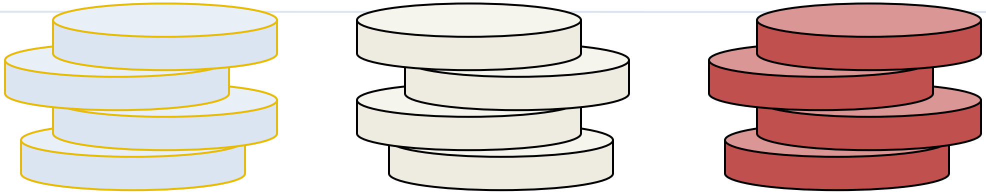
STACKS



Presentation for use with the textbook
1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
2. Data Structures Abstraction and Design Using Java, 2nd Edition by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

ABSTRACT DATA TYPES (ADTS)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
    - Data stored
    - Operations on the data
    - Error conditions associated with operations
- Example: ADT modeling a simple stock trading system
    - The data stored are buy/sell orders
    - The operations supported are
        - order buy(stock, shares, price)
        - order sell(stock, shares, price)
        - void cancel(order)
    - Error conditions:
        - Buy/sell a nonexistent stock
        - Cancel a nonexistent order

STACK ABSTRACT DATA TYPE

- A stack is one of the most commonly used data structures in computer science
- A stack can be compared to a Pez dispenser
  - Only the top item can be accessed
  - You can extract only one item at a time
- The top element in the stack is the last added to the stack (most recently)
- The stack's storage policy is *Last-In*, *First-Out*, or *LIFO*

- Main stack operations:
  - **push(object):** inserts an element
  - **object pop():** removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

A STACK OF STRINGS

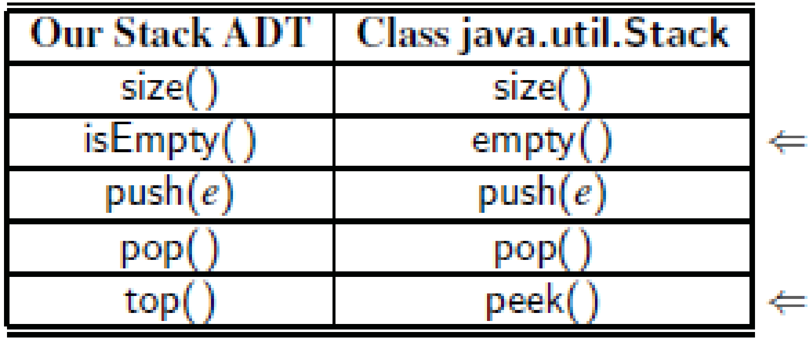
(a) (b) (c)

- “Rich” is the oldest element on the stack and “Jonathan” is the youngest (Figure a)
- `String last = names.top();` stores a reference to “Jonathan” in `last`
- `String temp = names.pop();` removes “Jonathan” and stores a reference to it in `temp` (Figure b)
- `names.push(“Philip”);` pushes “Philip” onto the stack (Figure c)

STACK INTERFACE IN JAVA

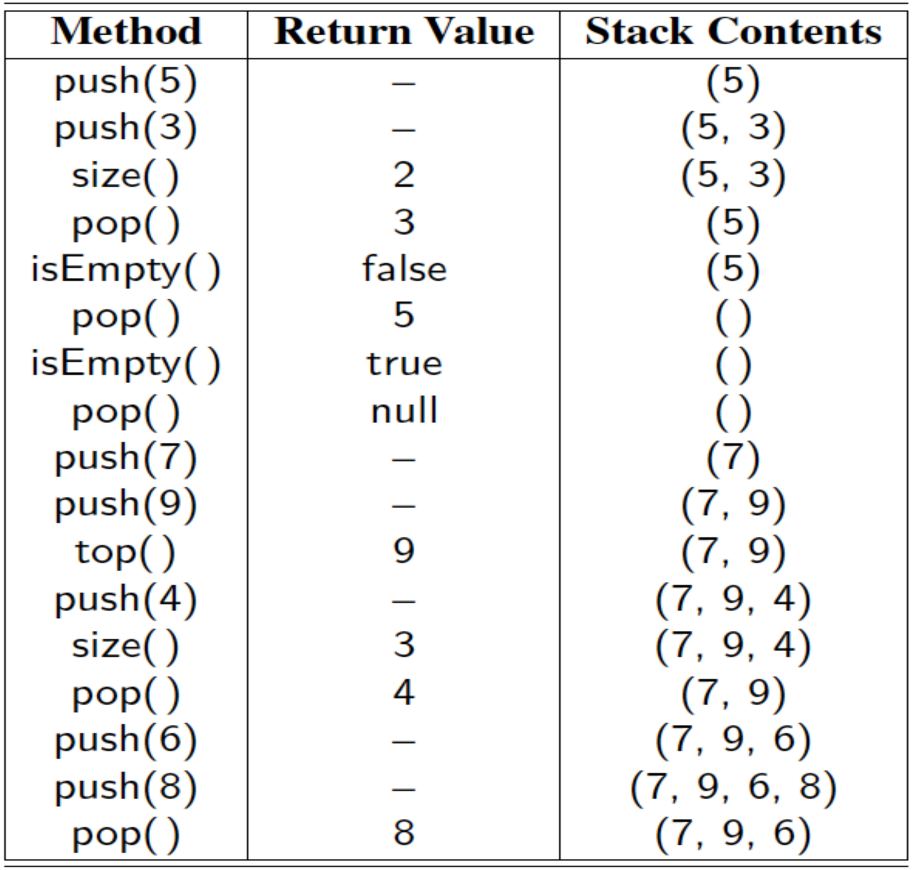
- Java interface corresponding to our Stack ADT
- Assumes null is returned from top() and pop() when stack is empty
- Different from the built-in Java class java.util.Stack

```
public interface Stack<E> {
   int size();
   boolean isEmpty();
   E top();
   void push(E element);
   E pop();
}
```



Our Stack ADT	Class java.util.Stack	
size()	size()	
isEmpty()	empty()	⇐
push(e)	push(e)	
pop()	pop()	
top()	peek()	⇐

ANOTHER EXAMPLE



Method	**Return Value**	**Stack Contents**
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

EXCEPTIONS VS. RETURNING NULL

- Attempting the execution of an operation of an ADT may sometimes cause an error condition
- Java supports a general abstraction for errors, called **exception**
- An exception is said to be "thrown" by an operation that cannot be properly executed
- In our Stack ADT, we do not use exceptions
- Instead, we allow operations pop and top to be performed even if the stack is empty
- For an empty stack, pop and top simply return null

APPLICATIONS OF STACKS

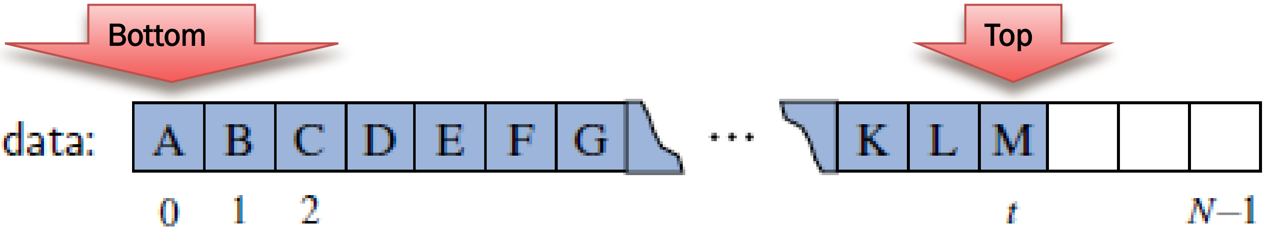
- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

ARRAY-BASED STACK

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable (t) keeps track of the index of the top element

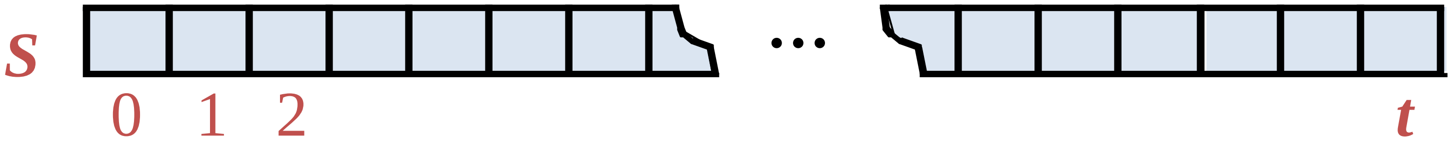
```
Algorithm size()
    return $t + 1$

Algorithm pop()
    if isEmpty() then
        return null
    else
        $t \leftarrow t - 1$
        return $S[t + 1]$
```



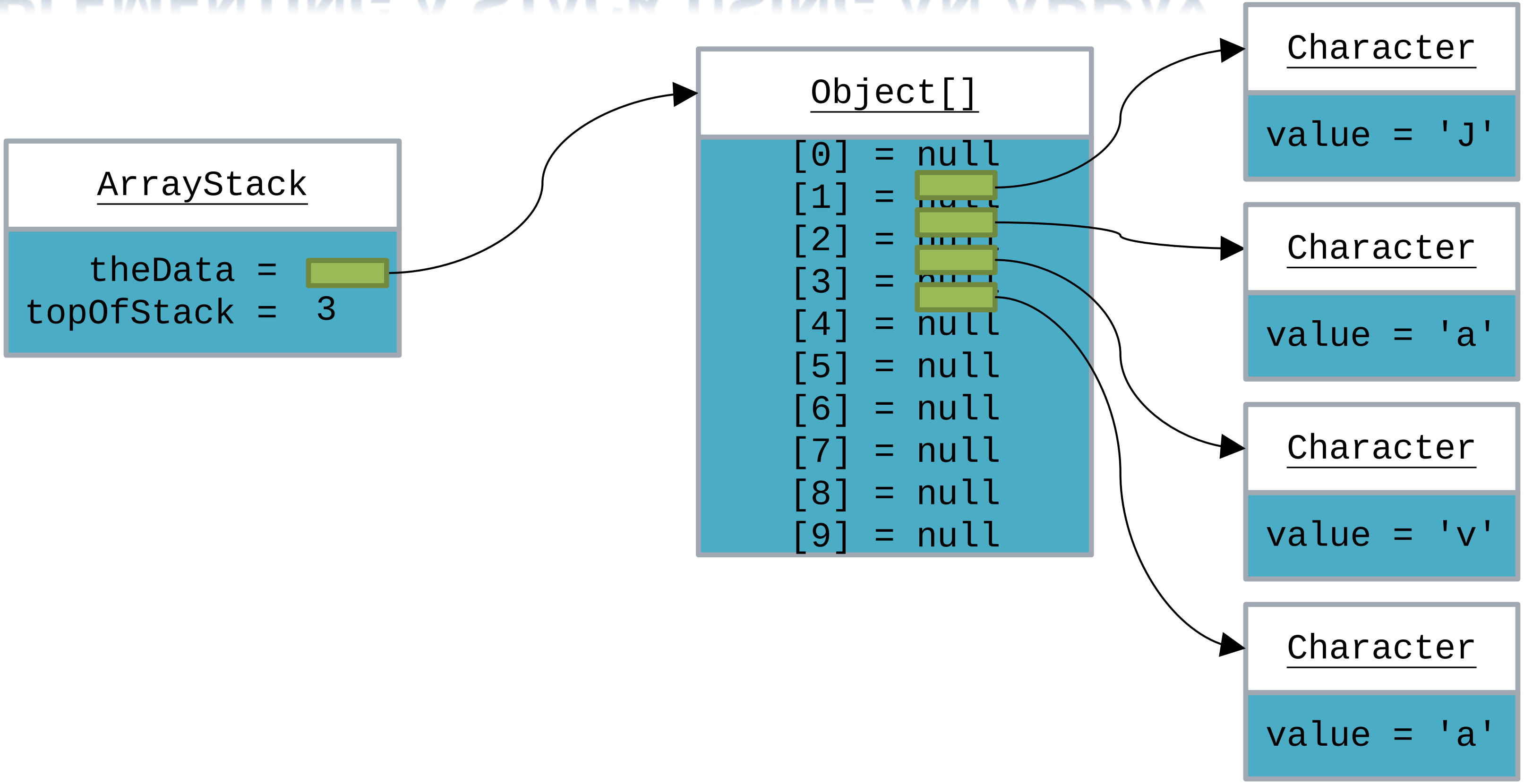
ARRAY-BASED STACK (CONT.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

```
Algorithm push(o)
    if $t = S.length - 1$ then
        throw IllegalStateException
    else
        $t \leftarrow t + 1$
        $S[t] \leftarrow o$
```



IMPLEMENTING A STACK USING AN ARRAY



ArrayStack

theData =
topOfStack = 3

Object[]

[0] = null
[1] = null
[2] = null
[3] = null
[4] = null
[5] = null
[6] = null
[7] = null
[8] = null
[9] = null

Character

value = 'J'

Character

value = 'a'

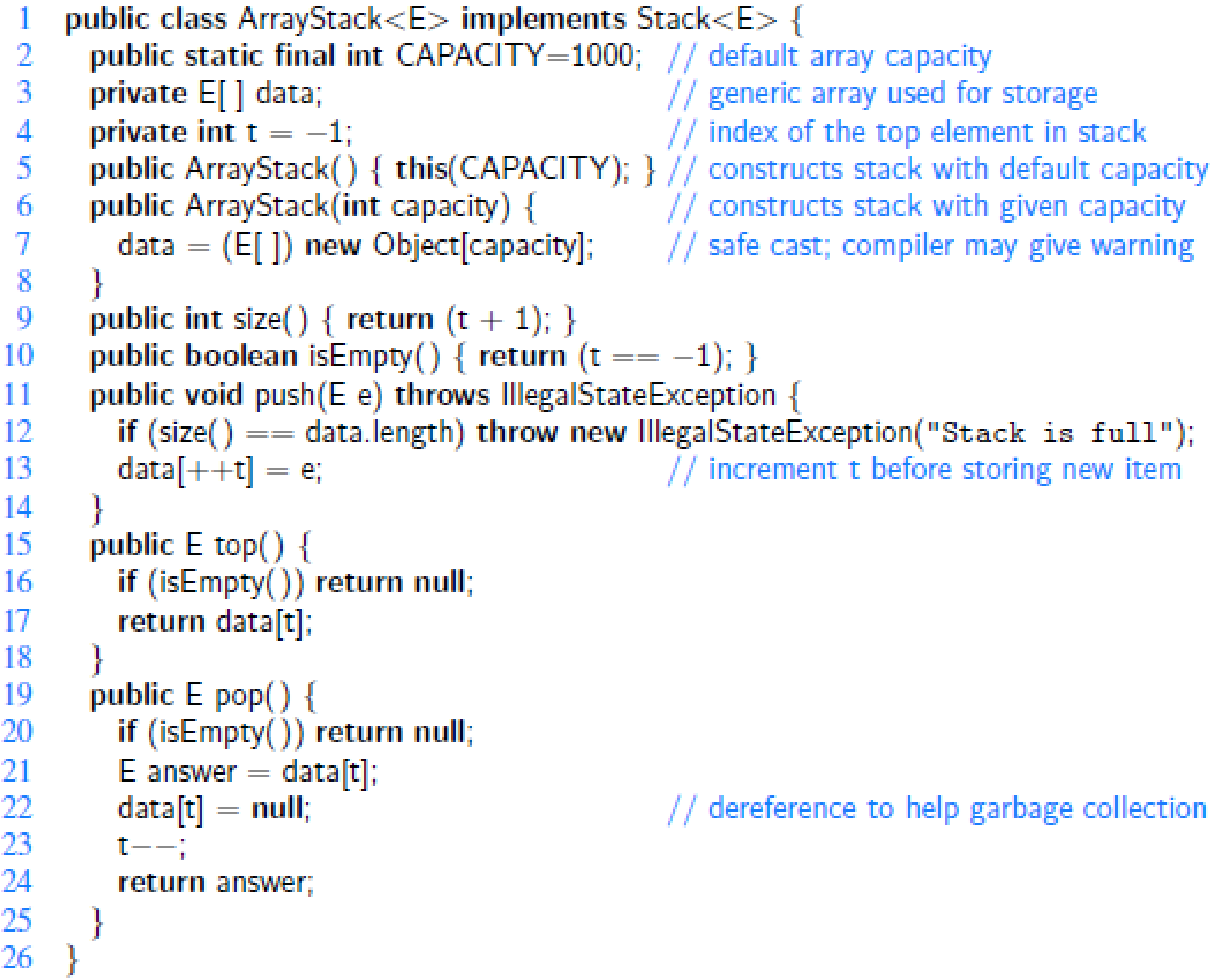
Character

value = 'v'

Character

value = 'a'

ARRAY-BASED STACK IN JAVA



```
public class ArrayStack<E> implements Stack<E> {
  public static final int CAPACITY=1000;  // default array capacity
  private E[ ] data;                       // generic array used for storage
  private int t = -1;                      // index of the top element in stack
  public ArrayStack( ) { this(CAPACITY); } // constructs stack with default capacity
  public ArrayStack(int capacity) {        // constructs stack with given capacity
    data = (E[ ]) new Object[capacity];    // safe cast; compiler may give warning
  }
  public int size( ) { return (t + 1); }
  public boolean isEmpty( ) { return (t == -1); }
  public void push(E e) throws IllegalStateException {
    if (size( ) == data.length) throw new IllegalStateException("Stack is full");
    data[++t] = e;                         // increment t before storing new item
  }
  public E top( ) {
    if (isEmpty( )) return null;
    return data[t];
  }
  public E pop( ) {
    if (isEmpty( )) return null;
    E answer = data[t];
    data[t] = null;                        // dereference to help garbage collection
    t--;
    return answer;
  }
}
```

ARRAY-BASED STACK IN JAVA

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

   // holds the stack elements
   private E[ ] S;

   // index to top element
   private int top = -1;

   // constructor
   public ArrayStack(int capacity) {
       S = (E[ ]) new Object[capacity]);
    }
```

```
    public E pop() {
      if isEmpty()
             return null;
       E temp = S[top];
       // facilitate garbage collection:
       S[top] = null;
       top = top – 1;
       return temp;
    }

 …  (other methods of Stack interface)
```

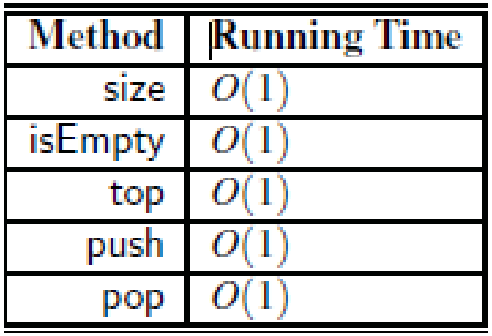
EXAMPLE USE IN JAVA

```
public class Tester {

   // … other methods
   public intReverse(Integer a[]) {
       Stack<Integer> s;
       s = new ArrayStack<Integer>();

       … (code to reverse array a) …
   }
```

```
public floatReverse(Float f[]) {
    Stack<Float> s;
    s = new ArrayStack<Float>();

    … (code to reverse array f) …
}
```

PERFORMANCE & LIMITATIONS OF ARRAY-BASED STACK

- Performance
  - Let *n* be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$



Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

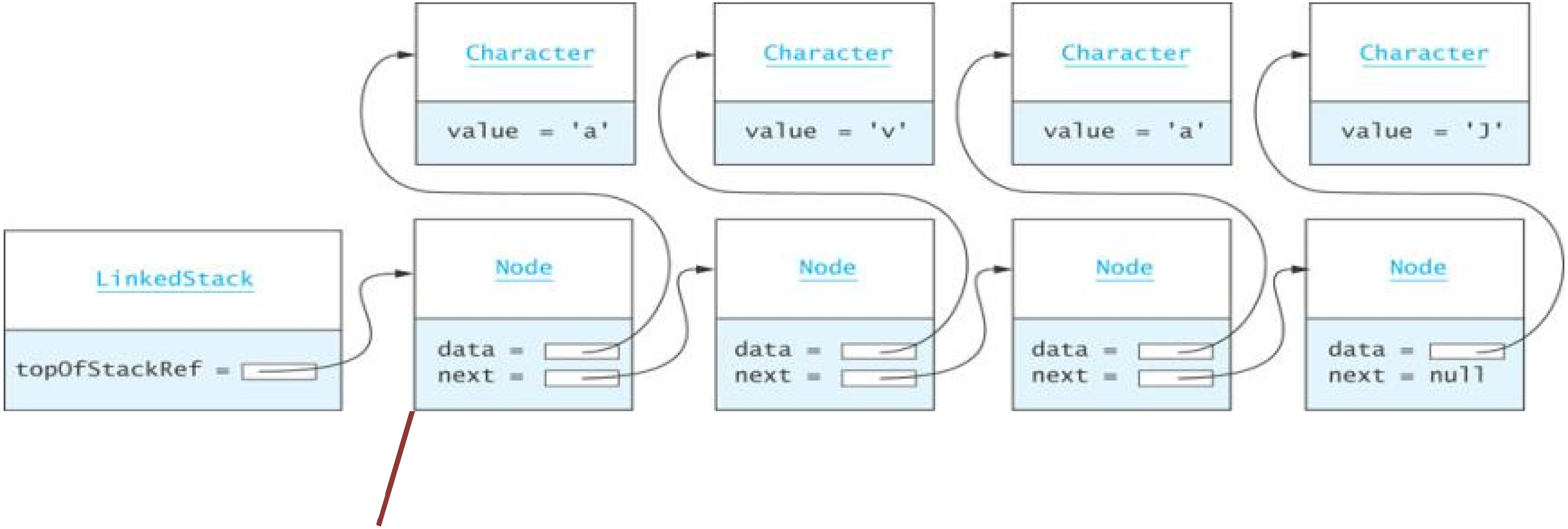
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed (fixed size array)
  - Trying to push a new element into a full stack causes an implementation-specific exception

IMPLEMENTING A STACK WITH A SINGLY LINKED LIST

- The linked-list approach has memory usage that is always proportional to the number of actual elements currently in the stack, and without an arbitrary capacity limit
- Q: What the best choice for the top of the stack: the front or back of the list?
  - With the top of the stack stored at the front of the list, all methods execute in constant time.

IMPLEMENTING A STACK AS A LINKED DATA STRUCTURE

- We can also implement a stack using a linked list of nodes



when the list is empty, pop returns null

THE *ADAPTER* DESIGN PATTERN

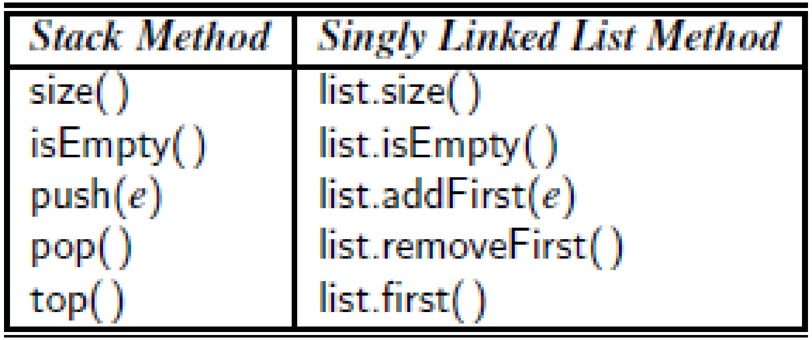
- We want to effectively modify an existing class so that its methods match those of a related, but different, class or interface.

- Define a new class in such a way that it contains an instance of the existing class as a hidden field, and then to implement each method of the new class using methods of this hidden instance variable.

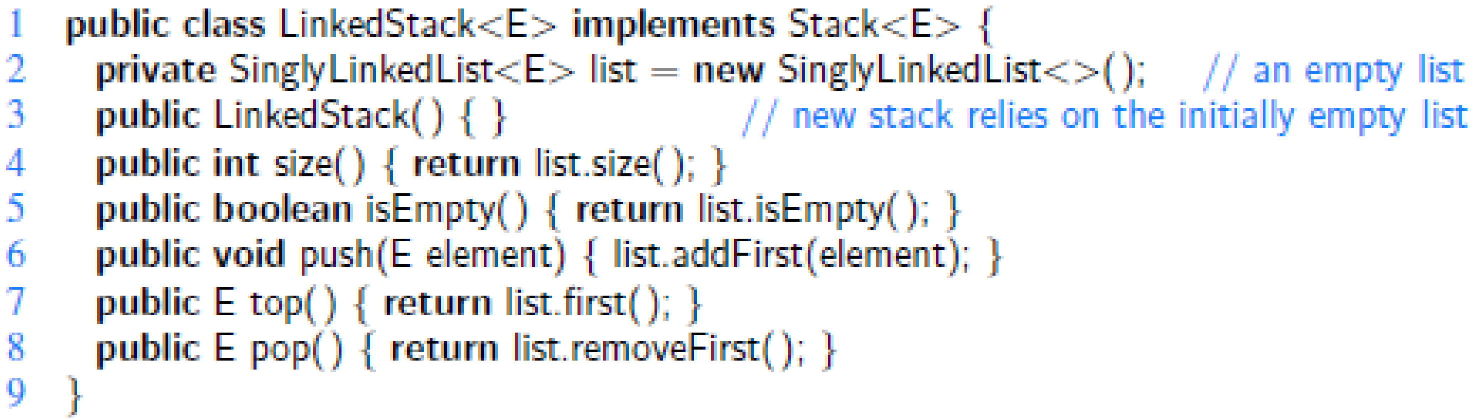
ADAPTING SINGLE LINKED LIST ON STACK ADT

- We will adapt SinglyLinkedList class of Section 3.2.1 to define a new LinkedStack class

SinglyLinkedList is named *list* as a private field, and uses the following correspondences:



Stack Method	*Singly Linked List Method*
size( )	list.size( )
isEmpty( )	list.isEmpty( )
push(e)	list.addFirst(e)
pop( )	list.removeFirst( )
top( )	list.first( )

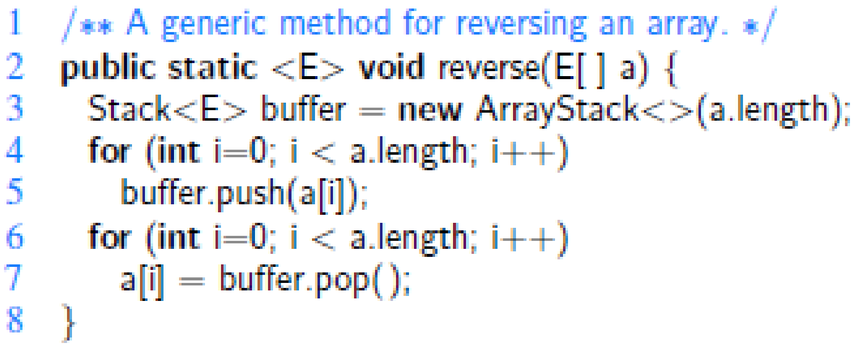


```
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>( );     // an empty list
  public LinkedStack( ) { }                    // new stack relies on the initially empty list
  public int size( ) { return list.size( ); }
  public boolean isEmpty( ) { return list.isEmpty( ); }
  public void push(E element) { list.addFirst(element); }
  public E top( ) { return list.first( ); }
  public E pop( ) { return list.removeFirst( ); }
}
```

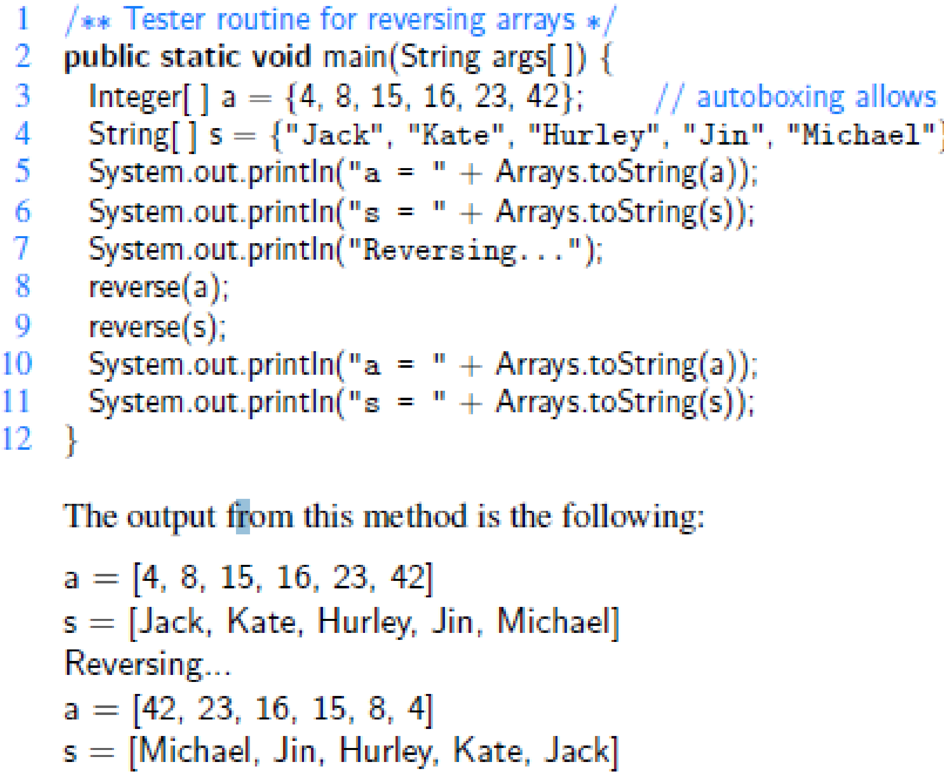
EXAMPLE: REVERSING AN ARRAY USING A STACK

A stack can be used as a general toll to reverse a data sequence.

reversing the elements of an array.



```
/** A generic method for reversing an array. */
public static <E> void reverse(E[ ] a) {
  Stack<E> buffer = new ArrayStack<>(a.length);
  for (int i=0; i < a.length; i++)
    buffer.push(a[i]);
  for (int i=0; i < a.length; i++)
    a[i] = buffer.pop( );
}
```



```
/** Tester routine for reversing arrays */
public static void main(String args[ ]) {
  Integer[ ] a = {4, 8, 15, 16, 23, 42};      // autoboxing allows
  String[ ] s = {"Jack", "Kate", "Hurley", "Jin", "Michael"
  System.out.println("a = " + Arrays.toString(a));
  System.out.println("s = " + Arrays.toString(s));
  System.out.println("Reversing...");
  reverse(a);
  reverse(s);
  System.out.println("a = " + Arrays.toString(a));
  System.out.println("s = " + Arrays.toString(s));
}
```

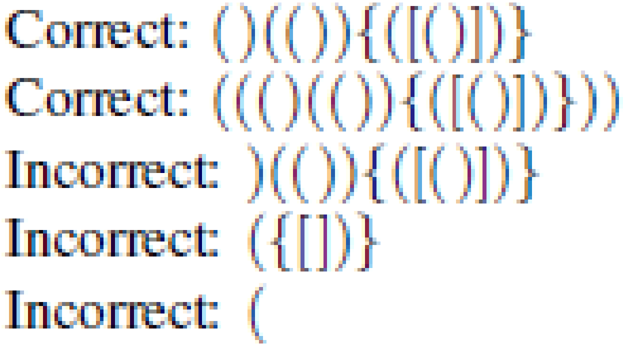
The output from this method is the following:

```
a = [4, 8, 15, 16, 23, 42]
s = [Jack, Kate, Hurley, Jin, Michael]
Reversing...
a = [42, 23, 16, 15, 8, 4]
s = [Michael, Jin, Hurley, Kate, Jack]
```

EXAMPLE: MATCHING PARENTHESES AND HTML TAGS

- Consider arithmetic expressions that may contain various pairs of grouping symbols:
  - Parentheses: “(” and “)”
  - Braces: “{” and “}”
  - Brackets: “[” and “]”

$[(5+x)-(y+z)$



Correct: ( )(( )){ ([( )]) }

Correct: ((( )(( )){ ([( )]) }))

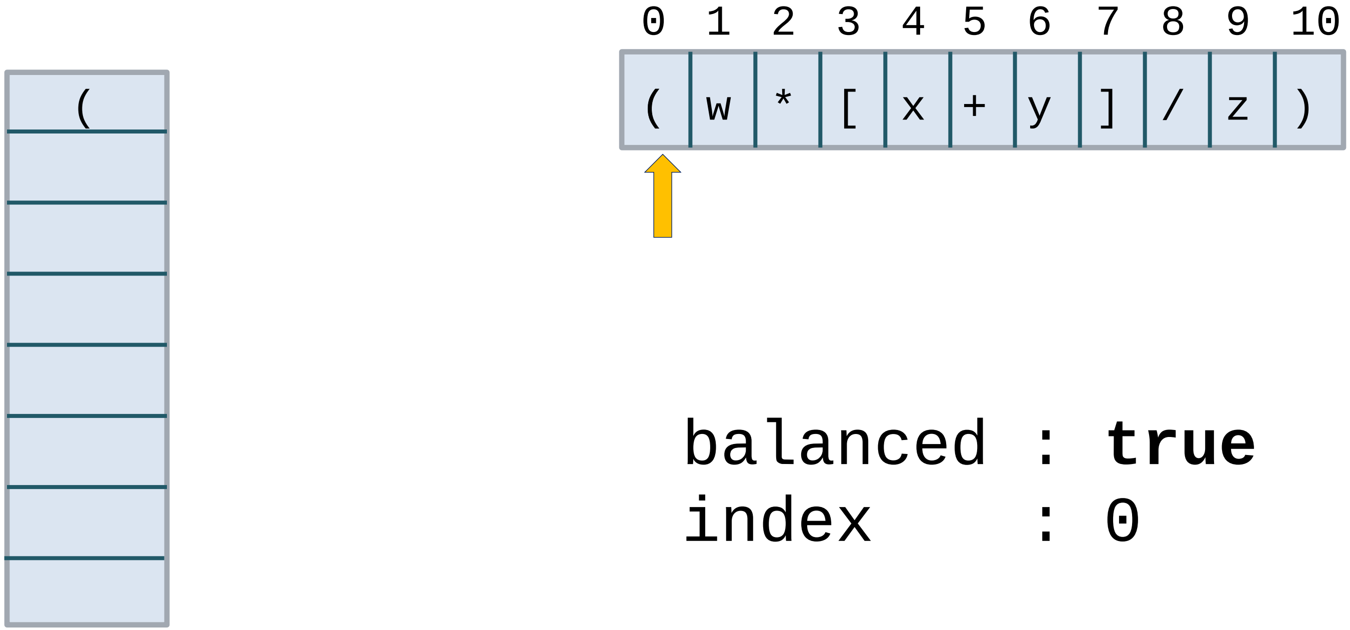
Incorrect:)(( )){ ([( )]) }

Incorrect: ({ [ ]) }

Incorrect: (

BALANCED PARENTHESES (CONT.)

Expression:`(w * [x + y] / z)`

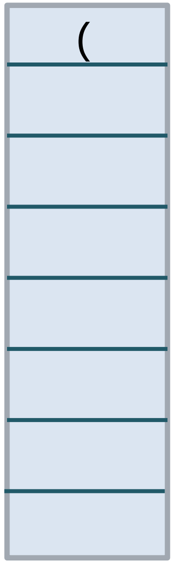


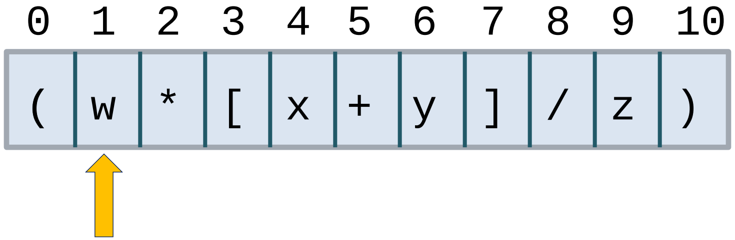
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 0
```

BALANCED PARENTHESES (CONT.)

Expression: `(w * [x + y] / z)`





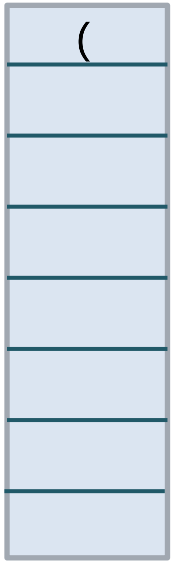
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

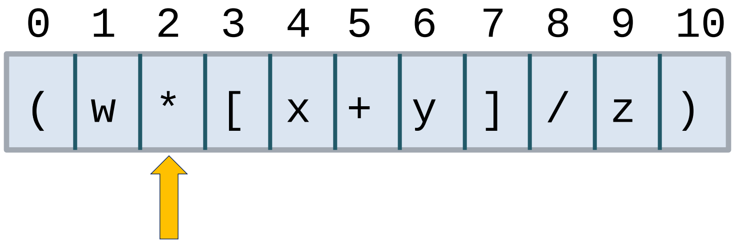
```
balanced : true
index    : 1
```

Data Structures Abstraction and Design Using Java, 2nd Edition
by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

BALANCED PARENTHESES (CONT.)

Expression:`(w * [x + y] / z)`



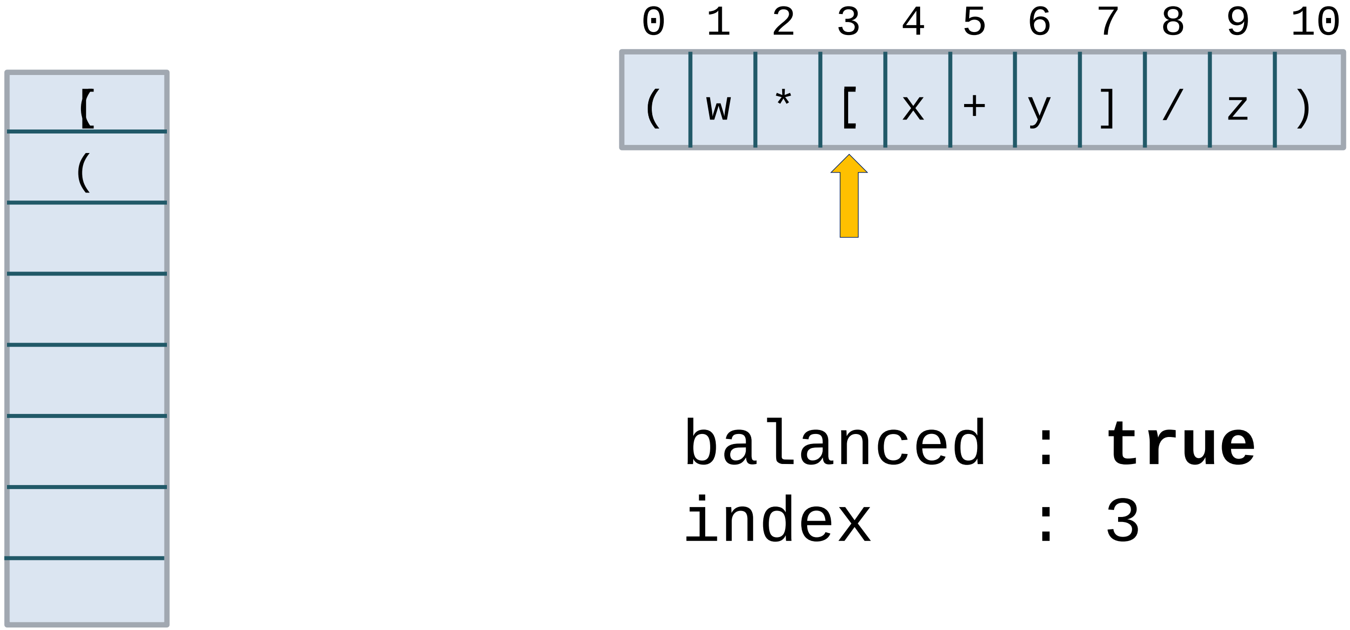


0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 2
```

BALANCED PARENTHESES (CONT.)

Expression:`(w * [x + y] / z)`



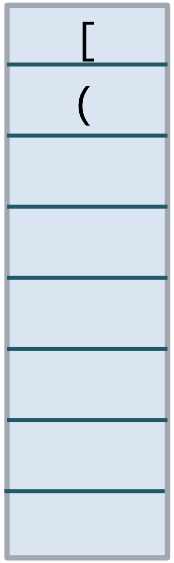
[
(

0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

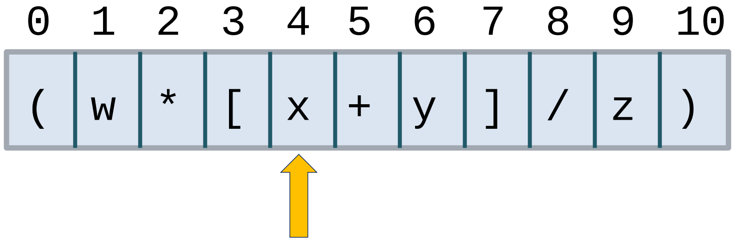
```
balanced : true
index    : 3
```

BALANCED PARENTHESES (CONT.)

Expression: `(w * [x + y] / z)`



[
(



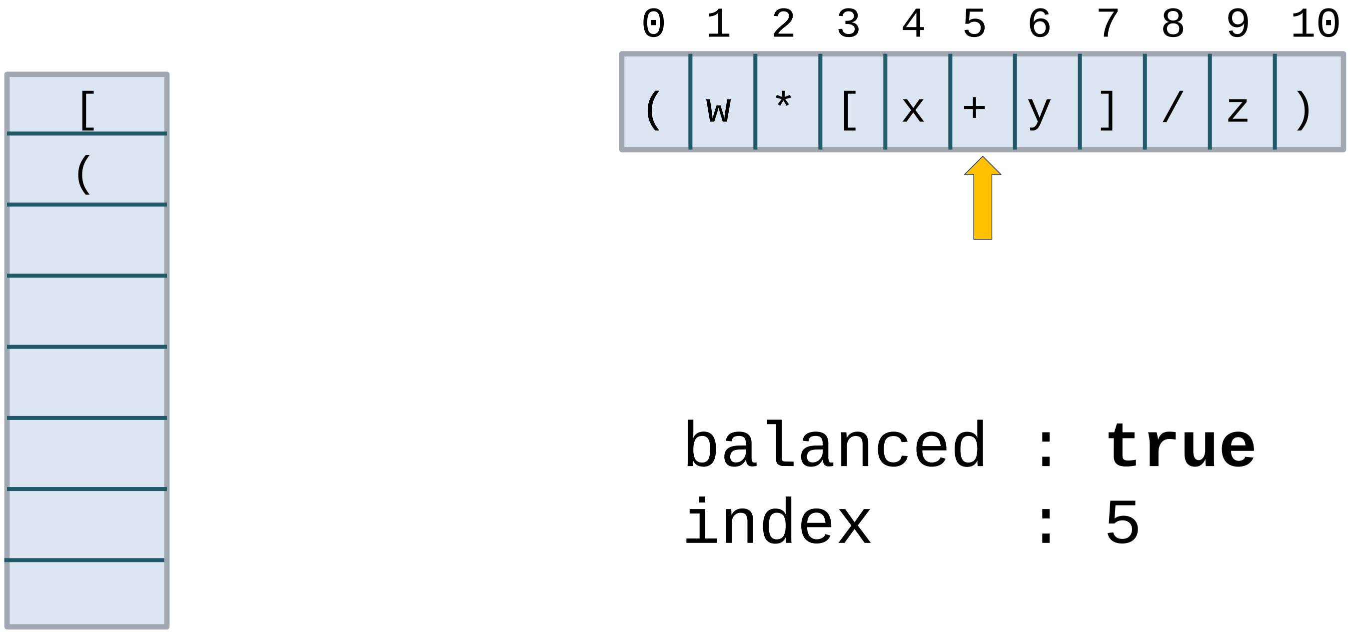
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

`balanced : `**`true`**

`index    : 4`

BALANCED PARENTHESES (CONT.)

Expression:`(w * [x + y] / z)`



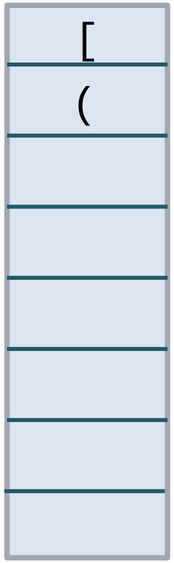
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

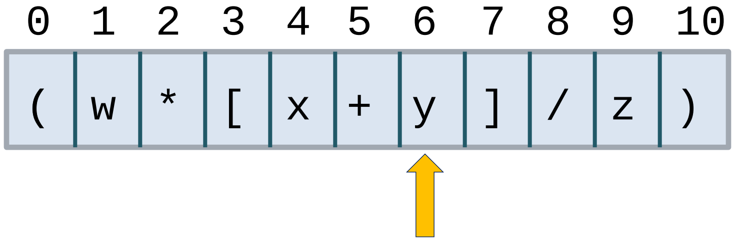
[
(

```
balanced : true
index    : 5
```

BALANCED PARENTHESES (CONT.)

Expression: `(w * [x + y] / z)`



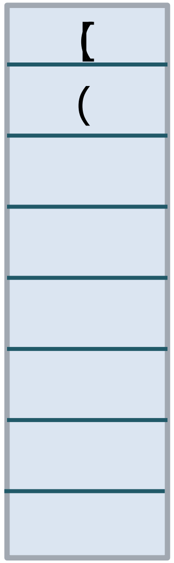


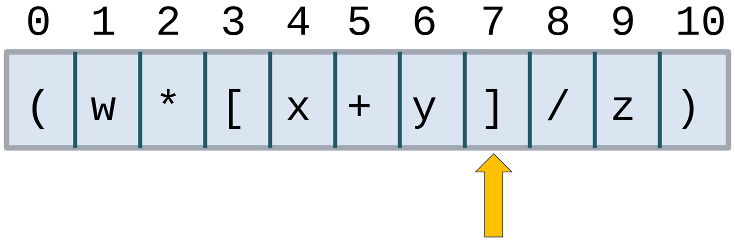
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

balanced : **true**
index   : 6

BALANCED PARENTHESES (CONT.)

Expression: `(w * [x + y] / z)`





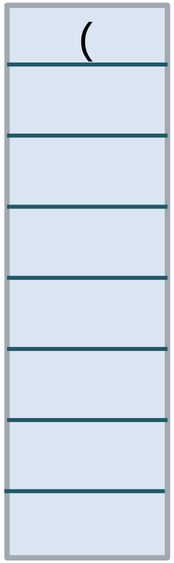
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

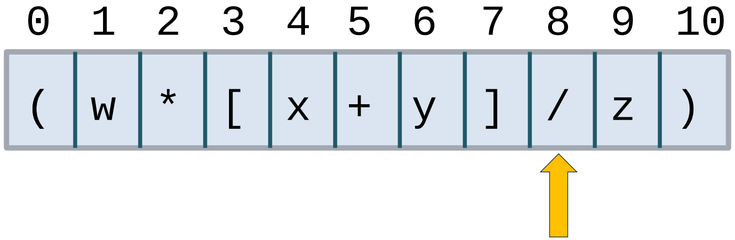
Matches!
`Balanced` still `true`

`balanced : `**`true`**
`index    : 7`

BALANCED PARENTHESES (CONT.)

Expression:`(w * [x + y] / z)`



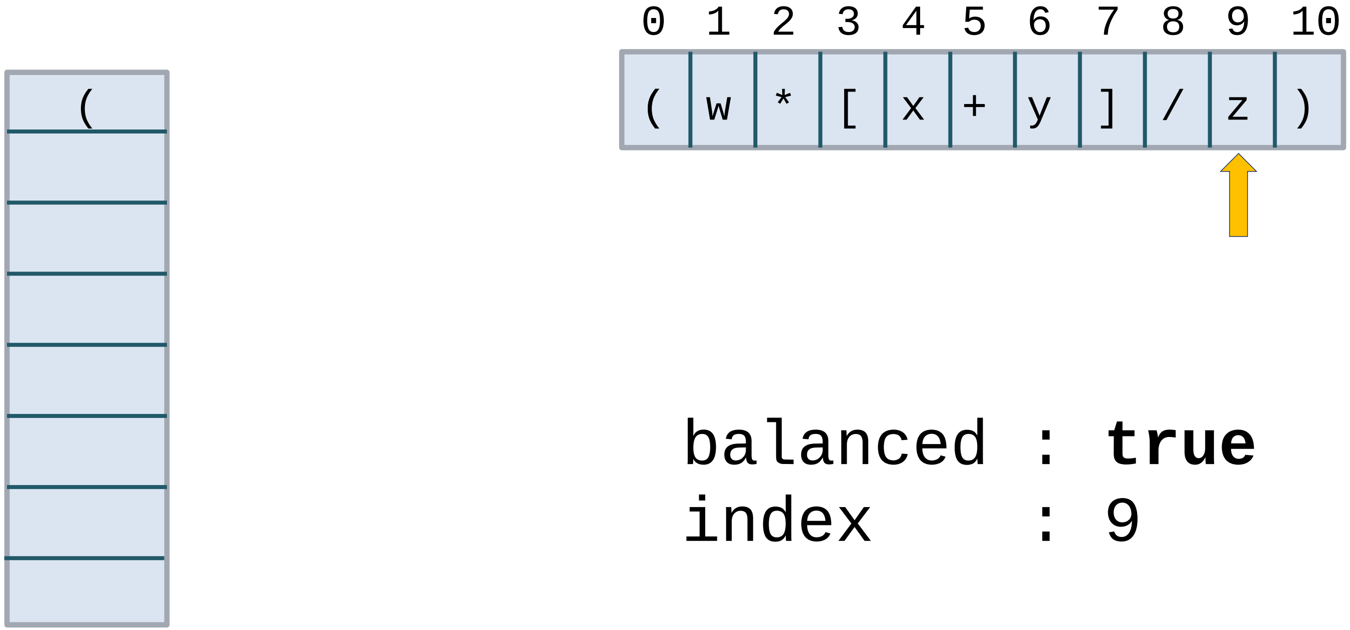


0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 8
```

BALANCED PARENTHESES (CONT.)

Expression:`(w * [x + y] / z)`

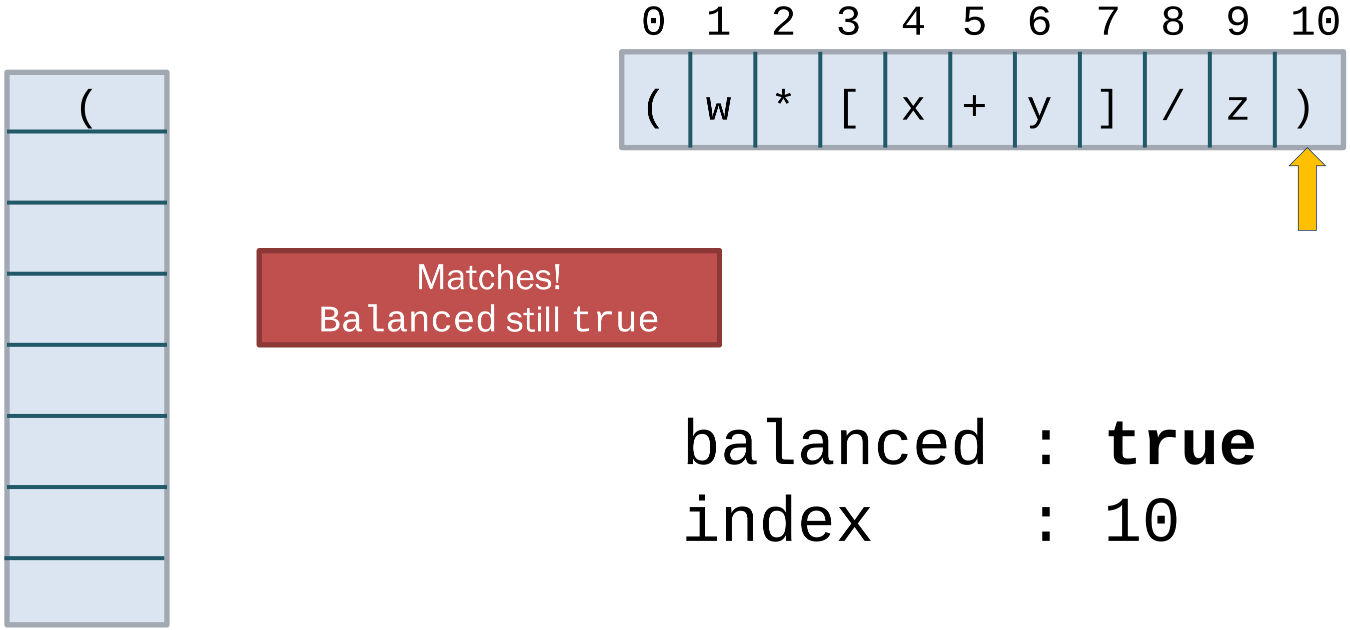


0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 9
```

BALANCED PARENTHESES (CONT.)

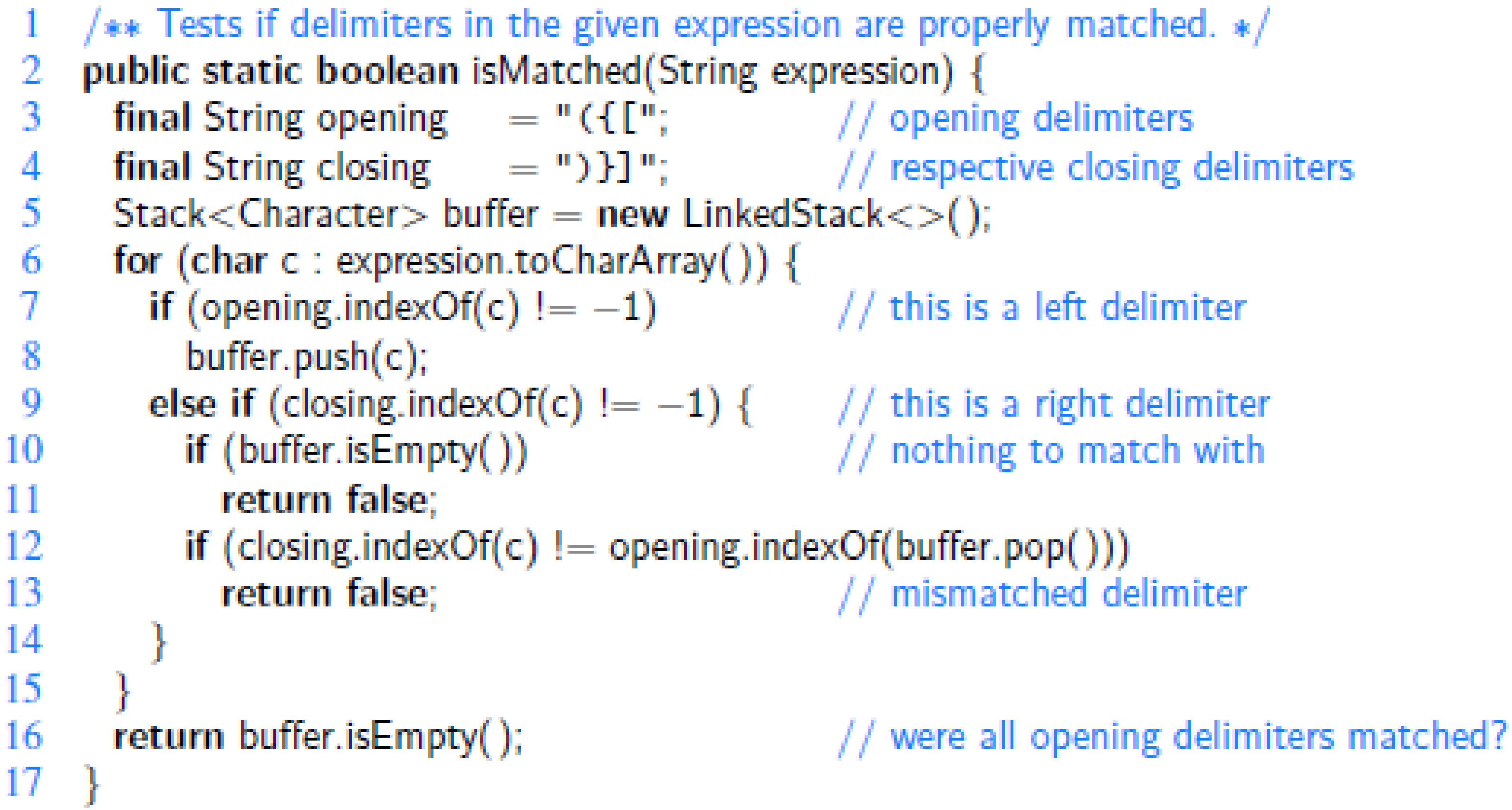
Expression: `(w * [x + y] / z)`



Matches!
Balanced still `true`

```
balanced : true
index    : 10
```

PARENTHESIS MATCHING IN AN ARITHMETIC EXPRESSION



```
/** Tests if delimiters in the given expression are properly matched. */
public static boolean isMatched(String expression) {
  final String opening = "({[";                      // opening delimiters
  final String closing = ")}]";                      // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>( );
  for (char c : expression.toCharArray( )) {
    if (opening.indexOf(c) != −1)                     // this is a left delimiter
      buffer.push(c);
    else if (closing.indexOf(c) != −1) {              // this is a right delimiter
      if (buffer.isEmpty( ))                          // nothing to match with
        return false;
      if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
        return false;                                 // mismatched delimiter
    }
  }
  return buffer.isEmpty( );                           // were all opening delimiters matched?
}
```

HTML TAG MATCHING

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

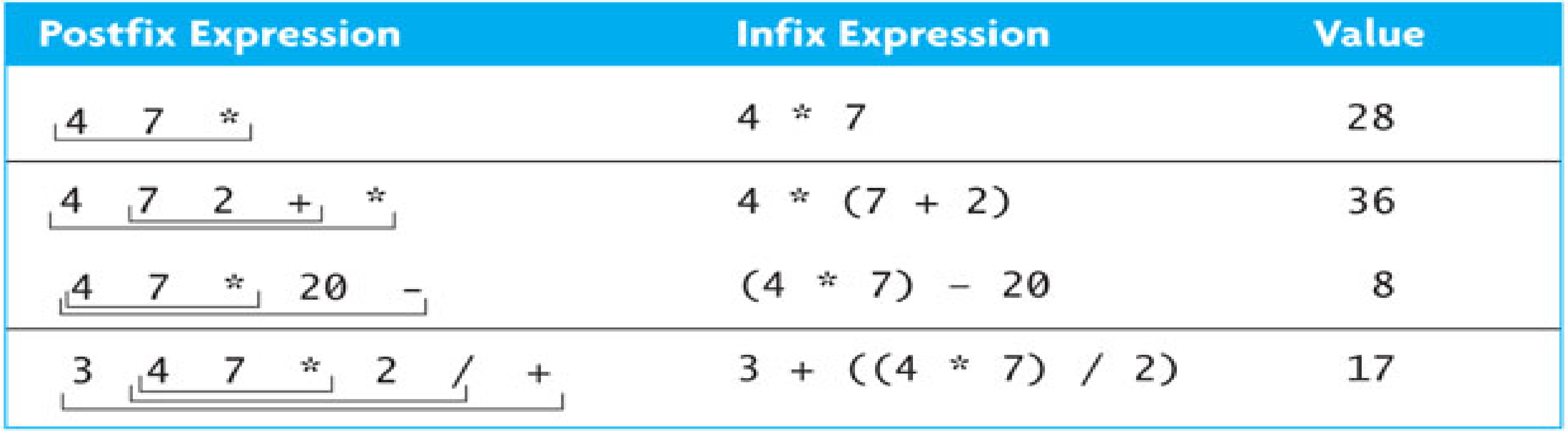
1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

HTML TAG MATCHING (JAVA)

```
/** Tests if every opening tag has a matching closing tag in HTML string. */
public static boolean isHTMLMatched(String html) {
  Stack<String> buffer = new LinkedStack<>();
  int j = html.indexOf('<');                          // find first '<' character (if any)
  while (j != −1) {
    int k = html.indexOf('>', j+1);                   // find next '>' character
    if (k == −1)
      return false;                                   // invalid tag
    String tag = html.substring(j+1, k);              // strip away < >
    if (!tag.startsWith("/"))                         // this is an opening tag
      buffer.push(tag);
    else {                                            // this is a closing tag
      if (buffer.isEmpty( ))
        return false;                                 // no tag to match
      if (!tag.substring(1).equals(buffer.pop( )))
        return false;                                 // mismatched tag
    }
    j = html.indexOf('<', k+1);                       // find next '<' character (if any)
  }
  return buffer.isEmpty( );                           // were all opening tags matched?
}
```

ADDITIONAL STACK APPLICATIONS

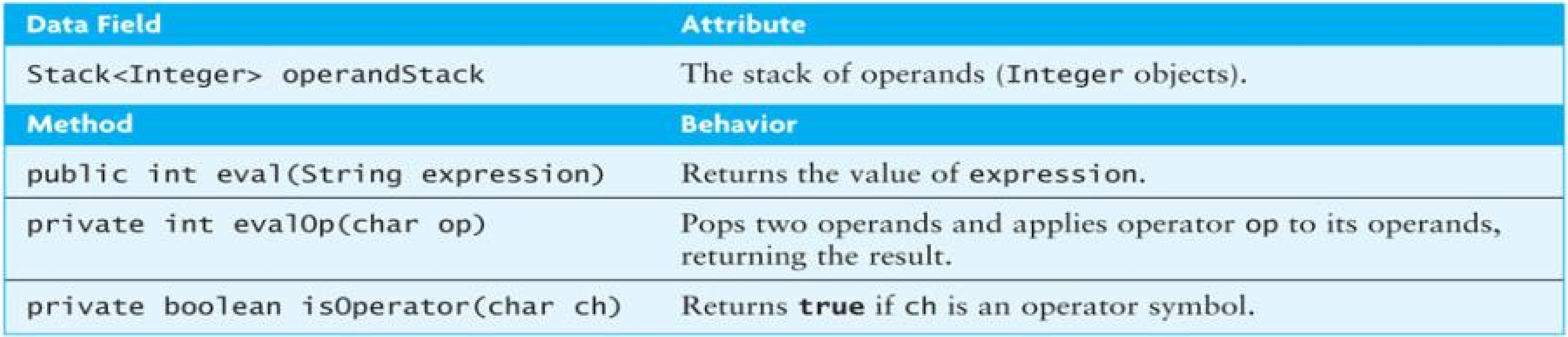
- Postfix and infix notation
  - Expressions normally are written in **infix** form, but
  - it easier to evaluate an expression in **postfix** form since there is no need to group sub-expressions in parentheses or worry about operator precedence



Postfix Expression	Infix Expression	Value
4 7 *	4 * 7	28
4 7 2 + *	4 * (7 + 2)	36
4 7 * 20 -	(4 * 7) - 20	8
3 4 7 * 2 / +	3 + ((4 * 7) / 2)	17

EVALUATING POSTFIX EXPRESSIONS

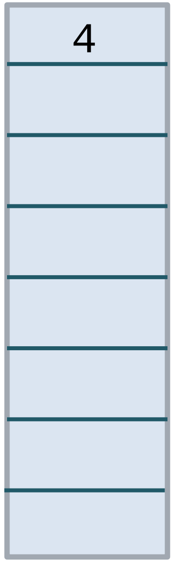
- Write a class that evaluates a postfix expression
- Use the space character as a delimiter between tokens

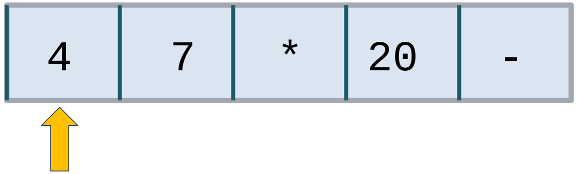


Data Field	**Attribute**
`Stack<Integer> operandStack`	The stack of operands (`Integer` objects).
Method	**Behavior**
`public int eval(String expression)`	Returns the value of `expression`.
`private int evalOp(char op)`	Pops two operands and applies operator `op` to its operands, returning the result.
`private boolean isOperator(char ch)`	Returns **`true`** if `ch` is an operator symbol.

EVALUATING POSTFIX EXPRESSIONS (CONT.)

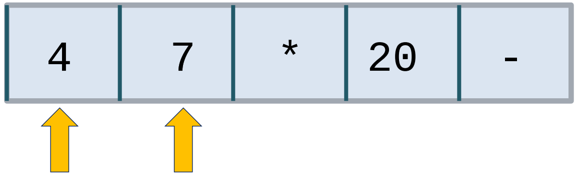
Stack of integers

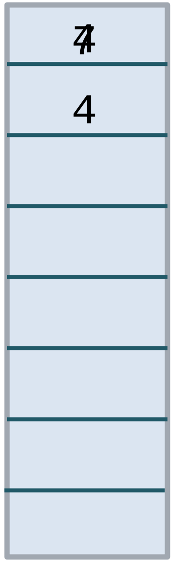




```
1.  create an empty stack of integers
2.  while there are more tokens
3.     get the next token
4.     if the first character of the token is a digit
5.        push the token on the stack
6.     else if the token is an operator
7.        pop the right operand off the stack
8.        pop the left operand off the stack
9.        evaluate the operation
10.       push the result onto the stack
11. pop the stack and return the result
```

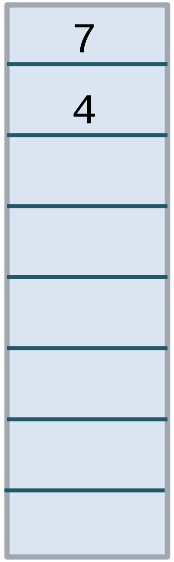
EVALUATING POSTFIX EXPRESSIONS (CONT.)



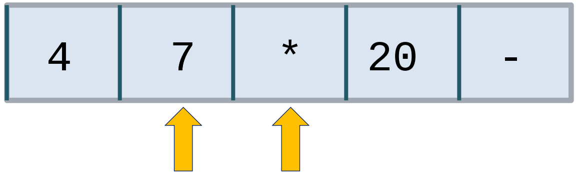


```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

EVALUATING POSTFIX EXPRESSIONS (CONT.)

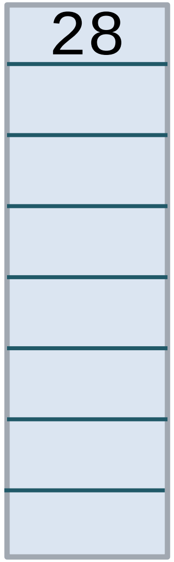


4 * 7

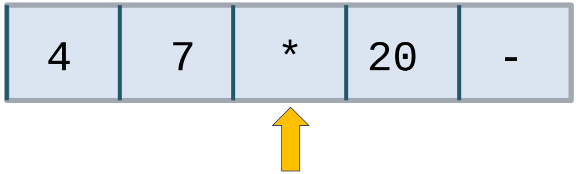


```
1. create an empty stack of integers
2. while there are more tokens
3.   get the next token
4.   if the first character of the token is a digit
5.      push the token on the stack
6.   else if the token is an operator
7.      pop the right operand off the stack
8.      pop the left operand off the stack
9.      evaluate the operation
10.     push the result onto the stack
11. pop the stack and return the result
```

EVALUATING POSTFIX EXPRESSIONS (CONT.)

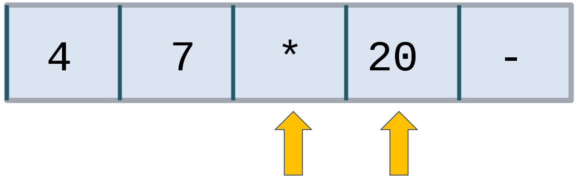


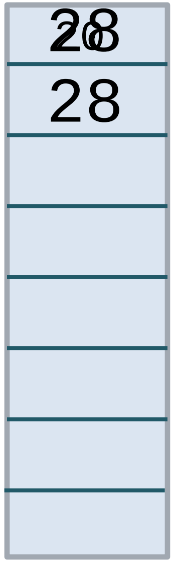
28



```
1.  create an empty stack of integers
2.  while there are more tokens
3.     get the next token
4.     if the first character of the token is a digit
5.        push the token on the stack
6.     else if the token is an operator
7.        pop the right operand off the stack
8.        pop the left operand off the stack
9.        evaluate the operation
10.       push the result onto the stack
11. pop the stack and return the result
```

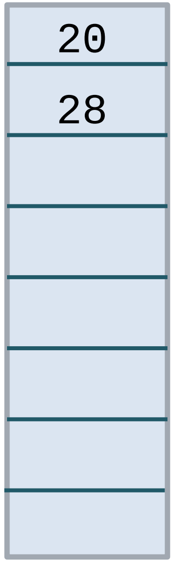
EVALUATING POSTFIX EXPRESSIONS (CONT.)



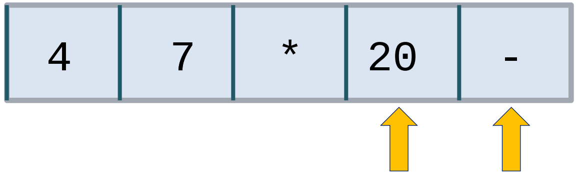


```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

EVALUATING POSTFIX EXPRESSIONS (CONT.)

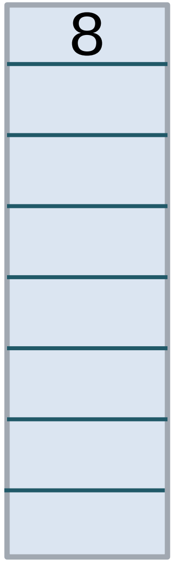


28 - 20

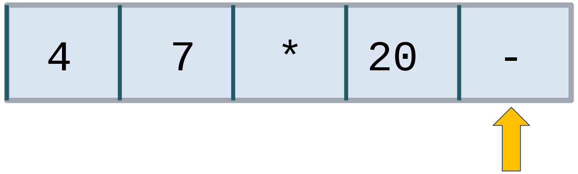


```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

EVALUATING POSTFIX EXPRESSIONS (CONT.)



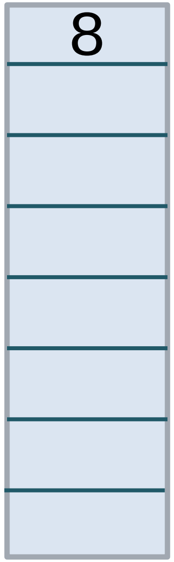
8

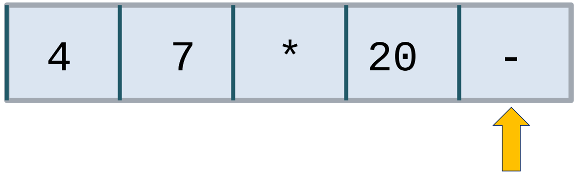


4	7	*	20	-

```
create an empty stack of integers
while there are more tokens
   get the next token
   if the first character of the token is a digit
      push the token on the stack
   else if the token is an operator
      pop the right operand off the stack
      pop the left operand off the stack
      evaluate the operation
      push the result onto the stack
pop the stack and return the result
```

EVALUATING POSTFIX EXPRESSIONS (CONT.)



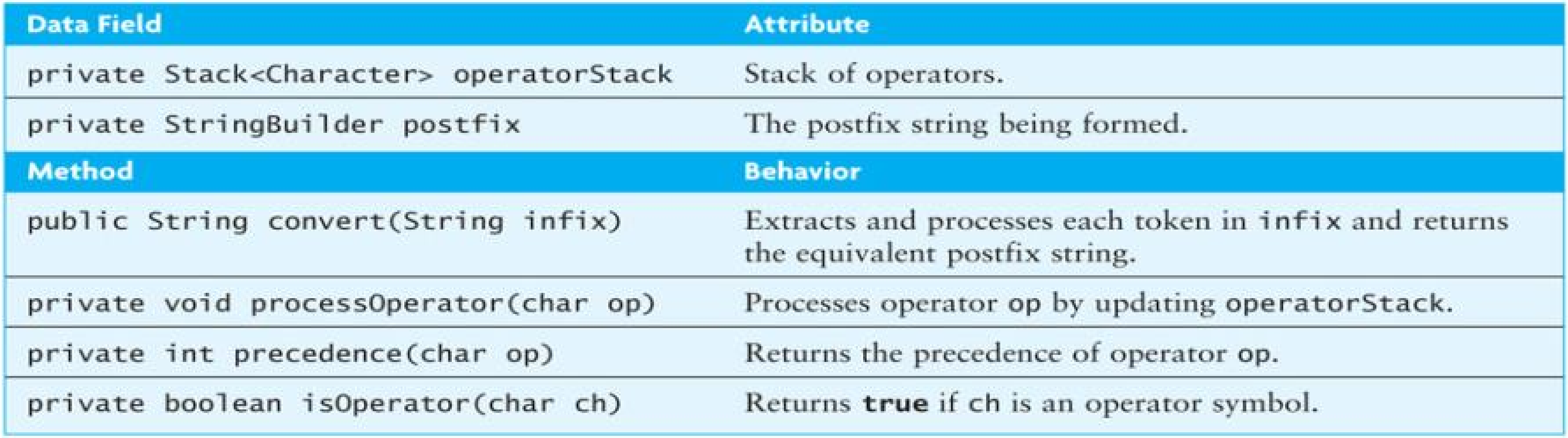


```
1.  create an empty stack of integers
2.  while there are more tokens
3.     get the next token
4.     if the first character of the token is a digit
5.        push the token on the stack
6.     else if the token is an operator
7.        pop the right operand off the stack
8.        pop the left operand off the stack
9.        evaluate the operation
10.       push the result onto the stack
11. pop the stack and return the result
```

Data Structures Abstraction and Design Using Java, 2nd Edition
by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

CONVERTING FROM INFIX TO POSTFIX

- Convert infix expressions to postfix expressions
- Assume:
    - Expressions consists of only spaces, operands, and operators
    - Space is a delimiter character
    - All operands that are identifiers begin with a letter or underscore
    - All operands that are numbers begin with a digit



Data Field	Attribute
`private Stack<Character> operatorStack`	Stack of operators.
`private StringBuilder 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.

CONVERTING FROM INFIX TO POSTFIX (CONT.)

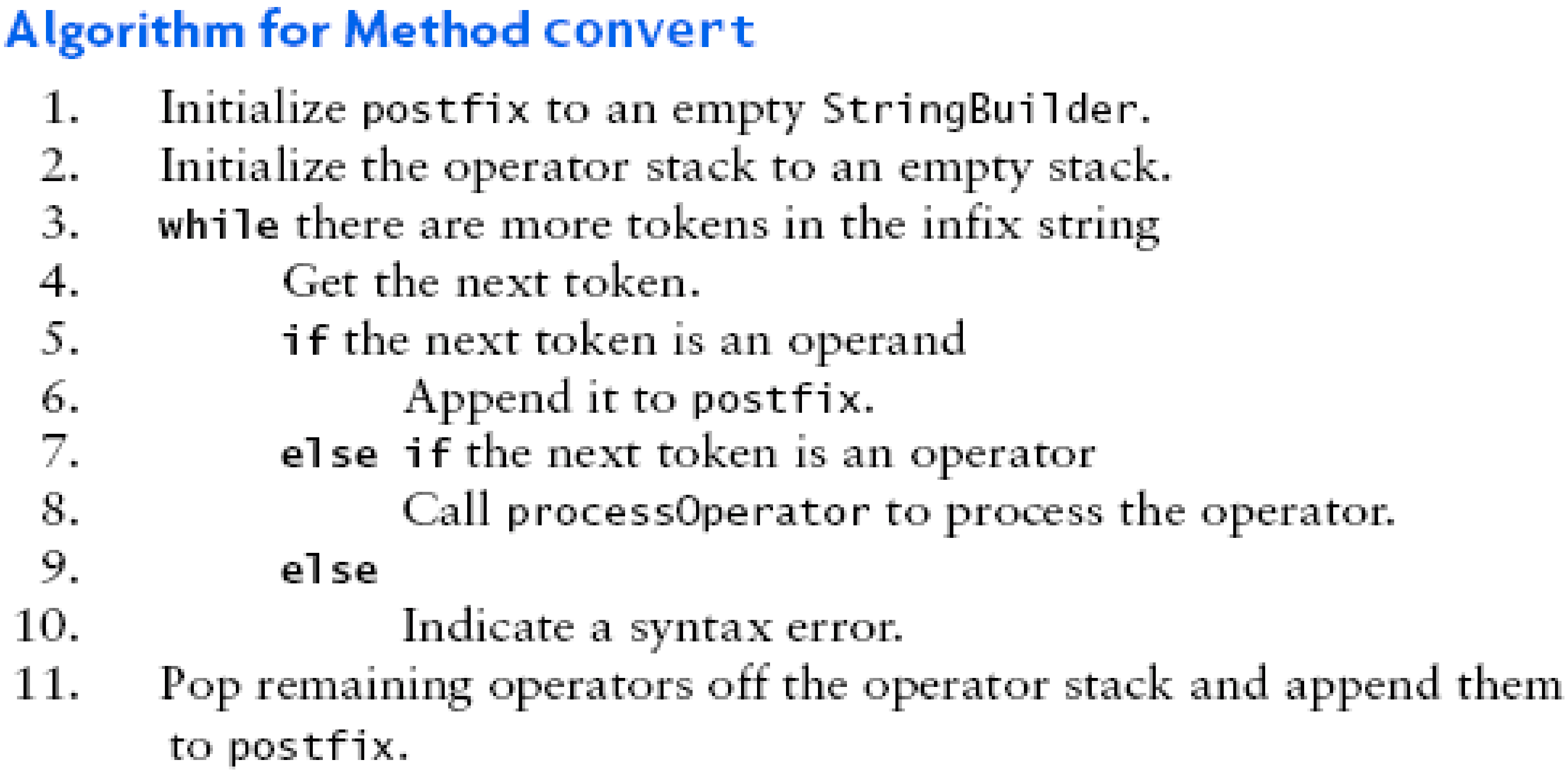
- Example: convert

  w – 5.1 / sum * 2

  to its postfix form

  w 5.1 sum / 2 * -

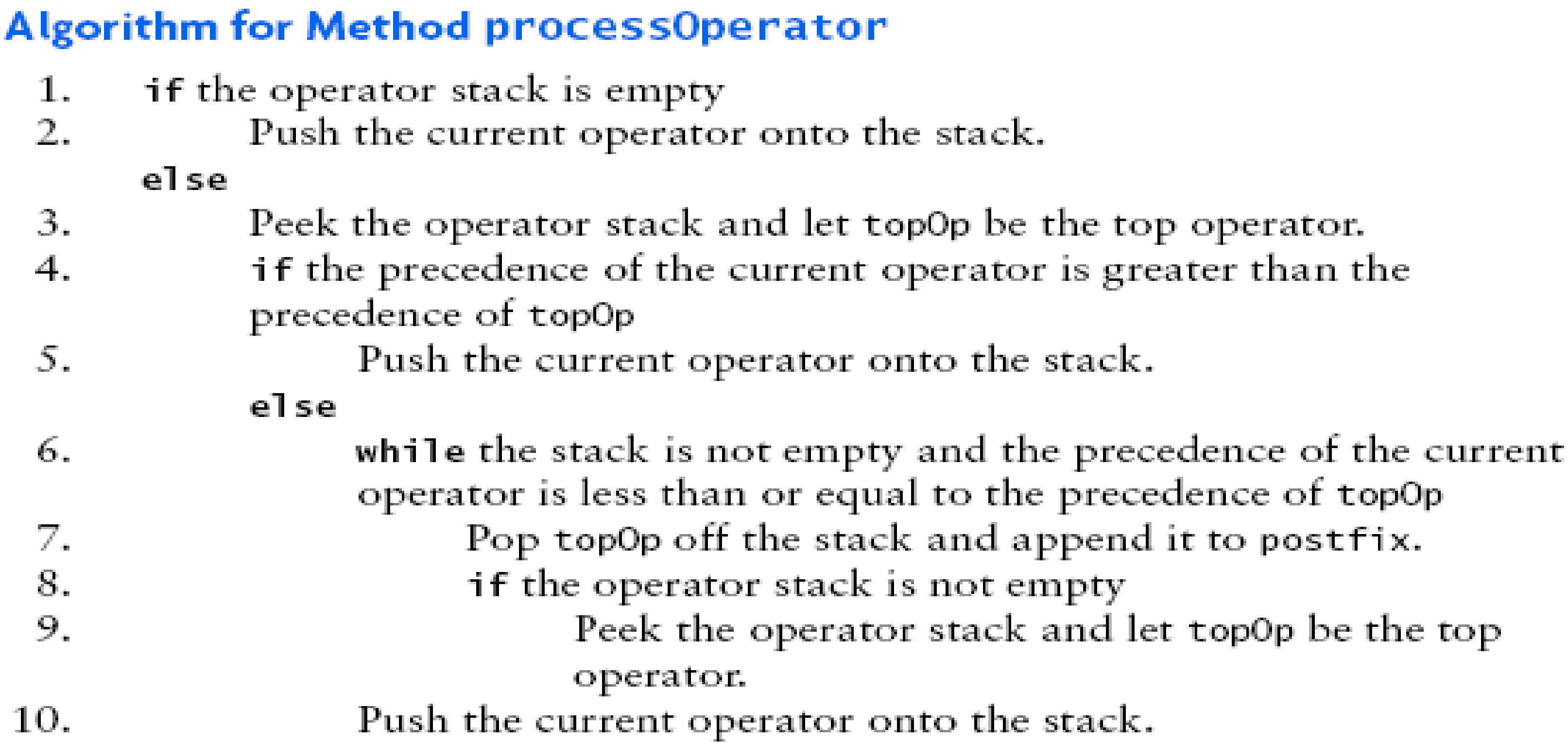
CONVERTING FROM INFIX TO POSTFIX (CONT.)



Algorithm for Method convert

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

CONVERTING FROM INFIX TO POSTFIX (CONT.)

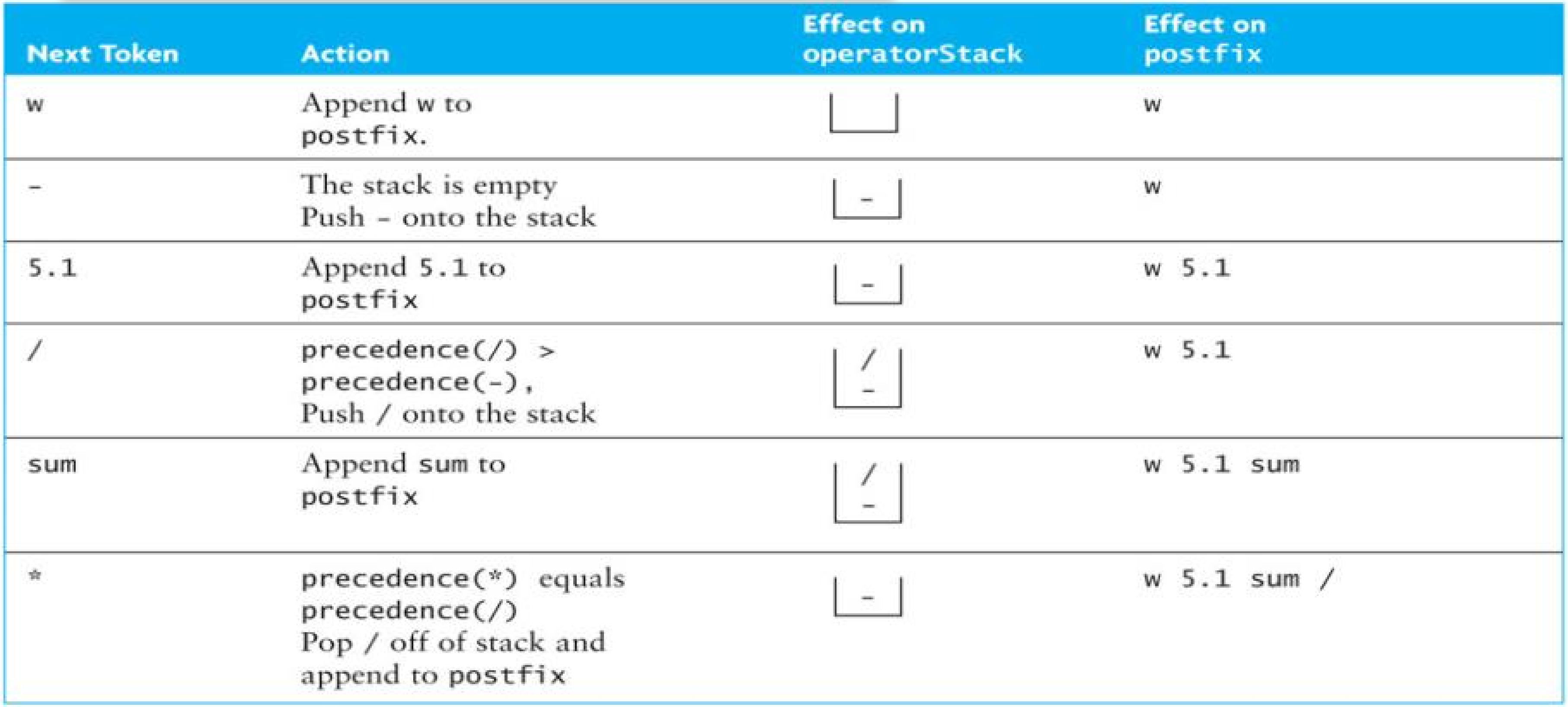


Algorithm for Method processOperator

```
if the operator stack is empty
    Push the current operator onto the stack.
else
    Peek the operator stack and let topOp be the top operator.
    if the precedence of the current operator is greater than the
    precedence of topOp
        Push the current operator onto the stack.
    else
        while the stack is not empty and the precedence of the current
        operator is less than or equal to the precedence of topOp
            Pop topOp off the stack and append it to postfix.
            if the operator stack is not empty
                Peek the operator stack and let topOp be the top
                operator.
        Push the current operator onto the stack.
```

CONVERTING FROM INFIX TO POSTFIX (CONT.)

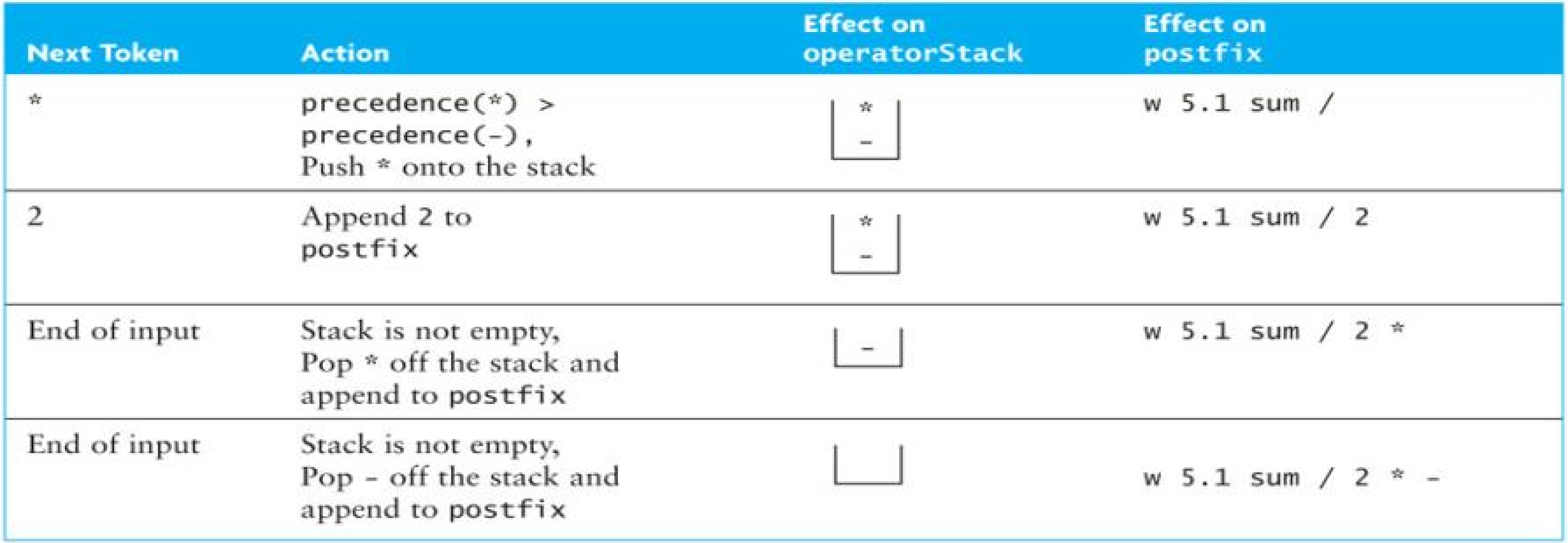
INPUT: w – 5.1 / sum * 2



Next Token	Action	Effect on operatorStack	Effect on postfix
w	Append w to postfix.		w
-	The stack is empty Push - onto the stack	-	w
5.1	Append 5.1 to postfix	-	w 5.1
/	precedence(/) > precedence(-), Push / onto the stack	/ -	w 5.1
sum	Append sum to postfix	/ -	w 5.1 sum
*	precedence(*) equals precedence(/) Pop / off of stack and append to postfix	-	w 5.1 sum /

Data Structures Abstraction and Design Using Java, 2nd Edition
by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

CONVERTING FROM INFIX TO POSTFIX (CONT.)



Next Token	**Action**	**Effect on operatorStack**	**Effect on postfix**
*	`precedence(*) > precedence(-)`, Push * onto the stack	* -	`w 5.1 sum /`
2	Append 2 to `postfix`	* -	`w 5.1 sum / 2`
End of input	Stack is not empty, Pop * off the stack and append to `postfix`	-	`w 5.1 sum / 2 *`
End of input	Stack is not empty, Pop - off the stack and append to `postfix`		`w 5.1 sum / 2 * -`

CONVERTING EXPRESSIONS WITH PARENTHESES

- The ability to convert expressions with parentheses is an important (and necessary) addition
- Modify `processOperator` to push each opening parenthesis onto the stack as soon as it is scanned
- When a closing parenthesis is encountered, pop off operators until the opening parenthesis is encountered