Required Actions for Next Lecture on Tuesday, Oct. 29

Source: https://imgflip.com/memetemplate/122501176/Baby-question

Actions Needed for Next Lecture

- Please prepare 1-min pitch of the final project
- The pitch should include answers to following questions:
  - What is the problem you want to address?
  - What is your idea?
  - Who are the target users?
  - Why is it a good idea?
- We will run an activity on Tuesday (10/29). Please be prepared!!!

Northeastern University

Lecture 16: Stacks - 1

Prof. Chen-Hsiang (Jones) Yu, Ph.D.

College of Engineering

Materials are edited by Prof. Jones Yu from

Data Structures and Abstractions with Java, 5th edition. By Frank M. Carrano and Timothy M. Henry.
ISBN-13 978-0-13-483169-5 © 2019 Pearson Education, Inc.

Stacks

- Add item on top of stack
- Remove item that is topmost
  - Last In, First Out (LIFO)

Specifications of the ADT Stack

- Data:
  - A collection of objects in reverse chronological order and having the same data type.
- Operations:
  - Push
  - Pop
  - Peak
  - Clear
  - Check if it is empty

Specifications of the ADT Stack (cont.)

Pseudocode	UML	Description
push(newEntry)	**+push(newEntry: T): void**	Task: Adds a new entry to the top of the stack. Input: newEntry is the new entry. Output: None.
pop()	**+pop(): T**	Task: Removes and returns the stack's top entry. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty before the operation.
peek()	**+peek(): T**	Task: Retrieves the stack's top entry without changing the stack in any way. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty.
isEmpty()	**+isEmpty(): boolean**	Task: Detects whether the stack is empty. Input: None. Output: Returns true if the stack is empty.
clear()	**+clear(): void**	Task: Removes all entries from the stack. Input: None. Output: None.

Design Decision

- When stack is empty
  - What to do with `pop` and `peek`?
- Possible actions
  - Assume that the ADT is not empty
  - Return `null`
  - Throw an exception (which type?)

Interface for the ADT Stack

```
/** An interface for the ADT stack. */
public interface StackInterface<T>
{
   /** Adds a new entry to the top of this stack.
       @param newEntry  An object to be added to the stack. */
   public void push(T newEntry);

   /** Removes and returns this stack's top entry.
       @return  The object at the top of the stack.
       @throws  EmptyStackException if the stack is empty before the operation. */
   public T pop();

   /** Retrieves this stack's top entry.
       @return  The object at the top of the stack.
       @throws  EmptyStackException if the stack is empty. */
   public T peek();

   /** Detects whether this stack is empty.
       @return  True if the stack is empty. */
   public boolean isEmpty();

   /** Removes all entries from this stack. */
   public void clear();
} // end StackInterface
```

Listing 5-1: An interface for the ADT stack

Example

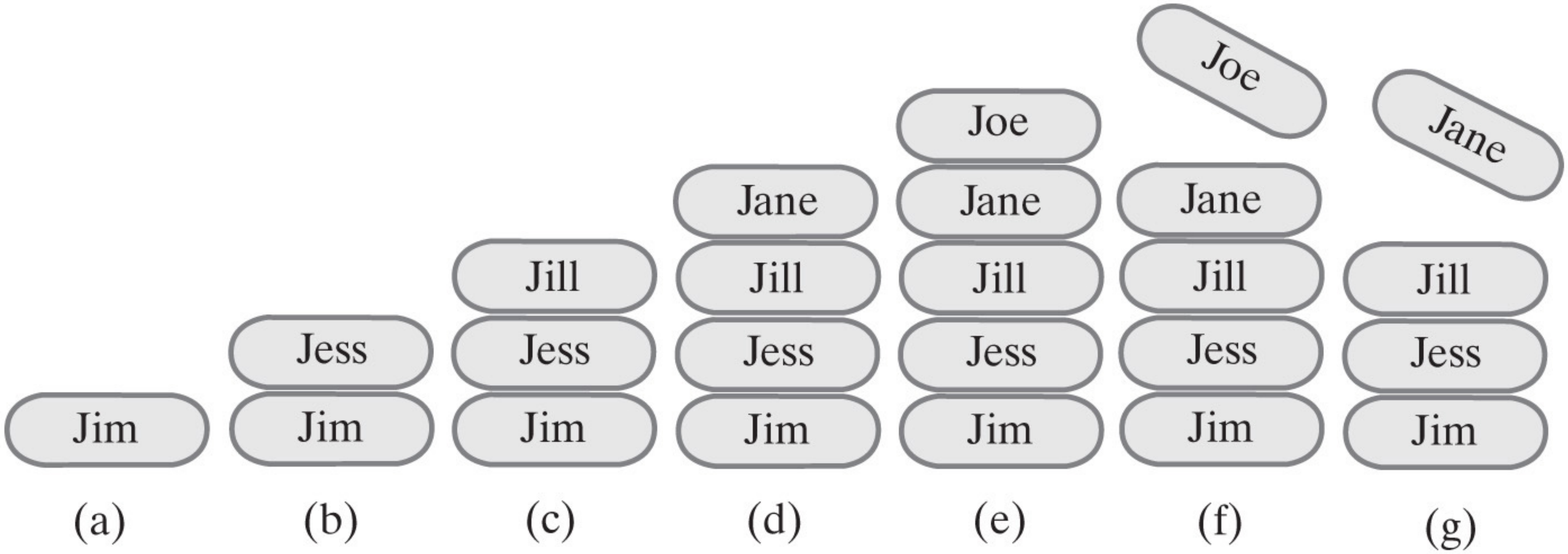

Figure 5-2: A stack of strings after (a) push adds Jim; (b) push adds Jess; (c) push adds Jill; (d) push adds Jane; (e) push adds Joe; (f) pop retrieves and removes Joe; (g) pop retrieves and removes Jane

Security Note

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

Exercise

- After the following statements execute, what string is at the top of the stack and what string is at the bottom?

```
StackInterface<String> stringStack = new OurStack<>();

stringStack.push(“Iron Man”);
stringStack.push(“Captain America”);
stringStack.pop();
stringStack.push(“Hulk”);
stringStack.push(“Black Widow”);
stringStack.push(“Thor”);
stringStack.pop();
```

Answer

- *Black Widow* is at the top and *Iron Man* is at the bottom.

Using a Stack to Process Algebraic Expression

Processing Algebraic Expressions

- Infix: each binary operator appears between its operands *a* + *b*

- Prefix: each binary operator appears before its operands + *a* *b*

- Postfix*:* each binary operator appears after its operands *a* *b* +

Processing Algebraic Expressions (cont.)

- Balanced expressions
  - Delimiters are paired correctly

{ [( )] }

{ [( ]) }

Processing Algebraic Expressions

{ [( )] }

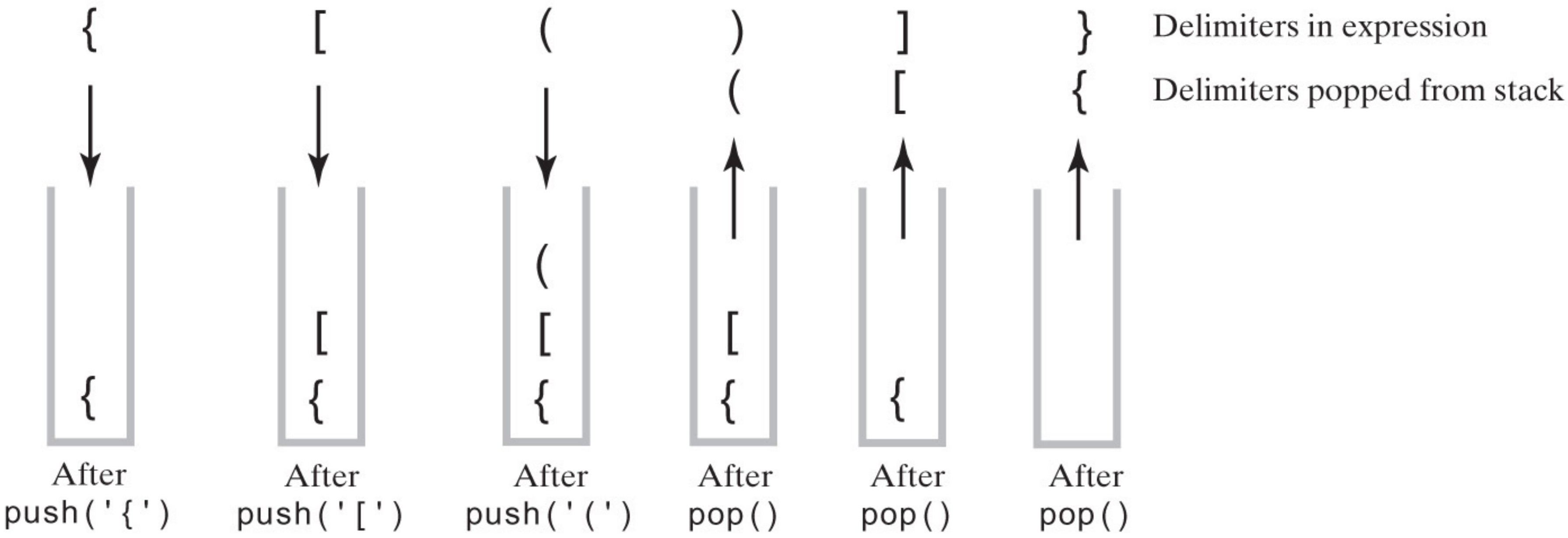

Figure 5-3: The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }

Processing Algebraic Expressions

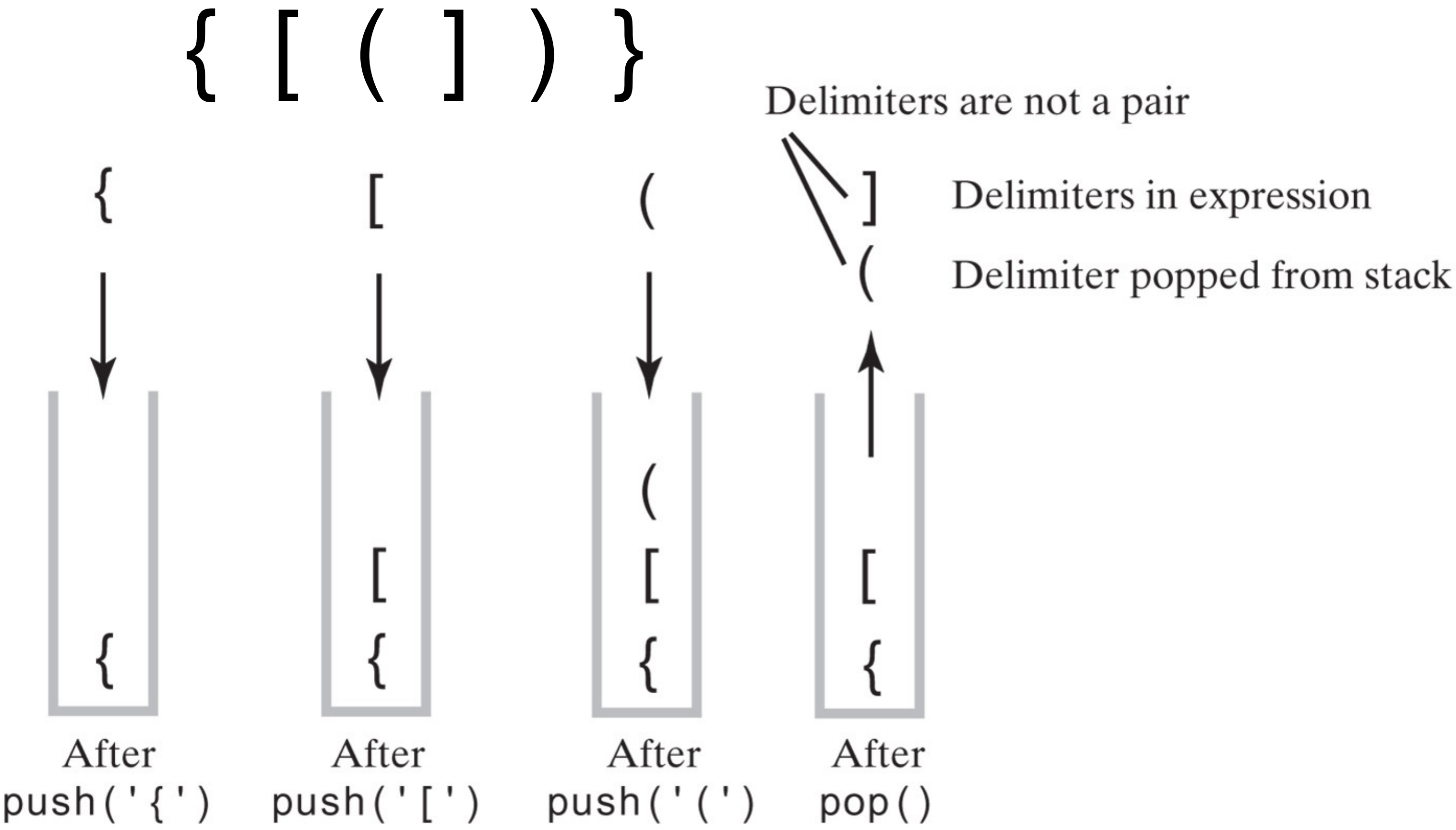

FIGURE 5-4 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( ]) }

Processing Algebraic Expressions

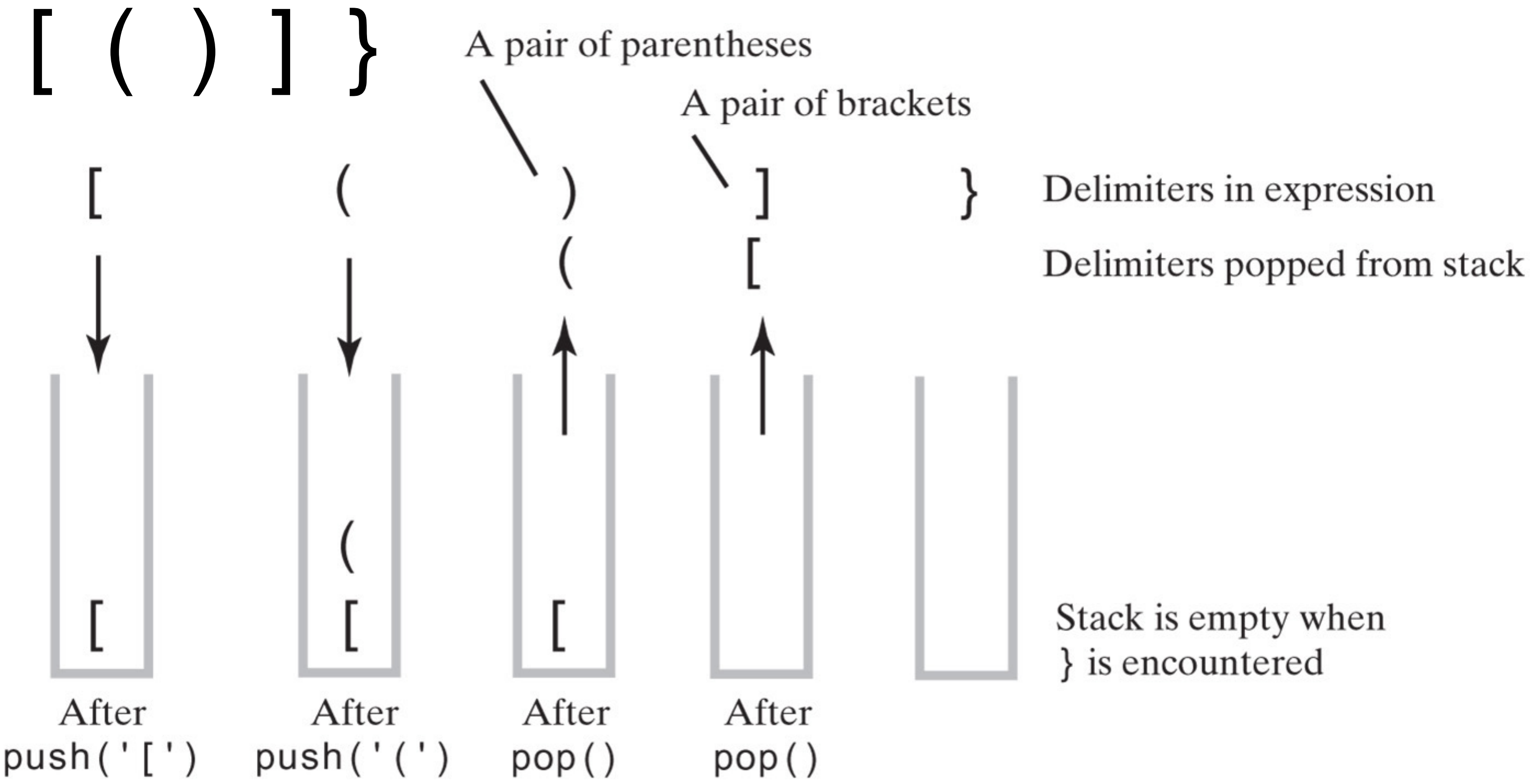

Figure 5-5: The contents of a stack during the scan of an expression that contains the unbalanced delimiters [( )] }

Processing Algebraic Expressions

{ [( )]

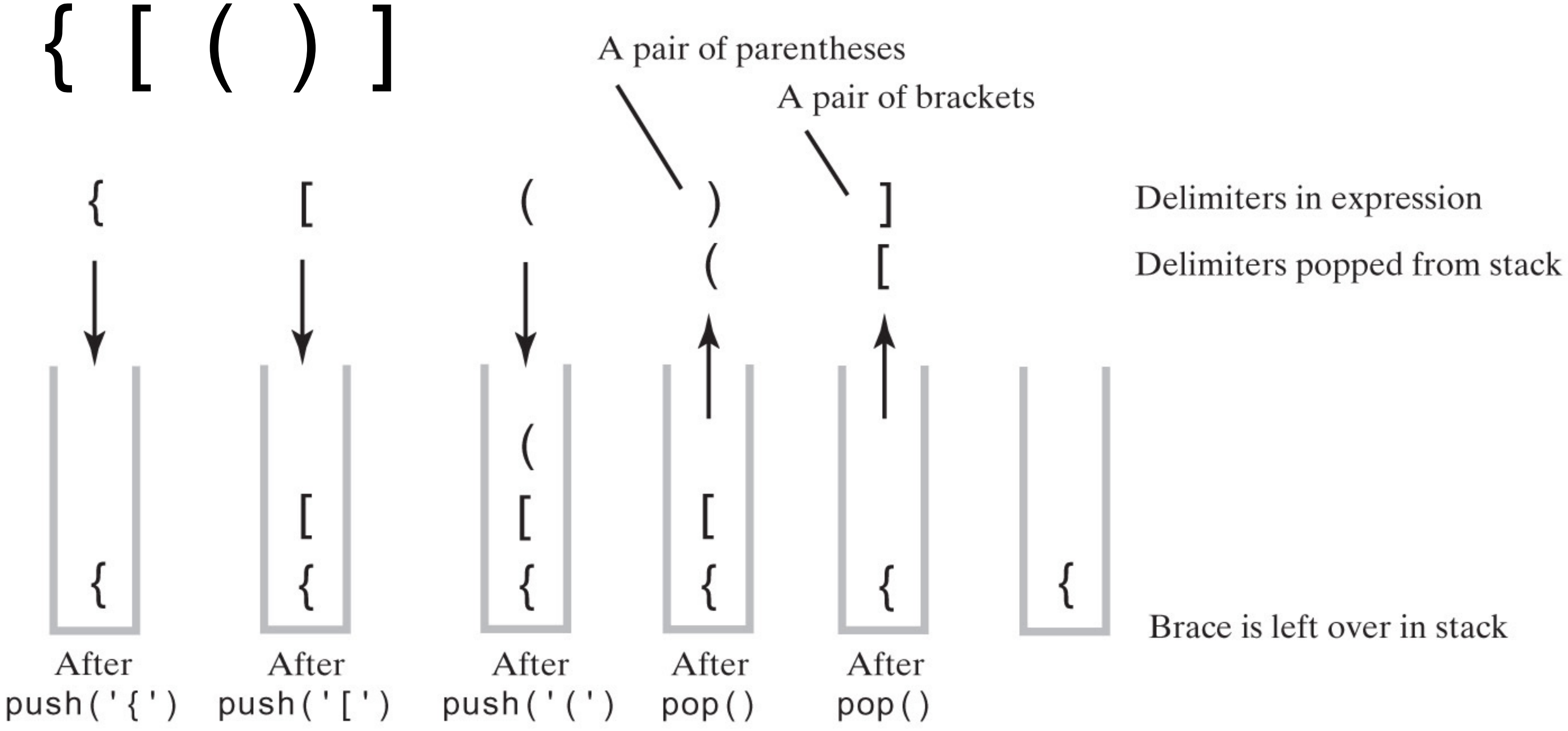

Figure 5-6: The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( )]

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

isBalanced = true // The absence of delimiters is balanced
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 entry of stack // Pop stack
        isBalanced = true or false according to whether openDelimiter and
                     nextCharacter are a pair of delimiters
      }
      break
  }
}

if (stack is not empty)
  isBalanced = false

return isBalanced
```

Algorithm to process for balanced expression.

Exercise

- Show the contents of the stack as you trace the algorithm `checkBalance` for each expression.
- What does `checkBalance` return in each case?
  - » [a { b / (c - d) + e / (f + g) } - h]
  - » { a [b + (c + 2) / d] + e) + f }

Answer

- [a { b / (c - d) + e / (f + g) } - h]

  [

  [{

  [{ (

  [{

  [{ (

  [{

  [

  empty

- The algorithm returns *true* for the expression

Answer

- { a [b + (c + 2) / d] + e) + f } -> { [( )]) }

  {

  { [

  { [(

  { [

  {

- The algorithm returns *false* for the expression

Java Implementation - Part 1

```
/** A class that checks whether the parentheses, brackets, and braces
   in a string occur in left/right pairs. */
public class BalanceChecker
{

   // Returns true if the given characters, open and close, form a pair
   // of parentheses, brackets, or braces.
   private static boolean isPaired(char open, char close)
   {
      return (open == '(' && close == ')') ||
             (open == '[' && close == ']') ||
             (open == '{' && close == '}');
   } // end isPaired

   /** Decides whether the parentheses, brackets, and braces
       in a string occur in left/right pairs.
       @param expression  A string to be checked.
       @return  True if the delimiters are paired correctly. */
   public static boolean checkBalance(String expression)
   {
      [ SEE NEXT SLIDE FOR IMPLEMENTATION]
   } // end checkBalance

} // end BalanceChecker
```

Listing 5-2: The class **`BalanceChecker`**

Java Implementation - Part 2

```java
StackInterface<Character> openDelimiterStack = new LinkedStack<>();
int characterCount = expression.length();
boolean isBalanced = true;
int index = 0;
char nextCharacter = ' ';

while (isBalanced && (index < characterCount)) {
   nextCharacter = expression.charAt(index);
   switch (nextCharacter) {
      case '(': case '[': case '{':
         openDelimiterStack.push(nextCharacter);
         break;
      case ')': case ']': case '}':
         if (openDelimiterStack.isEmpty())
            isBalanced = false;
         else {
            char openDelimiter = openDelimiterStack.pop();
            isBalanced = isPaired(openDelimiter, nextCharacter);
         } // end if
         break;
      default:
         break; // Ignore unexpected characters
   } // end switch
   index++;
} // end while

if (!openDelimiterStack.isEmpty())
   isBalanced = false;

return isBalanced;
```

Exercise

- Consider the following Java statements, assuming that MyStack is a class that implements the interface StackInterface.

```
int n = 4;
StackInterface<Integer> stack = new MyStack<>();

while (n > 0){
   stack.push(n);
   n--;
} // end while

int result = 1;
while (!stack.isEmpty()){
   int integer = myStack.pop();
   result = result * integer;
} // end while

System.out.println("result = " + result);
```

Q1: What value is displayed when this code executes?

Q2: What mathematical function does the code evaluate?

Answer

Q1: What value is displayed when this code executes?

A1: The value 24 is displayed.

Q2: What mathematical function does the code evaluate?

A2: The code evaluates factorial n when n is 4.