Stacks

Based on Koffmann and Wolfgang

Chapter 5

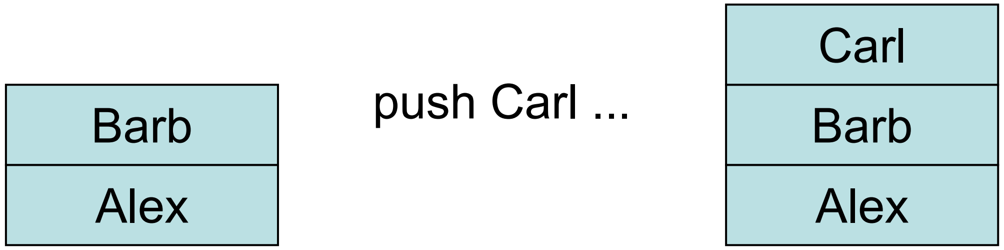
Chapter Outline

- The **`Stack<E>`** data type and its four methods:
    - **`push(E)`**, **`pop()`**, **`peek()`**, and **`empty()`**
- How the Java libraries implement **`Stack`**
- How to implement **`Stack`** using:
    - An array
    - A linked list
- Using **`Stack`** in applications
    - Finding palindromes
    - Testing for balanced (properly nested) parentheses
    - Evaluating arithmetic expressions

The **`Stack<E>`** Abstract Data Type

- A stack can be compared to a Pez dispenser
  - Only the top item can be accessed
  - Can only extract one item at a time
- A *stack* allows access only to the top element
- A stack's storage policy is *Last-In*, *First-Out*

FIGURE 5.1
A Pez Dispenser



Specification of **`Stack<E>`** As An Abstract Data Type

- Since only the top element of a stack is visible ...
  - There are only a few operations
- Need the ability to
  - *Inspect* the top element: **`peek()`**
  - *Retrieve & remove* the top element: **`pop()`**
  - *Push* a new element on the stack: **`push(E)`**
  - *Test* for an empty stack: **`empty()`**

API for **`Stack<E>`** (old)

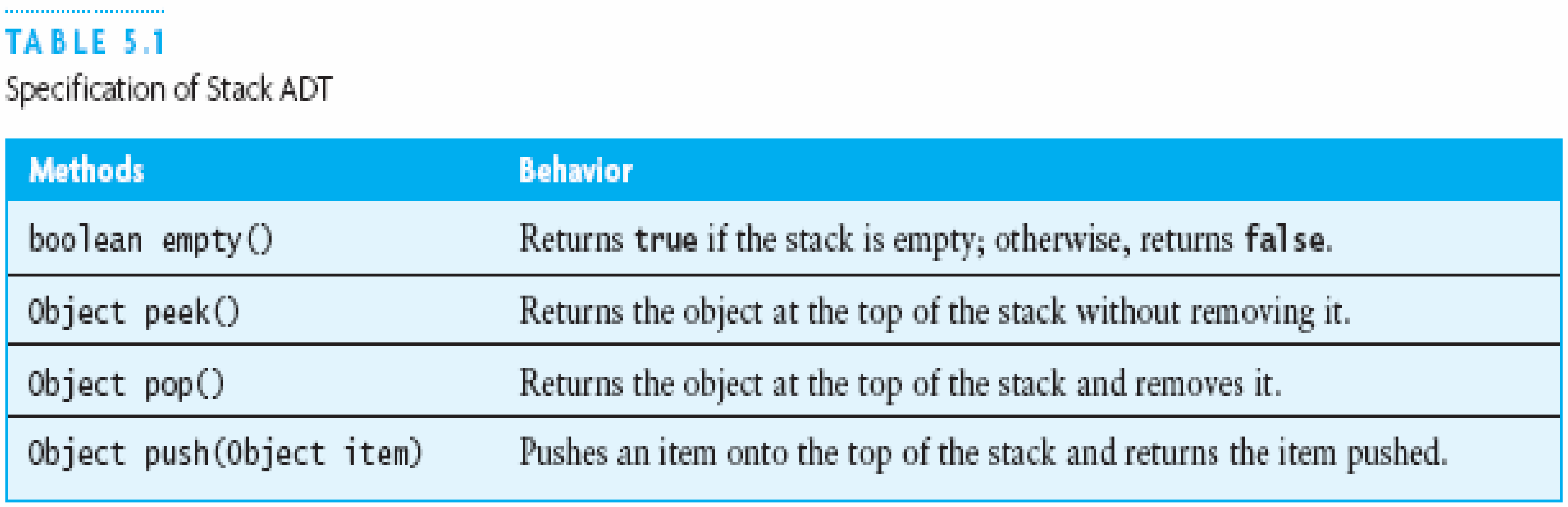


TABLE 5.1
Specification of Stack ADT

Methods	Behavior
`boolean empty()`	Returns **`true`** if the stack is empty; otherwise, returns **`false`**.
`Object peek()`	Returns the object at the top of the stack without removing it.
`Object pop()`	Returns the object at the top of the stack and removes it.
`Object push(Object item)`	Pushes an item onto the top of the stack and returns the item pushed.

Stack Applications

- Text studies two client programs using stacks
    - Palindrome finder
    - Parenthesis (bracket) matcher
- A <u>*palindrome*</u> is a string that is the same in either direction
    - “Able was I ere I saw Elba”
    - “A man, a plan, a canal: Panama”
    - Abba

Stack Applications (continued)

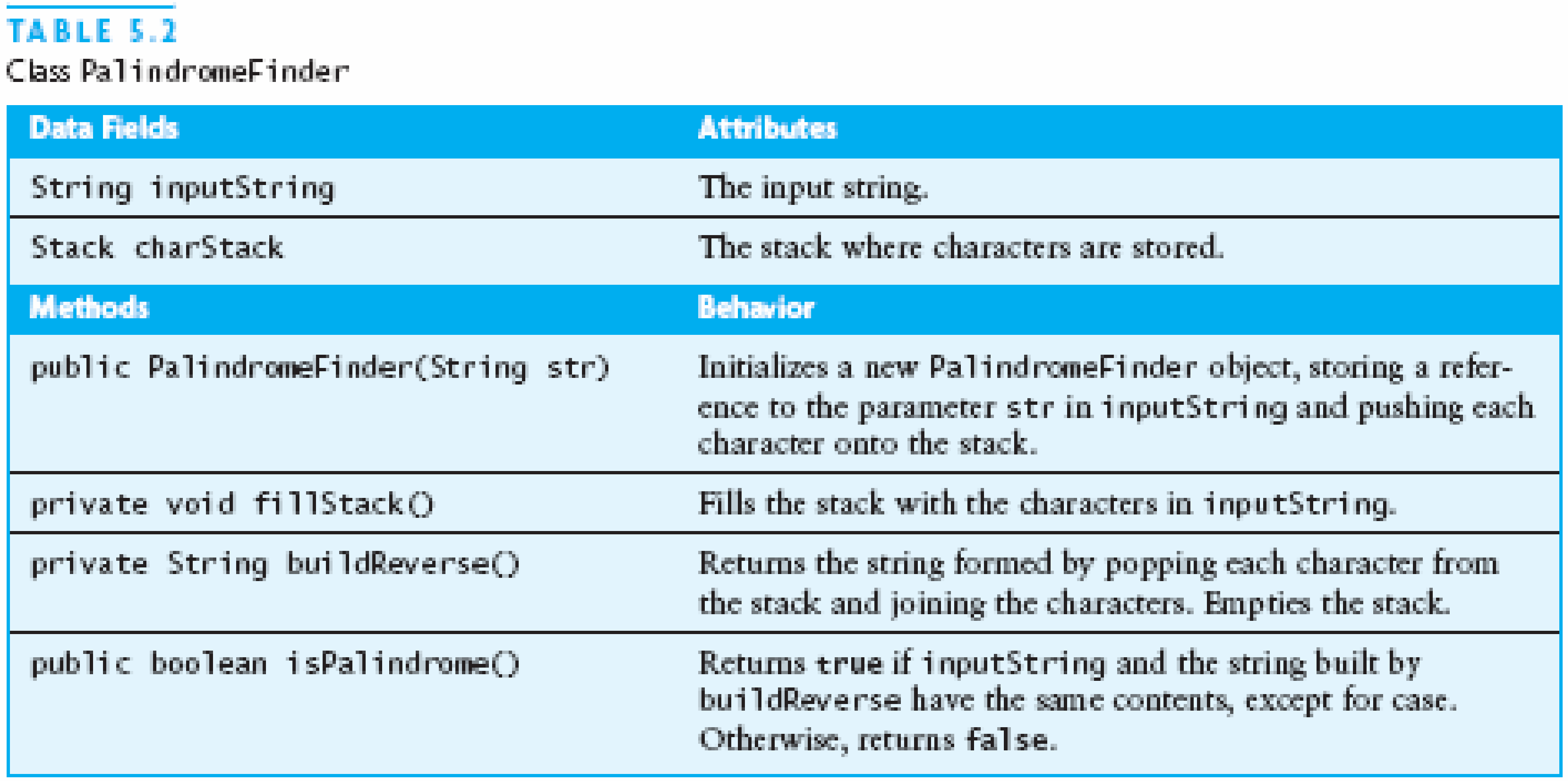


TABLE 5.2

Class `PalindromeFinder`

Data Fields	Attributes
`String inputString`	The input string.
`Stack charStack`	The stack where characters are stored.
Methods	**Behavior**
`public PalindromeFinder(String str)`	Initializes a new `PalindromeFinder` object, storing a reference to the parameter `str` in `inputString` and pushing each character onto the stack.
`private void fillStack()`	Fills the stack with the characters in `inputString`.
`private String buildReverse()`	Returns the string formed by popping each character from the stack and joining the characters. Empties the stack.
`public boolean isPalindrome()`	Returns `true` if `inputString` and the string built by `buildReverse` have the same contents, except for case. Otherwise, returns `false`.

Palindrome Code

```
public class Palindrome {
  private String input;
  private Stack<Character> charStack =
    new Stack<Character>();

  public Palindrome (String input) {
    this.input = input;
    fillStack();
  }

  ...

}
```

Palindrome Code (2)

```
// pushes chars of input String, such
// that last ends up on top, and they
// can be popped in reverse order

private void fillStack () {
  for (int i = 0, len = input.length();
       i < len;
       ++i) {
    charStack.push(input.charAt(i));
  }
}
```

Palindrome Code (3)

```
// adds chars, in reverse order, to result
// uses a StringBuilder to reduce allocation
// that would happen with String +

private String buildReverse () {
  StringBuilder result = new StringBuilder();
  while (!charStack.empty()) {
    result.append(charStack.pop());
  }
  return result.toString();
}
```

Palindrome Code (4)

```
public boolean isPalindrome () {
  return input.equalsIgnoreCase(buildReverse());
}
```

Testing this class:

- Single character, empty string: both always palindromes
- Multiple characters, one word
- Multiple words
- Different cases (upper/lower)
- Both even- and odd-length strings

Stack Applications: Balancing Brackets

- Arithmetic expressions should balance parentheses:

  (a+b*(c/(d-e)))+(d/e)
- Not too hard if limited to parentheses only:
  - Increment depth counter on (
  - Decrement depth counter on)
  - If always ≥ 0, then balanced properly
- Problem harder if also use brackets: [] {}
  - A stack provides a good solution!

Balancing Brackets: Overview

- Start with empty stack of currently open brackets
- Process each char of an expression String
- If it is an opener, push it on the stack
- If it is a closer, check it against the top stack element
  - If stack empty, or not a matching bracket:

    not balanced, so return false
  - Otherwise, it matches: pop the opener
- If stack is empty at the end, return true
- If not empty, then some bracket unmatched: false

isBalanced Code

```
public static boolean isBalanced (String expr) {
  Stack<Character> s = new Stack<Character>();
  try {
    for (int idx = 0, len = expr.length();
             idx < len;
             idx++) {
      ... process each char of expr ...
    }
  } catch (EmptyStackException ex) {
    return false;
  }
  return s.empty();
}
```

`isBalanced` Code: Loop Body

```
// process each char of expr:
char ch = expr.charAt(idx);
if (isOpen(ch)) {
  s.push(ch);
} else if (isClose(ch)) {
  char top = s.pop();
  if (!matches(ch, top)) {
    return false;
  }
}
```

isBalanced Code: Helper Routines

```
private static final String OPEN  = “([{“;
private static final String CLOSE = “)]}”;

private static boolean isOpen (char ch) {
  return OPEN.indexOf(ch) >= 0;
}

private static boolean isClose (char ch) {
  return CLOSE.indexOf(ch) >= 0;
}

private static boolean matches (char open,
                                char close) {
  return OPEN .indexOf(open) ==
         CLOSE.indexOf(close);
}
```

Testing **`isBalanced`**

- Expressions without brackets
- Expressions with just one level
- Expressions with multiple levels, same kind
- Expressions with multiple levels, different kinds
- Expressions with same # open/close, but bad order:
  -) (
- Expressions with open/close order ok, wrong bracket:
  - ( ]
- Expressions with too many openers
- Expressions with too many closers

Implementing **`Stack`** as Extension of **`Vector`**

- Collection hierarchy includes **`java.util.Stack`**
- The **`Vector`** class offers a growable array of objects
- Elements of a **`Vector`** accessed by index
- Size can grow or shrink as needed
- *That is:* **`Vector`** is just like **`ArrayList`**

Implementing **`Stack`** as Extension of **`Vector`** (2)

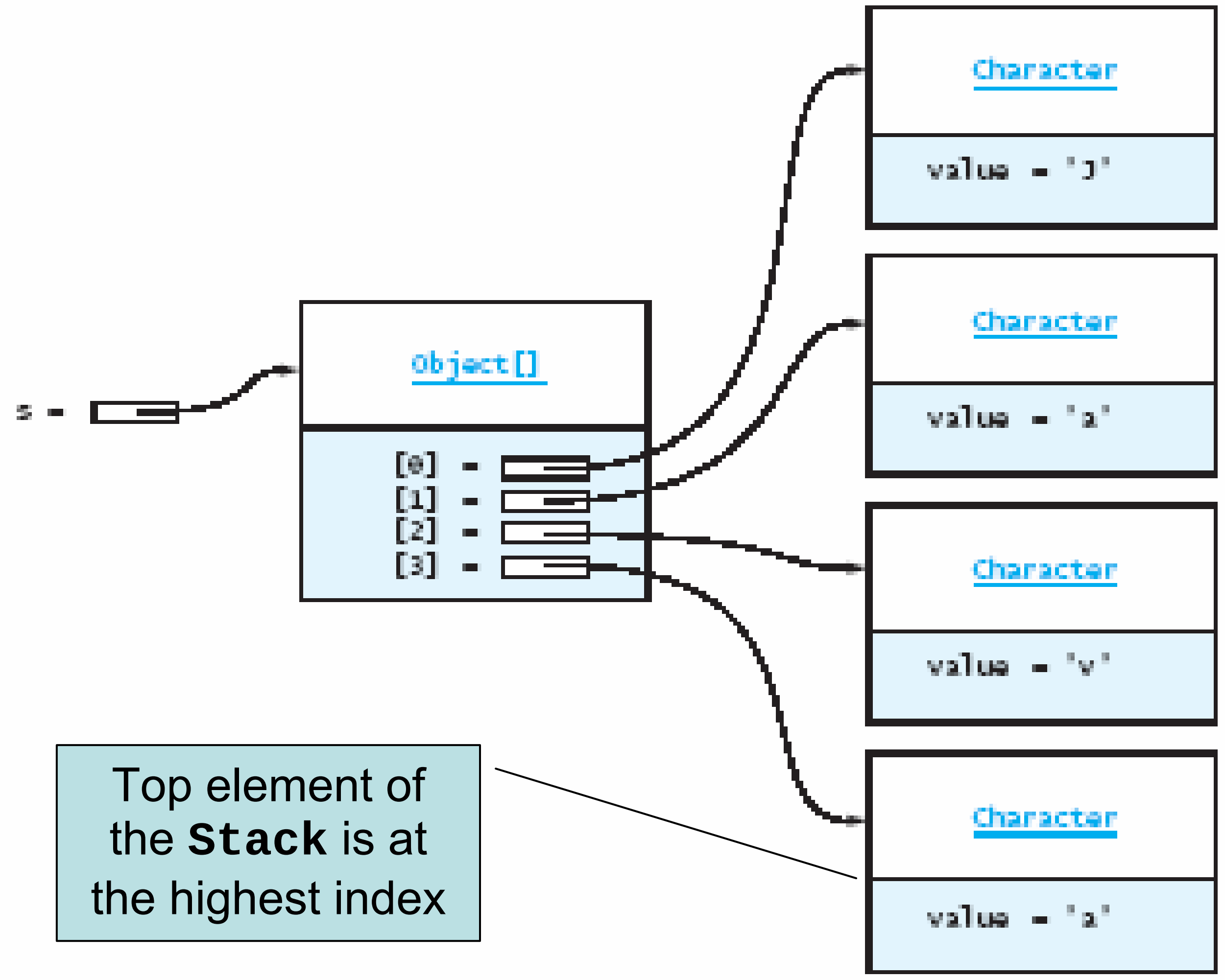


FIGURE 5.3
Characters of "Java" stored in Stack s (a Vector)

Top element of the **`Stack`** is at the highest index

Stack Code

```
public class Stack<E> extends Vector<E> {

  public E push (E e) {
    add(e);
    return e;
  }

  public E pop () throws EmptyStackException {
    try {
      return remove(size()-1);
    } catch (ArrayIndexOutOfBoundsException ex) {
      throw new EmptyStackException();
    }
  }
  ...
```

Stack Code (2)

```
public class Stack<E> extends Vector<E> {
  ...

  public E peek () throws EmptyStackException {
    try {
      return get(size()-1);
    } catch (ArrayIndexOutOfBoundsException ex) {
      throw new EmptyStackException();
    }
  }

  public boolean empty () {
    return size() == 0;
  }
}
```

Implementing **`Stack`** with a **`List`**

- Can use **`ArrayList`**, **`Vector`**, or **`LinkedList`**:
  - All implement the **`List`** interface
- Name of class illustrated in text is **`ListStack`**
- **`ListStack`** is an *adapter class*
  - Adapts methods of another class to ...
  - Interface its clients expect by ...
  - Giving different names to essentially the same operations

`ListStack` Code

```
public class ListStack<E>
    implements StackInterface<E> {

  private List<E> list;

  public ListStack () {
    list = new ArrayList<E>();
    // or new Vector<E> or new LinkedList<E>
  }

  public E push (E e) {
    list.add(e);
    return e;
  }
  ...
```

ListStack Code (2)

```
public E peek () {
  if (empty())
    throw new EmptyStackException();
  return list.get(list.size()-1);
}

public E pop () {
  if (empty())
    throw new EmptyStackException()
  return list.remove(list.size()-1);
}

public boolean empty () {
  return list.size() == 0;
}
```

Implementing **`Stack`** Using an Array

- Must allocate array with some default capacity
- Need to keep track of the top of the stack
- Have no size method, so must track size
- Similar to growable **`PhoneDirectory`**

Implementing **`Stack`** Using an Array (2)

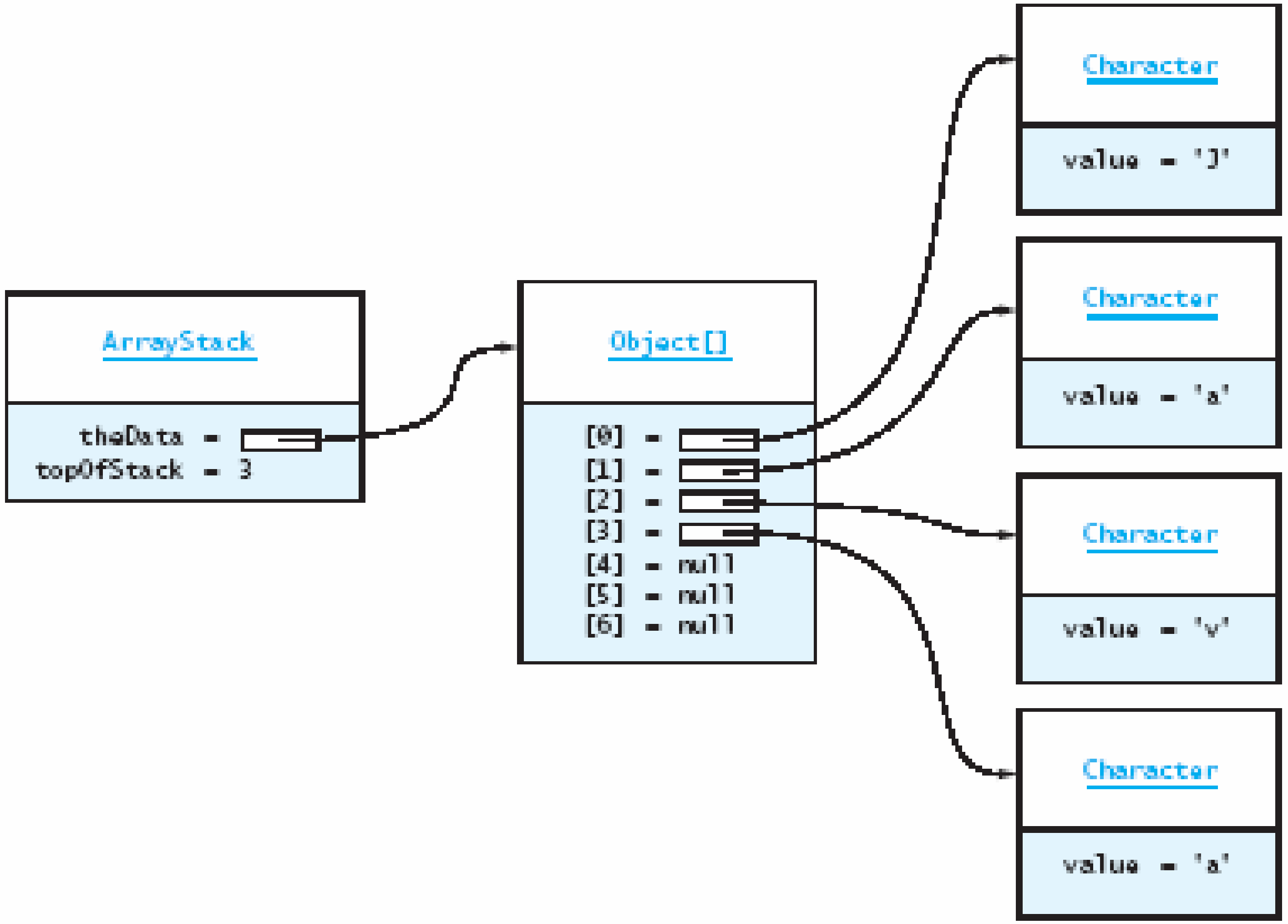


FIGURE 5.4
Stack of Character Objects in an Array

ArrayStack Code

```
public class ArrayStack<E>
    implements StackInterface<E> {

  private static final int INITIAL_CAPACITY = 10;

  private E[] data =
    (E[]) new Object[INITIAL_CAPACITY];

  private int top = -1;

  public ArrayStack () { }

  public boolean empty () { return top < 0; }
  ...
}
```

ArrayStack Code (2)

```
public E push (E e) {
  if (++top == data.length) reallocate();
  return data[top] = e;
}

public E pop () {
  if (empty()) throw new EmptyStackException();
  return data[top--];
}

public E peek () {
  if (empty()) throw new EmptyStackException();
  return data[top];
}
```

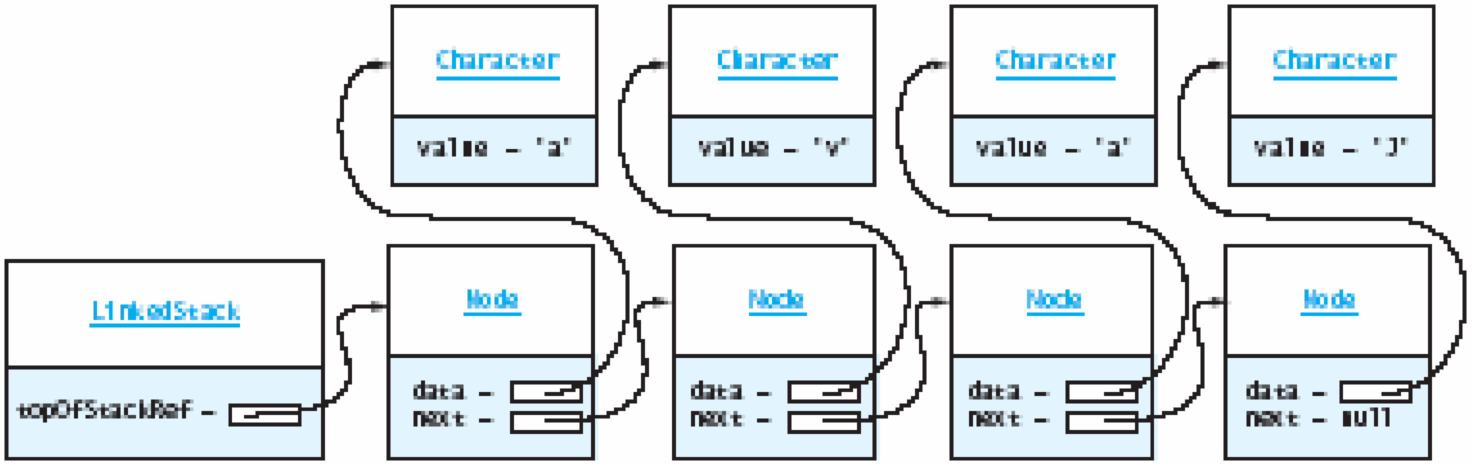
ArrayStack Code (3)

```
private reallocate () {
  E[] newData = (E[]) new Object[data.length*2];
  System.arraycopy(data, 0, newData, 0,
                   data.length);
  data = newData;
}
```

Implementing **`Stack`** as a Linked Structure

- We can implement **`Stack`** using a linked list:

FIGURE 5.5
Stack of Character Objects in a Linked List



`LinkedStack` Code

```
public class LinkedStack<E>
    implements StackInterface<E> {

  private Node<E> top = null;

  public LinkedStack () { }

  public boolean empty () { return top == null; }

  public E push (E e) {
    top = new Node<E>(e, top);
    return e;
  }
  ...
}
```

LinkedStack Code (2)

```
public E pop () {
  if (empty()) throw new EmptyStackException();
  E result = top.data;
  top = top.next;
  return result;
}

public E peek () {
  if (empty()) throw new EmptyStackException();
  return top.data;
}
```

LinkedStack Code (3)

```
private static class Node<E> {

  private E data;
  private Node<E> next;

  Node (E data, Node<E> next) {
    this.data = data;
    this.next = next;
  }

}
```

Comparison of **`Stack`** Implementations

- **`Vector`** is a poor choice: exposes **`Vector`** methods
- Likewise, *extending* other **`List`** classes exposes
- Using a **`List`** as a component avoid exposing
  - **`LinkedStack`** operations are O(1)
    - But Node objects add much space overhead
  - **`ArrayList`** component is perhaps easiest
    - Still *some* space overhead (**`Stack+ArrayList`**)
  - Array as a component best in space and time
    - But somewhat harder to implement

Additional Stack Applications: Evaluating Arithmetic Expressions

- Expressions normally written in infix form
  - Binary operators appear between their operands:

    a + b   c * d   e * f - g
- A computer normally scans in input order
  - One must always compute operand values first
  - So easier to evaluate in *postfix* form:

    a b +   c d *   e f * g -

Postfix Expression Evaluation Examples

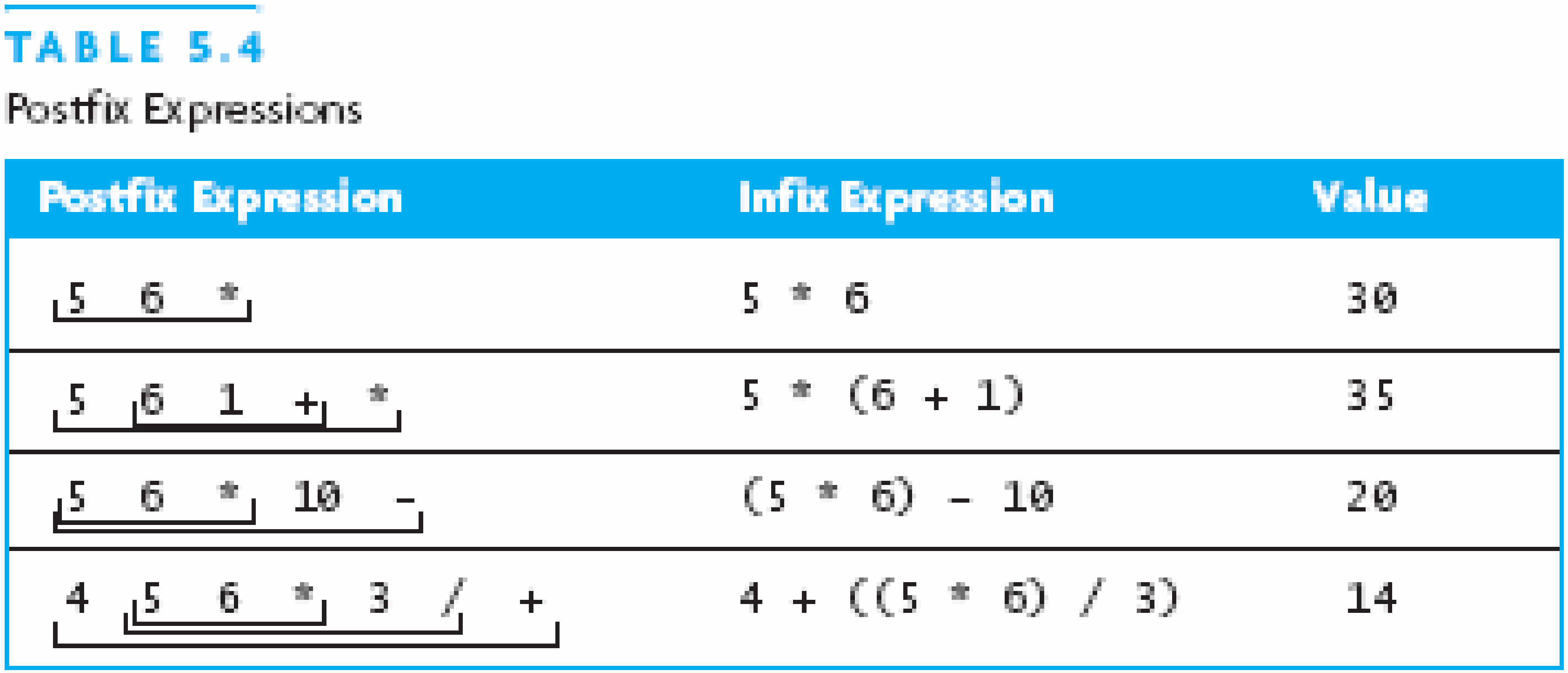


TABLE 5.4

Postfix Expressions

Postfix Expression	Infix Expression	Value
5 6 *	5 * 6	30
5 6 1 + *	5 * (6 + 1)	35
5 6 * 10 -	(5 * 6) - 10	20
4 5 6 * 3 / +	4 + ((5 * 6) / 3)	14

Advantages of Postfix Form

- No need to use parentheses for grouping
- No need to use precedence rules:
  - Usual meaning of a + b * c is a + (b * c)
  - * *takes precedence over* + ... or ...
  - * has *higher precedence* than +
  - Postfix: a b c * +
  - For (a+b)*c, postfix is: a b + c *

Program: Evaluating Postfix Expressions

- Input: **`String`** with integers and operators separated by spaces (for **`StringTokenizer`**)
- Operators: + - * /   (all binary, usual meaning)
- Strategy:
  - Use stack for input integers, evaluated operands
  - Operator pops operands, computes, pushes result
  - At end, last item on stack is the answer

Program: Evaluating Postfix Expressions (2)

```
1.  Create empty Stack<Integer>
2.  For each token of the expression String do:
3.      If the token is an Integer (starts with a digit)
4.          Push the token’s value on the stack
5.      Else if the token is an operator
6.          Pop the right (second) operand off the stack
7.          Pop the left (first) operand off the stack
8.          Evaluate the operation
9.          Push the result onto the stack
10. Pop the stack and return the result
```

Program: Evaluating Postfix Expressions (3)

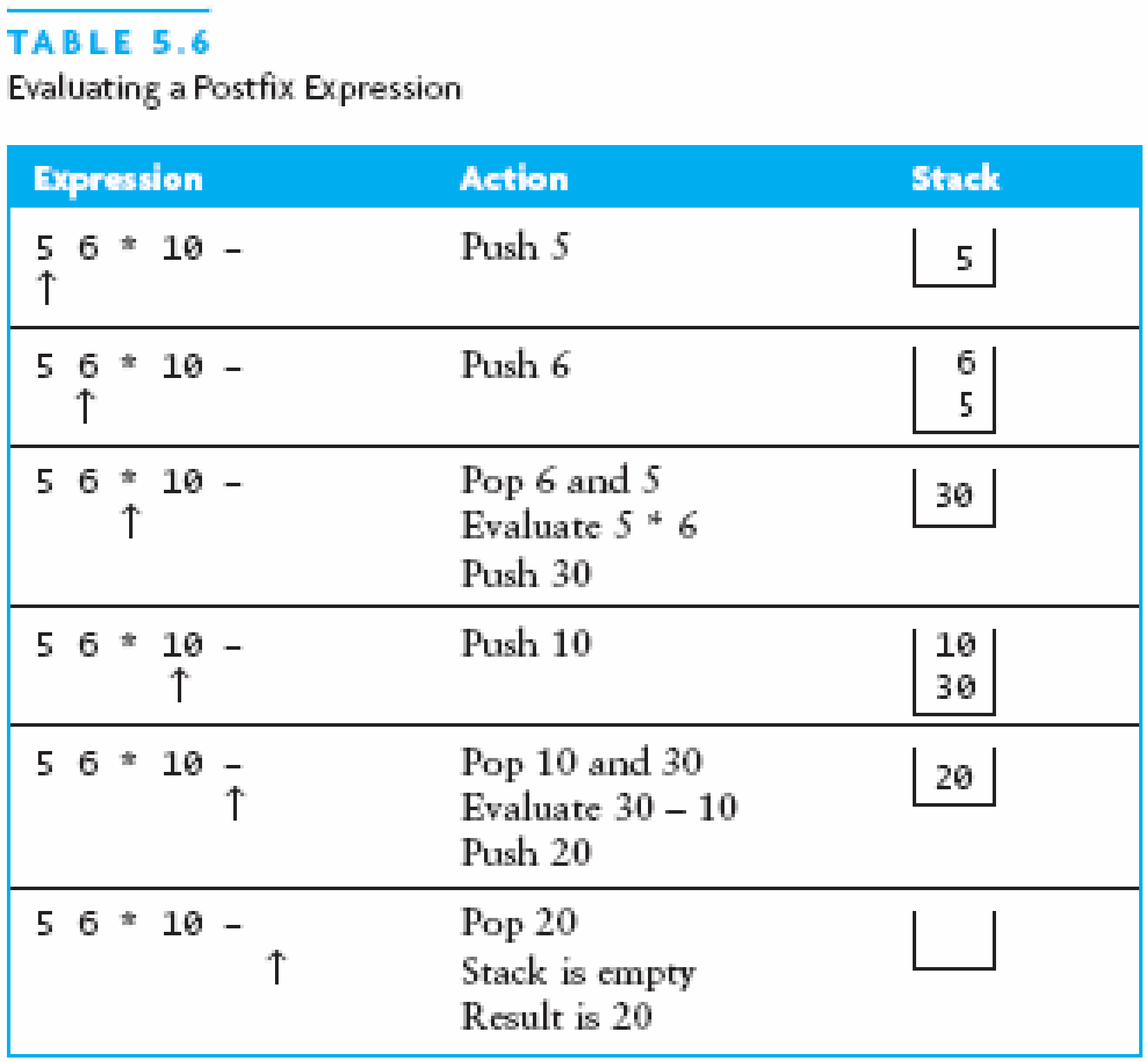


TABLE 5.6

Evaluating a Postfix Expression

Expression	Action	Stack

`5 6 * 10 -`
`↑`
	Push 5	5

`5 6 * 10 -`
`  ↑`
	Push 6	6
		5

`5 6 * 10 -`
`    ↑`
	Pop 6 and 5
	Evaluate 5 * 6
	Push 30
		30

`5 6 * 10 -`
`      ↑`
	Push 10	10
		30

`5 6 * 10 -`
`         ↑`
	Pop 10 and 30
	Evaluate 30 – 10
	Push 20
		20

`5 6 * 10 -`
`           ↑`
	Pop 20
	Stack is empty
	Result is 20	

Program: Evaluating Postfix Expressions (4)

```
import java.util.*;
public class PostfixEval {
  public static class SyntaxErrorException
      extends Exception {
    SyntaxErrorException (String msg) {
      super(msg);
    }
  }

  private final static String
    OPERATORS = “+-*/”;
```

Program: Evaluating Postfix Expressions (5)

```
private Stack<Integer> operandStack =
  new Stack<Integer>();

private int evalOp (char op) {
  int rhs = operandStack.pop();
  int lhs = operandStack.pop();
  int result = 0;
  switch (op) {
    case ‘+’: result = lhs + rhs; break;
    case ‘-’: result = lhs - rhs; break;
    case ‘*’: result = lhs * rhs; break;
    case ‘/’: result = lhs / rhs; break;
  }
  return result;
}
```

Program: Evaluating Postfix Expressions (6)

```
private boolean isOperator (char ch) {
  return OPERATORS.indexOf(ch) >= 0;
}

public int eval (String expr)
    throws SyntaxErrorException {
  operandStack.reset();
  StringTokenizer tokens =
    new StringTokenizer(expr);
  ...;
}
```

Program: Evaluating Postfix Expressions (7)

```
try {
  while (tokens.hasMoreTokens()) {
    String token = tokens.nextToken();
    ... loop body ...
  }
  ... after loop ...
} catch (EmptyStackException exc) {
  throw new SyntaxErrorException(
    “Syntax Error: The stack is empty”);
}
```

Program: Evaluating Postfix Expressions (8)

```
// loop body: work done for each token
if (Character.isDigit(token.charAt(0))) {
  int value = Integer.parseInt(token);
  operandStack.push(value);

} else if (isOperator(token.charAt(0))) {
  int result = evalOp(token.charAt(0));
  operandStack.push(result);

} else {
  throw new SyntaxErrorException(
    “Invalid character encountered”);
}
```

Program: Evaluating Postfix Expressions (9)

```
// Work after loop completes
int answer = operandStack.pop();
if (operandStack.empty()) {
  return answer;
} else {
  throw new SyntaxErrorException(
    “Syntax Error: Stack not empty”);
}
```

Converting Infix to Postfix

- Need to handle precedence
- Need to handle parentheses
  - First implementation handles only precedence

- Input: Infix with variables, integers, operators
  - (No parentheses for now)
- Output: Postfix form of same expression

Converting Infix to Postfix (2)

- Analysis:
  - *Operands* are in same order in infix and postfix
  - *Operators* occur later in postfix
- Strategy:
  - Send operands straight to output
  - Send higher precedence operators first
  - If same precedence, send in left to right order
  - Hold pending operators on a stack

Converting from Infix to Postfix: Example

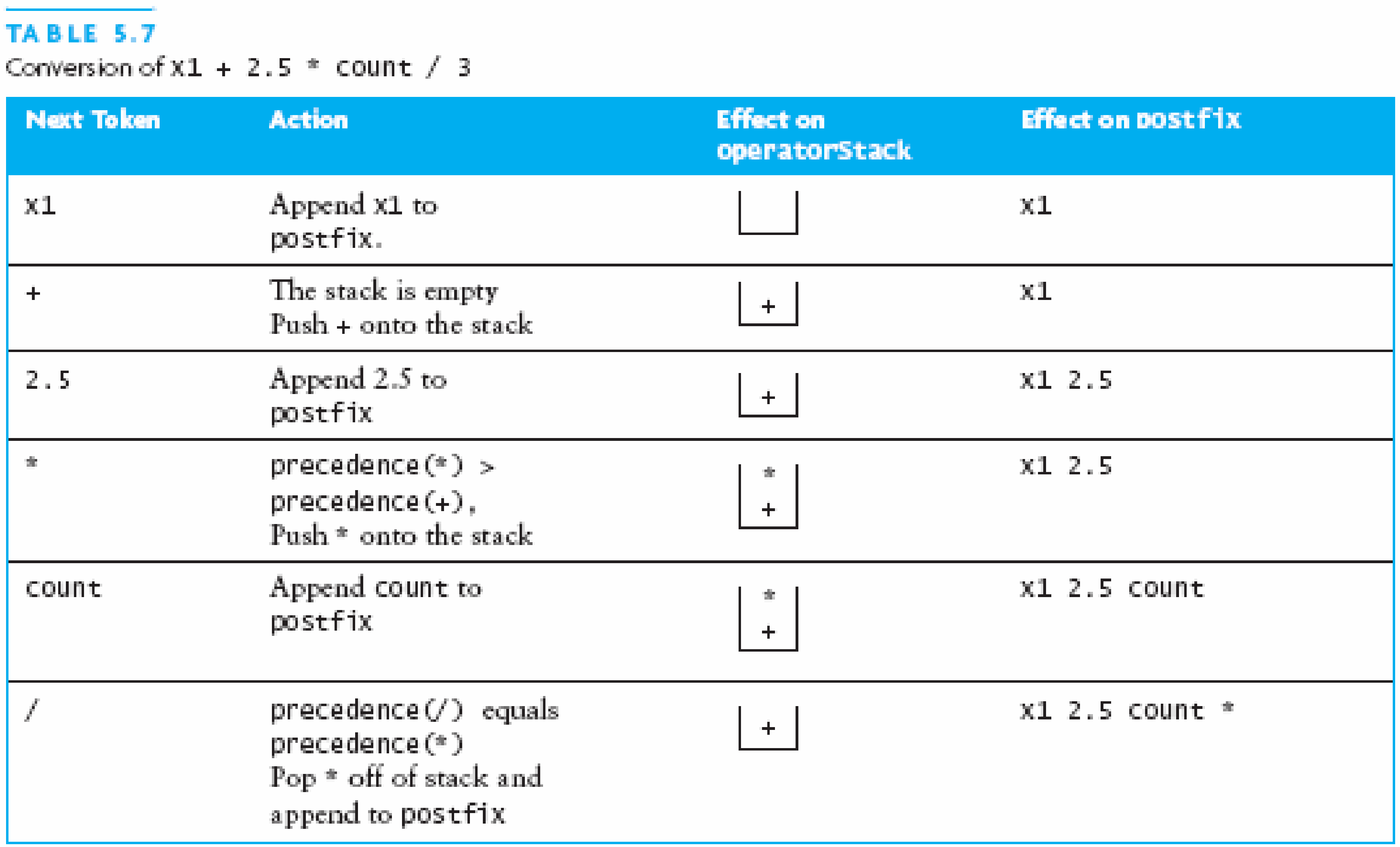


TABLE 5.7
Conversion of `x1 + 2.5 * count / 3`

Next Token	Action	Effect on operatorStack	Effect on postfix
x1	Append x1 to postfix.		x1
+	The stack is empty Push + onto the stack	+	x1
2.5	Append 2.5 to postfix	+	x1 2.5
*	precedence(*) > precedence(+), Push * onto the stack	* +	x1 2.5
count	Append count to postfix	* +	x1 2.5 count
/	precedence(/) equals precedence(*) Pop * off of stack and append to postfix	+	x1 2.5 count *

Converting from Infix to Postfix: Example (2)

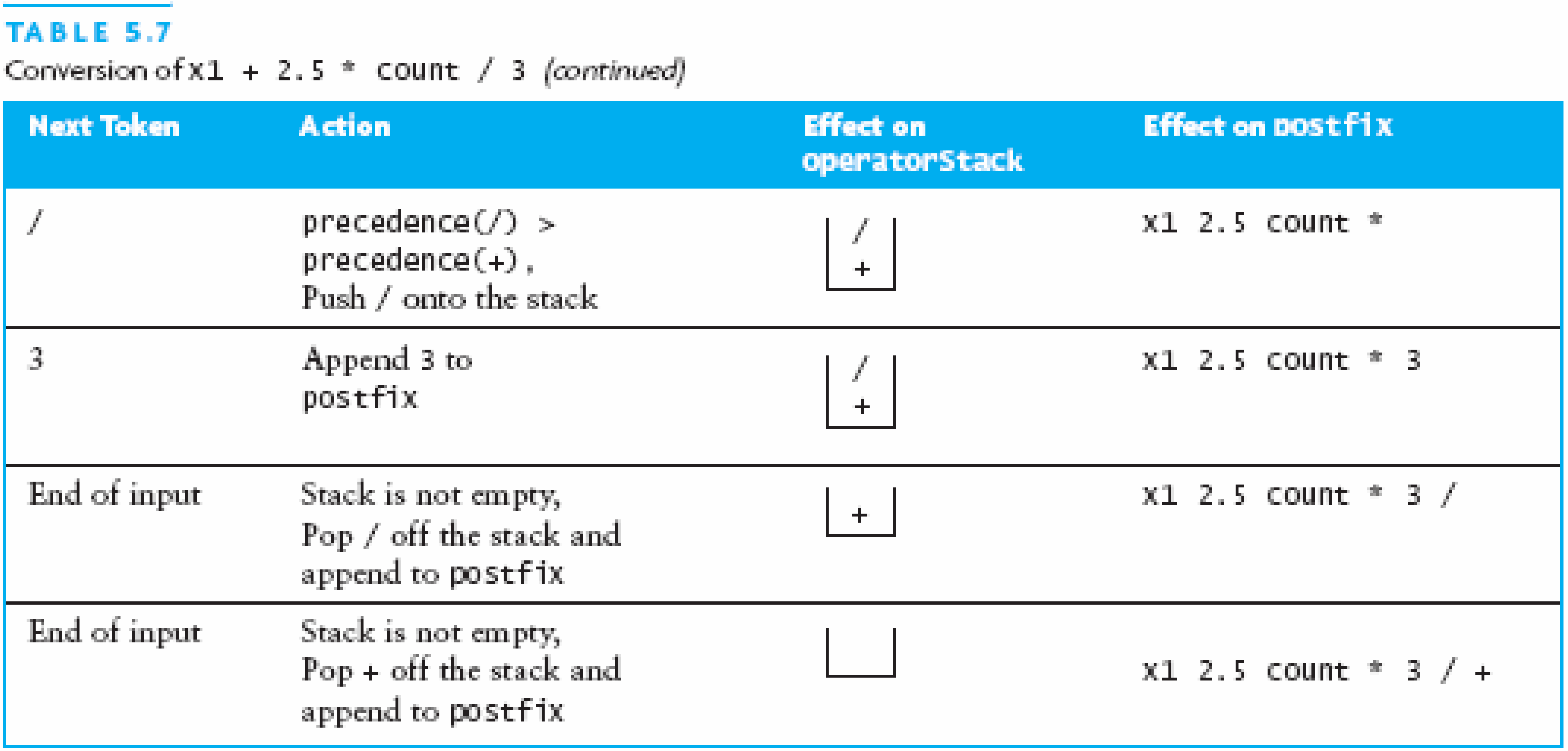


TABLE 5.7
Conversion of `x1 + 2.5 * count / 3` *(continued)*

Next Token	Action	Effect on `operatorStack`	Effect on `postfix`
`/`	`precedence(/) > precedence(+)`, Push / onto the stack	/ +	`x1 2.5 count *`
3	Append 3 to `postfix`	/ +	`x1 2.5 count * 3`
End of input	Stack is not empty, Pop / off the stack and append to `postfix`	+	`x1 2.5 count * 3 /`
End of input	Stack is not empty, Pop + off the stack and append to `postfix`		`x1 2.5 count * 3 / +`

Converting Infix to Postfix: **`convert`**

```
1.  Set postfix to an empty StringBuilder
2.  Set operator stack to an empty stack
3.  while more tokens in infix string
4.      Get the next token
5.      if the token is an operand
6.          Append the token to postfix
7.      else if the token is an operator
8.          Call processOperator to handle it
9.      else
10.         Indicate a syntax error
11. Pop remaining operators and add them to postfix
```

Converting Infix to Postfix: **`processOperator`**

```
1. while the stack is not empty & prec(new) ≤ prec(top)
2.     pop top off stack and append to postfix
3. push the new operator
```

Converting Infix to Postfix

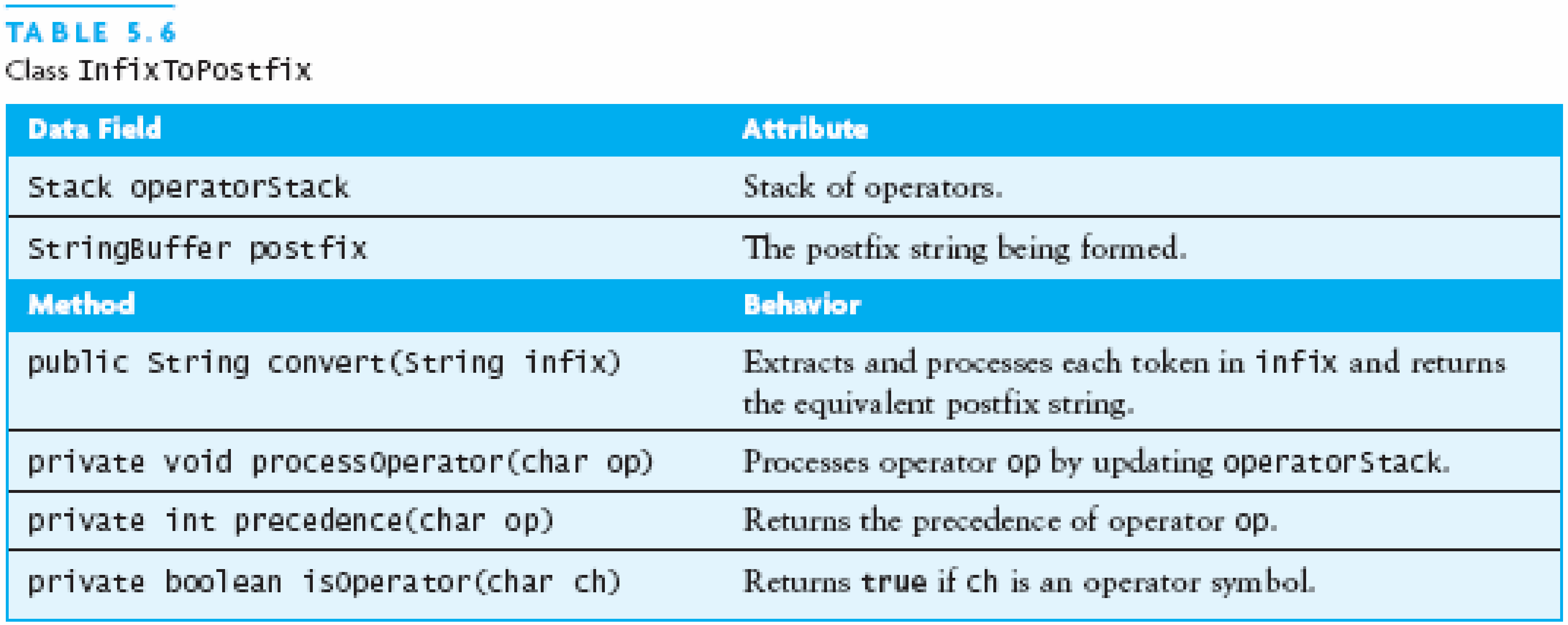


TABLE 5.6
Class `InfixToPostfix`

Data Field	Attribute
`Stack operatorStack`	Stack of operators.
`StringBuffer postfix`	The postfix string being formed.
Method	**Behavior**
`public String convert(String infix)`	Extracts and processes each token in `infix` and returns the equivalent postfix string.
`private void processOperator(char op)`	Processes operator `op` by updating `operatorStack`.
`private int precedence(char op)`	Returns the precedence of operator `op`.
`private boolean isOperator(char ch)`	Returns `true` if `ch` is an operator symbol.

Infix toPostfix: Code

```
import java.util.*;
public class InfixToPostfix {
  private Stack<Character> opStack;
  private static final String OPERATORS = “+-*/”;
  private static final int[] PRECEDENCE =
    {1, 1, 2, 2};
  private StringBuilder postfix;
  private boolean isOperator (char ch) {
    return OPERATORS.indexOf(ch) >= 0;
  }
  private int precedence (Character ch) {
    return (ch == null) ? 0 :
           PRECEDENCE[OPERATORS.indexOf(ch)];
  }
```

Infix toPostfix: Code (2)

```
public String convert (String infix)
    throws SyntaxErrorException {
  opStack = new Stack<Character>();
  postfix = new StringBuilder();
  StringTokenizer tokens =
    new StringTokenizer(infix);
  try {
    while (tokens.hasMoreTokens()) { loop body }
    ... after loop ...
  } catch (EmptyStackException exc) {
    throw new SyntaxErrorException(...);
  }
}
```

Infix toPostfix: Code (3)

```
// loop body
String token = tokens.nextToken();
Character first = token.charAt(0);
if (Character.isJavaIdentifierStart(first) ||
    Character.isDigit(first)) {
  postfix.append(token);
  postfix.append(‘ ‘);

} else if (isOperator(first)) {
  processOperator(first)

} else {
  throw new SyntaxErrorException(...);
}
```

Infix toPostfix: Code (4)

```
// after loop
while (!opStack.empty()) {
  char op = opStack.pop();
  postfix.append(op);
  postfix.append(‘ ‘);
}
return postfix.toString();
```

Infix toPostfix: Code (5)

```
private Character top () {
  return opStack.empty() ? null : opStack.peek();
}

private void processOperator (Character op) {
  while (precedence(op) <= precedence(top())) {
    postfix.append(opStack.pop());
    postfix.append(‘ ‘);
  }
  opStack.push(op);
}
```

Infix toPostfix: Handling Parentheses

```
private static final String OPERATORS = “+-*/()”;
  private static final int[] PRECEDENCE =
    {2, 2, 3, 3, 1, 1};

private int precedence (Character ch) {
  return (ch == null) ? 0 :
         PRECEDENCE[OPERATORS.indexOf(ch)];
}
```

Infix toPostfix: Handling Parentheses (2)

```
private void processOperator (Character op) {
  if (op.charValue() != ‘(‘) {
    while (precedence(op) <= precedence(top())) {
      char top = opStack.pop();
      if (top == ‘(‘) break;
      postfix.append(top);
      postfix.append(‘ ‘);
    }
  }
  if (op.charValue() != ‘)’) {
    opStack.push(op);
  }
}
```

Infix toPostfix: Handling Parentheses (3)

```
// in convert, after token loop:
while (!opStack.empty()) {
  char op = opStack.pop();
  if (op == ‘(‘)
    throw new SyntaxErrorException(...);
    // because ( is unmatched
  postfix.append(op);
  postfix.append(‘ ‘);
}
```

Infix to Postfix: Making It Even Cleaner

- Can push a special “open bracket” first
  - Stack is never empty
  - This will not pop off: we give it very low precedence
- Push the same special bracket at the end:
  - Will cause other operators to pop off
- Make bracket/parenthesis handling less special
  - Add notion of *left* and *right* *precedence*
  - Left value is for already scanned operator (on left)
  - Right value is for not yet pushed operator (on right)

Infix to Postfix: Making It Even Cleaner (2)

Operator	+	*	(	)
Left Precedence	3	4	1	---
Right Precedence	3	4	5	2

```
while (leftPred(top) >= rightPred(newer))
  pop(top);
if (top == ‘(‘)
  pop(top);
else
  push(newer);
```