CHAPTER 3

Stacks

Chapter Objectives

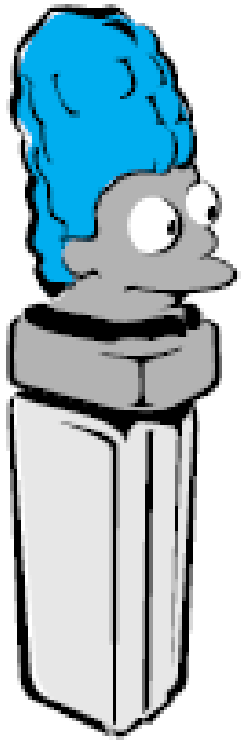
- To learn about the stack data type and how to use its four methods:
  - push
  - pop
  - peek
  - empty
- To understand how Java implements a stack
- To learn how to implement a stack using an underlying array or linked list
- To see how to use a stack to perform various applications, including finding palindromes, testing for balanced (properly nested) parentheses, and evaluating arithmetic expressions

Stack Abstract Data Type

Section 3.1

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*



Specification of the Stack Abstract Data Type

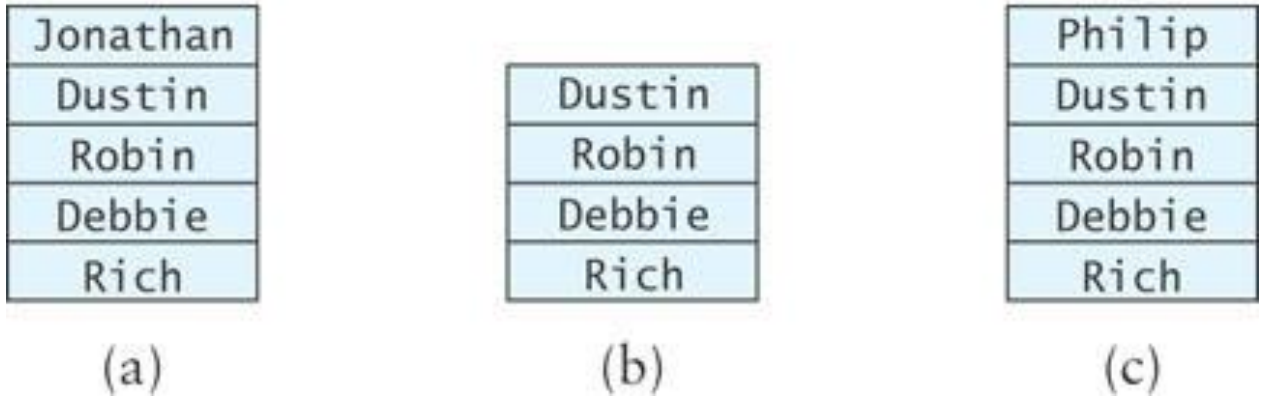
- Only the top element of a stack is visible; therefore the number of operations performed by a stack are few
- We need the ability to
  - test for an empty stack (empty)
  - inspect the top element (peek)
  - retrieve the top element (pop)
  - put a new element on the stack (push)

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

Specification of the Stack Abstract Data Type (cont.)


```
public interface StackInt<E> {
  /* comments are deleted */
  E push(E obj);
  E peek();
  E pop();
  boolean empty();
}
```

A Stack of Strings



- "Rich" is the oldest element on the stack and "Jonathan" is the youngest (Figure a)
- `String last = names.peek();` 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 Applications

Section 3.2

Balanced Parentheses

- When analyzing arithmetic expressions, it is important to determine whether an expression is balanced with respect to parentheses

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

- The problem is further complicated if braces or brackets are used in conjunction with parentheses
- The solution is to use stacks!

Balanced Parentheses (cont.)

Method	Behavior
`public static boolean isBalanced(String expression)`	Returns **`true`** if `expression` is balanced with respect to parentheses and **`false`** if it is not.
`private static boolean isOpen(char ch)`	Returns **`true`** if `ch` is an opening parenthesis.
`private static boolean isClose(char ch)`	Returns **`true`** if `ch` is a closing parenthesis.

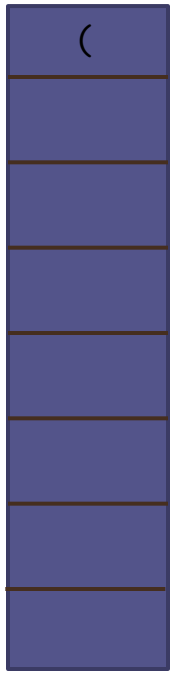
Balanced Parentheses (cont.)

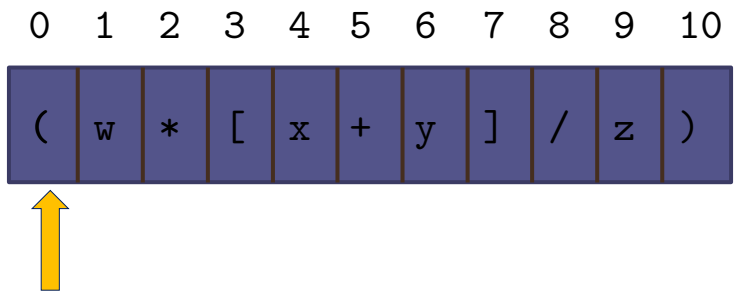
Algorithm for method `isBalanced`

```
1.  Create an empty stack of characters.
2.  Assume that the expression is balanced (balanced is true).
3.  Set index to 0.
4.  while balanced is true and index < the expression's length
5.      Get the next character in the data string.
6.      if the next character is an opening parenthesis
7.          Push it onto the stack.
8.      else if the next character is a closing parenthesis
9.          Pop the top of the stack.
10.         if stack was empty or its top does not match the closing
            parenthesis
11.             Set balanced to false.
12.     Increment index.
13. Return true if balanced is true and the stack is empty.
```

Balanced Parentheses (cont.)

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

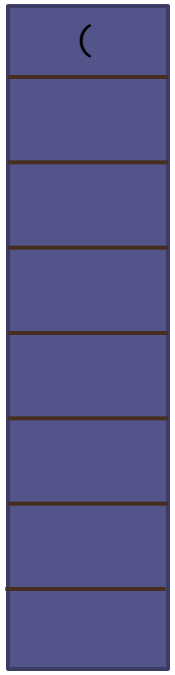


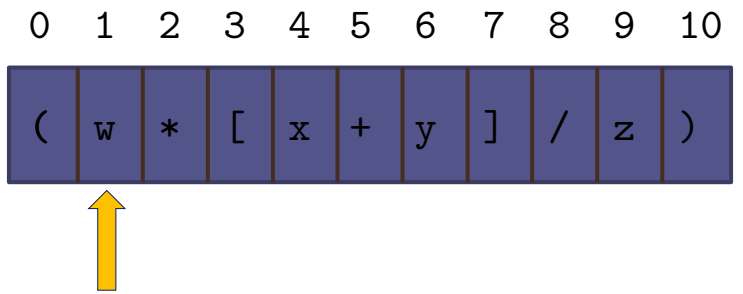



```
balanced : true
index    : 0
```

Balanced Parentheses (cont.)

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

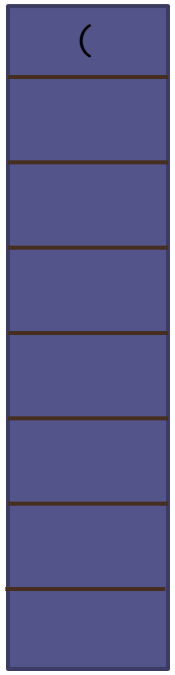


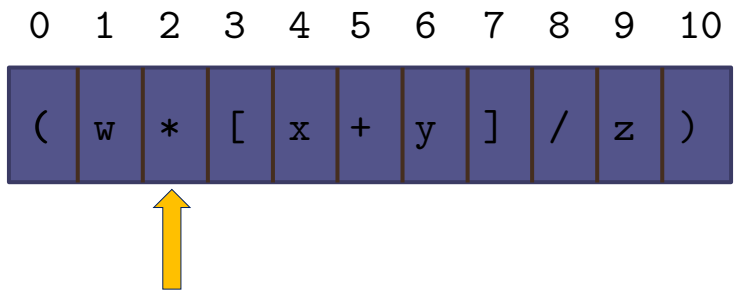



```
balanced : true
index    : 1
```

Balanced Parentheses (cont.)

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

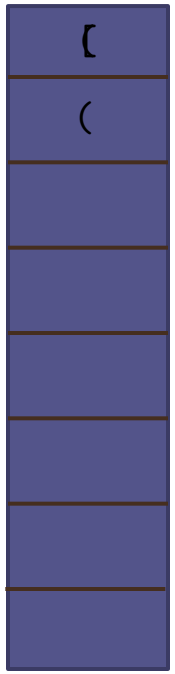


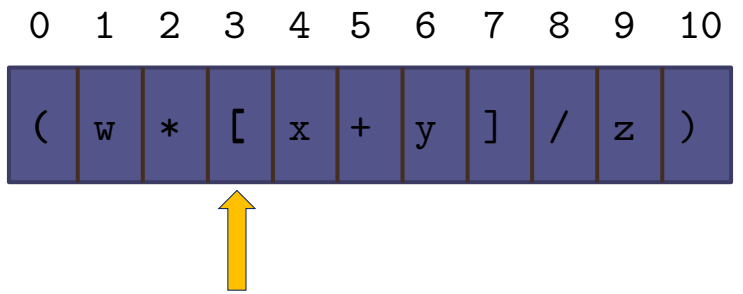



```
balanced : true
index    : 2
```

Balanced Parentheses (cont.)

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

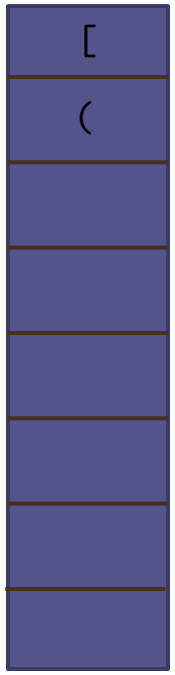


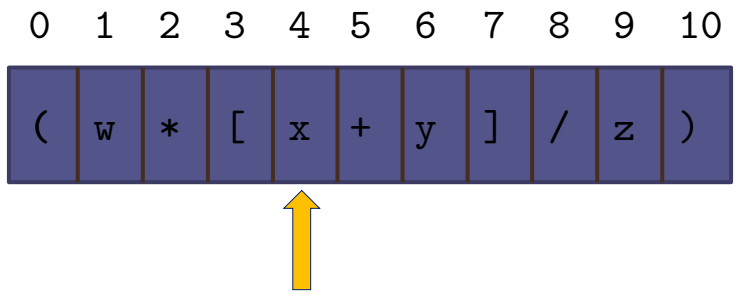



```
balanced : true
index    : 3
```

Balanced Parentheses (cont.)

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

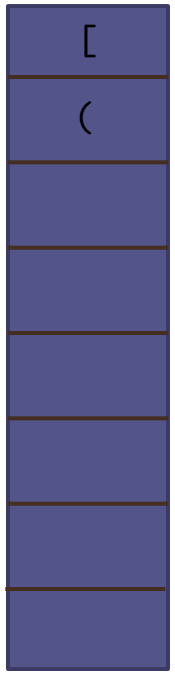


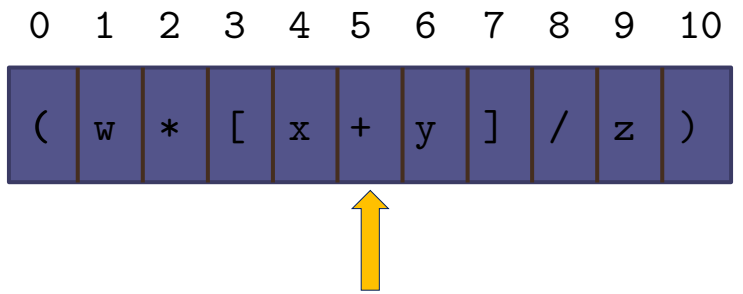



```
balanced : true
index    : 4
```

Balanced Parentheses (cont.)

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

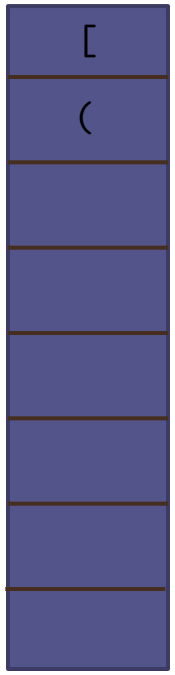


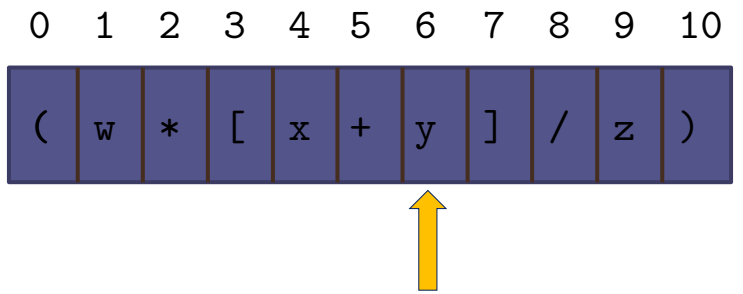



```
balanced : true
index    : 5
```

Balanced Parentheses (cont.)

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

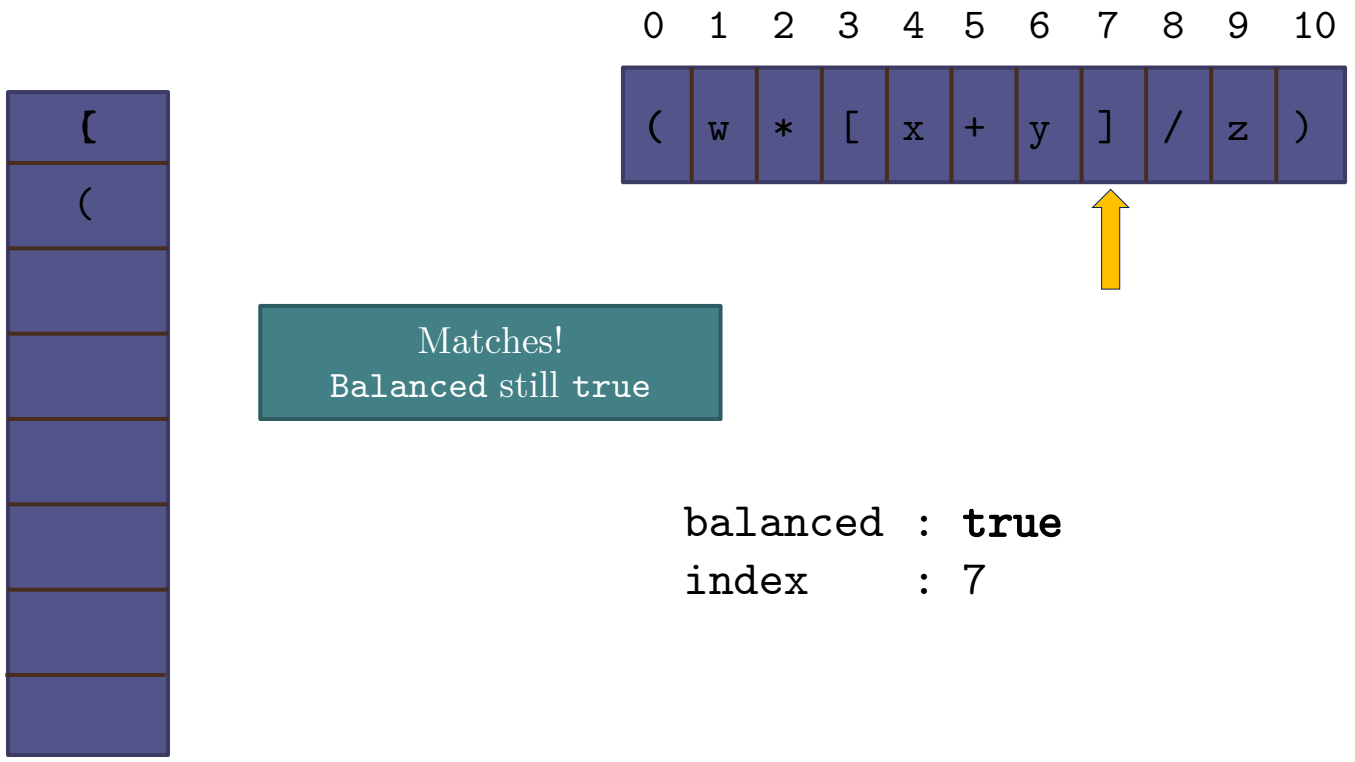





```
balanced : true
index    : 6
```

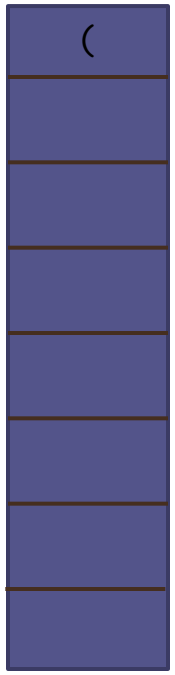
Balanced Parentheses (cont.)

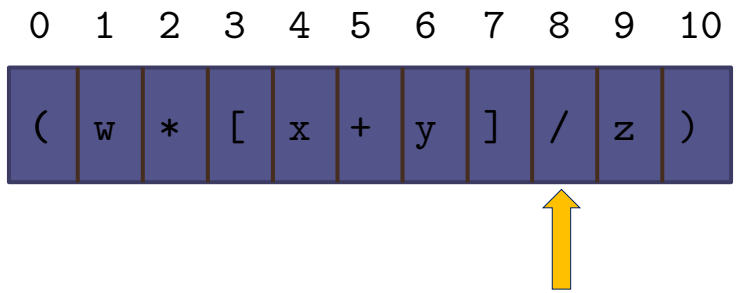
Expression: `(w * [x + y] / z)`



Balanced Parentheses (cont.)

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

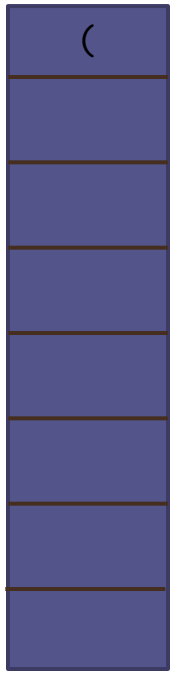


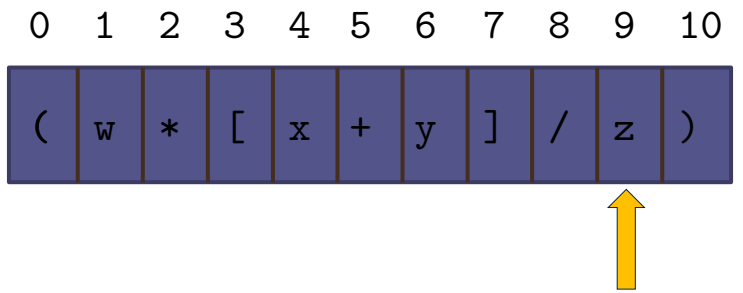



```
balanced : true
index    : 8
```

Balanced Parentheses (cont.)

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

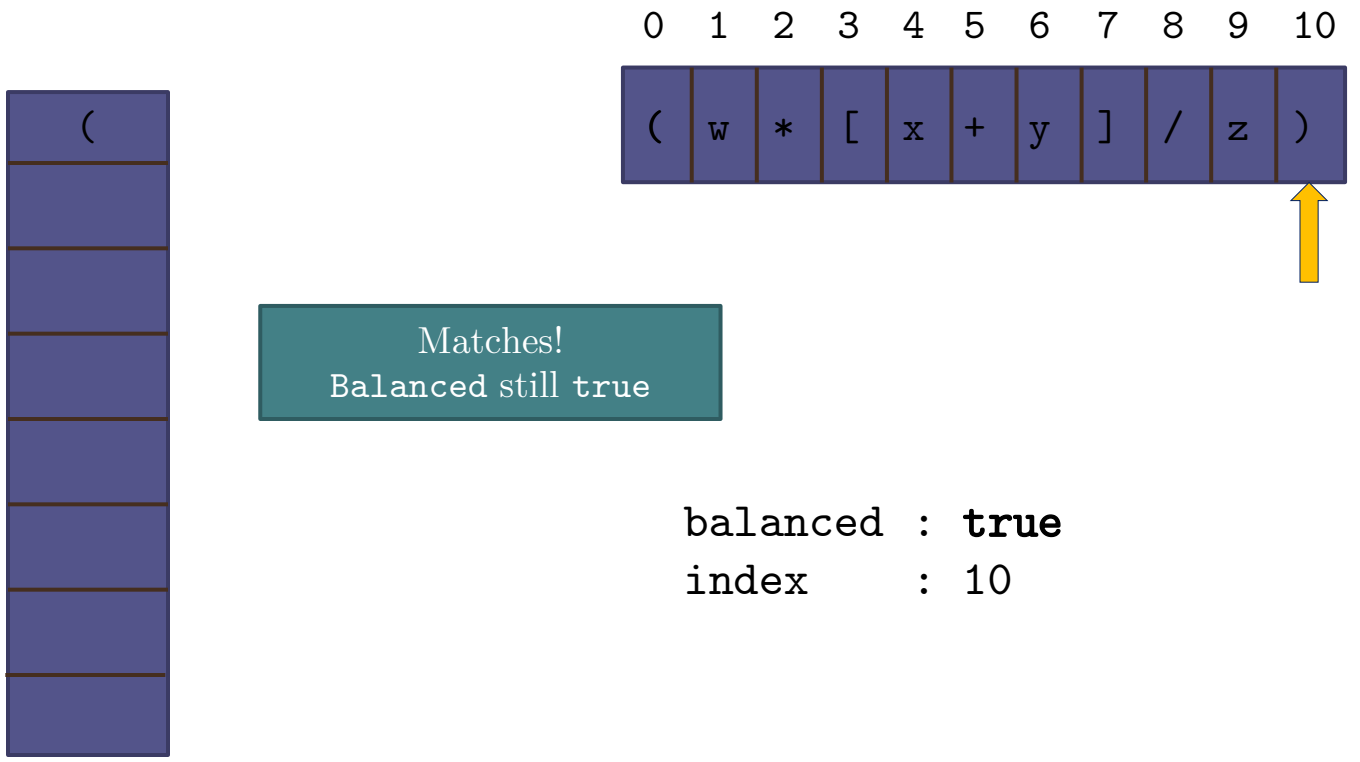





```
balanced : true
index    : 9
```

Balanced Parentheses (cont.)

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



Balanced Parentheses (cont.)

```
public static boolean isBalanced(String expression) {
    // Create an empty stack.
    Stack<Character> s = new Stack<Character>();
    boolean balanced = true;
    try {
        int index = 0;
        while (balanced && index < expression.length()) {
            char nextCh = expression.charAt(index);
            if (isOpen(nextCh)) {
                s.push(nextCh);
            } else if (isClose(nextCh)) {
                char topCh = s.pop();
                balanced =
                    OPEN.indexOf(topCh) == CLOSE.indexOf(nextCh);
            }
            index++;
        }
    } catch (EmptyStackException ex) {
        balanced = false;
    }
    return (balanced && s.empty());
}
```

```
private static final String OPEN = "([{";

private static final String CLOSE = ")]}";

private static boolean isOpen(char ch) {
    return OPEN.indexOf(ch) > -1;
}

private static boolean isClose(char ch) {
    return CLOSE.indexOf(ch) > -1;
}
```

Testing

- Provide a variety of input expressions displaying the result `true` or `false`
- Try several levels of nested parentheses
- Try nested parentheses where corresponding parentheses are not of the same type
- Try unbalanced parentheses
- No parentheses at all!

- PITFALL: attempting to pop an empty stack will throw an `EmptyStackException`. You can guard against this by either testing for an empty stack or catching the exception

Implementing a Stack

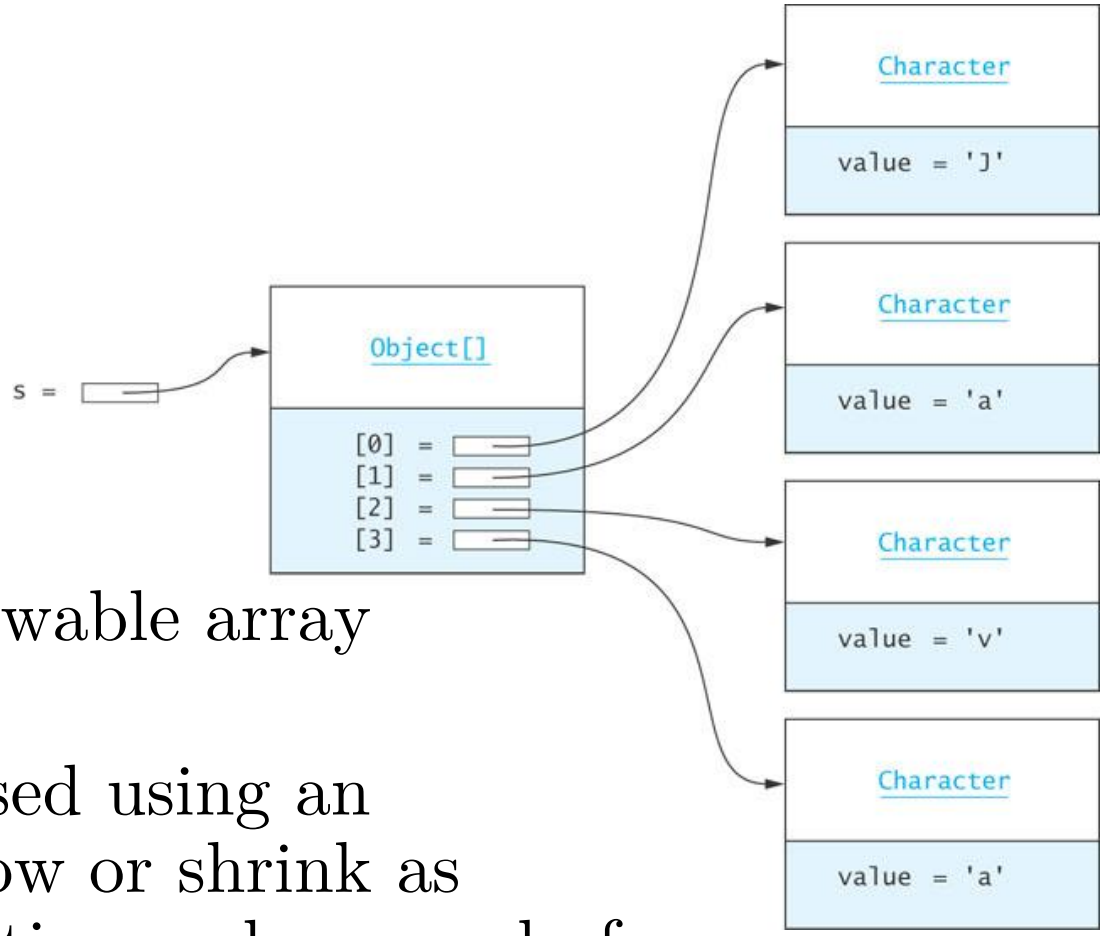
Section 3.3

Implementing a Stack as an Extension of Vector

- The Java API includes a `Stack` class as part of the package `java.util` :

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

- The `Vector` class implements a growable array of objects
- Elements of a `Vector` can be accessed using an integer index and the size can grow or shrink as needed to accommodate the insertion and removal of elements



Implementing a Stack as an Extension of Vector (cont.)

- We can use `Vector`'s `add` method to implement `push`:

```
public E push(obj E) {
    add(obj);
    return obj;
  }
```

- pop can be coded as

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

Implementing a Stack as an Extension of Vector (cont.)

- Because a `Stack` *is a* `Vector`, all of `Vector` operations can be applied to a `Stack` (such as searches and access by index)
- But, since only the top element of a stack should be accessible, this violates the principle of information hiding

Implementing a Stack with a List Component

- As an alternative to a stack as an extension of `Vector`, we can write a class, `ListStack`, that has a `List` component (in the example below, `theData`)
- We can use either the `ArrayList`, `Vector`, or the `LinkedList` classes, as all implement the `List` interface. The `push` method, for example, can be coded as

```
public E push(E obj) {
  theData.add(obj);
  return obj;
}
```

- A class which adapts methods of another class by giving different names to essentially the same methods (`push` instead of `add`) is called an ***adapter*** *class*
- Writing methods in this way is called *method* ***delegation***

Implementing a Stack with a List Component (cont.)


```
public class ListStack<E> implements StackInt<E> {
    private List<E> theData;

    public ListStack() {
        theData = new ArrayList<E>();
    }

    @Override
    public E push(E obj) {
        theData.add(obj);
        return obj;
    }

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

Implementing a Stack with a List Component (cont.)


```
public class ListStack<E> implements StackInt<E> {
    private List<E> theData;

    . . . .

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

    @Override
    public boolean empty() {
        return theData.size() == 0;
    }
}
```

Implementing a Stack Using an Array

- If we implement a stack as an array, we would need . . .

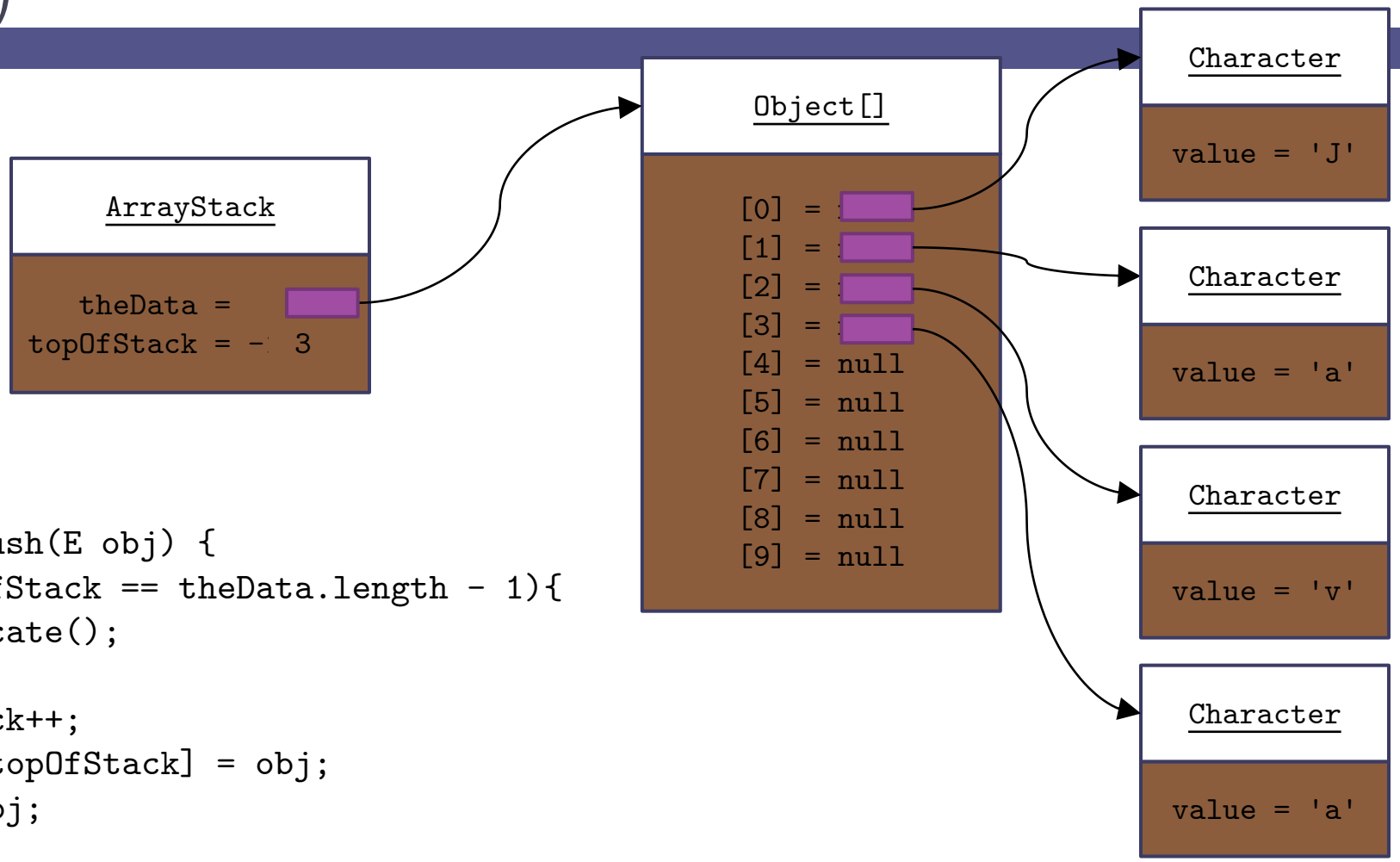
```
public class ArrayStack<E> implements StackInt<E> {
  private E[] theData;
  int topOfStack = -1;
  private static final int INITIAL_CAPACITY = 10;

  @SupressWarnings("unchecked")
  public ArrayStack() {
    theData = (E[])new Object[INITIAL_CAPACITY];
  }
```

Allocate storage for an array with a default capacity

Keep track of the top of the stack (subscript of the element at the top of the stack; for empty stack = -1)

There is no `size` variable or method

Implementing a Stack Using an Array (cont.)



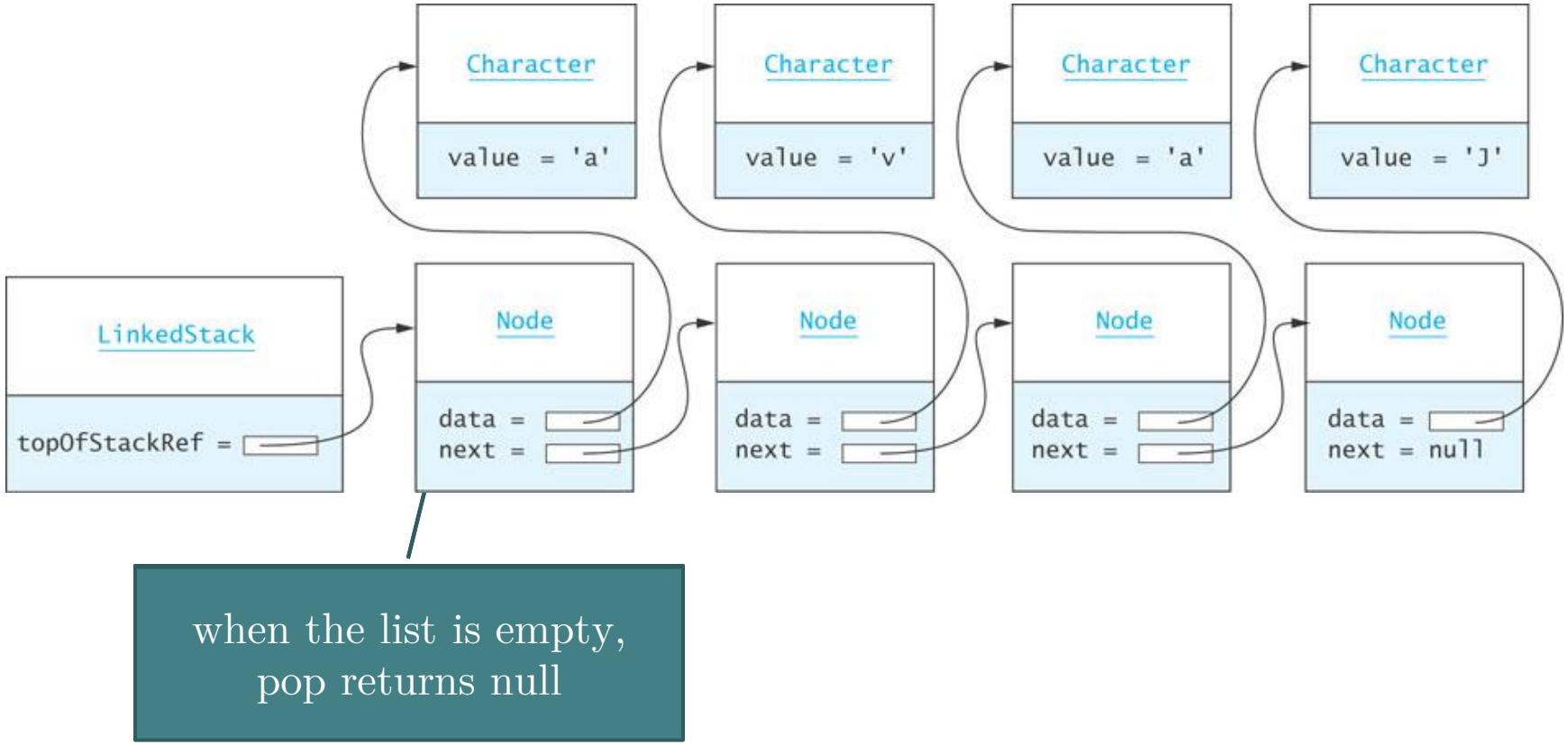

```
public E push(E obj) {
  if (topOfStack == theData.length - 1){
    reallocate();
  }
  topOfStack++;
  theData[topOfStack] = obj;
  return obj;
}
```

Implementing a Stack Using an Array (cont.)

```
@Override
public E pop() {
  if (empty()) {
    throw new EmptyStackException();
  }
  return theData[topOfStack--];
}
```

Implementing a Stack as a Linked Data Structure

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



Implementing a Stack as a Linked Data Structure (cont.)


```
import java.util.NoSuchElementException;

/** Class to implement interface StackInt<E> as a linked list.
*/
public class LinkedStack<E> implements StackInt<E> {
    private Node<E> topOfStackRef = null;

    public E push(E obj) {
        topOfStackRef = new Node<>(obj, topOfStackRef);
        return obj;
    }

    public E pop() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        } else {
            E result = topOfStackRef.data;
            topOfStackRef = topOfStackRef.next;
            return result;
        }
    }
    public E peek() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        } else {
            return topOfStackRef.data;
        }
    }
    public boolean isEmpty() {
        return topOfStackRef == null;
    }
}
```

Comparison of Stack Implementations

- Extending a `Vector` (as is done by Java) is a poor choice for stack implementation, since all `Vector` methods are accessible
- The easiest implementation uses a `List` component (`ArrayList` is the simplest) for storing data
  - An underlying array requires reallocation of space when the array becomes full, and
  - an underlying linked data structure requires allocating storage for links
  - As all insertions and deletions occur at one end, they are constant time, O(1), regardless of the type of implementation used

Additional Stack Applications

Section 3.4

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

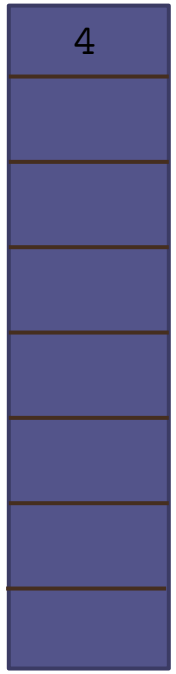
Postfix Expression	Infix Expression	Value
4 7 *	4 * 7	28
4 7 2 + *	4 * (7 + 2)	36
4 7 * 20 -	(4 * 7) - 20	8
3 4 7 * 2 / +	3 + ((4 * 7) / 2)	17

Evaluating Postfix Expressions

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

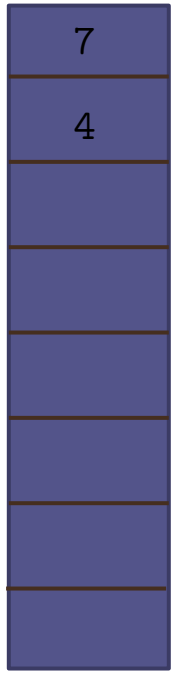
Data Field	Attribute
`Stack<Integer> operandStack`	The stack of operands (`Integer` objects).
Method	**Behavior**
`public int eval(String expression)`	Returns the value of `expression`.
`private int evalOp(char op)`	Pops two operands and applies operator `op` to its operands, returning the result.
`private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

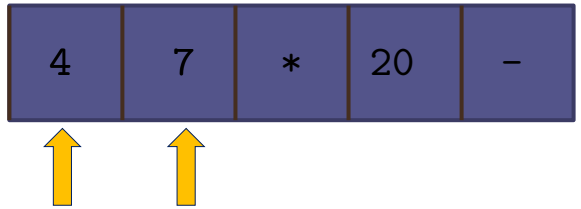
Evaluating Postfix Expressions (cont.)




```
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
```

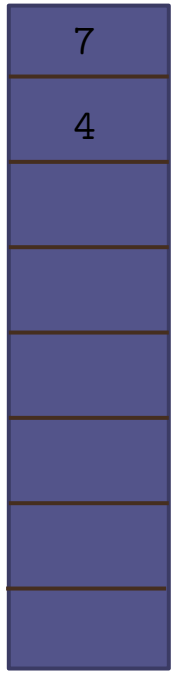
Evaluating Postfix Expressions (cont.)



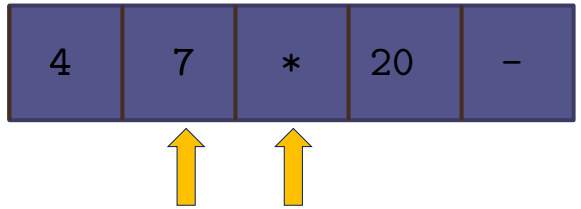



```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand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

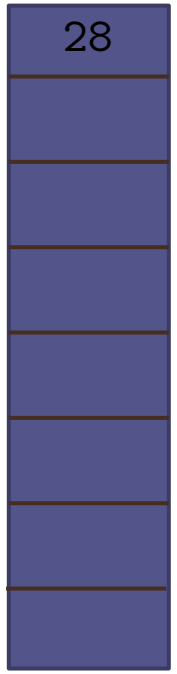


4 * 7

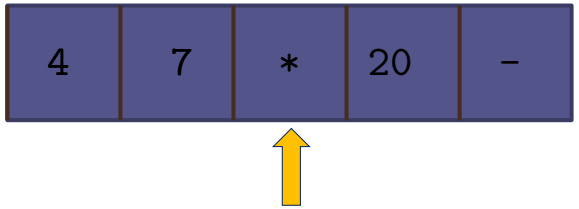



```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand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

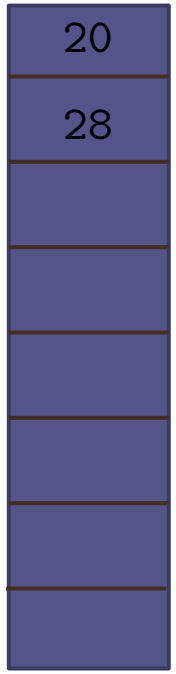


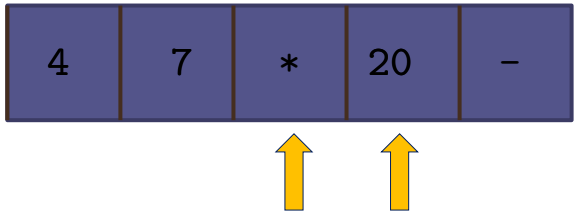
28




```
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
```

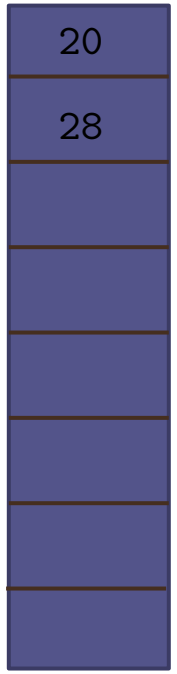
Evaluating Postfix Expressions (cont.)



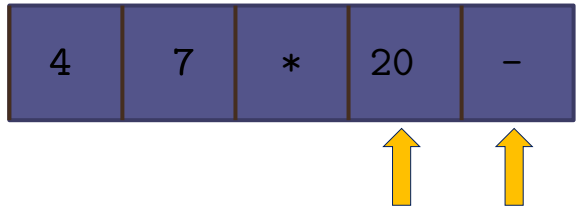



```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand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

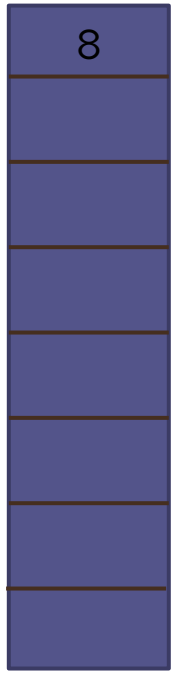


28 - 20




```
 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
```

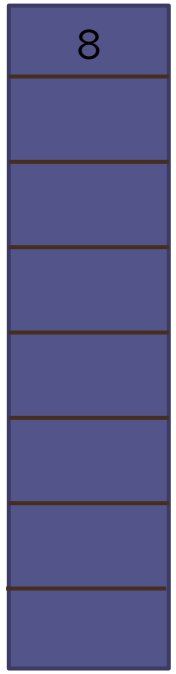
Evaluating Postfix Expressions (cont.)



8


```
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.)


```
import java.util.*;

/** Class that can evaluate a postfix expression. */
public class PostfixEvaluator {
        private static final String OPERATORS = "+-
*/";
private static int evalOp(char op, Deque<Integer>
operandStack) {
    // Pop the two operands off the stack.
    int rhs = operandStack.pop();
    int lhs = operandStack.pop();
    int result = 0;
    // Evaluate the operator.
    switch (op) {
    case '+' : result = lhs + rhs;
    break;
    case '-' : result = lhs - rhs;
    break;
    case '/' : result = lhs / rhs;
    break;
    case '*' : result = lhs * rhs;
    break;
    }
    return result;
    }
    private static boolean isOperator(char ch) {
    return OPERATORS.indexOf(ch) != -1;
    }
```

```
public static int eval(String expression) throws SyntaxErrorException {
    // Create an empty stack.
    Deque<Integer> operandStack = new ArrayDeque<>();
    // Process each token.
    String[] tokens = expression.split("\\s+");
    try {
        for (String nextToken : tokens) {
            char firstChar = nextToken.charAt(0);
            // Does it start with a digit?
            if (Character.isDigit(firstChar)) {
                // Get the integer value.
                int value = Integer.parseInt(nextToken);
                // Push value onto operand stack.
                operandStack.push(value);
            } // Is it an operator?
            else if (isOperator(firstChar)) {
                // Evaluate the operator.
                int result = evalOp(firstChar, operandStack);
                // Push result onto the operand stack.
                operandStack.push(result);
            } else {
                // Invalid character.
                throw new SyntaxErrorException("Invalid character encountered: " + firstChar);
            }
        } // End for.
          // No more tokens - pop result from operand stack.
```

Evaluating Postfix Expressions (cont.)


```
int answer = operandStack.pop();
            // Operand stack should be empty.
            if (operandStack.isEmpty()) {
                return answer;
            } else {
                // Indicate syntax error.
                throw new SyntaxErrorException("Syntax Err");
            }
        } catch (NoSuchElementException ex) {
            // Pop was attempted on an empty stack.
            throw new SyntaxErrorException("Syntax Err");
        }
    }
}
```

Evaluating Postfix Expressions (cont.)

- Testing: write a driver which
  - creates a `PostfixEvaluator` object
  - reads one or more expressions and report the result
  - catches `PostfixEvaluator.SyntaxErrorException`
  - exercises each path by using each operator
  - exercises each path through the method by trying different orderings and multiple occurrences of operators
  - tests for syntax errors:
    - an operator without any operands
    - a single operand
    - an extra operand
    - an extra operator
    - a variable name
    - the empty string

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.)

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 /

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 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

```
1.  if the operator stack is empty
2.      Push the current operator onto the stack.
    else
3.      Peek the operator stack and let topOp be the top operator.
4.      if the precedence of the current operator is greater than the
        precedence of topOp
5.          Push the current operator onto the stack.
        else
6.          while the stack is not empty and the precedence of the current
            operator is less than or equal to the precedence of topOp
7.              Pop topOp off the stack and append it to postfix.
8.              if the operator stack is not empty
9.                  Peek the operator stack and let topOp be the top
                    operator.
10.         Push the current operator onto the stack.
```

Converting from Infix to Postfix (cont.)

- Listing 3.7 (`InfixToPostfix.java`, pages 181 - 183)

Converting from Infix to Postfix (cont.)

- Testing
  - Use enough test expressions to satisfy yourself that the conversions are correct for properly formed input expressions
  - Use a driver to catch `InfixToPostfix.SyntaxErrorException`
- Listing 3.8 (`TestInfixToPostfix.java`, page 184)

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
- Listing 3.9 (`InfixToPostfixParens.java`, pages 186 - 188)