University of Pittsburgh

Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines:
  - Homework 4: this Friday @ 11:59 pm
  - Lab 3: next Monday @ 11:59 pm
  - Programming Assignment 1: Friday Oct. 7th
- **Live Remote Support Session** for Assignment 1
  - This Friday @ 2:00 pm
  - Session is recorded
- **Student Support Hours** of the teaching team are posted on the Syllabus page

Previous Lecture …

- ADT List
  - resizable array implementation: ArrayList
    - Rest of the methods
  - Linked implementation: LinkedList

Muddiest Points

- **Q: Assignment 1. It's due in a week, worth 10% of our final grade, and there has been literally no guidance on how to start or successfully complete it. I have no experience with two-dimensional arrays, or arrays of objects, or implementing a new interface (especially one that isn't included in the textbook or hasn't been taught in lecture.) This seems like we're being asked to run while still learning to walk.**
- I will host a live remote support session this Friday @ 2:00 pm

Muddiest Points

- **Q: When should you create your own exception instead of using one of the exception types that already exist in Java Libraries?**

Exception Basics

- Method creates and throws an exception object
  - We say “throws an exception”
- Signal to program
  - *Unexpected* situation has happened
  - e.g., client didn’t adhere to method’s preconditions
- Handle the exception
  - Detect and react

The Basics

Some **Checked** exceptions in the Java Class Library

- `ClassNotFoundException`
- `FileNotFoundException`
- `IOException`
- `NoSuchMethodException`
- `WriteAbortedException`

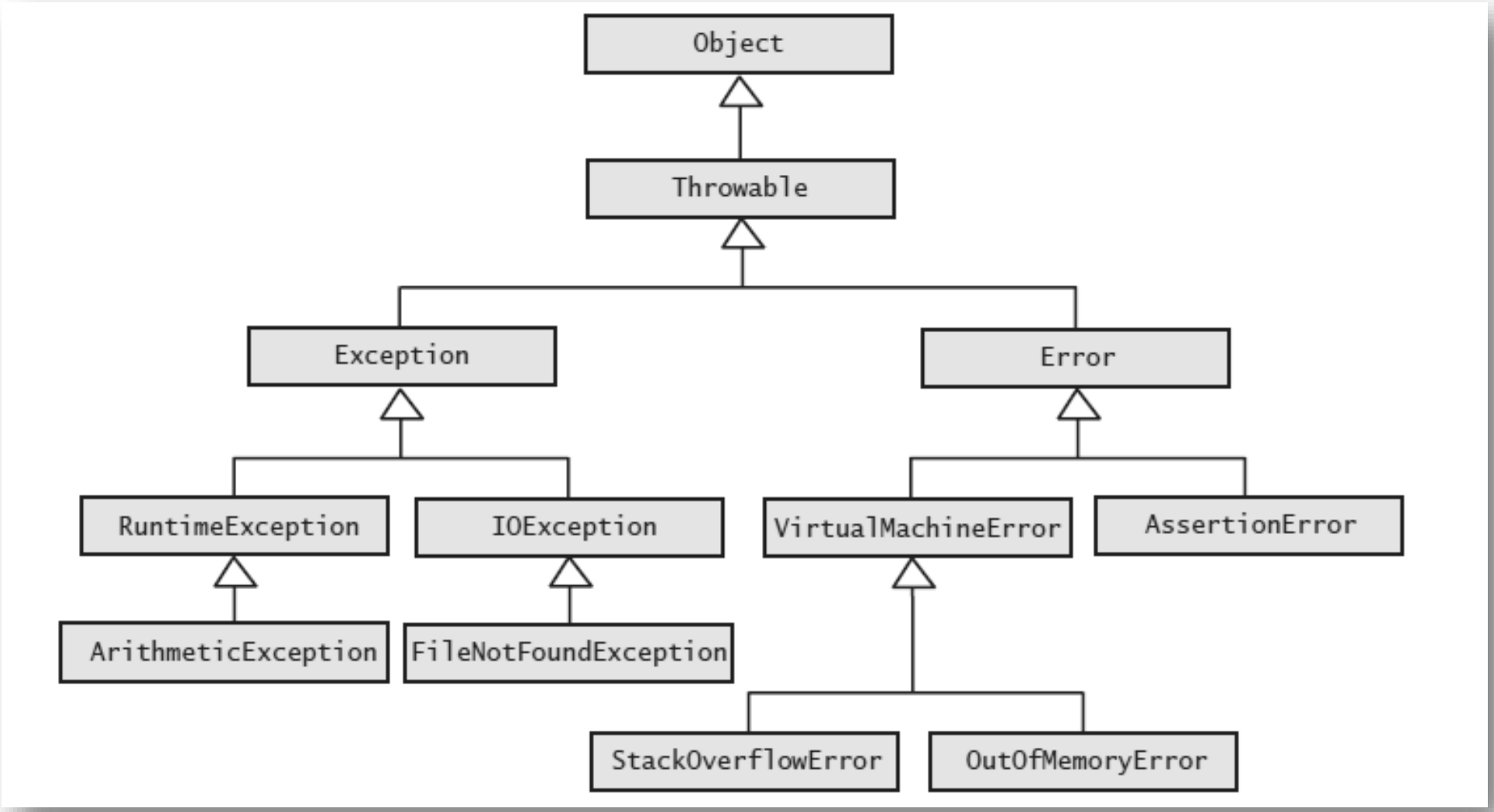
The Basics

Some **Runtime** exceptions in the Java Class Library

- `ArithmeticException`
- `ArrayIndexOutOfBoundsException`
- `ClassCastException`
- `IllegalArgumentException`
- `IllegalStateException`
- `IndexOutOfBoundsException`
- `NoSuchElementException`
- `NullPointerException`
- `StringIndexOutOfBoundsException`
- `UnsupportedOperationException`

The Basics

The hierarchy of some standard exception and error classes



Throwing an Exception

- A method intentionally throws an exception by executing a **`throw`** statement.
- Programmers usually create the object within the **`throw`** statement

```
throw new IOException();
```

Throwing an Exception

- If you can resolve unusual situation in a reasonable manner
  - likely can use a decision statement instead of throwing an exception
- If several resolutions to abnormal occurrence possible, and you want client to choose
  - Throw a checked exception
- If a programmer makes a coding mistake by using your method incorrectly
  - Throw a runtime exception

Handling an exception: The `try-catch` Blocks

- Code to handle an `IOException` as a result of invoking the method `readString`

```
try
{
    < Possibly some code >
    anObject.readString(. . .); //  Might throw an IOException
    < Possibly some more code >
}
catch (IOException e)
{
    < Code to react to the exception, probably including the following statement: >
    System.out.println(e.getMessage());
}
```

Multiple `catch` Blocks

- Order for `catch` blocks matters!
- Start with most specific exceptions

```
catch (FileNotFoundException e)
{
   . . .
}
catch (IOException e) // Handle all other IOExceptions
{
   . . .
}
```

Handling an Exception

- If programmer not sure what action is best for a client when an exception occurs
  - Leave the handling of the exception to the method's client
- A method that can cause but does not handle a checked exception must declare it in its header
- The **`throws`** clause must declare all checked exceptions thrown by a method

```
public String readString(. . .) throws IOException
```

Muddiest Points

- **Q: When should you create your own exception instead of using one of the exception types that already exist in Java Libraries?**
- When you want to report a situation that none of the exception classes already defined in the Java Class Library accurately describes
- To minimize dependencies and import statements in client code
  - one import: import edu.cs0445.*
  - vs. multiple imports:
    - import java.util.FileNotFoundException
    - import java.security.AccessControlException
    - …

Muddiest Points

- **Q: Are the only nodes identified with names first and last node, whereas the rest are identified by index?**
- From the client perspective, all nodes are identified by position only: First node is at position 1, last node at position getLength()
- From the **linked** implementation perspective, we have a variable for the first node only; the rest are reachable by traversing the chain

Muddiest Points

- **Q: I know you didn't use any in the code you wrote during lecture, but what exactly does the assert keyword in java? Because it's used in a lot of the methods throughout the slides.**
- The assert statement is a way to catch inconsistencies early. We use assert to explicitly define what we expect to be true at various points in the code
- If code is run with assert statements enabled (java **-ea** Main), when the assert condition is false, the program will stop and tell us which assertion was violated
- For example, a LinkedList object is empty if numberOfEntries == 0, in which case firstNode is expected to be null. We can use assert to make sure that this expectation is true.
  - if(numberOfEntries == 0){
    - assert firstNode == null;
  - …..
  - }

Muddiest Points

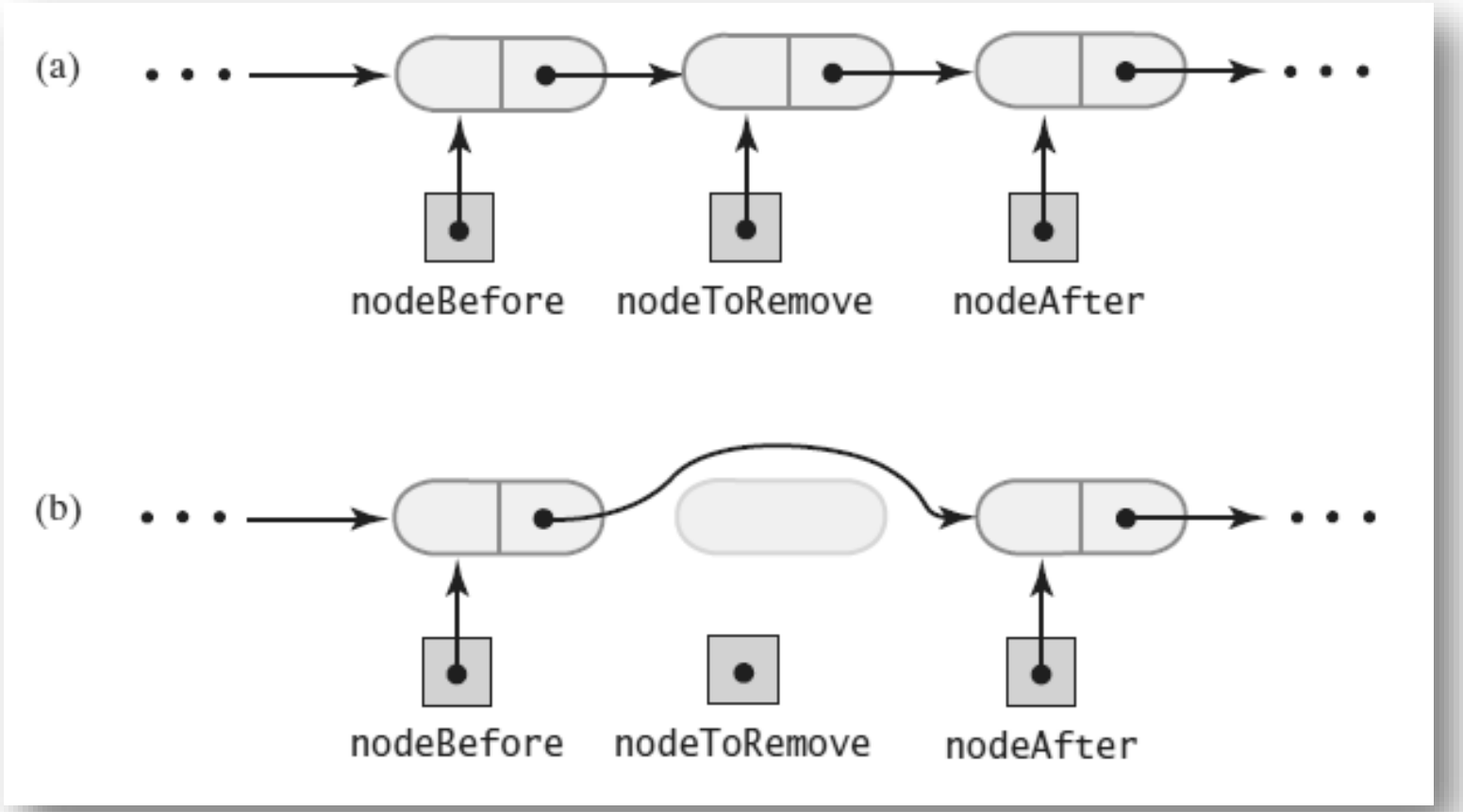
- **Q: Go over more runtime questions with code examples**
- Let's analyze the running time of the LinkedList methods

Today's Agenda

- ADT List
  - Refined Linked implementation with head and tail references
- ADT Stack
  - Implementation using ADT List
  - Array-based implementation
  - Linked implementation

Removing a Node other than first node

A chain of nodes (a) just prior to removing an interior node; (b) just after removing an interior node

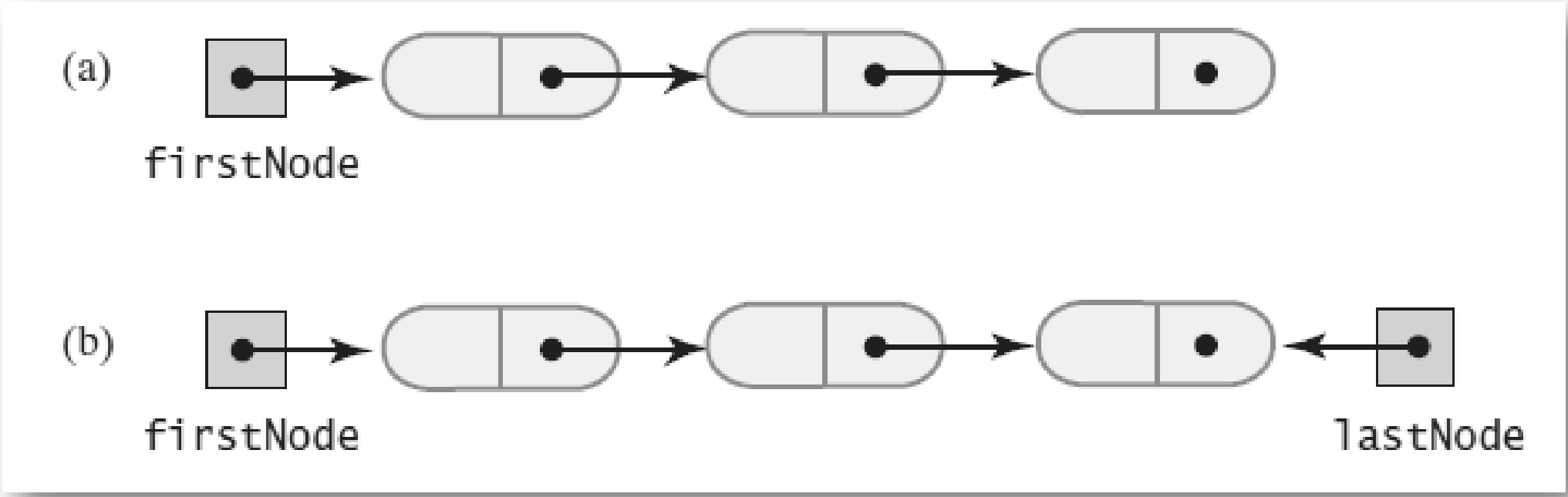


The `remove` method returns the entry that it deletes from the list

```
public T remove(int givenPosition)
{
   T result = null;                                  // Return value
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {
      assert !isEmpty();
      if (givenPosition == 1)                        // Case 1: Remove first entry
      {
         result = firstNode.getData();               // Save entry to be removed
         firstNode = firstNode.getNextNode();        // Remove entry
      }
      else                                           // Case 2: Not first entry
      {
         Node nodeBefore = getNodeAt(givenPosition - 1);
         Node nodeToRemove = nodeBefore.getNextNode();
         result = nodeToRemove.getData();            // Save entry to be removed
         Node nodeAfter = nodeToRemove.getNextNode();
         nodeBefore.setNextNode(nodeAfter);          // Remove entry
      } // end if
      numberOfEntries--;                             // Update count
      return result;                                 // Return removed entry
   }
   else
   throw new IndexOutOfBoundsException(
             "Illegal position given to remove operation.");
} // end remove
```

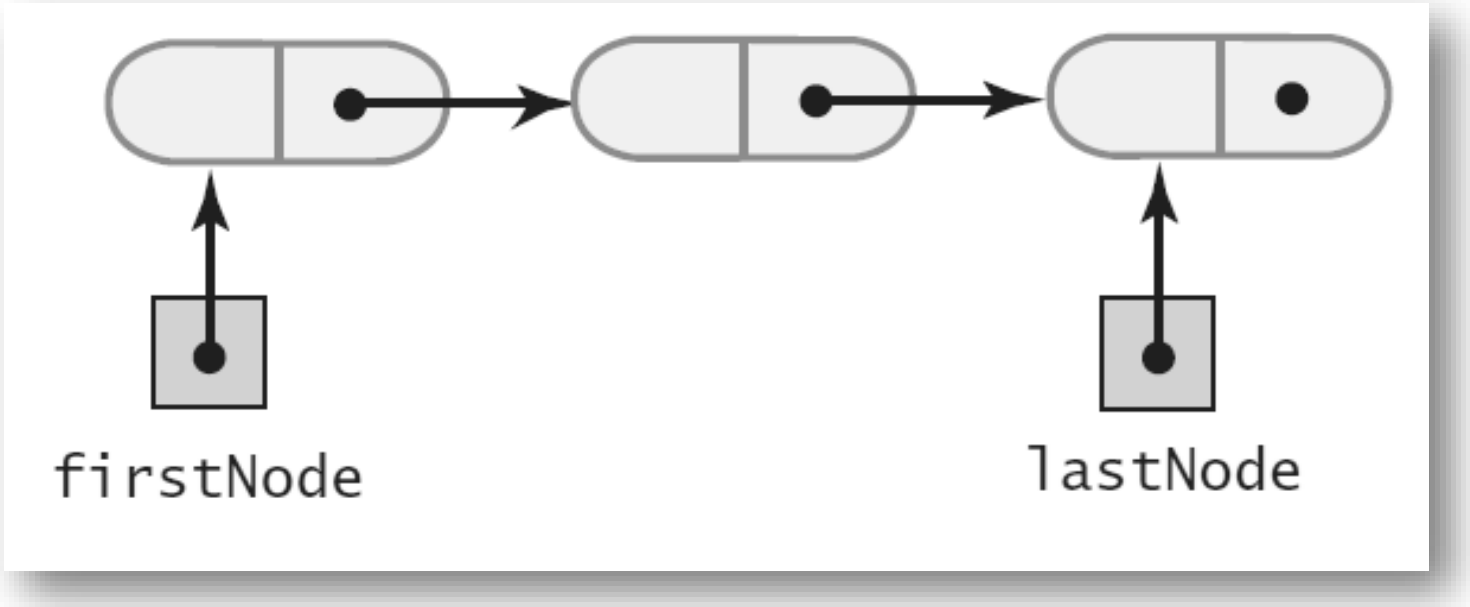
Design Decision: A Link to Last Node

A linked chain with (a) a head reference;
(b) both a head reference and a tail reference



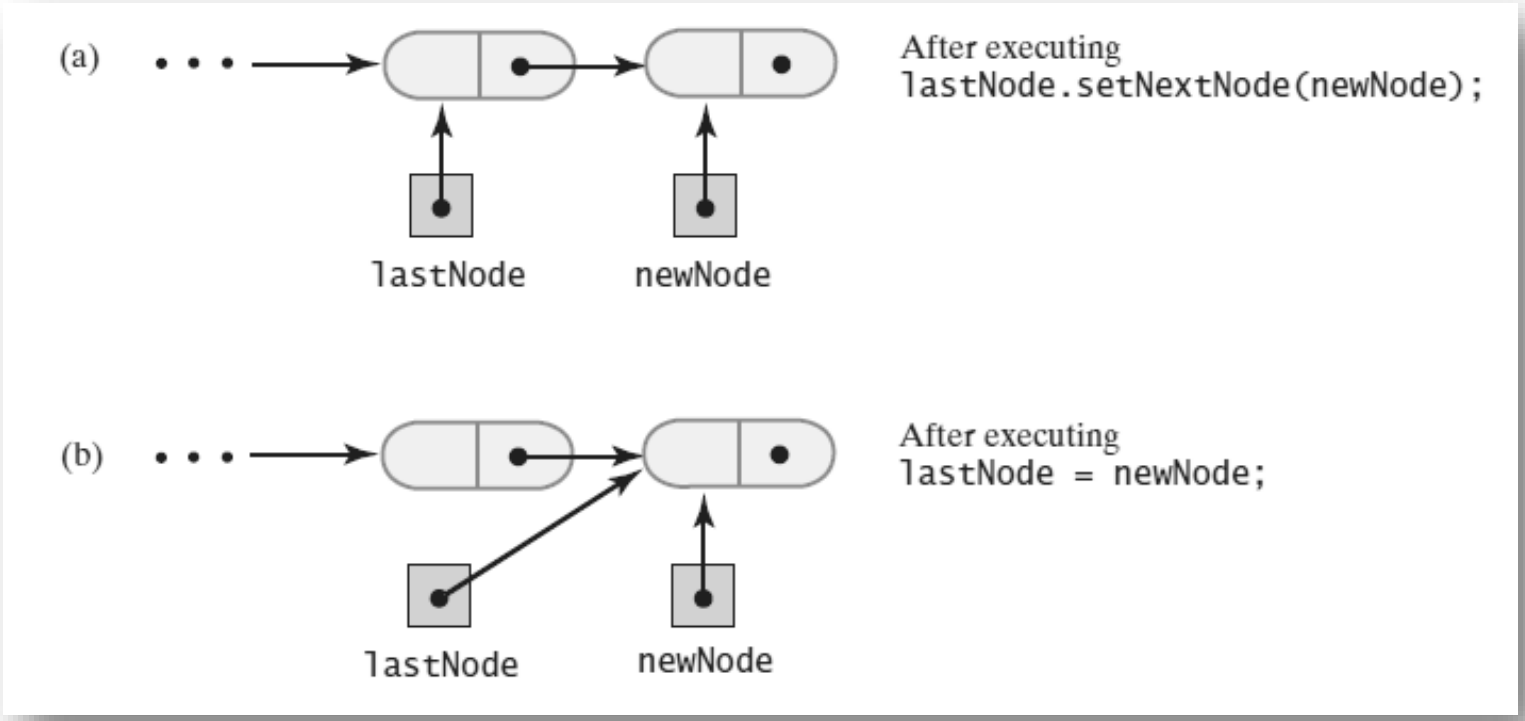
A Refined Implementation

A linked chain with both a head reference and a tail reference



A Refined Implementation

Adding a node to the end of a nonempty chain that has a tail reference



A Refined Implementation

Revision of the first `add` method

```
public void add(T newEntry)
{
   Node newNode = new Node(newEntry);

   if (isEmpty())
      firstNode = newNode;
   else
      lastNode.setNextNode(newNode);

   lastNode = newNode;

   numberOfEntries++;
} // end add
```

A Refined Implementation

Implementation of the method that adds by position.

```
public void add(int newPosition, T newEntry)
{
if ((newPosition >= 1) && (newPosition <= numberOfEntries + 1))
{
    Node newNode = new Node(newEntry);
    if (isEmpty())
    {
       firstNode = newNode;
       lastNode = newNode;
    }
    else if (newPosition == 1)
    {
       newNode.setNextNode(firstNode);
       firstNode = newNode;
    }
    else if (newPosition == numberOfEntries + 1)
```

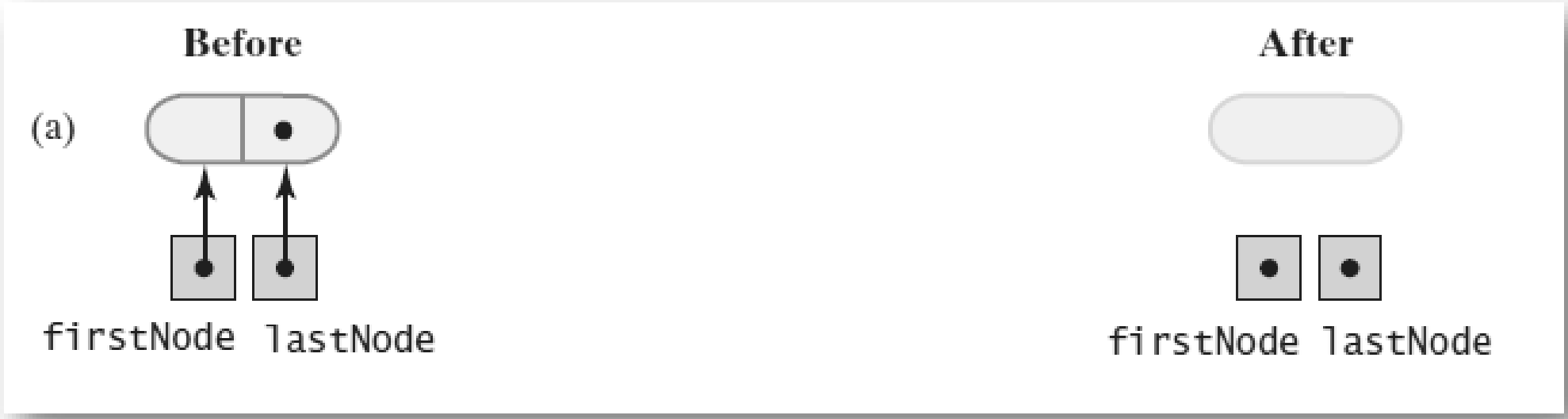
A Refined Implementation

Implementation of the method that adds by position.

```
            firstNode = newNode;
        }
        else if (newPosition == numberOfEntries + 1)
        {
            lastNode.setNextNode(newNode);
            lastNode = newNode;
        }
        else
        {
            Node nodeBefore = getNodeAt(newPosition - 1);
            Node nodeAfter = nodeBefore.getNextNode();
            newNode.setNextNode(nodeAfter);
            nodeBefore.setNextNode(newNode);
        } // end if

        numberOfEntries++;
    }
    else
        throw new IndexOutOfBoundsException(
                  "Illegal position given to add operation.");
} // end add
```

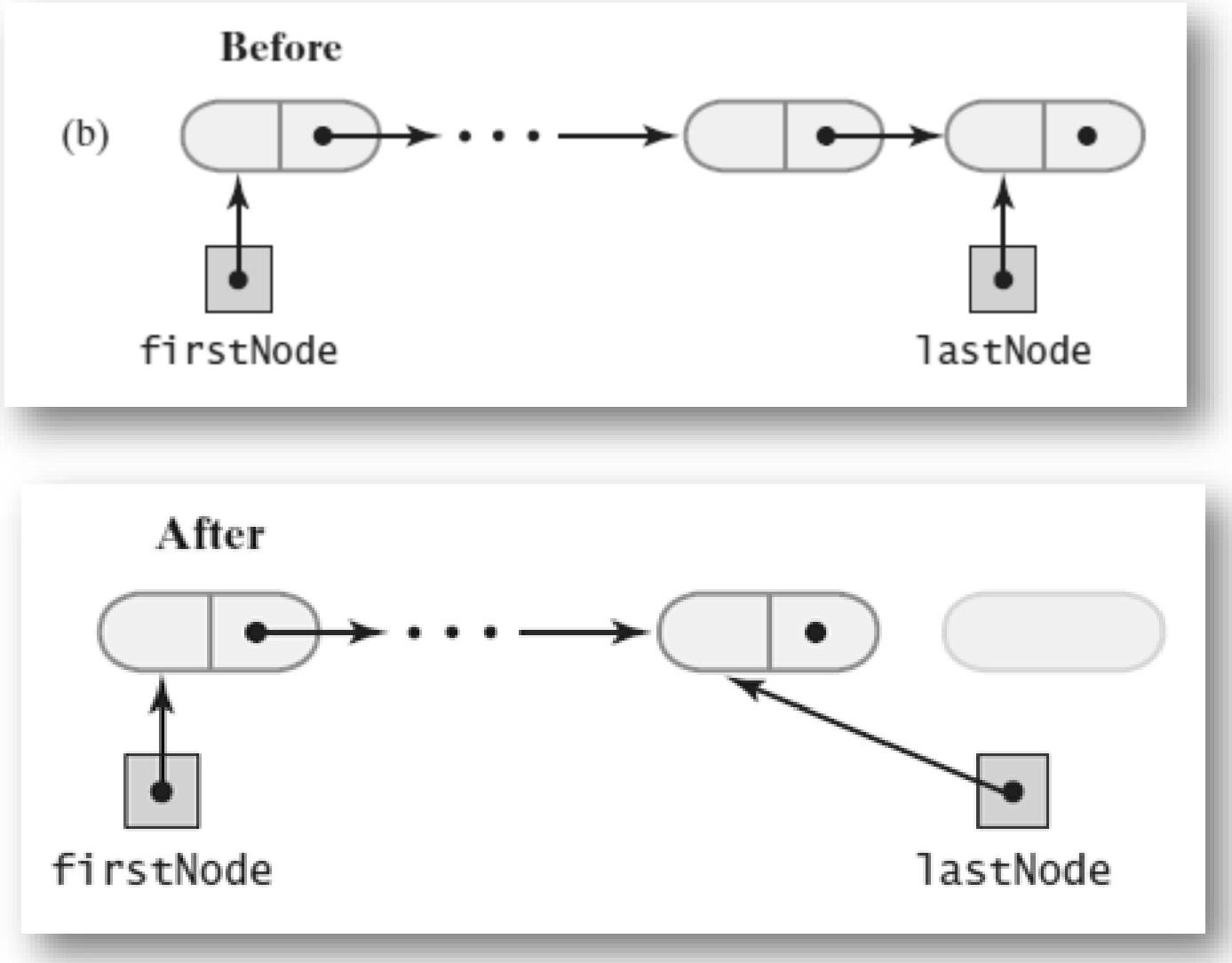
A Refined Implementation

Removing the last node from a chain that has both head and tail references when
the chain contains (a) one node



A Refined Implementation

Removing the last node from a chain that has both head and tail references when the chain contains (b) more than one node



A Refined Implementation

Implementation of the remove operation:

```
public T remove(int givenPosition)
{
   T result = null;                                  // Return value
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {
      assert !isEmpty();
      if (givenPosition == 1)                        // Case 1: Remove first entry
      {
         result = firstNode.getData();               // Save entry to be removed
         firstNode = firstNode.getNextNode();
         if (numberOfEntries == 1)
            lastNode = null;                         // Solitary entry was removed
      }
      else                                           // Case 2: Not first entry
      {
         Node nodeBefore = getNodeAt(givenPosition - 1);
         Node nodeToRemove = nodeBefore.getNextNode();
```

A Refined Implementation

Implementation of the remove operation:

```
            Node nodeBefore = getNodeAt(givenPosition - 1);
            Node nodeToRemove = nodeBefore.getNextNode();
            Node nodeAfter = nodeToRemove.getNextNode();
            nodeBefore.setNextNode(nodeAfter);
            result = nodeToRemove.getData();     // Save entry to be removed

            if (givenPosition == numberOfEntries)
               lastNode = nodeBefore;            // Last node was removed
         } // end if
         numberOfEntries--;
      }
      else
         throw new IndexOutOfBoundsException(
                   "Illegal position given to remove operation.");
      return result;                              // Return removed entry
   } // end remove
```

Efficiency of Using a Chain

The time efficiencies of the ADT list operations for three implementations, expressed in Big Oh notation

When 2 expressions are given: beginning of list and rest

When 3 expressions are given: beginning, middle, and end

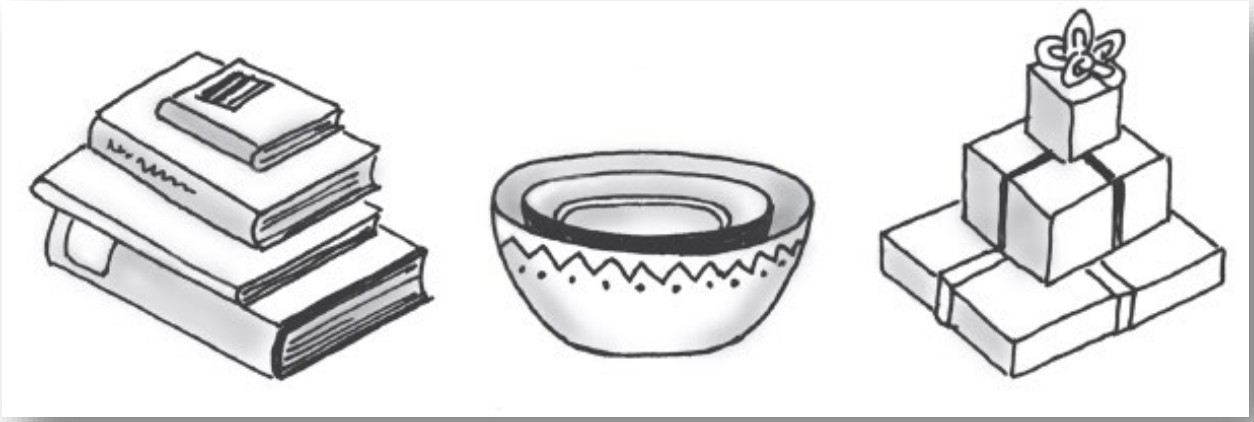
Operation	AList	LList	LList2
add(newEntry)	$O(1)$	$O(n)$	$O(1)$
add(newPosition, newEntry)	$O(n)$; $O(1)$	$O(1)$; $O(n)$	$O(1)$; $O(n)$; $O(1)$
toArray()	$O(n)$	$O(n)$	$O(n)$
remove(givenPosition)	$O(n)$; $O(1)$	$O(1)$; $O(n)$	$O(1)$; $O(n)$
replace(givenPosition, newEntry)	$O(1)$	$O(1)$; $O(n)$	$O(1)$; $O(n)$; $O(1)$
getEntry(givenPosition)	$O(1)$	$O(1)$; $O(n)$	$O(1)$; $O(n)$; $O(1)$
contains(anEntry)	$O(n)$	$O(n)$	$O(n)$
clear(), getLength(), isEmpty()	$O(1)$	$O(1)$	$O(1)$

Java Class Library: The Class `LinkedList`

- Implements the interface **`List`**
- **`LinkedList`** defines more methods than are in the interface **`List`**
- You can use the class **`LinkedList`** as implementation of ADT
  - queue
  - deque
  - or list.

Stacks

- Some familiar stacks



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

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?)
    - Can use `java.util.EmptyStackException`
    - or define our own Exception class

Interface

An interface for the ADT stack

```
1  public interface StackInterface<T>
2  {
3     /** Adds a new entry to the top of this stack.
4         @param newEntry  An object to be added to the stack. */
5     public void push(T newEntry);
6
7     /** Removes and returns this stack's top entry.
8         @return  The object at the top of the stack.
9         @throws  EmptyStackException if the stack is empty before
10        the operation. */
11    public T pop();
12
13    /** Retrieves this stack's top entry.
14        @return  The object at the top of the stack.
```

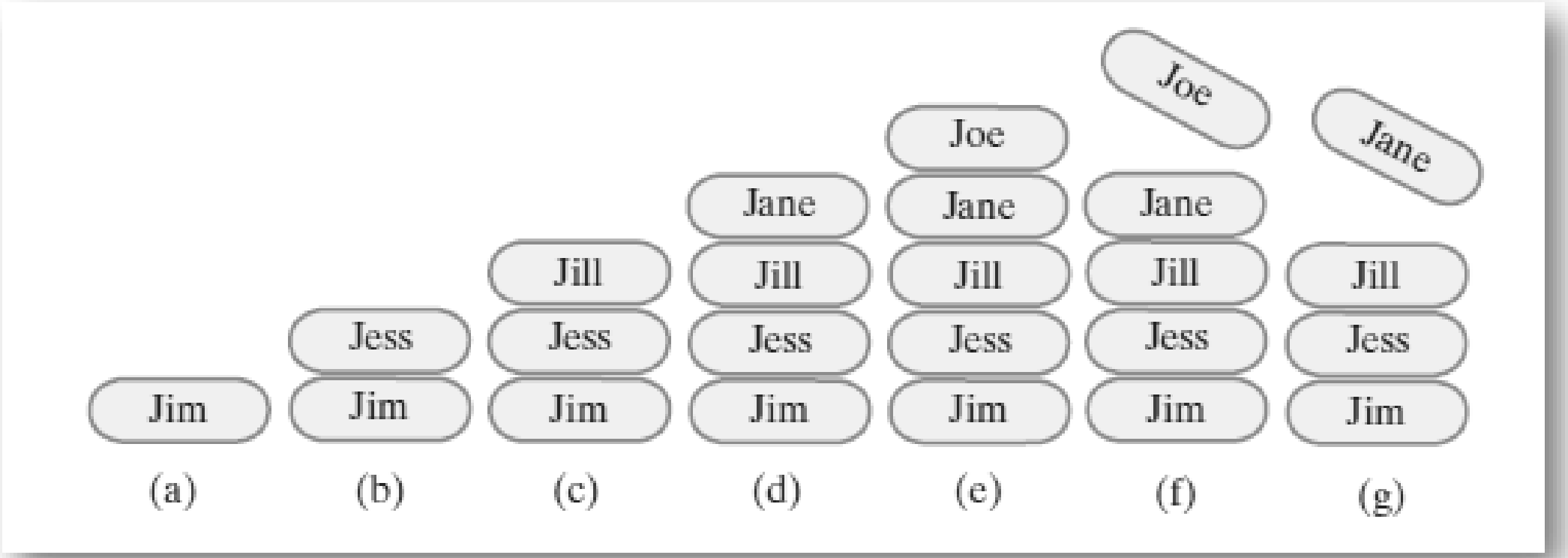
Interface

An interface for the ADT stack

```
13    /** Retrieves this stack's top entry.
14        @return  The object at the top of the stack.
15        @throws  EmptyStackException if the stack is empty. */
16    public T peek();
17
18    /** Detects whether this stack is empty.
19        @return  True if the stack is empty. */
20    public boolean isEmpty();
21
22    /** Removes all entries from this stack. */
23    public void clear();
24 } // end StackInterface
```

Example

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



Design guidelines for Interfaces

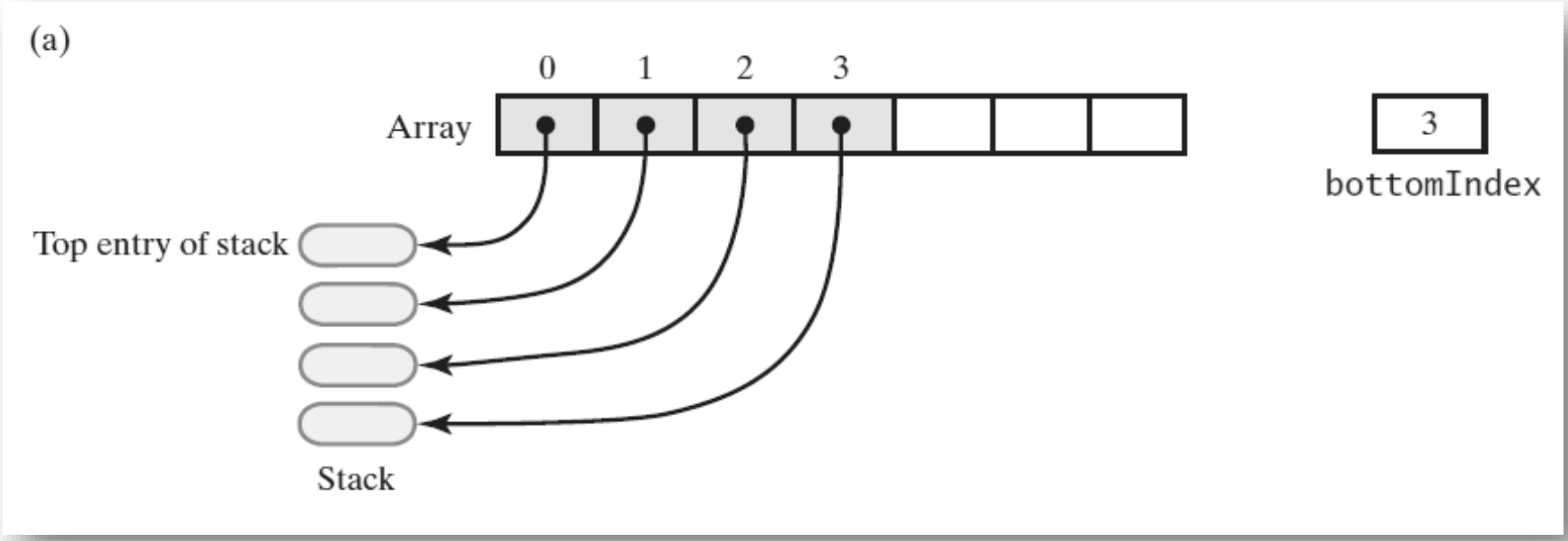
- Use preconditions and postconditions to document assumptions
- Do not trust client to use public methods correctly
- Avoid ambiguous return values
- Prefer throwing exceptions instead of returning values to signal problem

Array-Based 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

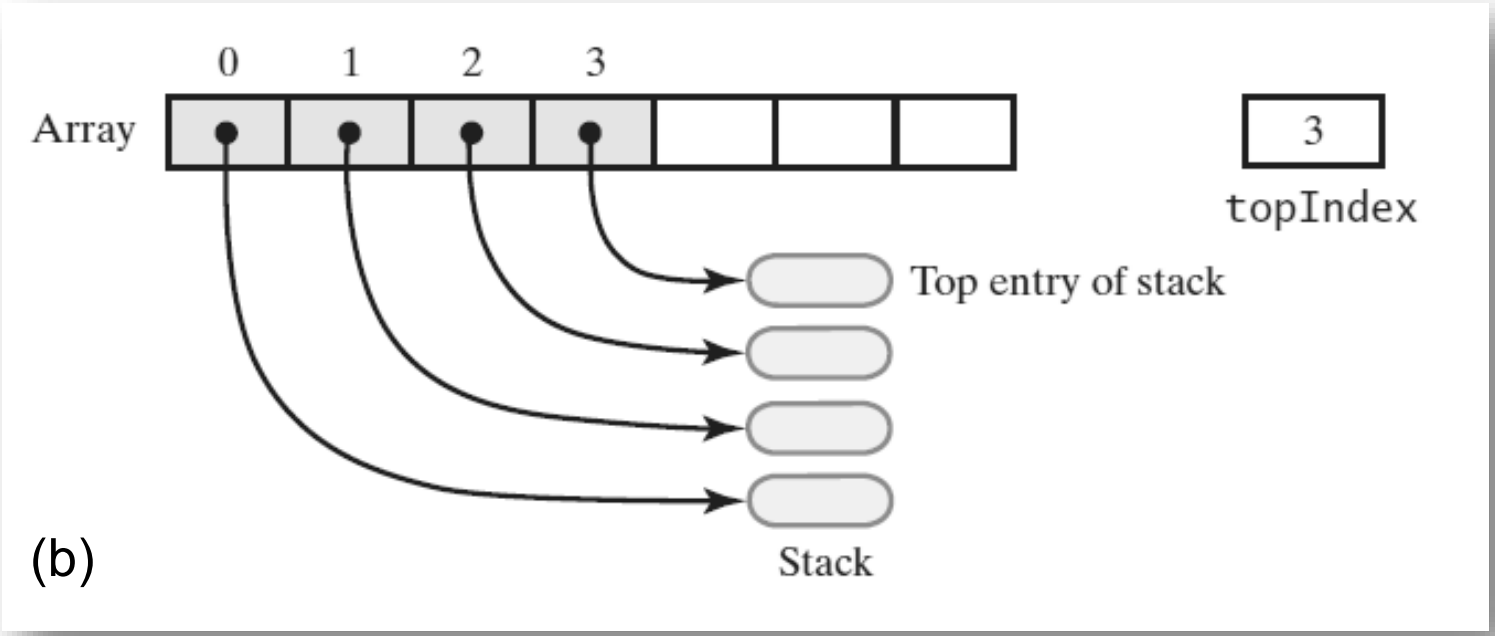
In-efficient Implementation

An array that implements a stack; its first location references (a) the top entry in the stack;



Efficient Array-Based Implementation

An array that implements a stack; its first location references (b) the bottom entry in the stack



Array-Based Implementation

An outline of an array-based implementation of the ADT stack

```
1  /**
2      A class of stacks whose entries are stored in an array.
3      @author Frank M. Carrano
4  */
5  public final class ArrayStack<T> implements StackInterface<T>
6  {
7     private T[] stack;     // Array of stack entries
8     private int topIndex; // Index of top entry
9     private boolean initialized = false;
10    private static final int DEFAULT_CAPACITY = 50;
11    private static final int MAX_CAPACITY = 10000;
12
13    public ArrayStack()
14    {
15       this(DEFAULT_CAPACITY);
16    } // end default constructor
17
18    public ArrayStack(int initialCapacity)
```

Array-Based Implementation

An outline of an array-based implementation of the ADT stack

```
18    public ArrayStack(int initialCapacity)
19    {
20        checkCapacity(initialCapacity);
21
22        // The cast is safe because the new array contains null entries
23        @SuppressWarnings("unchecked")
24        T[] tempStack = (T[])new Object[initialCapacity];
25        stack = tempStack;
26        topIndex = -1;
27        initialized = true;
28    } // end constructor
29
30    < Implementations of the stack operations go here. >
31    < Implementations of the private methods go here; checkCapacity and checkInitialization
32      are analogous to those in Chapter 2. >
33    . . .
34 } // end ArrayStack
```

Array-Based Implementation

- Adding to the top.

```
public void push(T newEntry)
{
   checkInitialization();
   ensureCapacity();
   stack[topIndex + 1] = newEntry;
   topIndex++;
} // end push

private void ensureCapacity()
{
   if (topIndex == stack.length - 1) // If array is full, double its size
   {
      int newLength = 2 * stack.length;
      checkCapacity(newLength);
      stack = Arrays.copyOf(stack, newLength);
   } // end if
} // end ensureCapacity
```

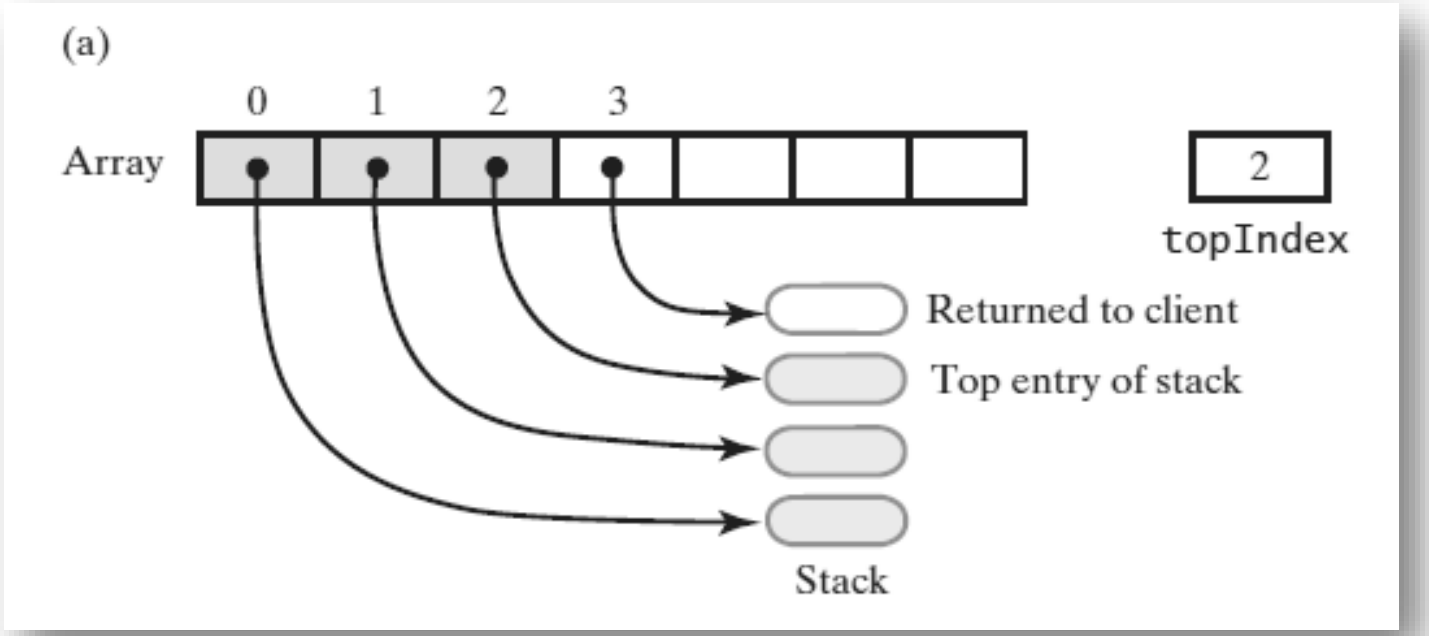
Array-Based Implementation

- Retrieving the top, operation is O(1)

```
public T peek()
{
   checkInitialization();
   if (isEmpty())
      throw new EmptyStackException();
   else
      return stack[topIndex];
} // end peek
```

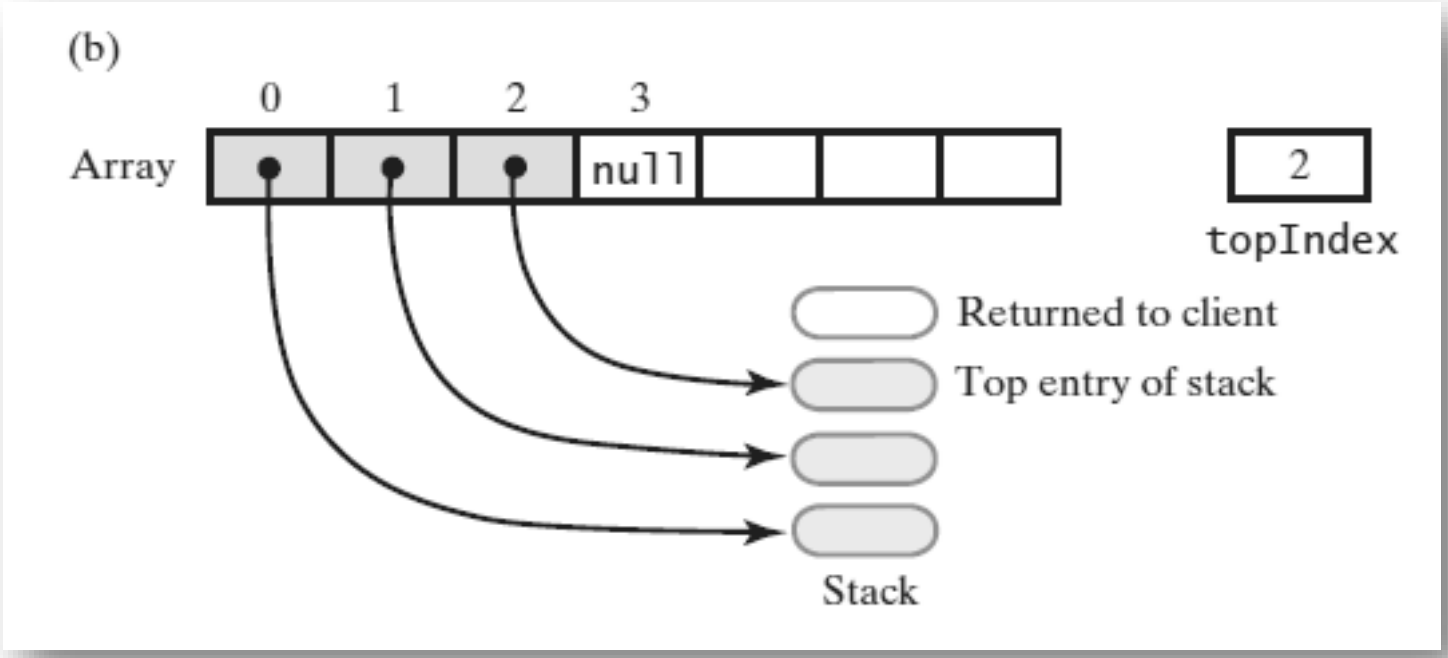
Array-Based Implementation

An array-based stack after its top entry is removed by (a) decrementing `topIndex`;



Array-Based Implementation

An array-based stack after its top entry is removed by (b) setting `stack[topIndex]` to null and then decrementing `topIndex`



Array-Based Implementation

- Removing the top

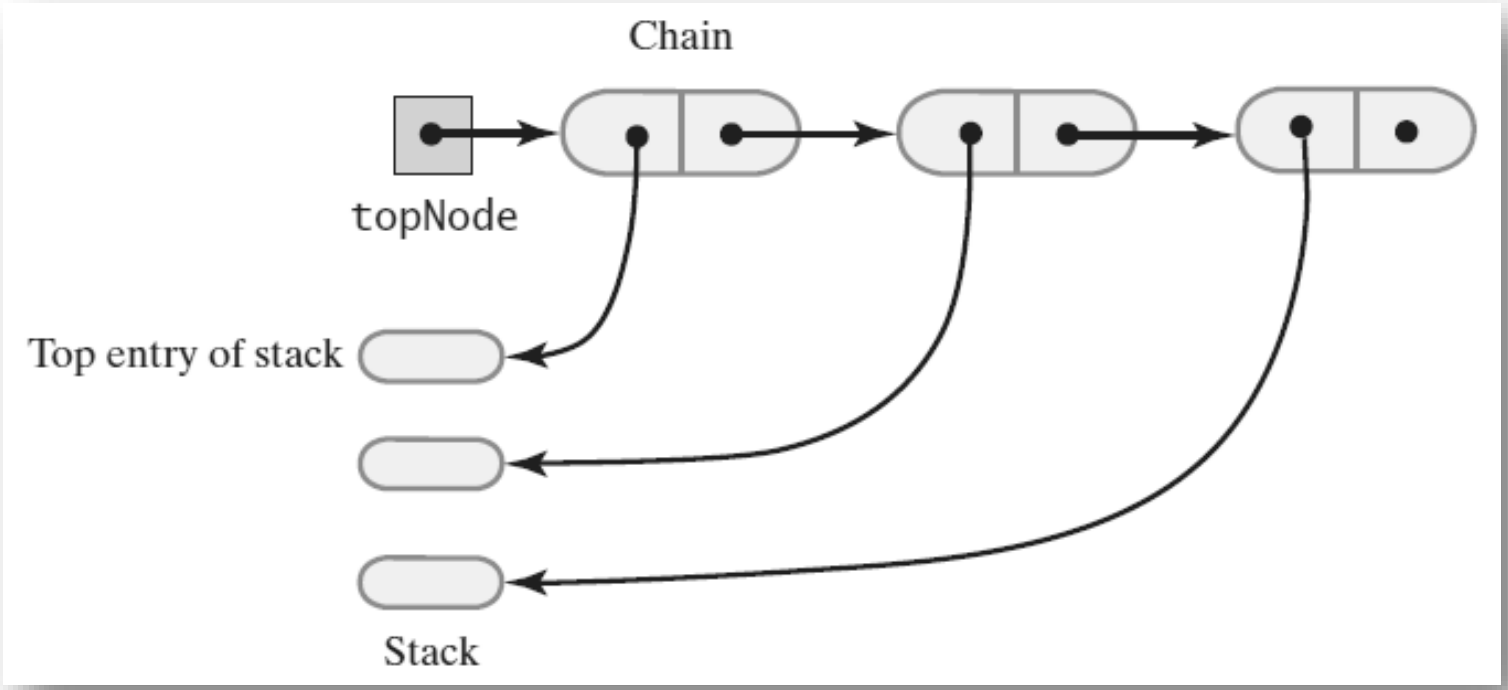
```
public T pop()
{
  checkInitialization();
  if (isEmpty())
     throw new EmptyStackException();
  else
  {
     T top = stack[topIndex];
     stack[topIndex] = null;
     topIndex--;
     return top;
  } // end if
} // end pop
```

Linked 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

A chain of linked nodes
that implements a stack



Linked Implementation

An outline of a linked implementation of the ADT stack

```
1  /**
2      A class of stacks whose entries are stored in a chain of nodes.
3      @author Frank M. Carrano
4  */
5  public final class LinkedStack<T> implements StackInterface<T>
6  {
7     private Node topNode; // References the first node in the chain
8
9     public LinkedStack()
10    {
11       topNode = null;
12    } // end default constructor
13    < Implementations of the stack operations go here. >
14    . . .
15
16    private class Node
17    {
18
```

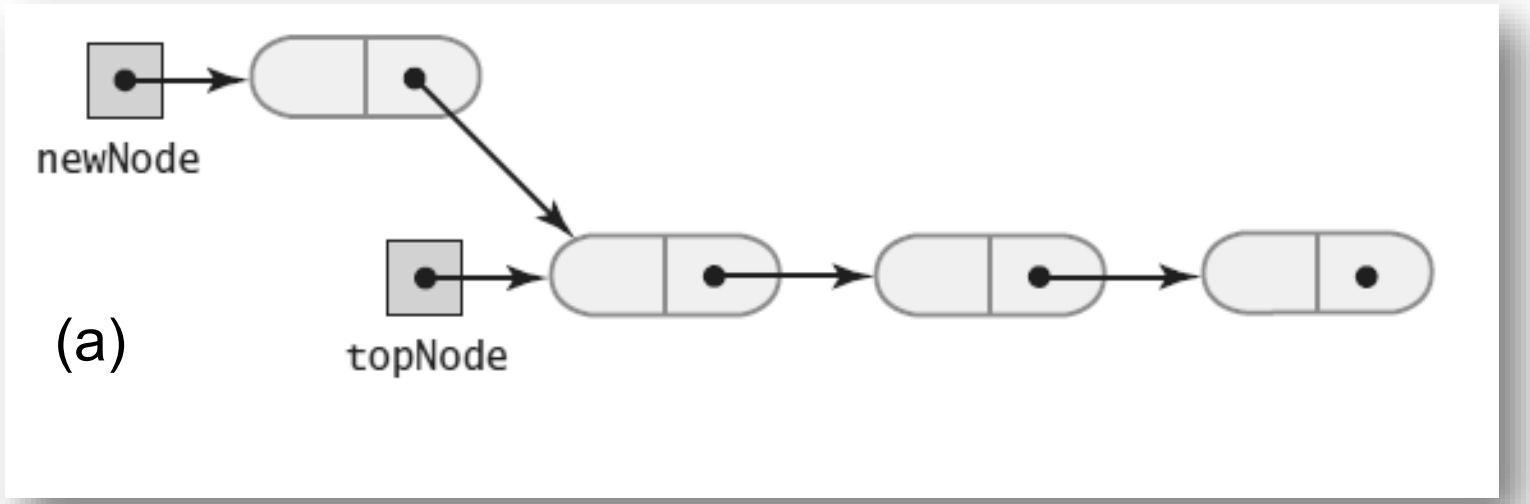
Linked Implementation

An outline of a linked implementation
of the ADT stack

```
19          private T    data; // Entry in stack
20          private Node next; // Link to next node
21
22          < Constructors and the methods getData, setData, getNextNode, and setNextNode
23            are here. >
24       } // end Node
25    } // end LinkedStack
```

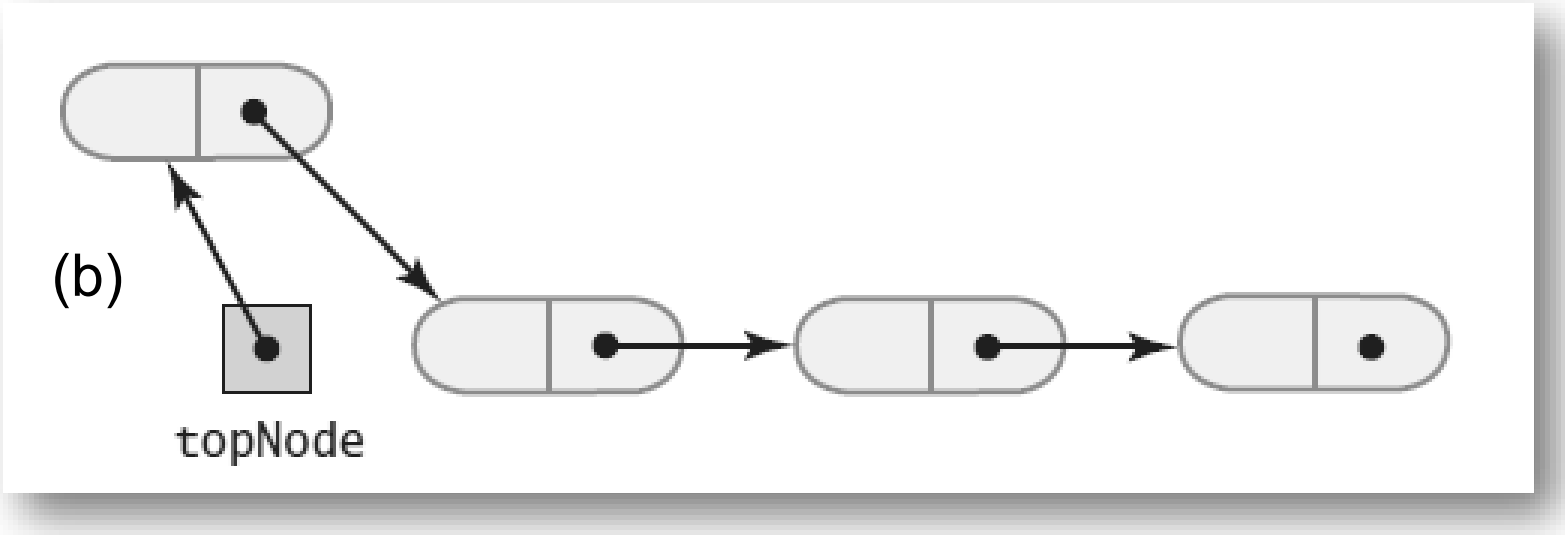
Linked Implementation

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



Linked Implementation

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



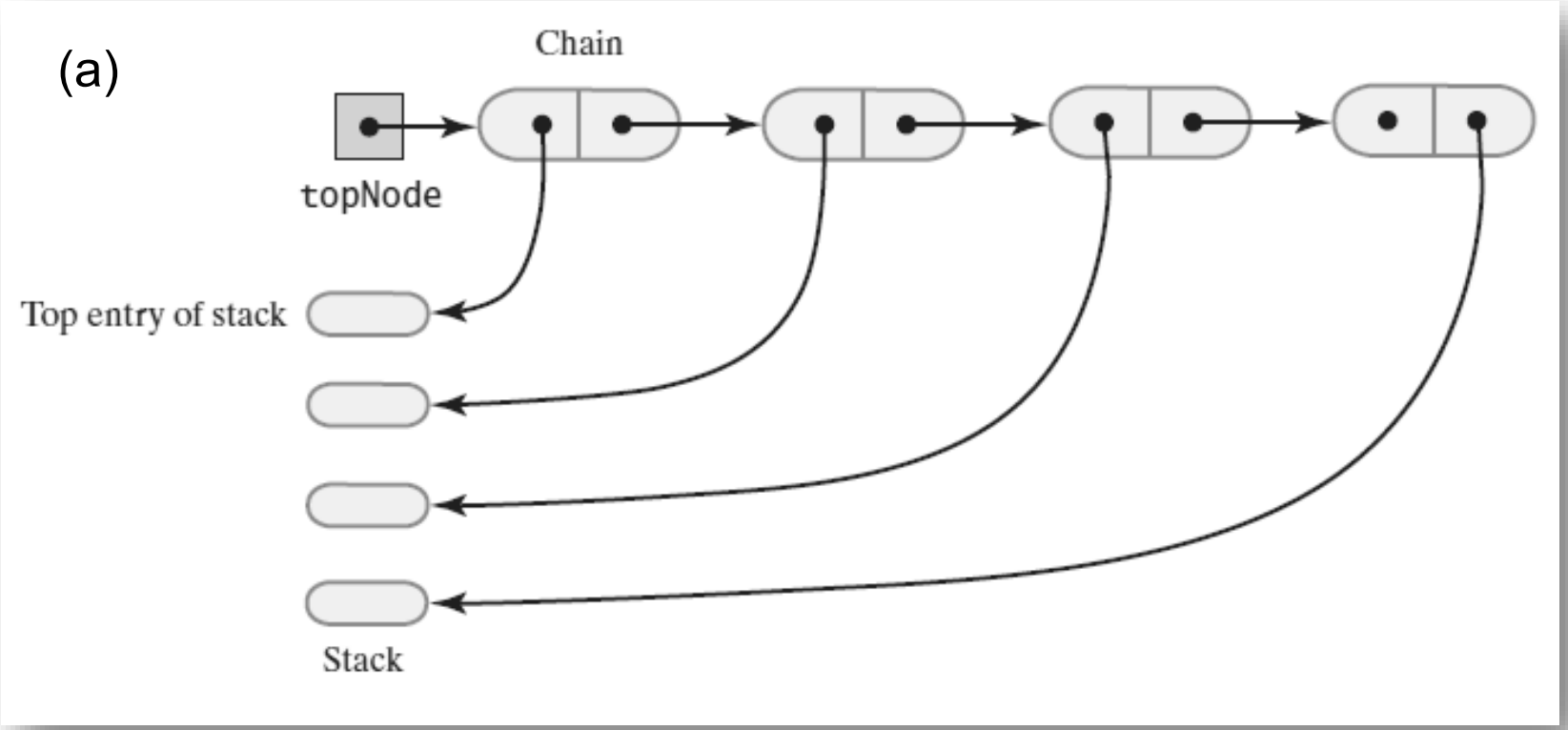
Linked Implementation

- Definition of **`push`**

```
public void push(T newEntry)
{
    Node newNode = new Node(newEntry, topNode);
    topNode = newNode;
} // end push
```

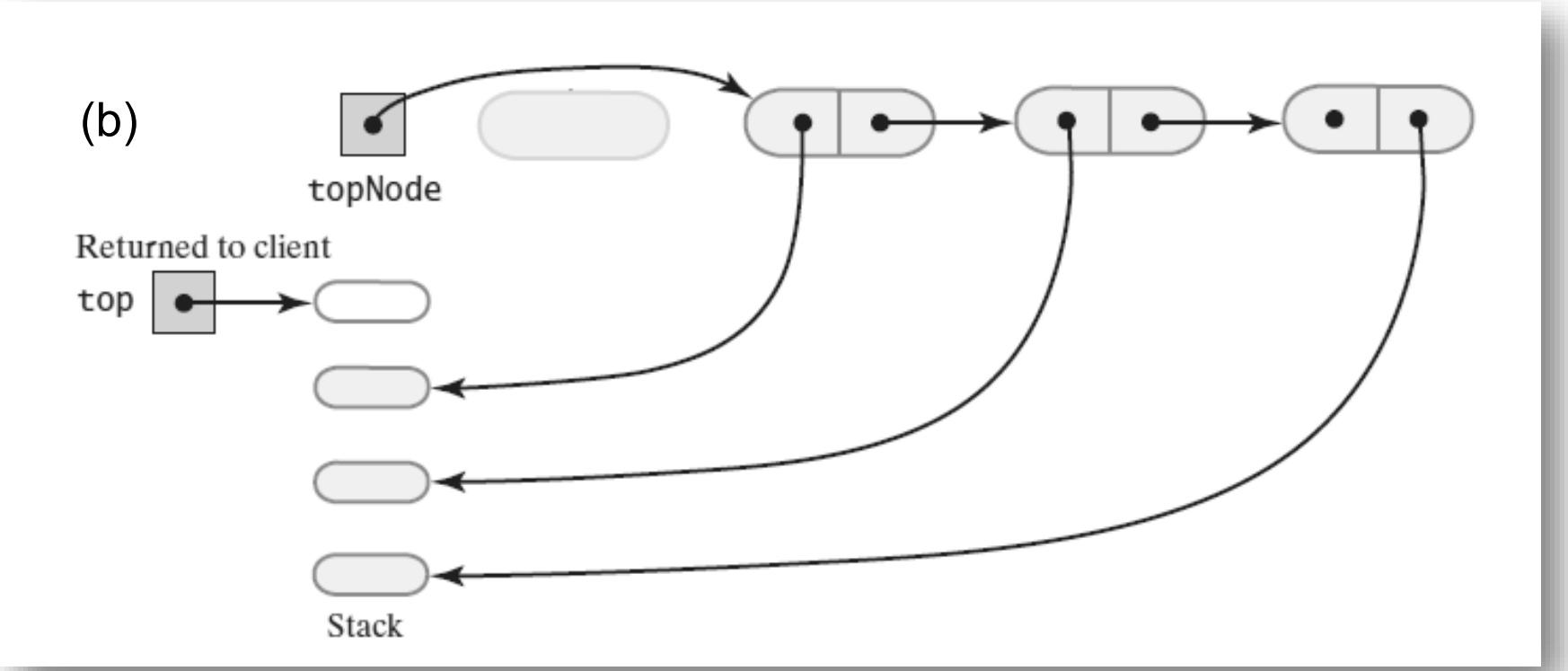
Linked Implementation

The stack (a) before the
first node in the chain is deleted



Linked Implementation

The stack (b) after the
first node in the chain is deleted



Linked Implementation

- Definition of `pop`

```
public T pop()
{
   T top = peek(); // Might throw EmptyStackException
   assert topNode != null;
   topNode = topNode.getNextNode();
   return top;
} // end pop
```

Linked Implementation

- Definition of rest of class.

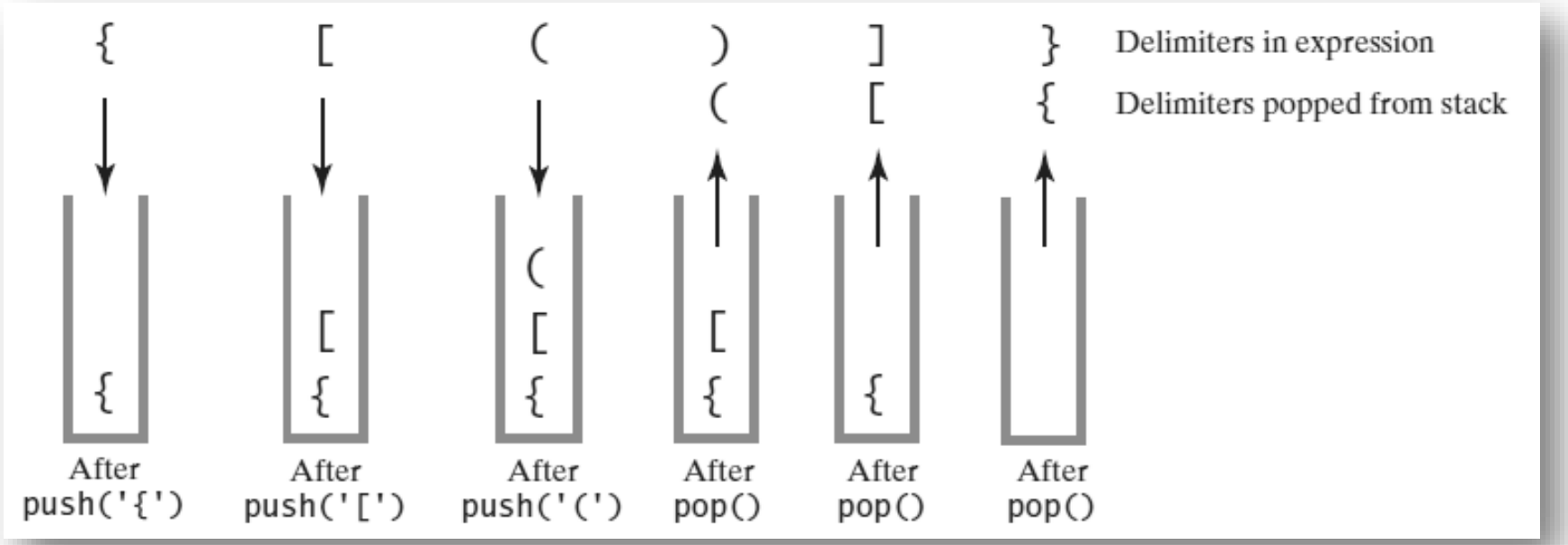
```
public boolean isEmpty()
{
   return topNode == null;
} // end isEmpty

public void clear()
{
   topNode = null;
} // end clear
```

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

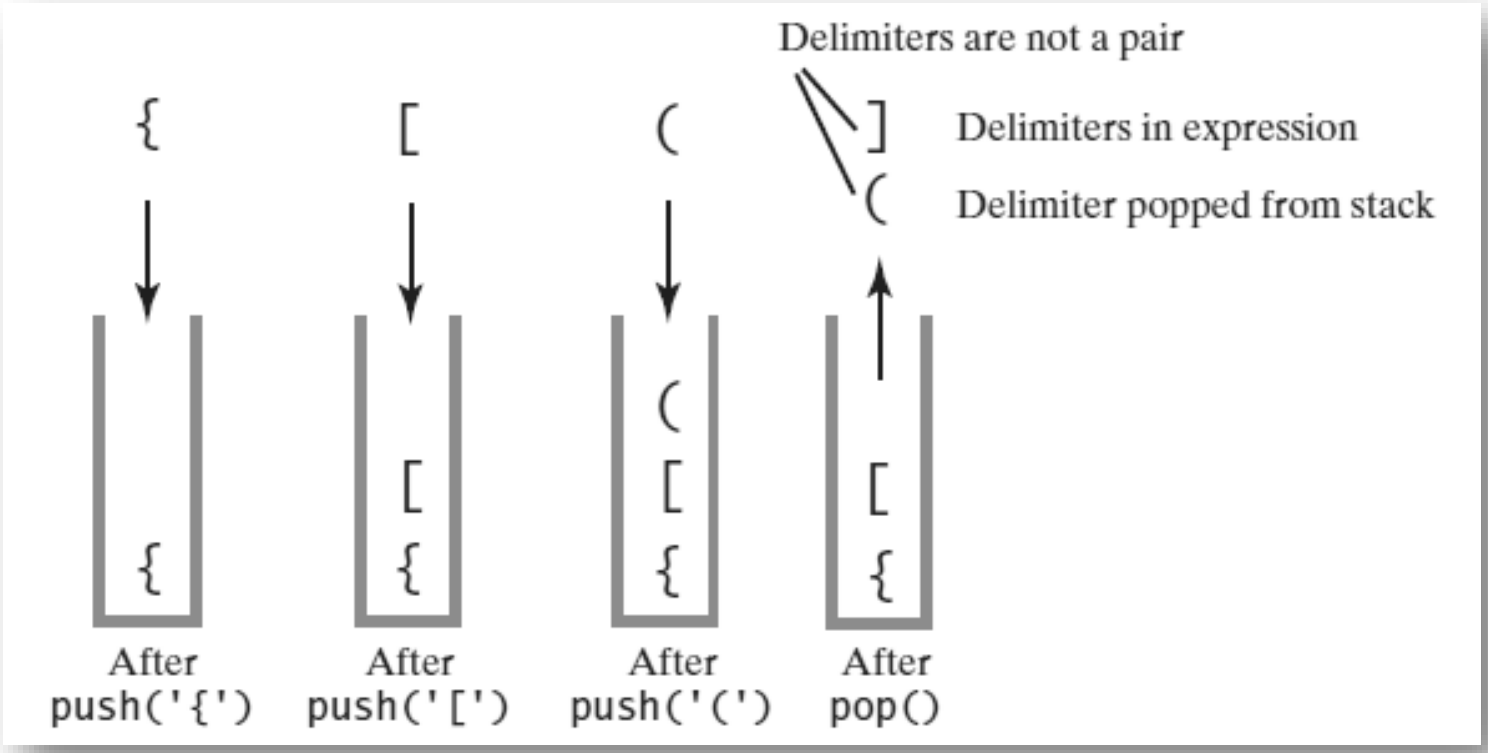
Processing Algebraic Expressions

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



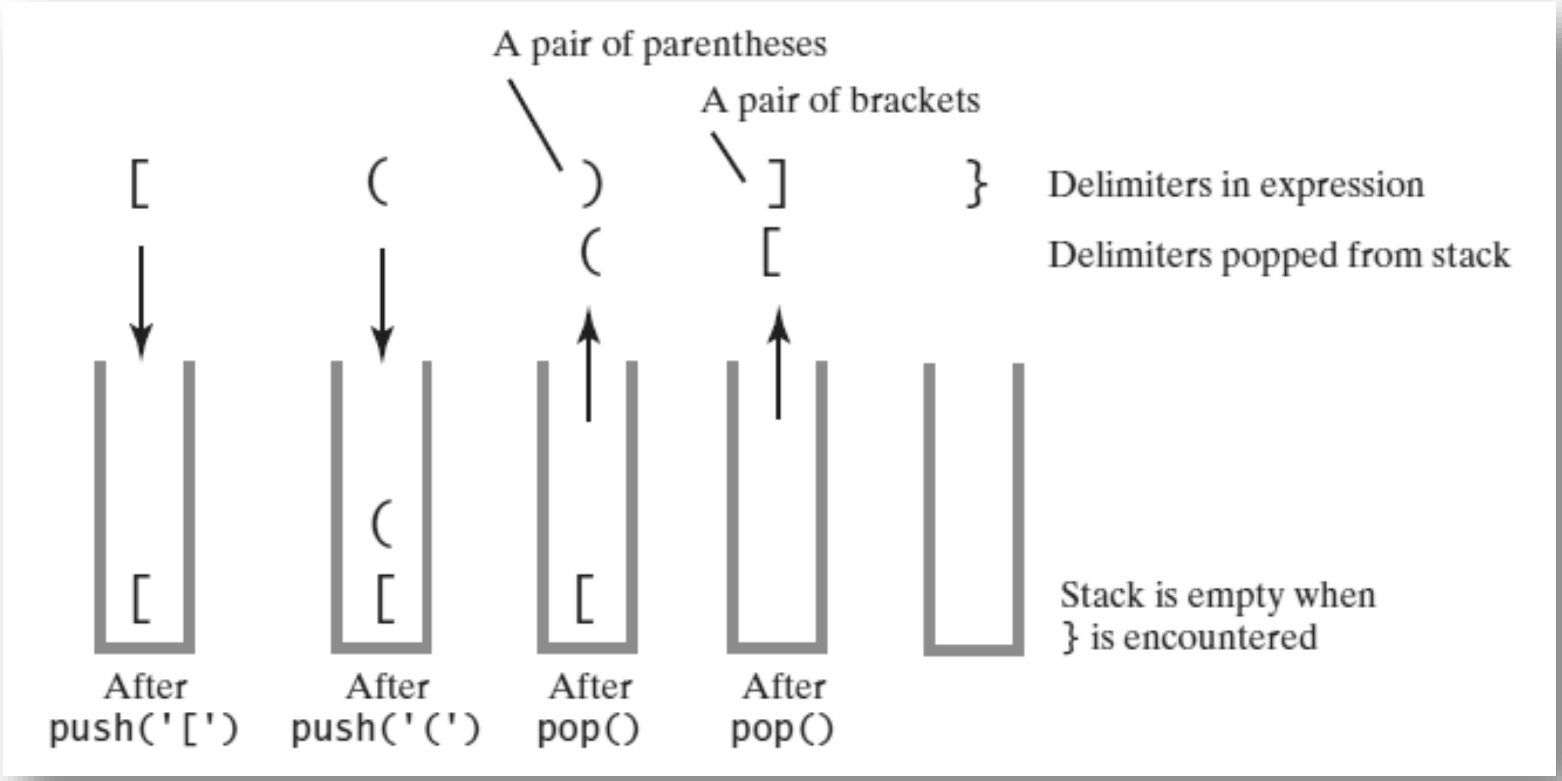
Processing Algebraic Expressions

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



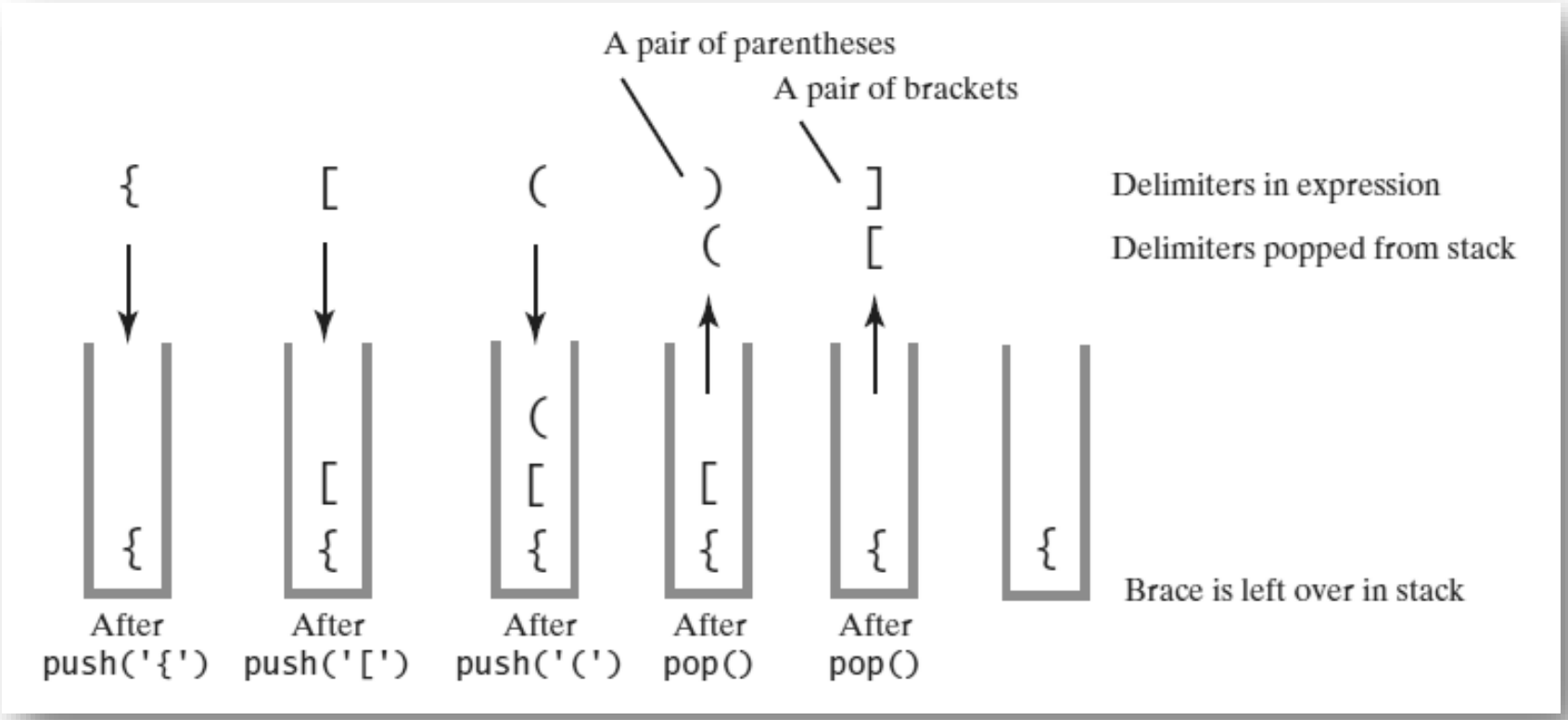
Processing Algebraic Expressions

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



Processing Algebraic Expressions

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



Processing Algebraic Expressions

- Algorithm to process for balanced expression.

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

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

- Algorithm to process for balanced expression.

```
            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
```

Java Implementation

The class `BalanceChecker`

```
1  public class BalanceChecker
2  {
3     /** Decides whether the parentheses, brackets, and braces
4         in a string occur in left/right pairs.
5         @param expression  A string to be checked.
6         @return  True if the delimiters are paired correctly. */
7     public static boolean checkBalance(String expression)
8     {
9        StackInterface<Character> openDelimiterStack = new OurStack<>();
10
11       int characterCount = expression.length();
12       boolean isBalanced = true;
13       int index = 0;
14       char nextCharacter = ' ';
15
16       while (isBalanced && (index < characterCount))
17       {
18          nextCharacter = expression.charAt(index);
19          switch (nextCharacter)
20          {
21             case '(': case '[': case '{':
```

Java Implementation

The class **`BalanceChecker`**

```
16        while (isBalanced && (index < characterCount))
17        {
18           nextCharacter = expression.charAt(index);
19           switch (nextCharacter)
20           {
21              case '(': case '[': case '{':
22                 openDelimiterStack.push(nextCharacter);
23                 break;
24              case ')': case ']': case '}':
25                 if (openDelimiterStack.isEmpty())
26                    isBalanced = false;
27                 else
28                 {
29                    char openDelimiter = openDelimiterStack.pop();
30                    isBalanced = isPaired(openDelimiter, nextCharacter);
31                 } // end if
```

Java Implementation

The class **`BalanceChecker`**

```
32                   break;
33                default: break; // Ignore unexpected characters
34             } // end switch
35             index++;
36          } // end while
37
38          if (!openDelimiterStack.isEmpty())
39             isBalanced = false;
40          return isBalanced;
41       } // end checkBalance
42
43       // Returns true if the given characters, open and close, form a pair
44       // of parentheses, brackets, or braces.
45       private static boolean isPaired(char open, char close)
46       {
47          return (open == '(' && close == ')') ||
48                 (open == '[' && close == ']') ||
49                 (open == '{' && close == '}');
50       } // end isPaired
51    } // end BalanceChecker
```

Infix to Postfix

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*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

Successive Operators with Same Precedence

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	$a\ b$	$-$
$+$	$a\ b\ -$	
	$a\ b\ -$	$+$
c	$a\ b\ -\ c$	$+$
	$a\ b\ -\ c\ +$	

Successive Operators with Same Precedence

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 ^ ^	

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
```

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
```

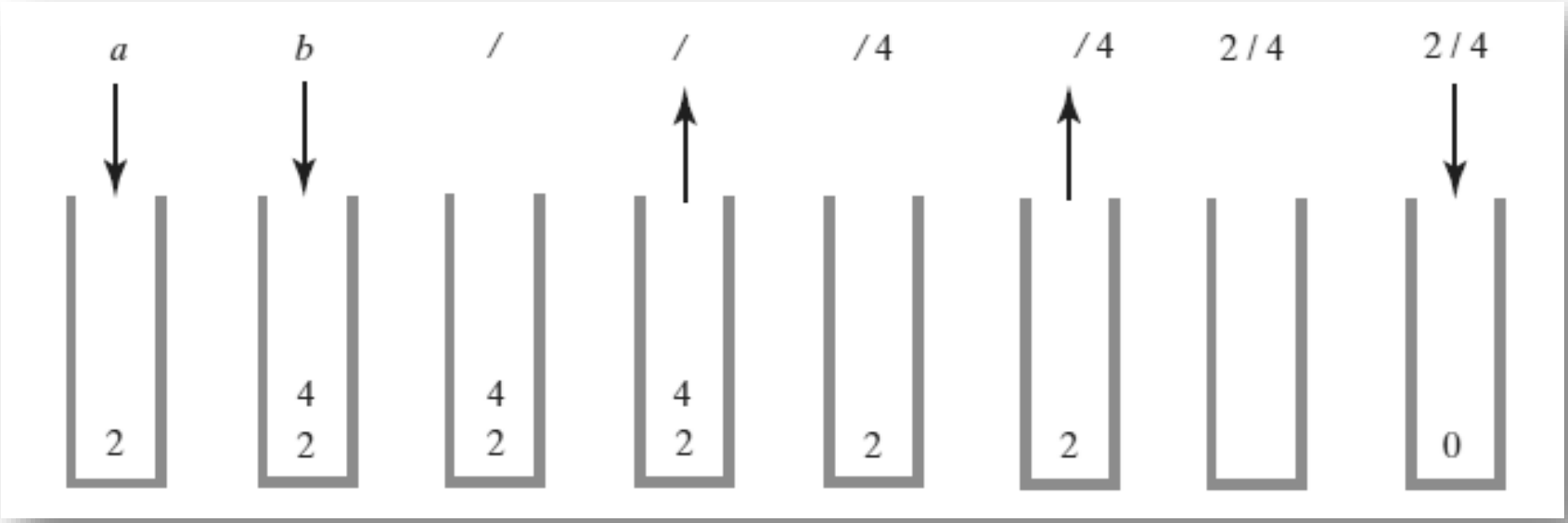
Infix to Postfix

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

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 − + **	

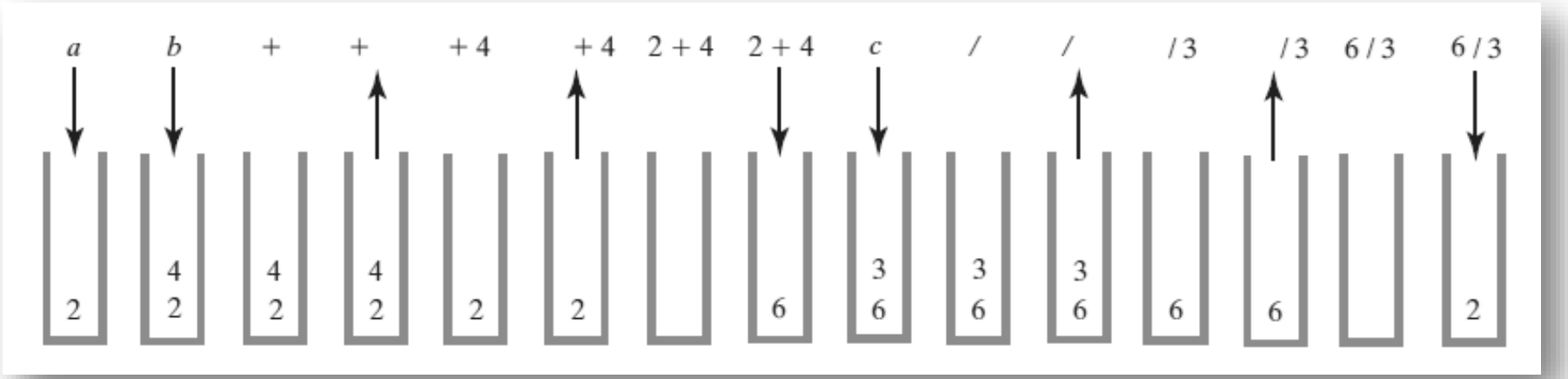
Evaluating Postfix Expressions

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



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



Evaluating Postfix Expressions

- Algorithm for 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 '^' :
```

Evaluating Postfix Expressions

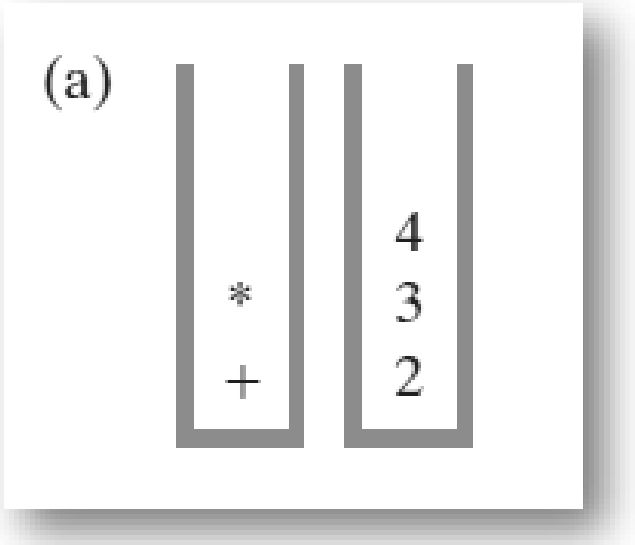
- Algorithm for evaluating postfix expressions.

```
            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
    }
}
```

Evaluating Infix Expressions

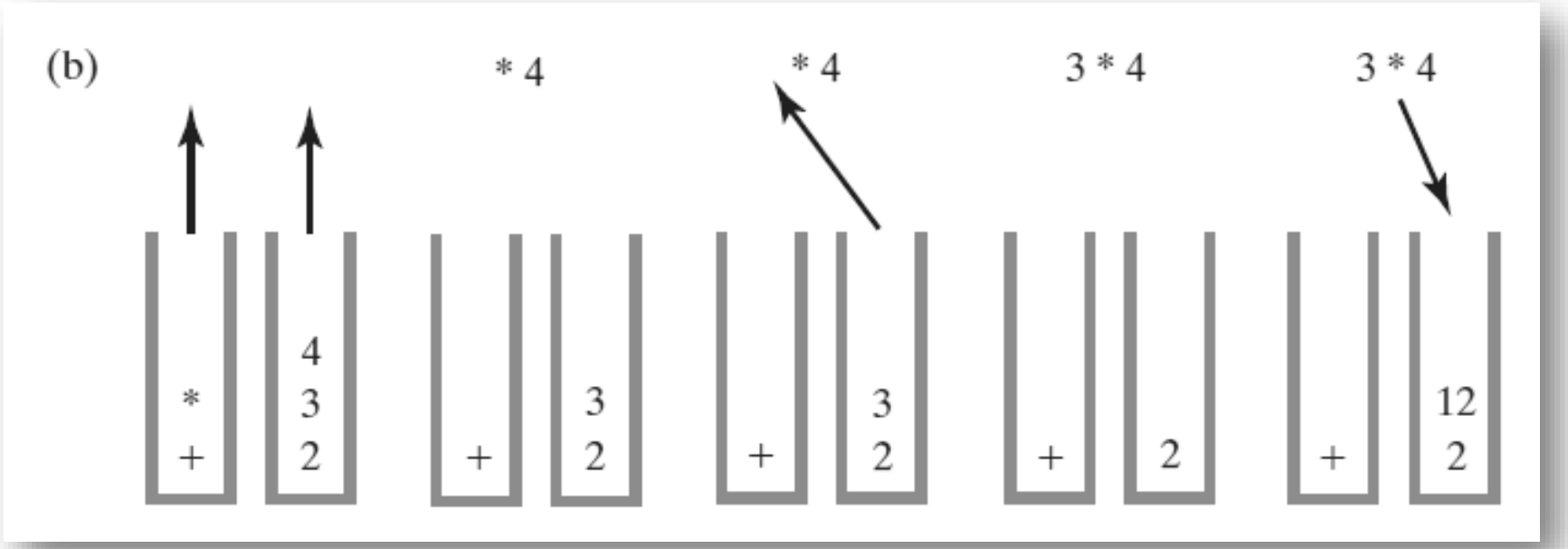
Two stacks during the evaluation of
a + b * c when a is 2, b is 3, and c is 4:

- (a) after reaching the end of the expression;



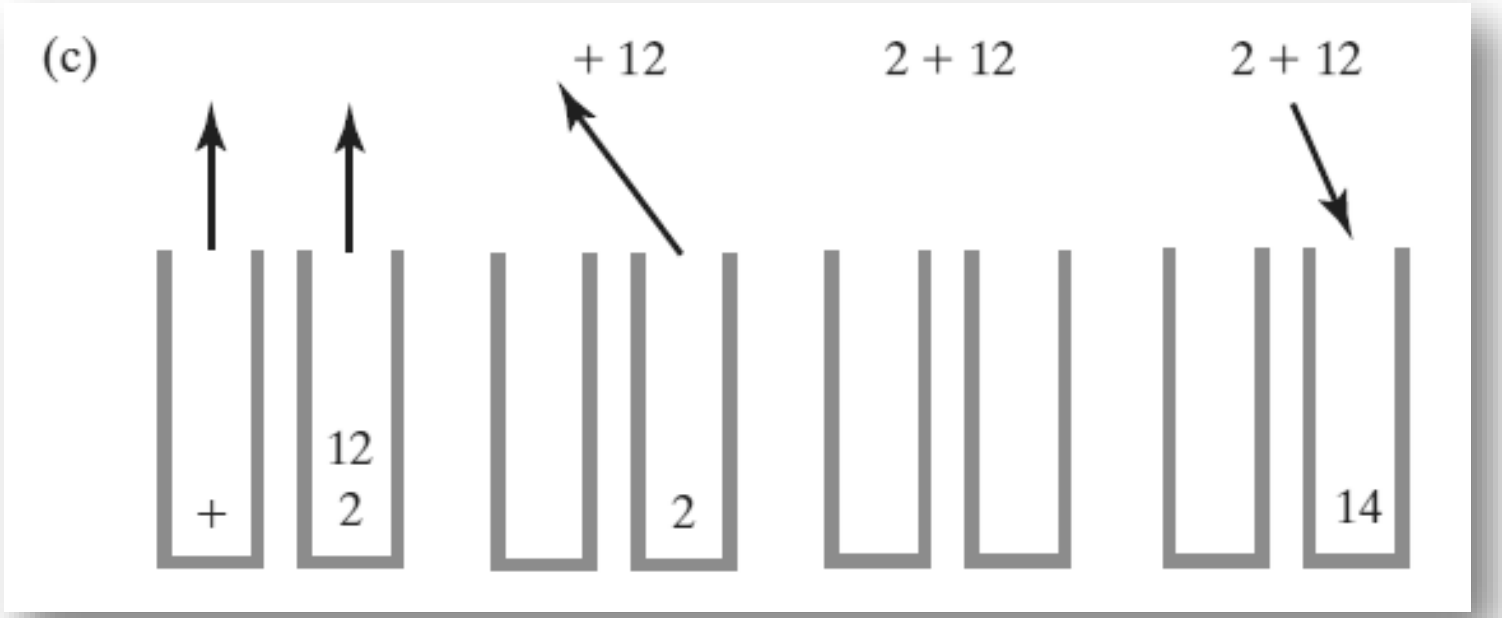
Evaluating Infix Expressions

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

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 to evaluate infix expression.

```
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

- Algorithm to evaluate infix expression.

```
    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

- Algorithm to evaluate infix expression.

```
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
```

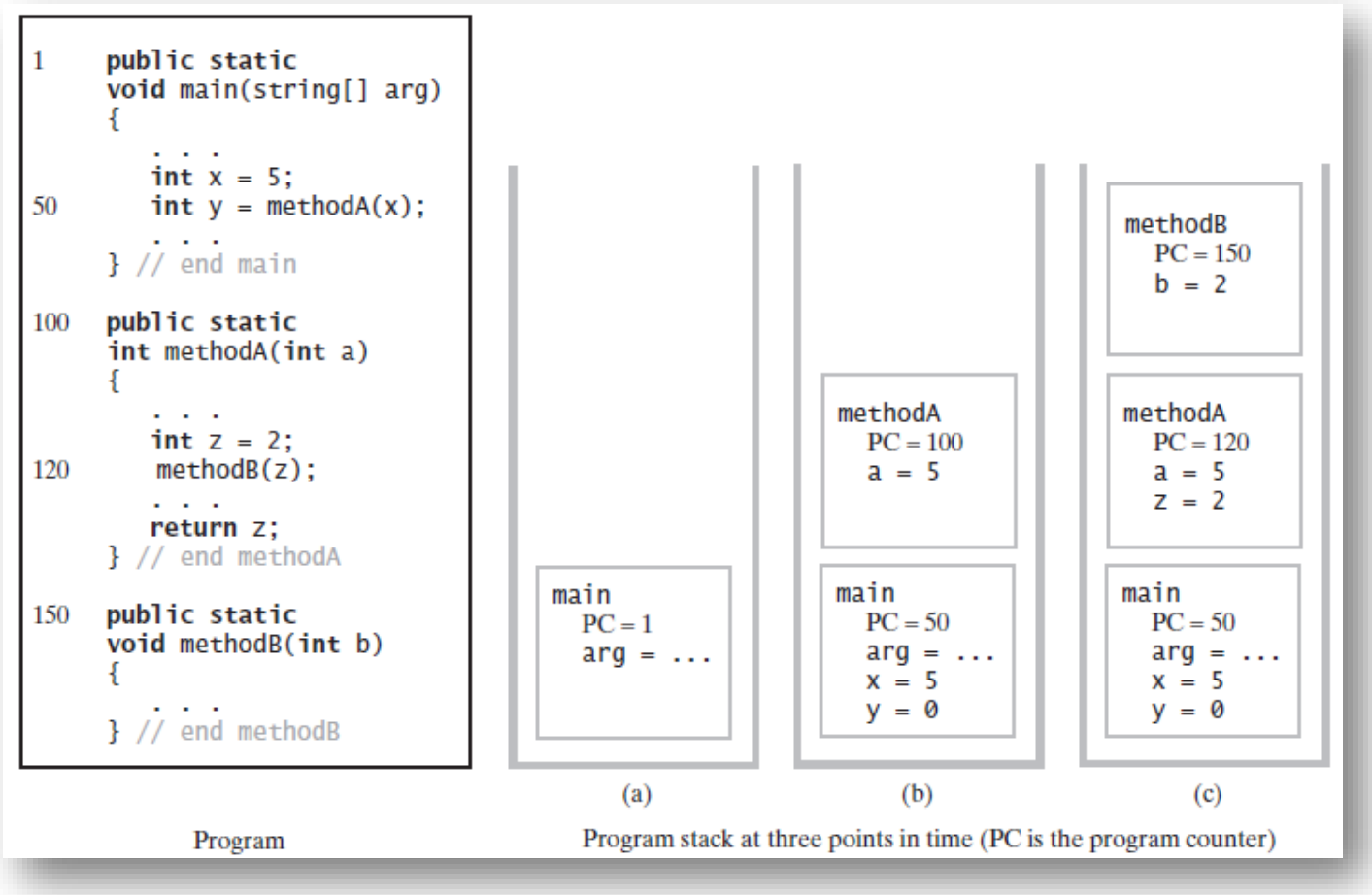
Evaluating Infix Expressions

- Algorithm to evaluate infix expression.

```
        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 Program Stack

The program stack at three points in time: (a) when main begins execution; (b) when methodA begins execution; (c) when methodB begins execution



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();`