More Collections: Bags, Sets, Stacks, Maps

Lecture 4

Menu

- More Collections
- Bags and Sets
- Stacks and Applications
- Maps and Applications

Collections library

Interfaces:

- *Collection <E >*
  = Bag (most general)
- *List <E >*
  = ordered collection
- *Set <E >*
  = unordered, no duplicates
- *Stack<E>*
  ordered collection, limited access
  (add/remove at end)
- *Map <K, V >*
  = key-value pairs (or mapping)
- *Queue <E >*
  ordered collection, limited access
  (add at end, remove from front)

Classes

- List classes:
  ArrayList, LinkedList,
- Set classes:
  HashSet, TreeSet, …
- Stack classes:
  ArrayStack, LinkedStack
- Map classes:
  HashMap, TreeMap, …
- …

Bags

- A Bag is a collection with
  - no structure or order maintained
  - no access constraints (access any item any time)
  - duplicates allowed
- Minimal Operations:
  - **add**(value) → returns true *iff* a collection was changed
  - **remove**(value) → returns true *iff* a collection was changed
  - **contains**(value) → returns true *iff* value is in bag
    uses equal to test.
  - **findElement**(value) → returns a matching item, *iff* in bag
- Plus
  - **size**(), **isEmpty**(), **iterator**(),
    clear(), **addAll**(collection), **removeAll**(collection),
    containsAll(collection), …

Bag Applications

- When to use a Bag?
  - When there is no need to order a collection, and duplicates are possible:
    - A collection of current logged-on users (can there be dups?)
    - The books in a book collection (can there be dups?)
    - …
- There are no standard implementations of Bag!!

Set ADT

- Set is a collection with:
  - no structure or order maintained
  - no access constraints (access any item any time)
  - Only property is that duplicates are excluded
- Operations:

  (Same as Bag, but different behaviour)
  - add(value) → true *iff* value *was added (ie, no duplicate)*
  - remove(value) → *true iff value removed (was in set)*
  - contains(value) → *true iff value is in the set*
  - findElement(value) → *matching item, iff value is in the set*
  - …
- Sets are as common as Lists

Stack

- Organizes entries according to the order in which added
- Additions are made to one end, the top
- The item most recently added is always on the top

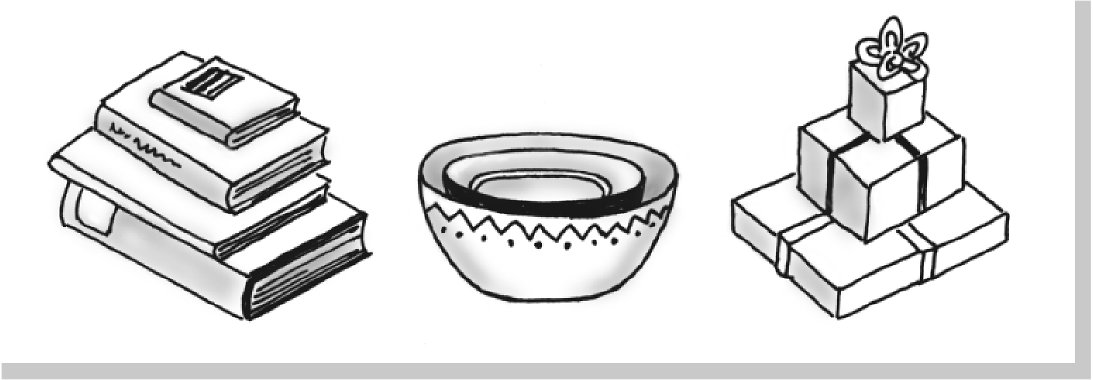


Fig. Some familiar stacks

Stack example

Fig. A stack of strings after

(a) push adds *Jim*;

(b) push adds *Jess*;

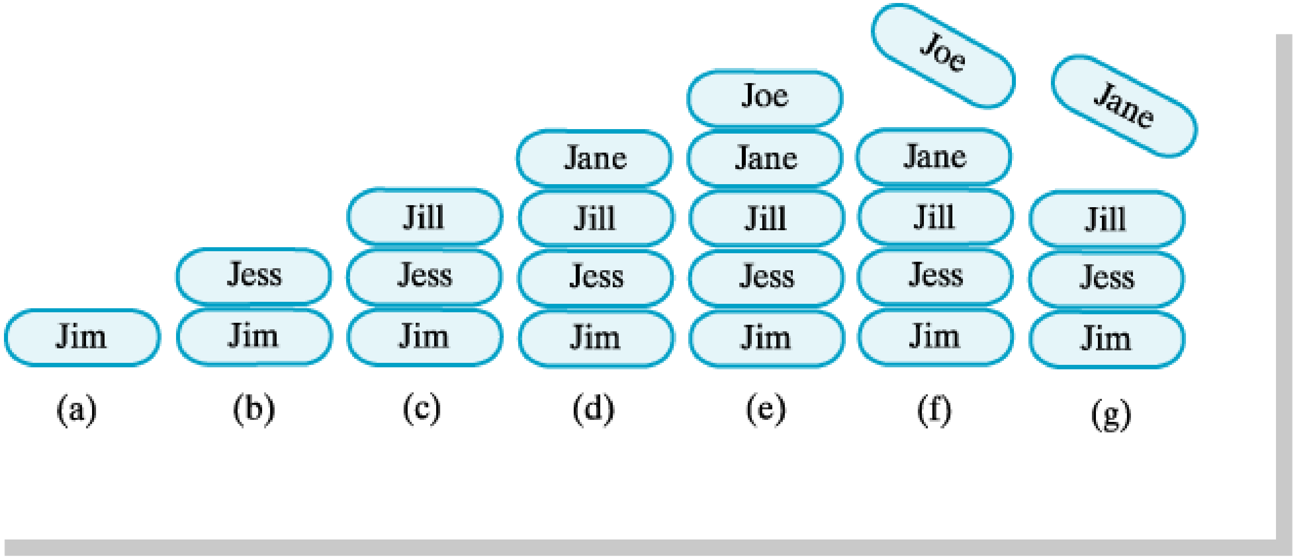
(c) push adds *Jill*;

(d) push adds *Jane*;

(e) push adds *Joe*;

(f) pop retrieves and removes *Joe*;

(g) pop retrieves and removes *Jane*



Stacks

- Stacks are a special kind of List:
  - Sequence of values, ('sequence' means?)
  - Constrained access: add, get, and remove only from one end.
  - There exists a Stack interface and different implementations of it (ArrayStack, LinkedStack, etc)
  - In Java Collections library:
    - Stack is a class that implements List
    - Has extra operations: **push**(value), **pop**(), **peek**()

- push(value): Put value on top of stack

- pop(): Removes and returns top of stack
- peek(): Returns top of stack, *without* removing

- plus the other List operations

Applications of Stacks

- Processing files of structured (nested) data.
  - E.g. reading files with structured markup (HTML, XML,…)

- Program execution, e.g. working on subtasks, then returning to previous task.

- Undo in editors.

- Expression evaluation,
  - (6 + 4) * ((12.1 *sin(15)) – (cos(20) / 38))

HTML & XML examples

- **HTML example**

```
<html>
<body>
The content of the body element is displayed in your browser.
</body>
</html>
```

- **XML examples**

```
<Person>
        <name>Henry Ford</name>
</Person>

<Book>
        <title>My Life and Work</title>
        <author>Henry Ford</author>
</Book>
```

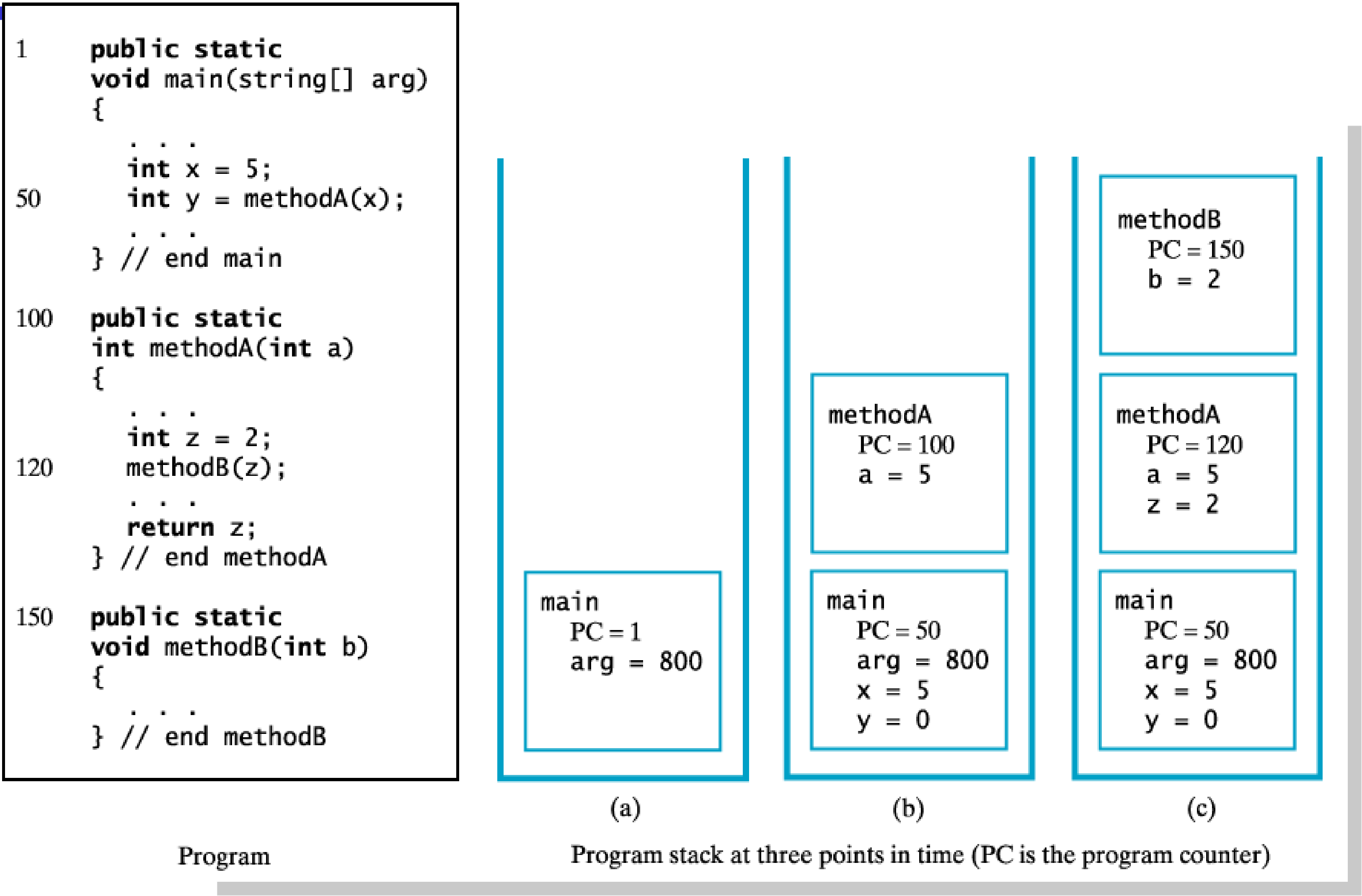
How do we make sure XML/HTML tags in a web document is properly nested?

The Program Stack for Program Execution

- When a method is called
  - Runtime environment creates activation record
  - Shows method's state during execution

- Activation record pushed onto the program stack (Java stack)
  - Top of stack belongs to currently executing method
  - Next record down the stack belongs to the one that called current method

The Program Stack



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

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

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

Program

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

Fig. The program stack at 3 points in time; (a) when **main** begins execution; (b) when **methodA** begins execution, (c) when **methodB** begins execution.

Recursive Methods

- A recursive method making many recursive calls
  - Places many activation records in the program stack
  - Explains why recursive methods can use much memory

- Possible to replace recursion with iteration by using a stack

Stack for evaluating expressions

- (6 + 4) * ((12.1 *sin(15)) – (cos(20) / 38))

- How does it work?

Using a Stack to Process Algebraic Expressions

- Checking for Balanced Parentheses, Brackets, and Braces in an Infix Algebraic Expression

- Transforming an Infix Expression to a Postfix Expression
- Evaluating Postfix Expressions

- Evaluating Infix Expressions

Using a Stack to Process Algebraic Expressions

- **Infix expressions**
  - Binary operators appear between operands
  - `a + b`
- **Prefix expressions**
  - Binary operators appear before operands
  - `+ a b`
- **Postfix expressions**
  - Binary operators appear after operands
  - `a b +`
  - Easier to process – no need for parentheses nor precedence (why?)

Checking for Balanced `(), [], {}`

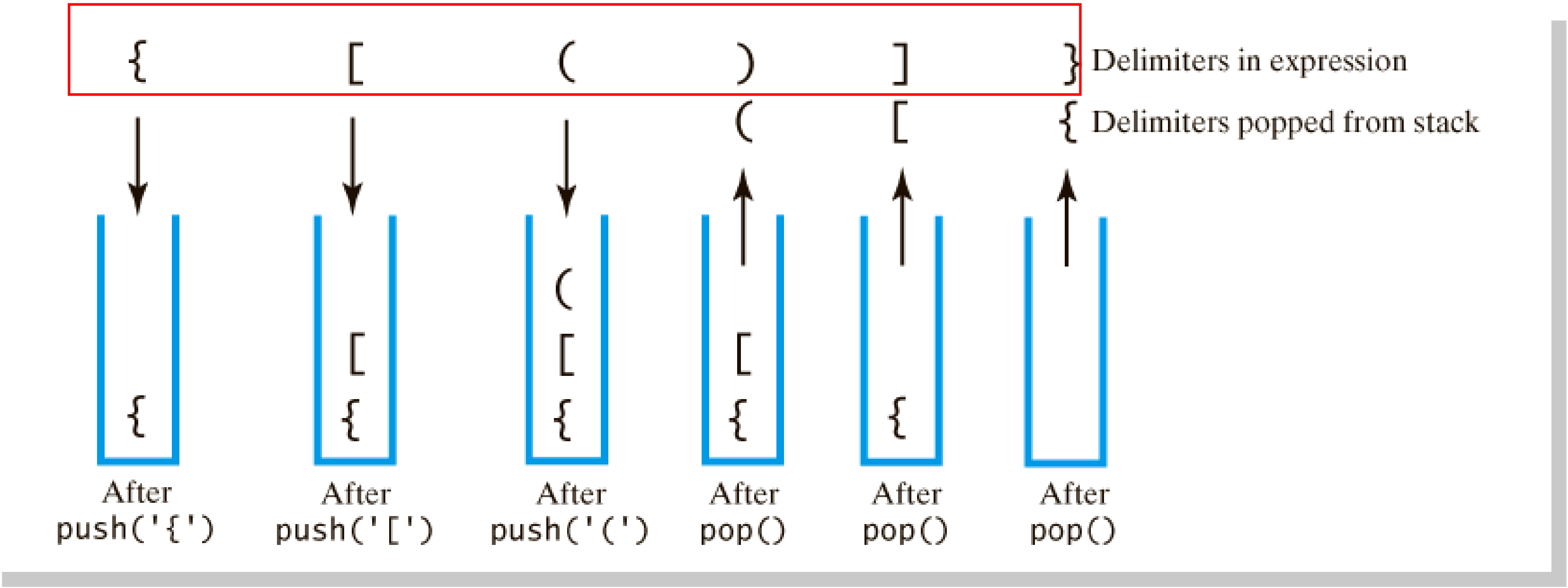


Fig. The contents of a stack during the scan of an expression that contains the balanced delimiters **`{[()]}`**

Checking for Balanced `()`, `[]`, `{}`

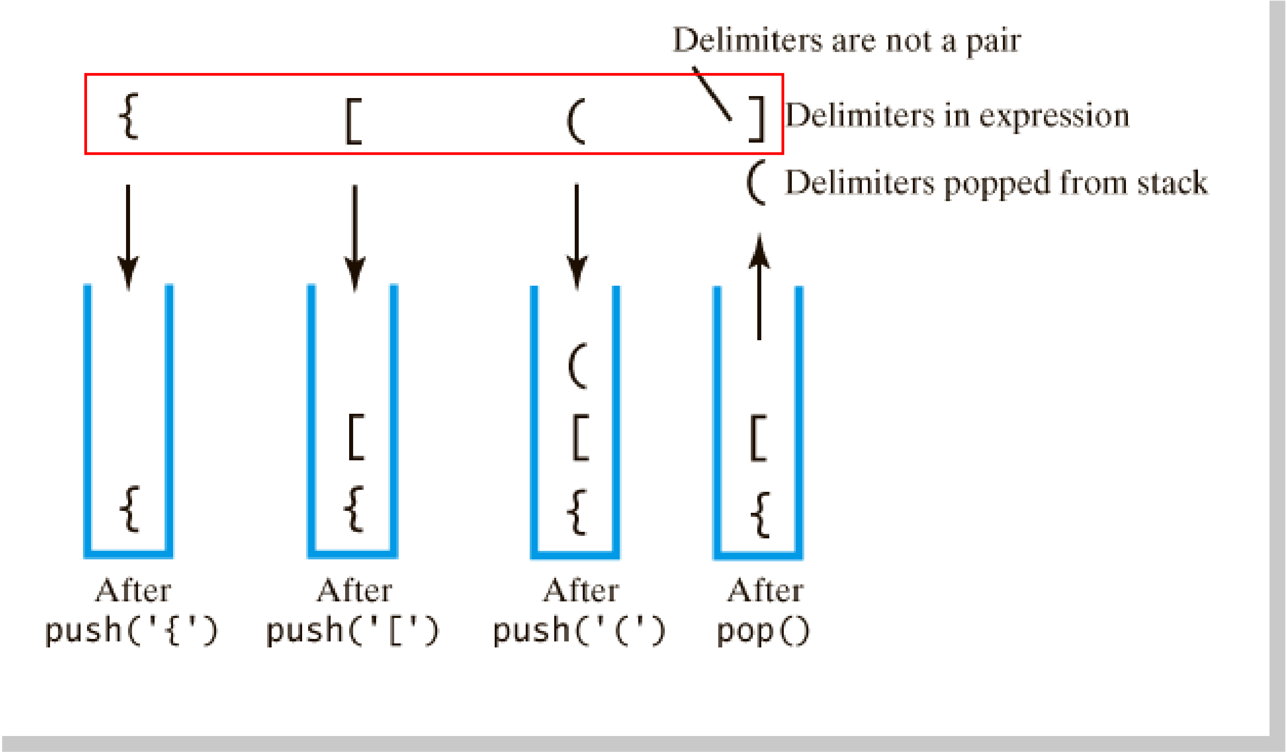


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters **`{[(])}`**

Q&A: Checking for Balanced (), [], {}

Show stack contents

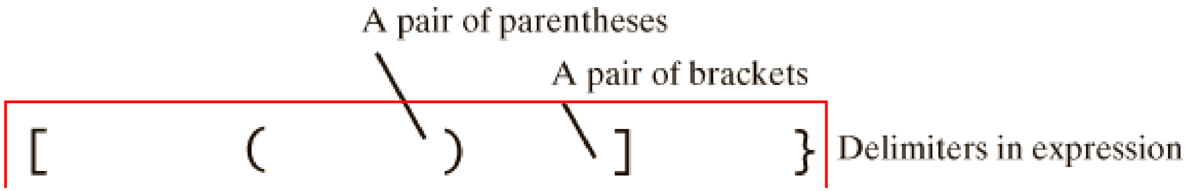


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters **[( )] }**

Checking for Balanced (), [], {}

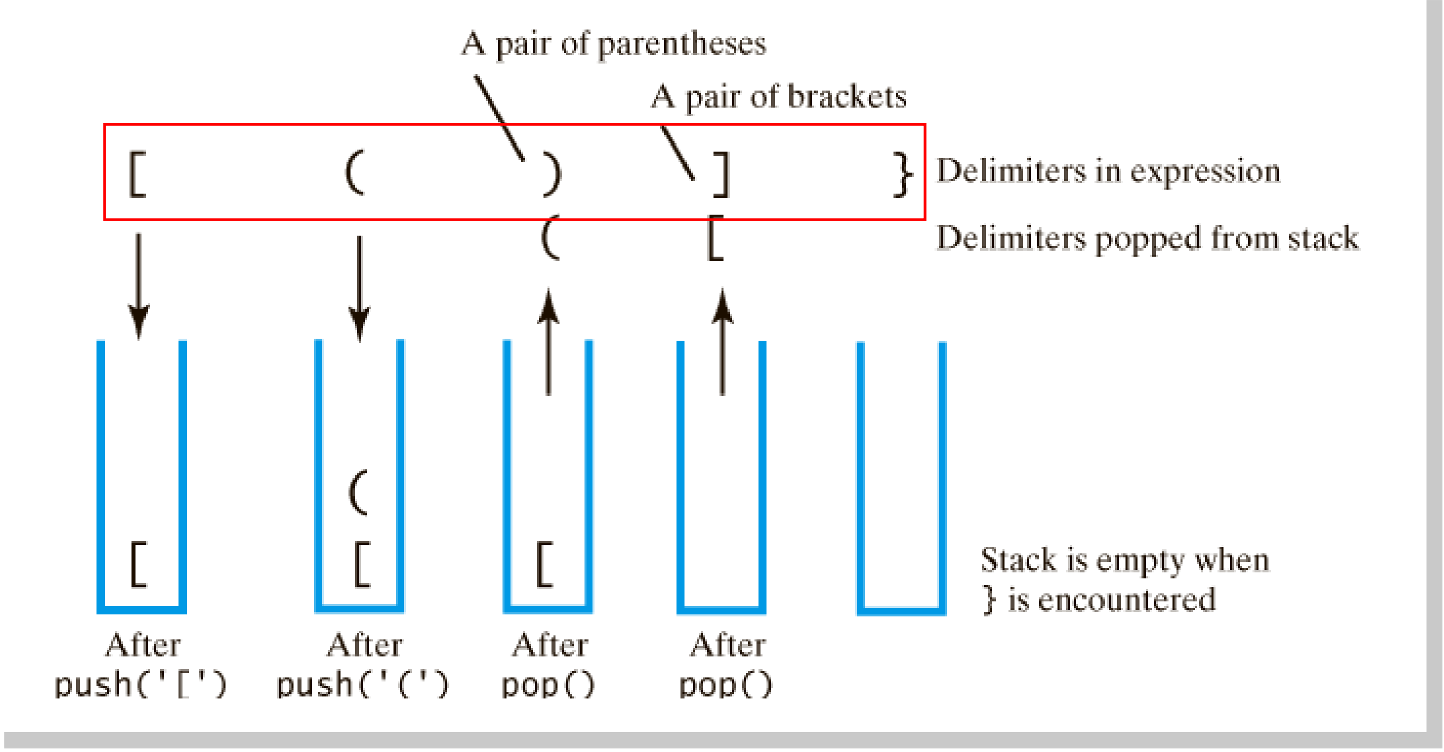


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters **[()]}**

Q&A: Checking for Balanced (), [], {}

Show stack contents

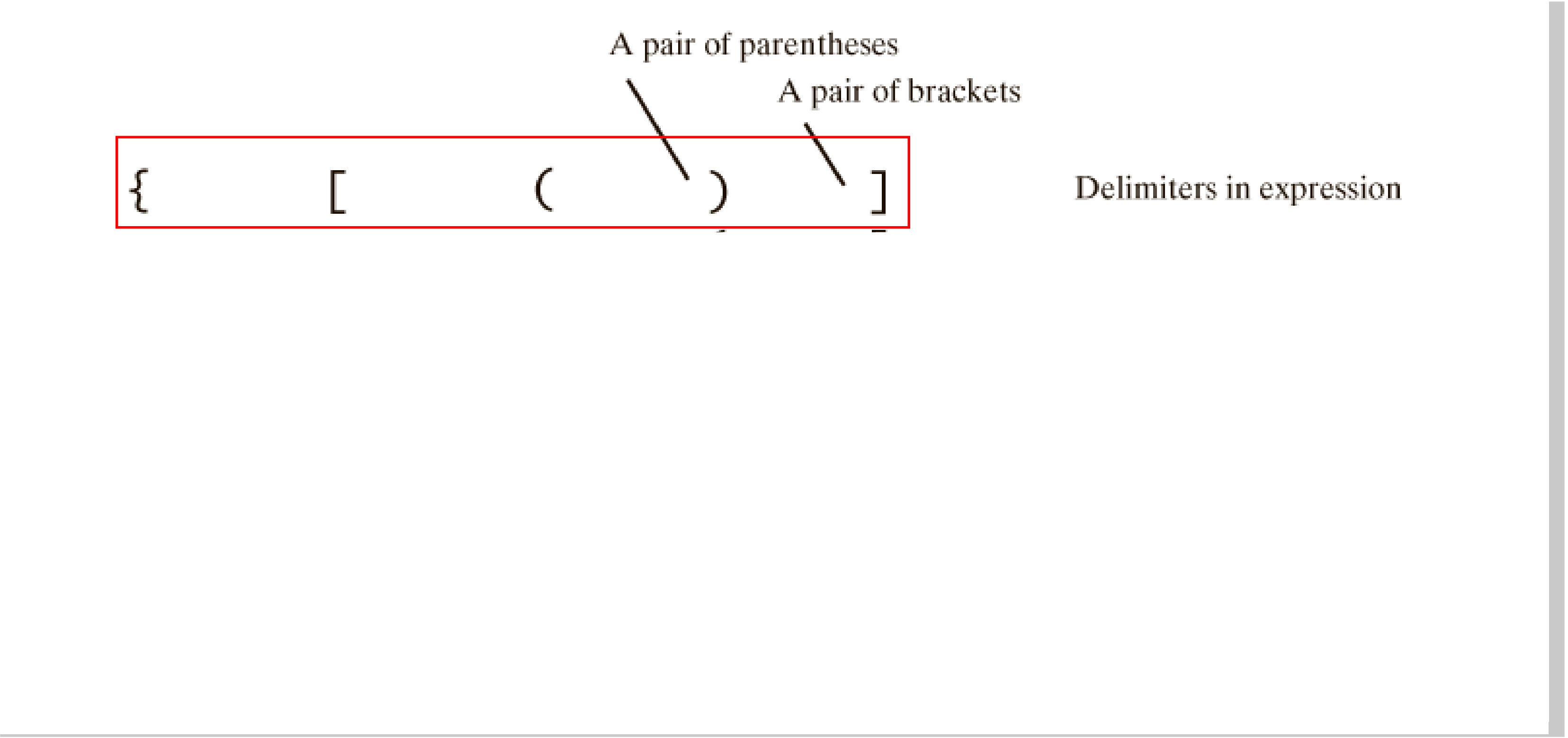


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( )]

Checking for Balanced `()`, `[]`, `{}`

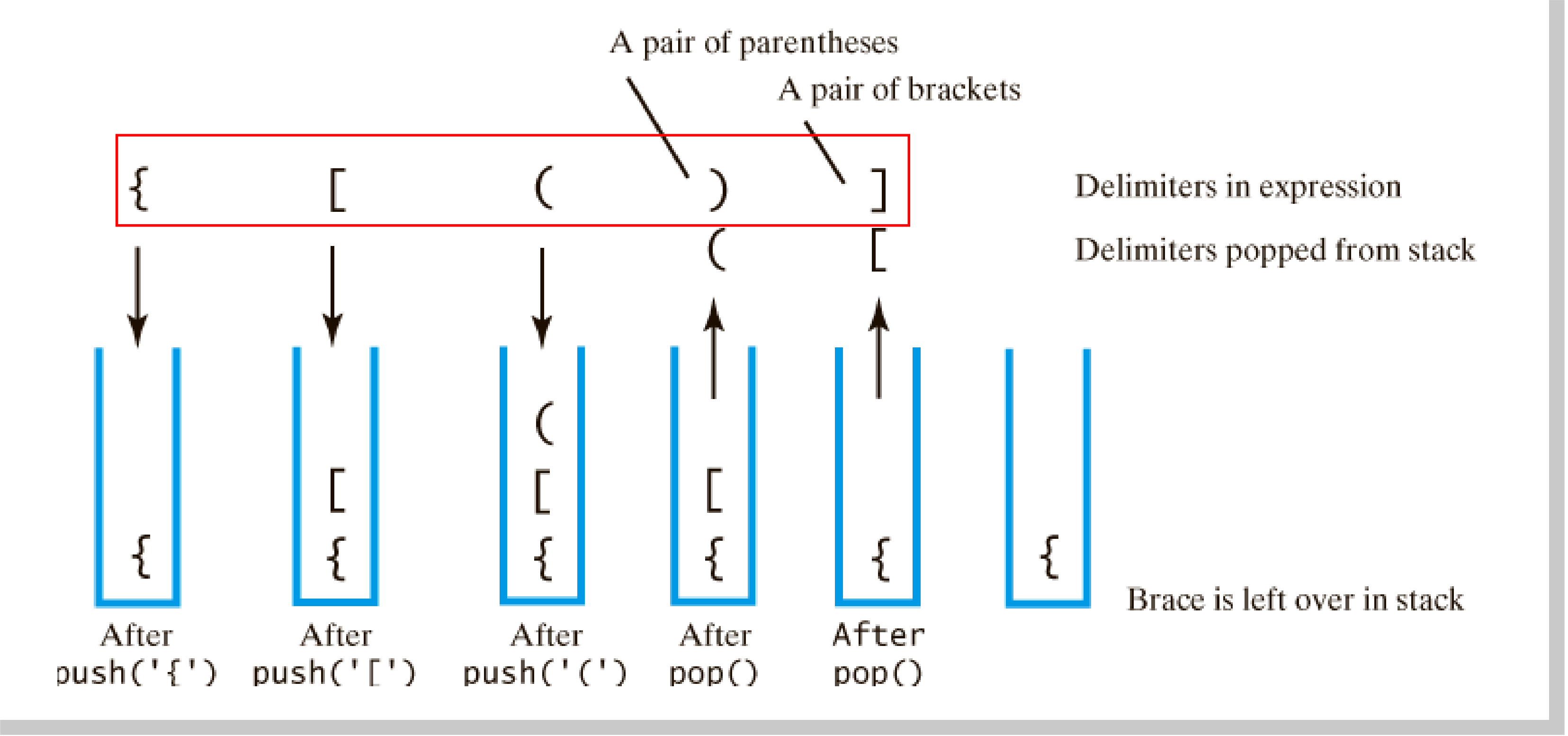


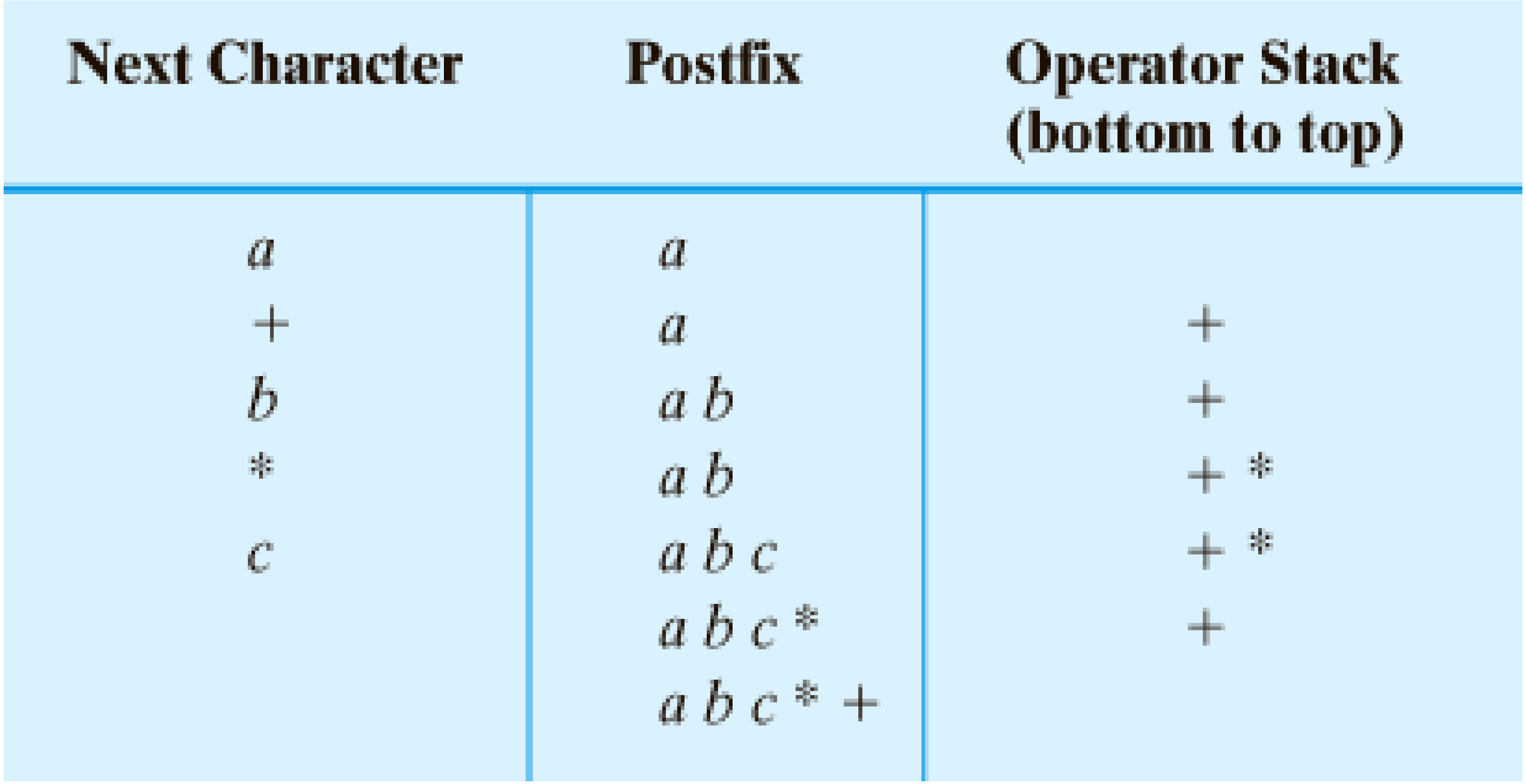
Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[()]`

Checking for Balanced ( ), [ ], { }

```
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
                  {     openDelimiter = top 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
```

Exercise: rewrite this algorithm in pseudo code

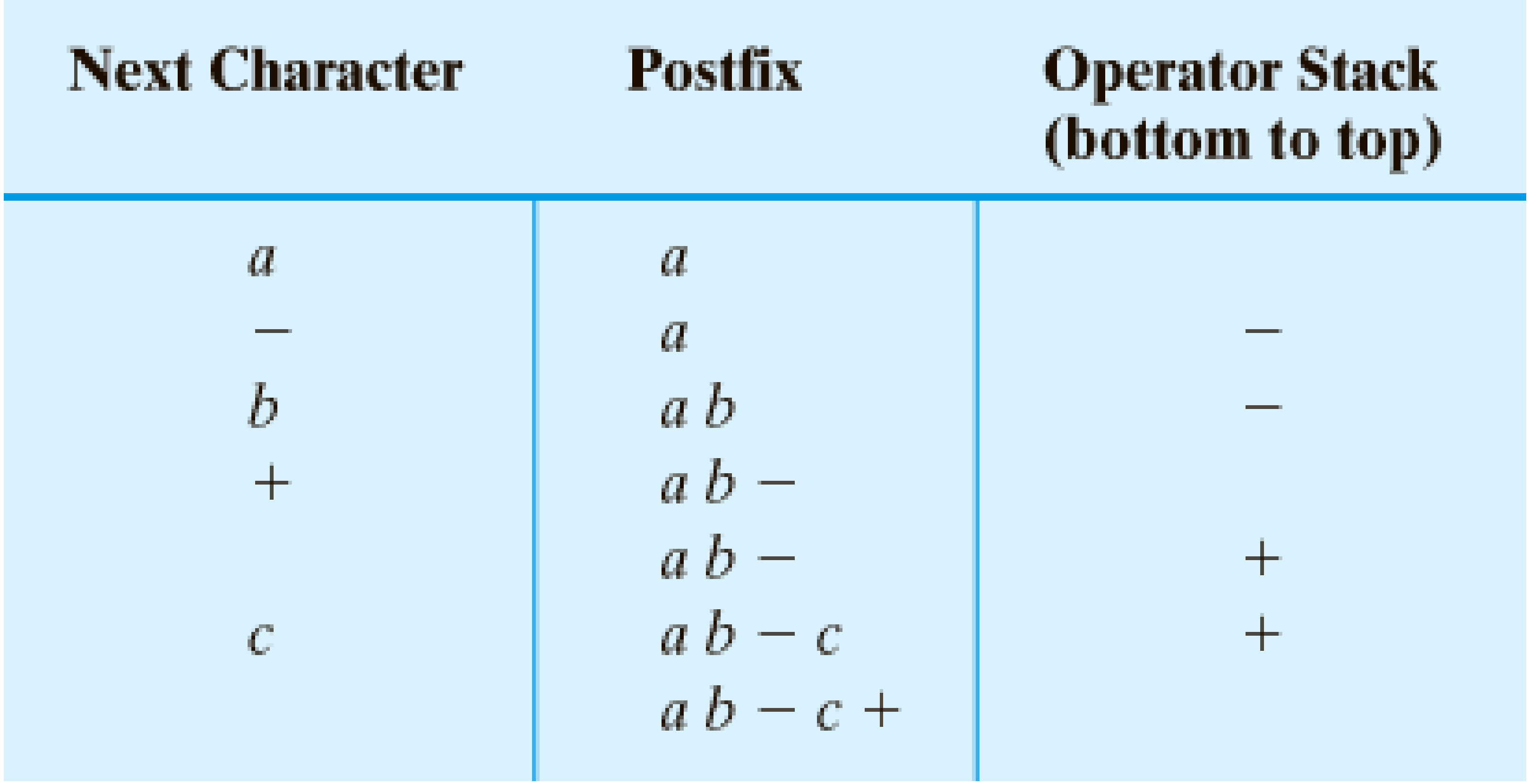
Transforming Infix to Postfix



Next Character	Postfix	Operator Stack (bottom to top)
a	*a*	
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

Fig. Converting the infix expression
a + b * c to postfix form a b c * +

Transforming Infix to Postfix

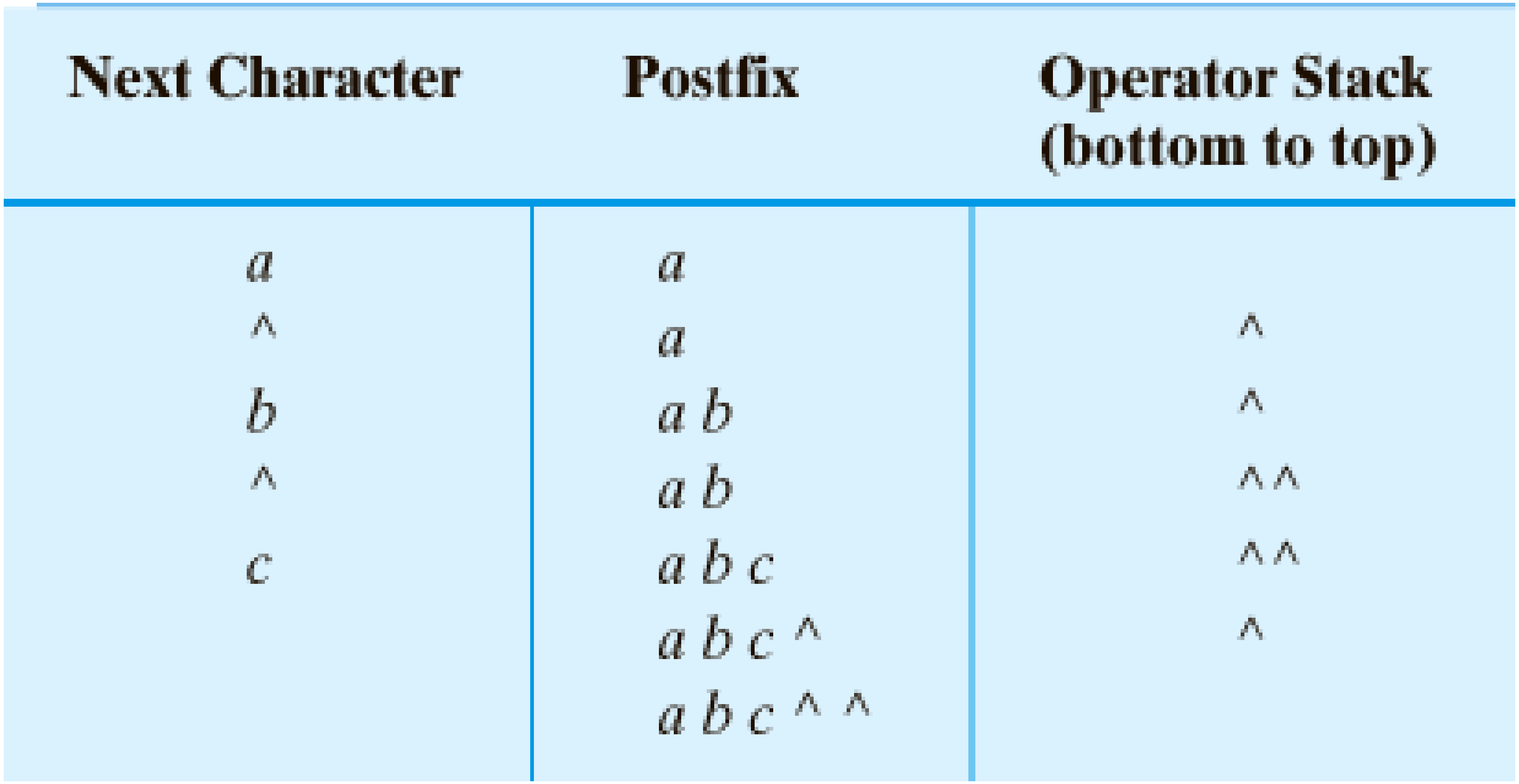


Next Character	Postfix	Operator Stack (bottom to top)
a	a	
$-$	a	$-$
b	$a\ b$	$-$
$+$	$a\ b\ -$	
	$a\ b\ -$	$+$
c	$a\ b\ -\ c$	$+$
	$a\ b\ -\ c\ +$	

Fig. Converting infix expression **a – b + c**

to postfix form: **a b – c +**

Transforming Infix to Postfix



Next Character	Postfix	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^ ^
c	*a b c*	^ ^
	a b c ^	^
	a b c ^ ^	

Fig. Converting infix expression **a ^ b ^ c** to postfix form: **a b c ^ ^**

Infix-to-Postfix Algorithm

Symbol in Infix	Action
Operand	Append to end of output expression
Operator ^	Push ^ onto stack
Operator +,-, *, or /	Pop operators from stack, append to output expression until stack empty or top has lower precedence than new operator. Then push new operator onto stack
Open parenthesis	Push (onto stack
Close parenthesis	Pop operators from stack, append to output expression until we pop a matching open parenthesis. Discard both parentheses.

Exercise

- Convert infix to postfix & show stack contents:

- (a+n)*(b-8*m)
- b/v^7
- {3+[d-7*(g+5)]/w}
- [(4+b]-2)

Evaluating Postfix Expression

Infix expression: **a/b** is converted into postfix expression: **ab/**

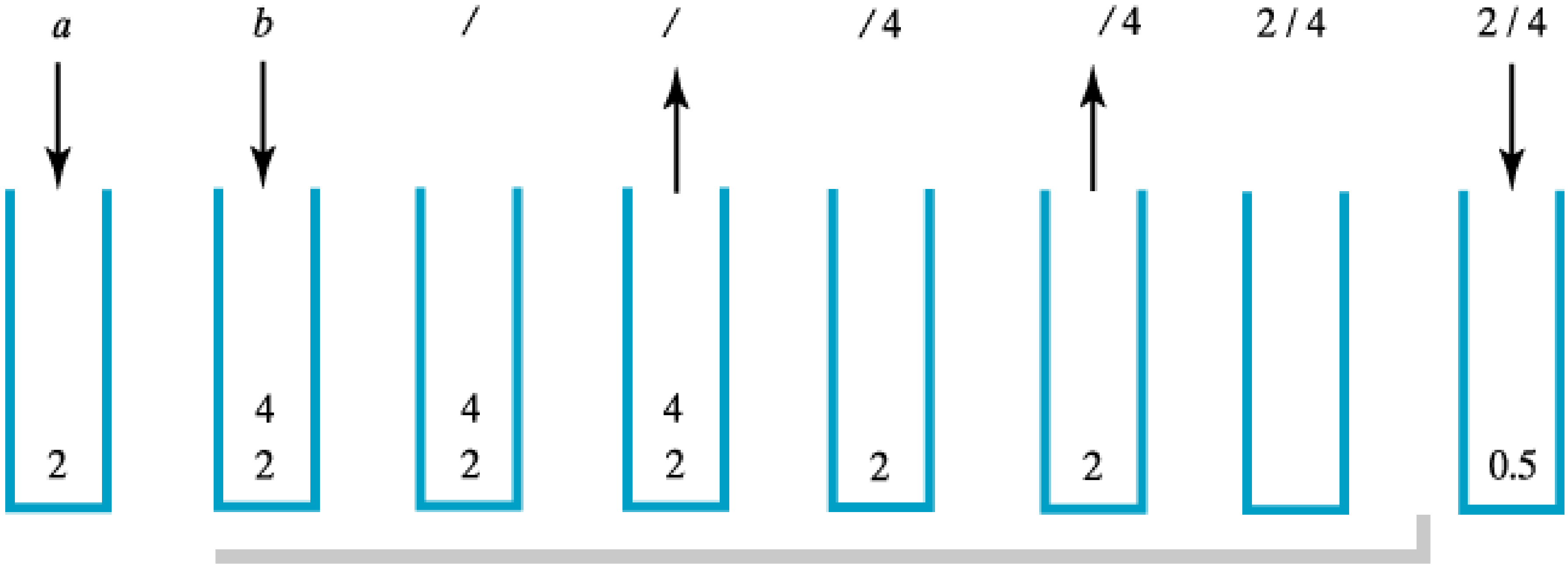


Fig. The stack during the evaluation of the postfix expression `a b /` when `a` is 2 and `b` is 4

Q&A: Evaluating Postfix Expression: show stack contents

Infix expression **(a+b)/c** is converted into the postfix expression **a b + c /**

Fig. 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 Expression

Infix expression **(a+b)/c** is converted into the postfix expression **a b + c /**

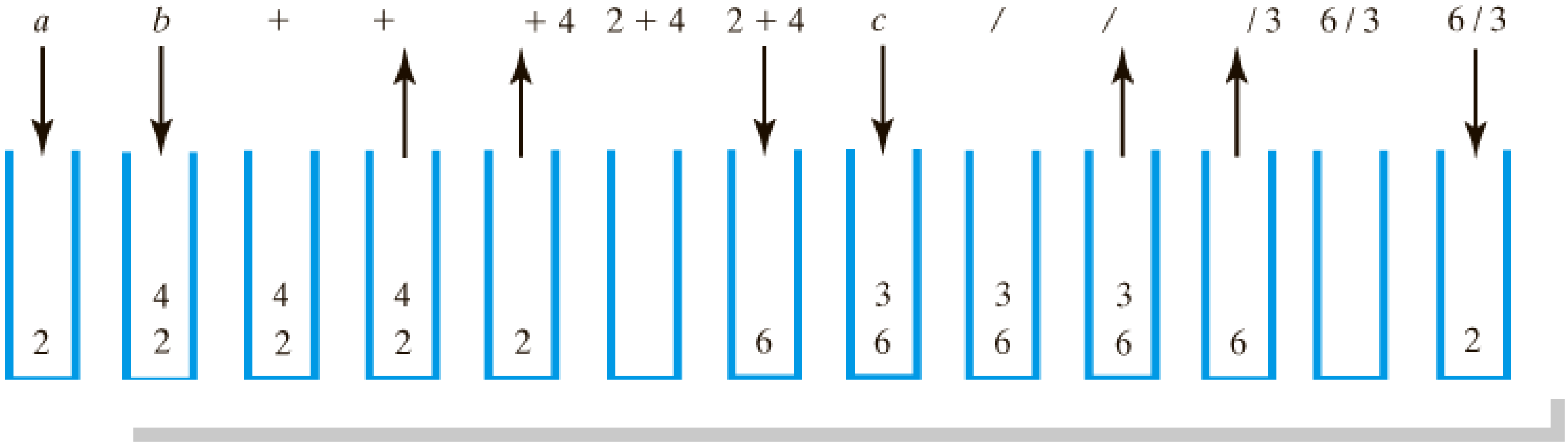


Fig. The stack during the evaluation of the postfix expression `a b + c /` when `a` is 2, `b` is 4 and `c` is 3

Evaluating Postfix Expression

```
Algorithm evaluatePostfix(postfix)      // Evaluates a postfix expression.
valueStack = a new empty stack
while (postfix has characters left to parse)
{    nextCharacter = next nonblank character of postfix
     switch (nextCharacter)
     {    case variable:
               valueStack.push(value of the variable nextCharacter)
               break
          case '+': case '-': case '*': case '/': case '^':
               operandTwo = valueStack.pop()
               operandOne = valueStack.pop()
               result = the result of the operation in nextCharacter and its
               operands   operandOne and operandTwo
               valueStack.push(result)
               break
          default: break
     }
}
return valueStack.peek()                          /* does not check errors in postfix */
```

Exercise:

add error processing to the pseudo code of Postfix Expression Evaluation

Evaluating Infix Expressions using Two Stacks

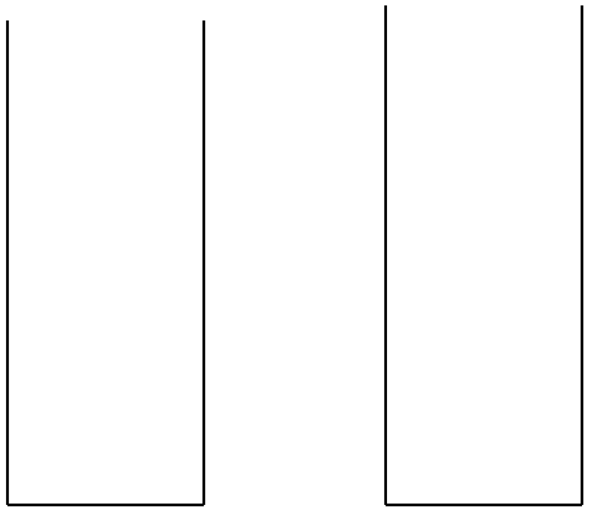


Fig. Two stacks during evaluation of **a + b * c** when **a = 2, b = 3, c = 4;** (a) after reaching end of expression; (b) while performing multiplication; (c) while performing the addition

Evaluating Infix Expressions using Two Stacks

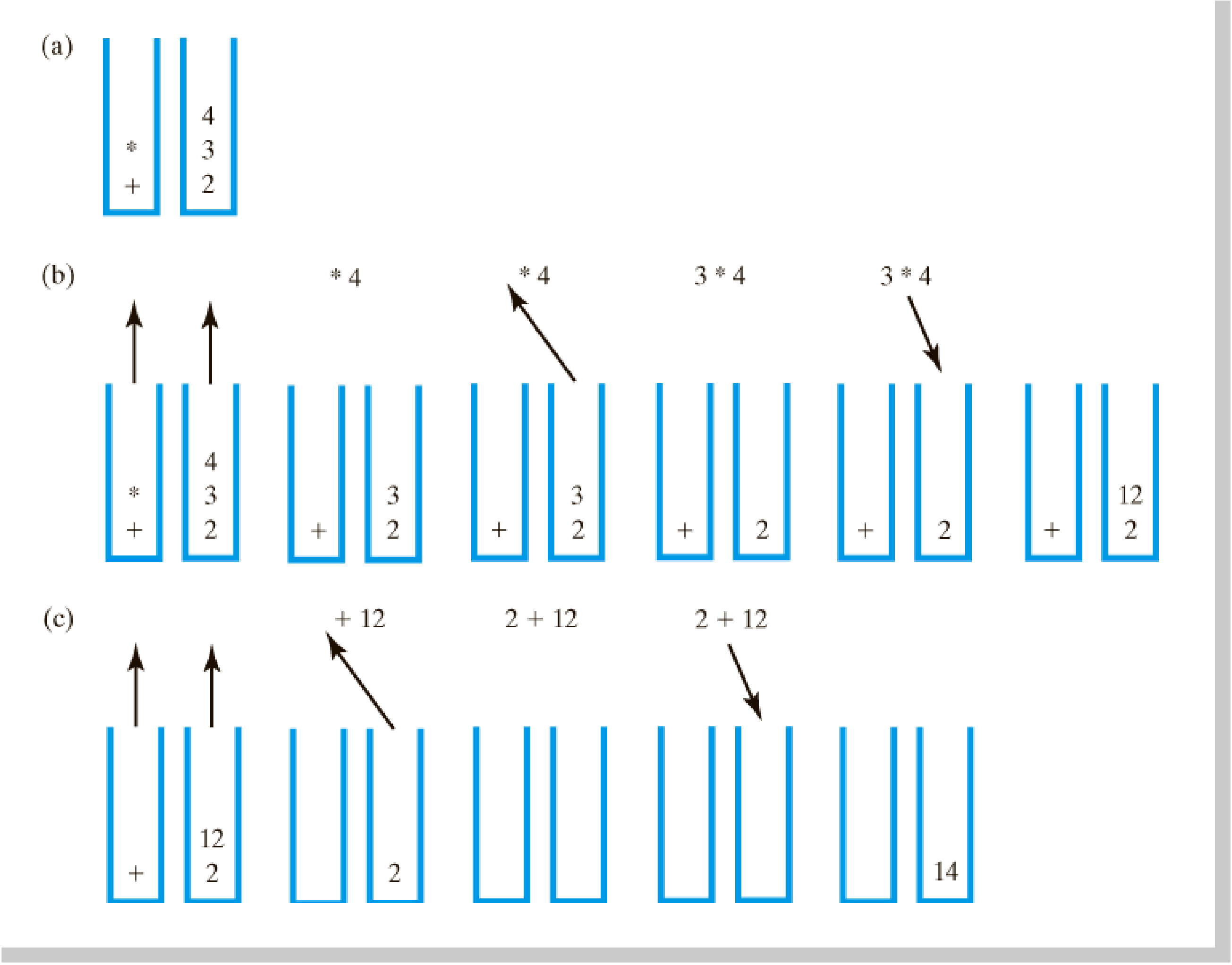


Fig. Two stacks during evaluation of **a + b * c** when **a = 2, b = 3, c = 4;** (a) after reaching end of expression; (b) while performing multiplication; (c) while performing the addition

Ex. Try infix evaluation with dual stacks on the following: a * b + c when **a = 2, b = 3, c = 4**

Java Class Library: The Class `Stack`

- Methods in class **`Stack`** in **`java.util`**

```
public void push(Object item);
public Object pop();
public Object peek();
public boolean isEmpty();
public void clear();
```

Q&A: What should be included in a Java Stack Interface Spec ?

Spec of the ADT Stack

- Specification of a stack of objects

```
public interface StackInterface
{       /** Task: Adds a new entry to the top of the stack.
        * @param newEntry an object to be added to the stack */
        public void push(Object newEntry);

        /** Task: Removes and returns the top of the stack.
        * @return either the object at the top of the stack or null if  the stack was
empty */
        public object pop();

        /** Task: Retrieves the top of the stack.
        * @return either the object at the top of the stack or null if the stack is
empty */
        public object peek();

        /** Task: Determines whether the stack is empty.
        * @return true if the stack is empty */
        public boolean isEmpty();

        /** Task: Removes all entries from the stack */
        public void clear();
} // end StackInterface
```

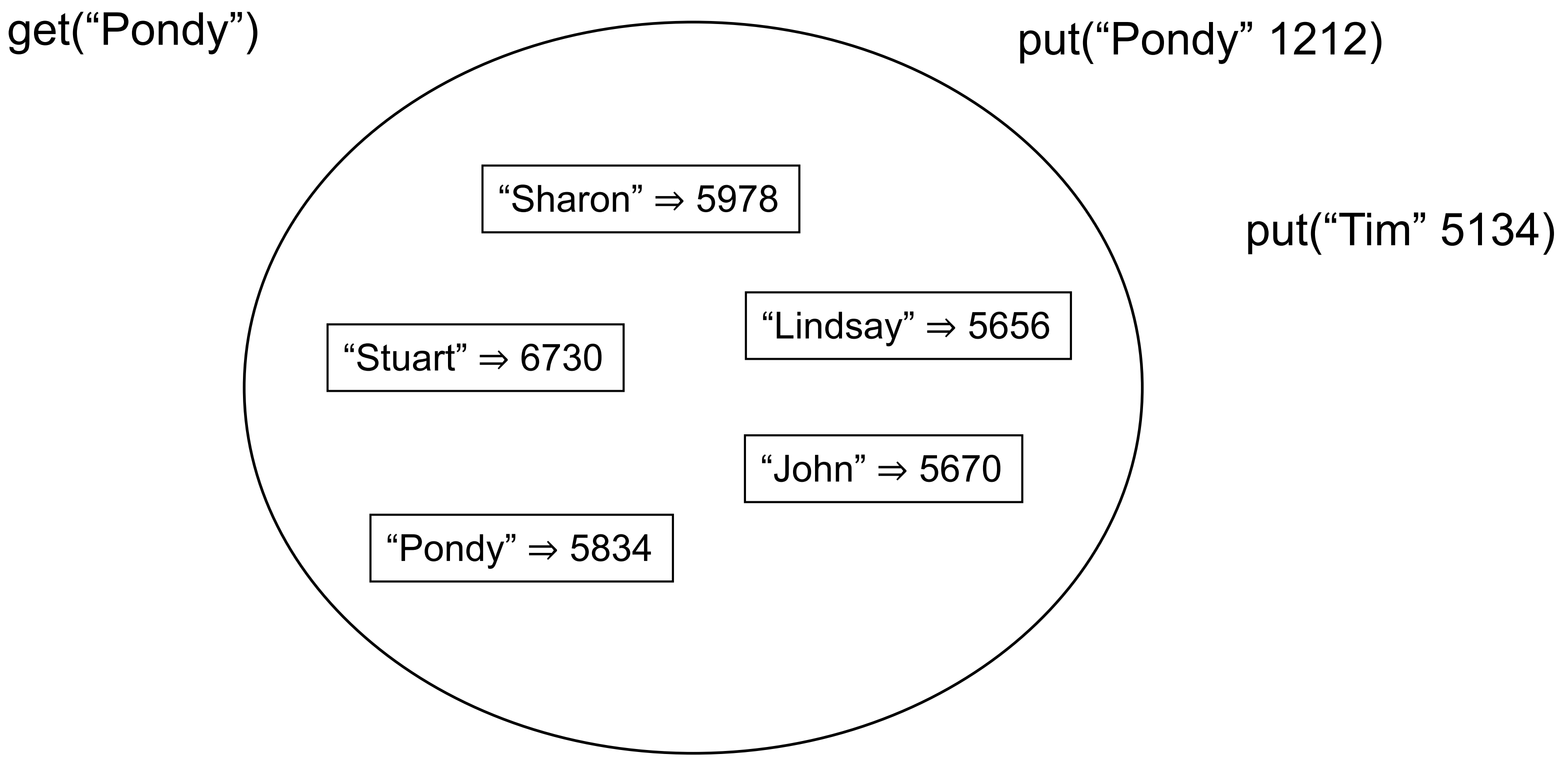
Stacks Example

- Reversing the items from a file:
  - Read and push onto a stack
  - Pop them off the stack

```
public void reverseNums(Scanner sc){
   Stack<Integer> myNums = new ArrayStack<Integer>();
   while (sc.hasNext())
      myNums.push(sc.Next())
   while (! myNums.isEmpty())
      textArea.append(myNums.pop() + “\n”);
}
```

Maps

- Collection of data, but not of single values:
  - Map = **Set** of pairs of keys to values
  - Constrained access: get values via keys.
  - **No duplicate keys**
  - Lots of implementations, most common is **HashMap**.



get(“Pondy”)

put(“Pondy” 1212)

put(“Tim” 5134)

“Sharon” ⇒ 5978

“Lindsay” ⇒ 5656

“Stuart” ⇒ 6730

“John” ⇒ 5670

“Pondy” ⇒ 5834

Maps

- When declaring and constructing, must specify two types:
  - Type of the key, and type of the value

```
private Map<String, Integer> phoneBook;
   :
phoneBook = new HashMap<String, Integer>();
```

- Central operations:
  - get(key), → returns value associated with key (or null)
  - put(key, value), → sets the value associated with key (and returns the old value, if any)
  - remove(key), → removes the key *and* associated value (and returns the old value, if any)
  - containsKey(key), → boolean
  - size()

Example of using Map

- Find the highest frequency word in a file

  ⇒ must count frequency of every word.

  ie, need to associate a count (int) with each word (String)

  ⇒ use a Map of word–count pairs:

- Two Steps:
  - construct the counts of each word: countWords(file) → map
  - find the highest count findMaxCount(map) → word

System.out.println(findMaxCount(countWords(file)));

Example of using Map – in pseudocode

```
/** Construct histogram of counts of all words in a file */
public Map<String, Integer> countWords(Scanner sc){
    // construct a new map
    // for each word in the file
    //     if word is in the map, increment its count
    //     else, add it to the map with a count of 1
    // return map
}

/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
    // for each word in map
    //    if has higher count than current max, record it
    // return current max word
}
```

Example of using Map

```
/** Construct histogram of counts of all words in a file */
public Map<String, Integer> countWords(Scanner scan){
    Map<String, Integer>  counts = new HashMap<String, Integer> ();
    for (String word : scan){
       if ( counts.containsKey(word) )
            counts.put(word, counts.get(word)+1);
       else
            counts.put(word, 1);
    return counts;
}

/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
    // for each word in map
    //    if has higher count than current max, record it
    // return current max word
}
```

Iterating through a Map

- How do you iterate through a Map? (eg, to print it out)
  - A Map isn't just a collection of items!

    ⇒ could iterate through the collection of keys

    ⇒ could iterate through the collection of values

    ⇒ could iterate through the collection of pairs

- Java Map allows all three!
  - keySet() → Set of all keys
    - **for** (String name : phonebook.keySet()){….
  - values() → Collection of all values
    - **for** (Integer num : phonebook.values()){….
  - entrySet() → Set of all Map.Entry's
    - **for** (Map.Entry<String, Integer> entry : phonebook.entrySet()){….

      … entry.getKey() …

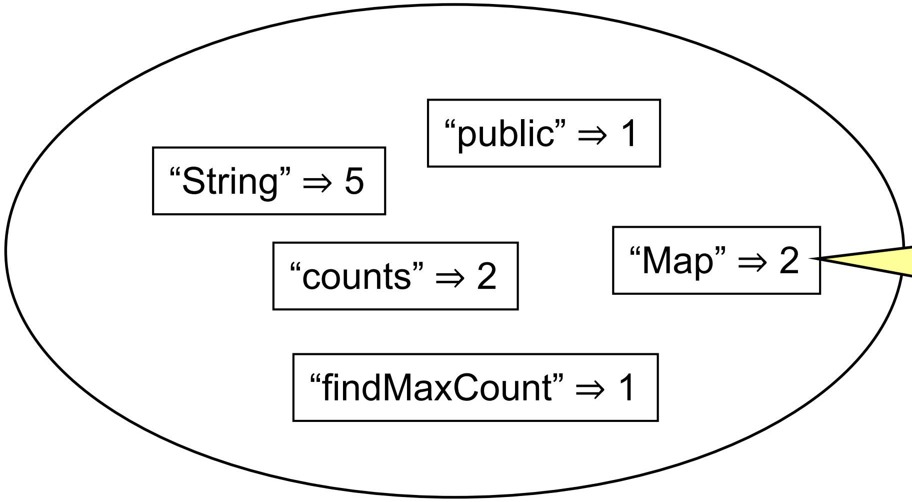
      … entry.getValue()…

Iterating through Map: keySet

```
/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
    String maxWord = null;
    int maxCount = -1;
    for (String word : counts.keySet() ){
      int count = counts.get(word);
      if (count > maxCount){
          maxCount = count;
          maxWord = word;
      }
    }
    return maxWord;
}
```

Iterating through Map: entrySet

```
public String findMaxCount(Map<String, Integer> counts){
    String maxWord = null;
    int maxCount = -1;
    for (Map.Entry<String, Integer> entry : counts.entrySet() ){
       if (entry.getValue() > maxCount){
           maxCount = entry.getValue();
           maxWord = entry.getKey();
       }
    }
    return maxWord;
}
```



Map.Entry<K,V>
- getKey()
- getValue()

Summary

- More Collections
- Bags and Sets
- Stacks and Applications
- Maps and Applications

Readings

- [Mar07] Read 3.6, 4.8
- [Mar13] Read 3.6, 4.8