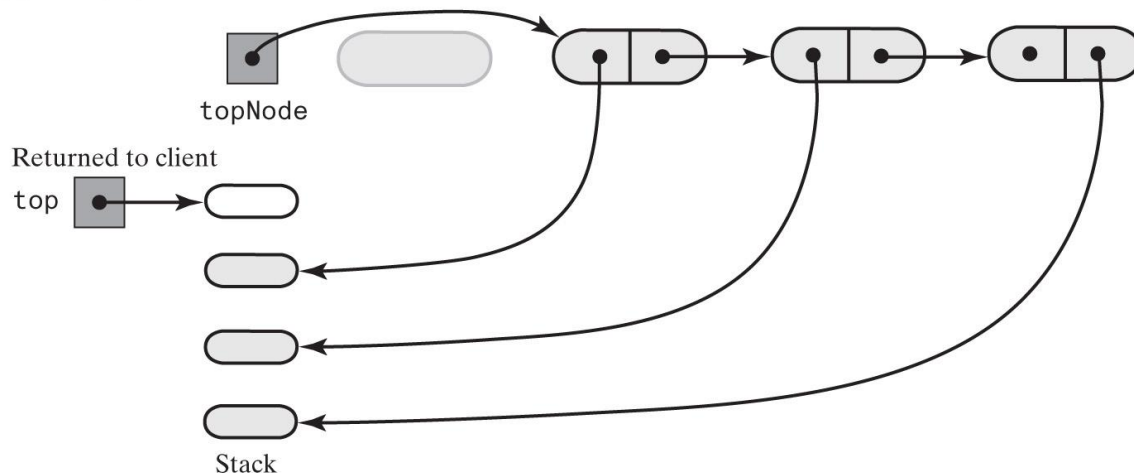
- The stack before and after pop deletes the first node in the chain

(a) Before pop



© 2019 Pearson Education, Inc.

(b) After pop



© 2019 Pearson Education, Inc.

pop()

- peek() and get the top node
- Set the top node to the next one in the chain

```
public T pop() {  
    if(isEmpty())  
        return null;  
  
    // get the top node  
  
    T top = peek();  
  
    // set the top to the next node  
  
    topNode = topNode.getNextNode();  
    numberOfEntries--;  
    return top;  
}  
  
public T peek() {  
    if (isEmpty())  
        return null;  
    else  
        return topNode.getData();  
}
```


Other LinkedStack methods

- `clear()`
 - note that we only have to set `top` to null and the number of entries to 0. Garbage collection takes care of cleaning things up.
- `toArray()`
 - to convert to array, we have to walk down the chain and copy each node's data to the array.

```
public void clear() {
    topNode = null; // Causes deallocation of nodes in the chain
    numberOfEntries = 0;
}

@Override
public int size() {
    return numberOfEntries;
}

@Override
public T[] toArray() {
    // create a new array
    @SuppressWarnings("unchecked")
    T[] tempStack = (T[]) new Object[size()];

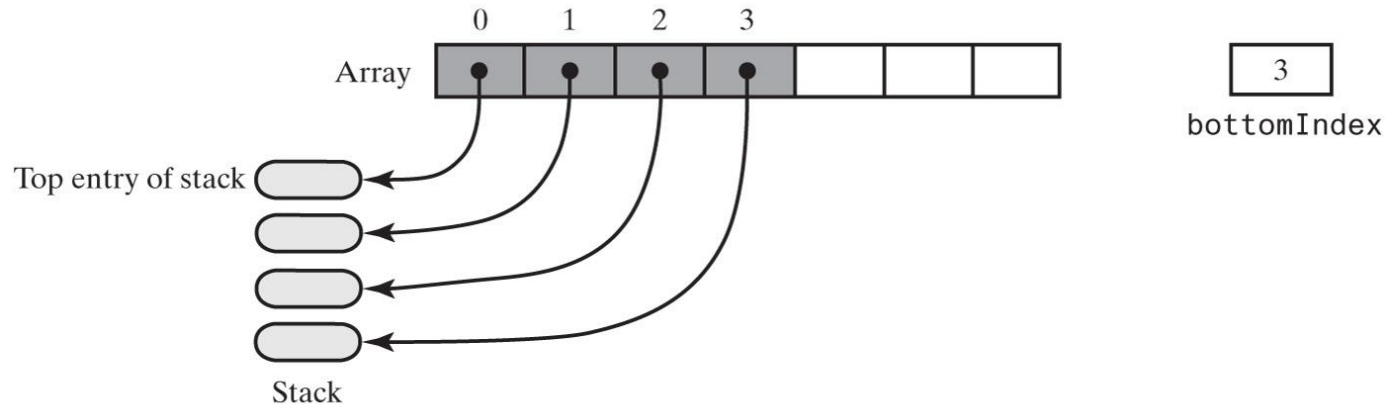
    // walk along the chain, copying the data to the array each iteration
     index = 0;
    (Node<T> currentNode = topNode; currentNode != null; currentNode = currentNode.getNextNode()) {
        tempStack[index] = currentNode.getData();
        index++;
    }
    return tempStack;
}
```

Array-Based Stack Implementation

- Each operation involves top of stack
 - **push**
 - **pop**
 - **peek**
- End of the array easiest to access
 - Let this be top of stack
 - Let first entry be bottom of stack

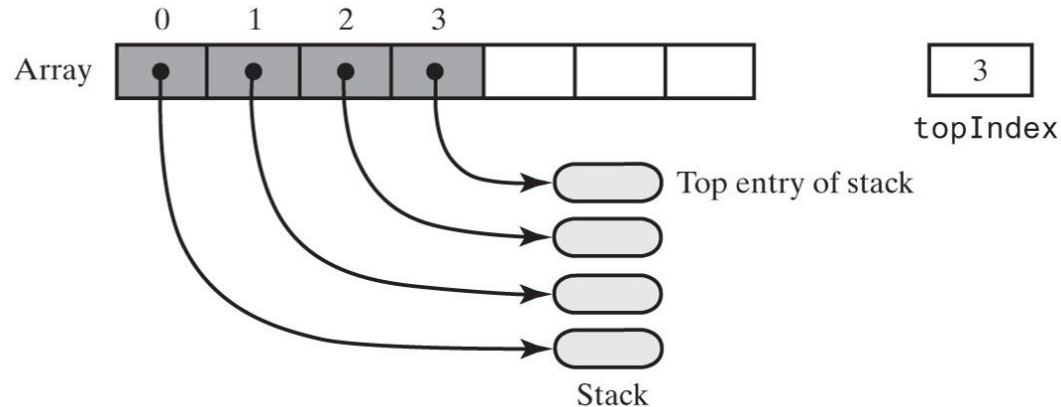
Two array representations of a stack: from the top or bottom

(a) Inefficient: The array's first element references the stack's top entry



© 2019 Pearson Education, Inc.

(b) Efficient: The array's first element references the stack's bottom entry




© 2019 Pearson Education, Inc.

ArrayStack implementation

- Uses a backing array and an index to the top of the stack.
- As we add items, the index will grow.
- Index starts at -1, so top of stack for first element added will be 0

```
public final class CompletedArrayStack<T> implements StackInterface<T> {  
    private T[] stack; // Array of stack entries  
    private int topIndex; // Index of top entry  
  
    private static final int DEFAULT_CAPACITY = 50;  
    private static final int MAX_CAPACITY = 10000;  
  
    public CompletedArrayStack() {  
        this(DEFAULT_CAPACITY);  
    }  
  
    public CompletedArrayStack(int initialCapacity) {  
        checkCapacity(initialCapacity); // do we have enough room?  
  
        // The cast is safe because the new array contains null entries  
        @SuppressWarnings("unchecked")  
        T[] tempStack = (T[]) new Object[initialCapacity];  
        stack = tempStack;  
        topIndex = -1; // note default value  
    }  
}
```



push()

- Make sure we have enough room.
 - If not, double the size of the array.
 - Copy the old elements to the new array. Expensive to do.
- Increment the top index, and store the data there

```
public void push(T newEntry) {
    ensureCapacity(); // make sure we have enough room

    // place the entry at the end, and increment the index
    stack[topIndex + 1] = newEntry;
    topIndex++;
}

/**
 * Throws an exception if the client requests a capacity that is too large.
 * @param capacity requested capacity
 */
private void checkCapacity(int capacity) {
    if (capacity > MAX_CAPACITY)
        throw new IllegalStateException(
            "Attempt to create a stack " + "whose capacity exceeds " + "allowed maximum.");
}

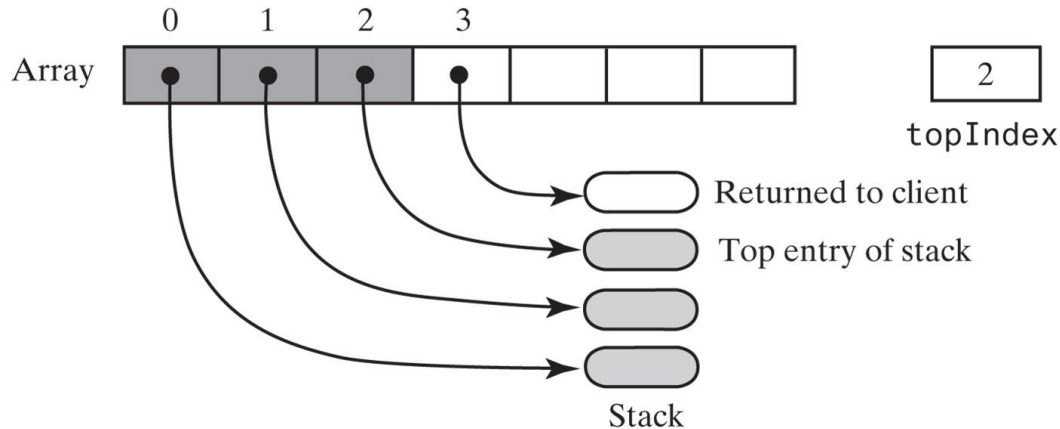
/**
 * Doubles the size of the array stack if it is full
 */
private void ensureCapacity() {
    if (topIndex >= stack.length - 1)
    {
        // the array is full, so double the size

        int newLength = 2 * stack.length;
        checkCapacity(newLength);

        // copyOf will create a new array with
        // the larger size and copy all elements
        stack = Arrays.copyOf(stack, newLength);
    }
}
```

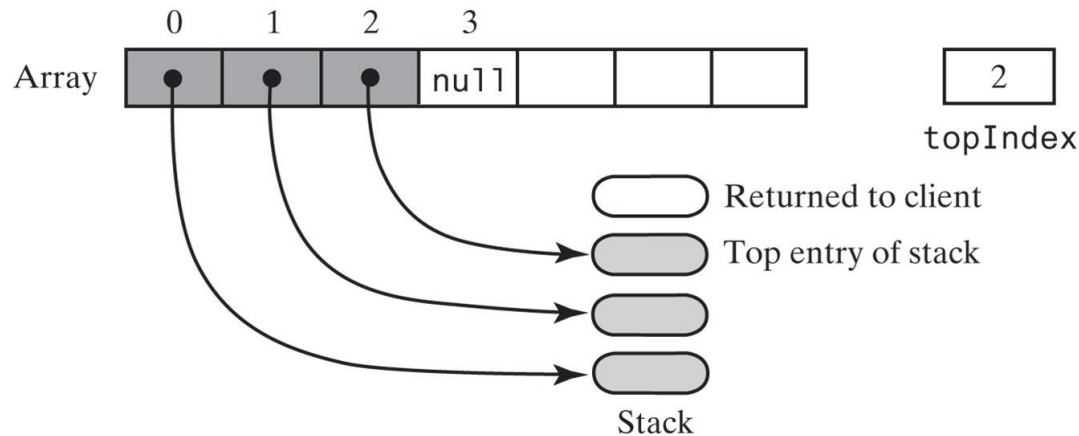
pop(): An array-based stack after its top entry is removed in two different ways

(a) By decrementing topIndex



© 2019 Pearson Education, Inc.

(b) By setting stack[topIndex] to null and then decrementing topIndex



© 2019 Pearson Education, Inc.

pop() and peek()

- Get the element at the top. Operation is $O(1)$
- For pop(), then decrement the index

```
public T pop() {
    if (isEmpty())
        return null;
    else {
        // get the top index
        T top = stack[topIndex];

        // now set that location to null
        // and decrement the index
        stack[topIndex] = null;
        topIndex--;

        return top;
    } // end if
}

public T peek() {
    if (isEmpty())
        return null;
    else
        return stack[topIndex];
}
```

Other methods

- `clear()` could just set `topIndex` to `-1` directly, but setting everything to `null` is safer
- Notice simple array copy for `toArray()`

```
public boolean isEmpty() {
    return topIndex < 0;
}

public void clear() {
    // Remove references to the objects in the stack,
    // but do not deallocate the array

    // note that topIndex reverts to -1

    while (topIndex > -1) {
        stack[topIndex] = null;
        topIndex--;
    }
}

public T[] toArray() {
    return Arrays.copyOf(stack, stack.size());
}

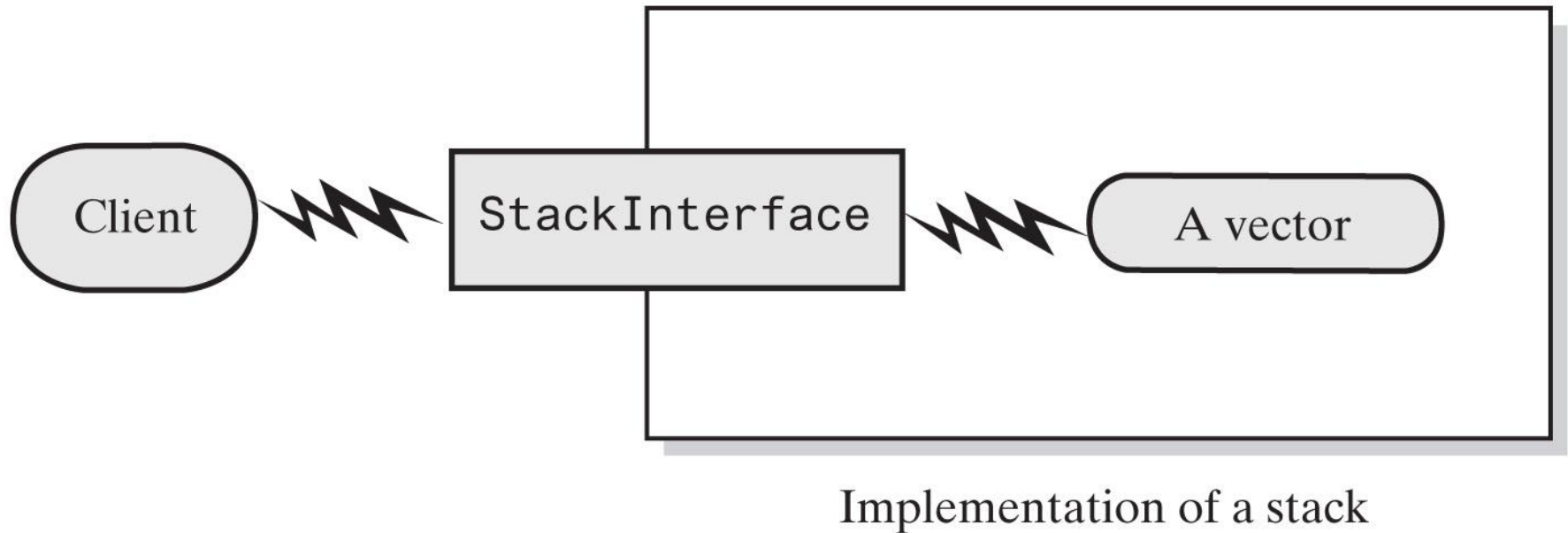
public int size() {
    return topIndex + 1;
}
```

Vector-Based Stack Implementation

- The class **Vector**
 - An object that behaves like a high-level array
 - Index begins with 0
 - Methods to access or set entries
 - Size will grow as needed
 - Has methods to add, remove, clear
 - Also methods to determine
 - Last element
 - Is the vector empty
 - Number of entries
- Use vector's methods to manipulate stack

Vector-Based Stack Implementation

- FIGURE 6-6 A client using the methods given in StackInterface; these methods interact with a vector's methods to perform stack operations



© 2019 Pearson Education, Inc.

VectorStack implementation

- Use Vector as backing object for the stack
- Use Vector methods to implement stack methods
- Vector will grow dynamically, no need to increase size

```
public final class CompletedVectorStack<T> implements StackInterface<T> {  
    private Vector<T> stack; // Last element is the top entry in stack  
  
    // Vector class can grow dynamically, so no need to increase  
  
    private static final int DEFAULT_CAPACITY = 50;  
  
    public CompletedVectorStack() {  
        this(DEFAULT_CAPACITY);  
    }  
  
    public CompletedVectorStack(int initialCapacity) {  
        stack = new Vector<>(initialCapacity); // Size doubles as needed  
    }  
}
```

push(), peek(), pop()

- Simply use Vector methods
- remove() needs an index, so we calculate the last one as size - 1

```
public void push(T newEntry) {
    stack.add(newEntry);
}

public T peek() {
    if (isEmpty())
        return null;
    else
        return stack.lastElement();
}

public T pop() {
    if (isEmpty())
        return null;
    else
        return stack.remove(stack.size() - 1);
}
```


Other methods

- Notice toArray() needs an array of the appropriate size to copy into.

```
public boolean isEmpty() {  
    return stack.isEmpty();  
}  
  
public void clear() {  
    stack.clear();  
}  
  
@Override  
public int size() {  
    return stack.size();  
}  
  
public T[] toArray() {  
    @SuppressWarnings("unchecked")  
    T[] tempStack = (T[]) new Object[size()];  
  
    // Vector toArray() requires an array as an arg to fill in  
    return stack.toArray(tempStack);  
}
```

Java library Stack class

- Has all of the methods of StackInterface more or less.
- See StackDemo for examples.

In class exercises

- Complete
 - `ArrayStack.java`
 - `VectorStack.java`
 - `LinkedStack.java`
- Test using `StackTestDriver.java` and `BalanceCheckerDemo.java`
- These classes will be used again in the future, so save your work.