WIA 1002 DATA STRUCTURE

SEM 2, SESSION 2024/205

NURUL JAPAR
nuruljapar@um.edu.my

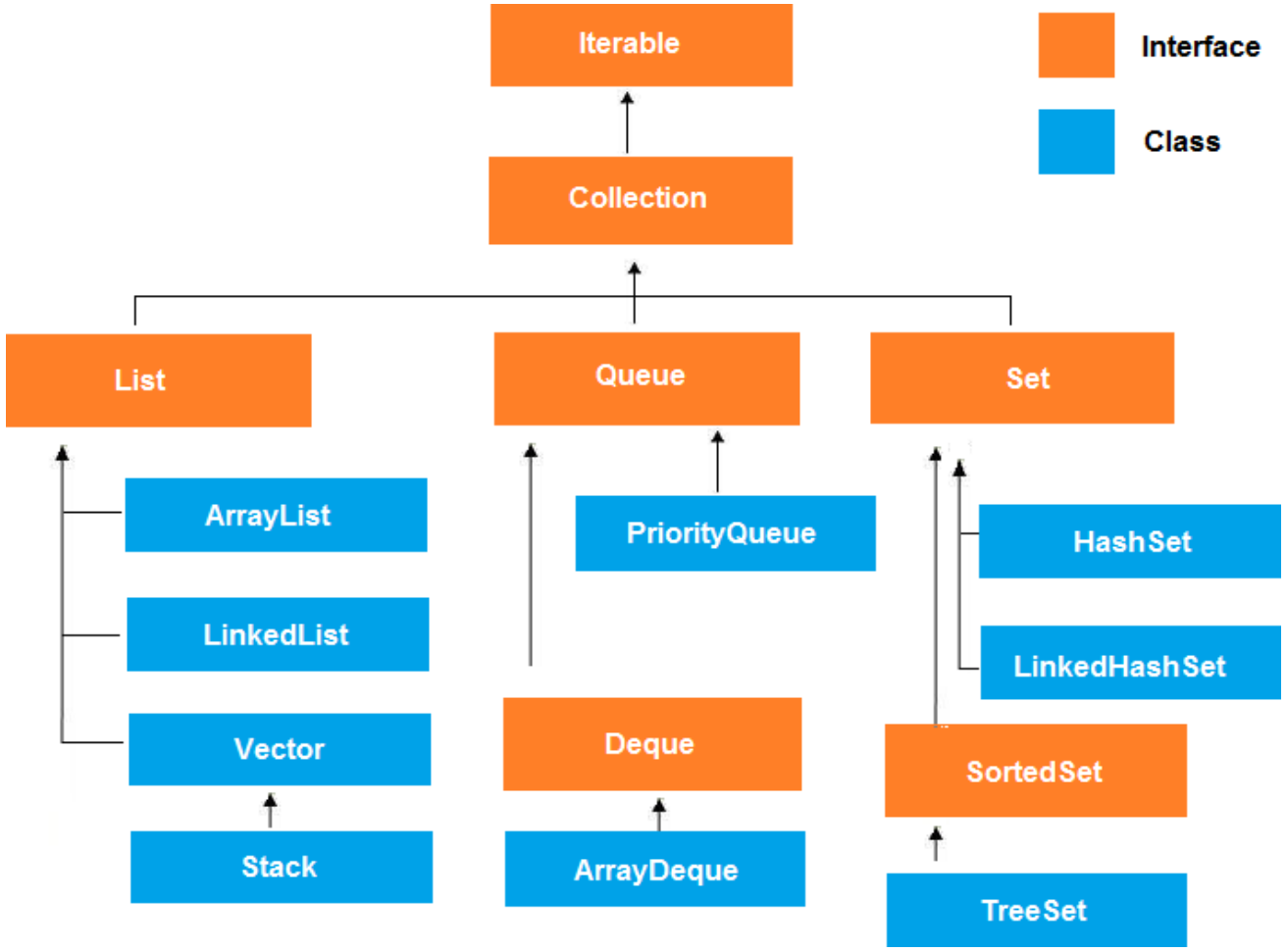
HOO WAI LAM
wlhoo@um.edu.my

Stack

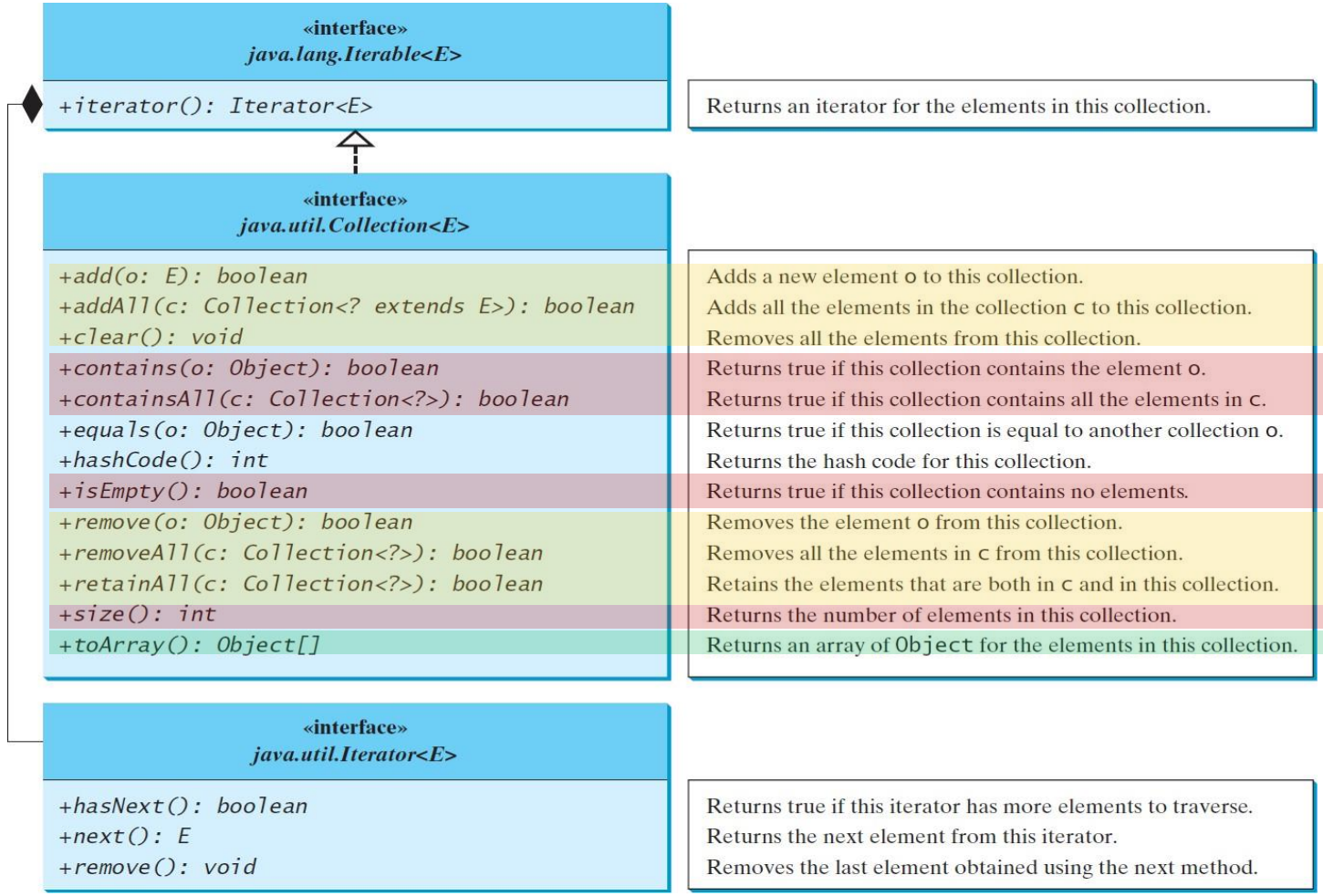
- Recap
- Introduction
- Implementation
- Postfix Evaluation

Recap

Java Collection Framework Hierarchy



The Collection Interface



The Collection interface is the root interface for manipulating a collection of objects.

Basic operations
Query Operations
Representation

List

- A list is a popular *Abstract Data Type* that stores data in sequential order.
- Examples: a list of students, a list of available rooms, a list of cities, and a list of books, etc.
- The common operations on a list are:
  - ✓ Retrieve an element from this list.
  - ✓ Insert a new element to this list.
  - ✓ Delete an element from this list.
  - ✓ Find how many elements are in this list.
  - ✓ Find if an element is in this list.
  - ✓ Find if this list is empty.

Two Ways to Implement Lists

1. Using an **array** to store the elements
   - » The array is dynamically created.
   - » If array capacity is exceeded, create a new larger array and copy all the elements from the current array to the new array.
2. Using a **linked list**
   - » A linked structure consists of nodes.
   - » Each node is dynamically created to hold an element.
   - » All the nodes are linked together to form a list.

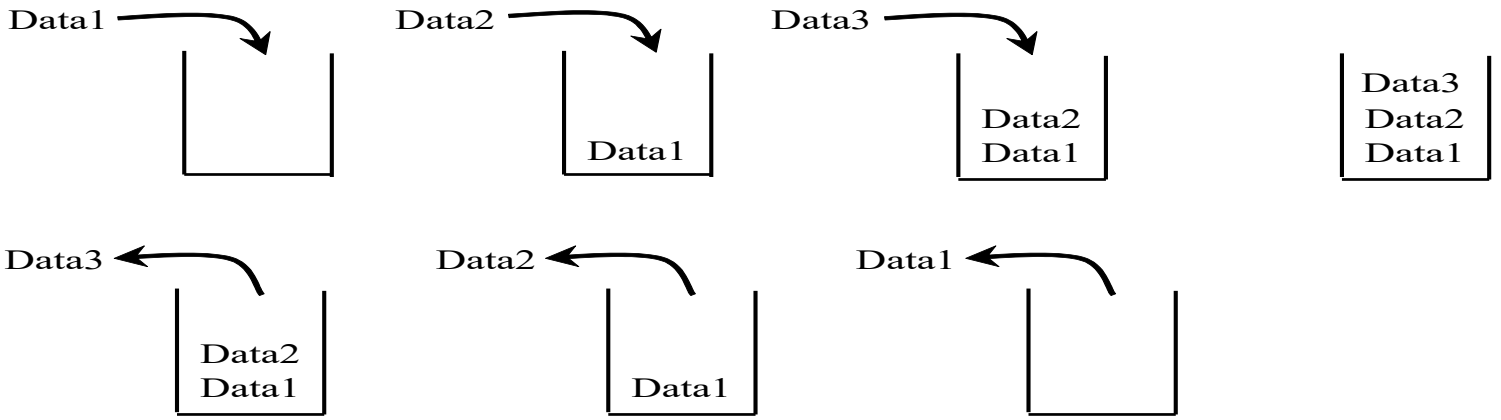
Array or Linked List?

1. Use an array
   - » get(int index) and set(int index, Object o) through an index and add(Object o) for adding an element at the end of the list are efficient.
   - » add(int index, Object o) and remove(int index) are inefficient - shift potentially large number of elements.
2. Use a linked list
   - » improve efficiency for adding and removing an element anywhere in a list.

Introduction

Introducing Stack

- A stack can be viewed as a special type of **list**, where the elements are accessed, inserted, and deleted only from the end, called the top, of the stack.

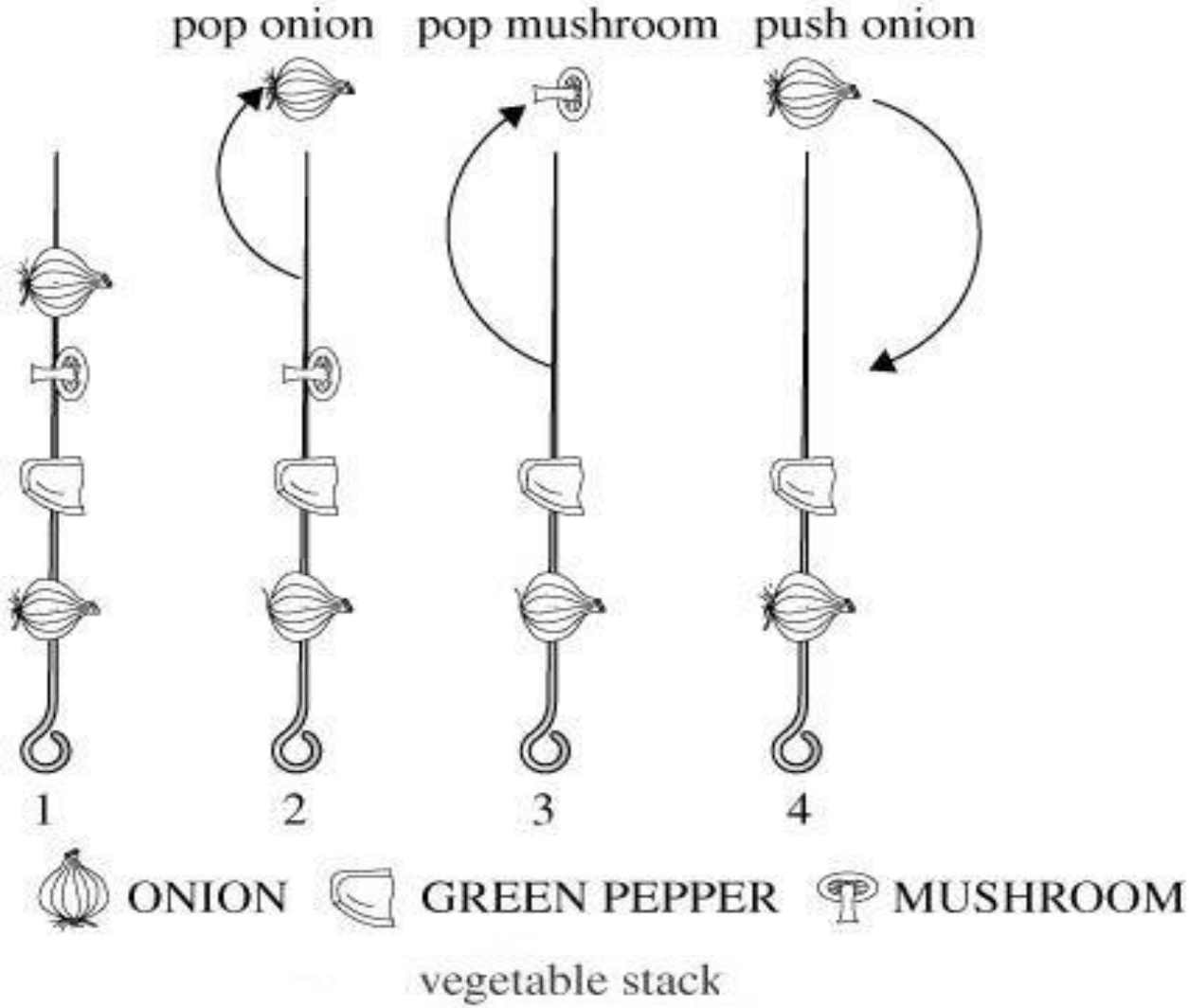


Introducing Stack

- It is a data structure that holds data in a last-in, first-out (LIFO) fashion.
- Stack - an adapter which defines a **restricted set of *list* methods**.

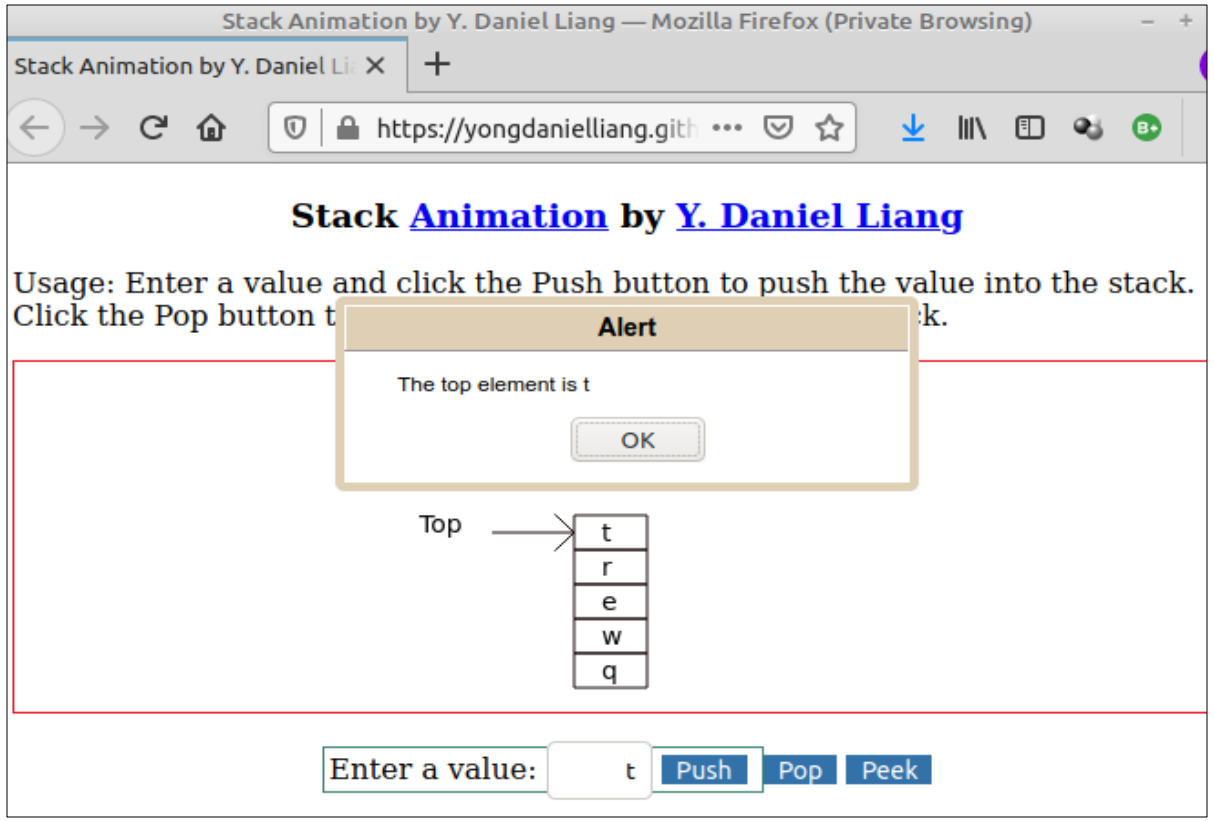
Methods in Stack

- Pop() = removes item on top
- Push() = adds item at the top
- Peek() or Top() = access value on top



vegetable stack

Stack Animation



https://yongdanielliang.github.io/animation/web/Stack.html

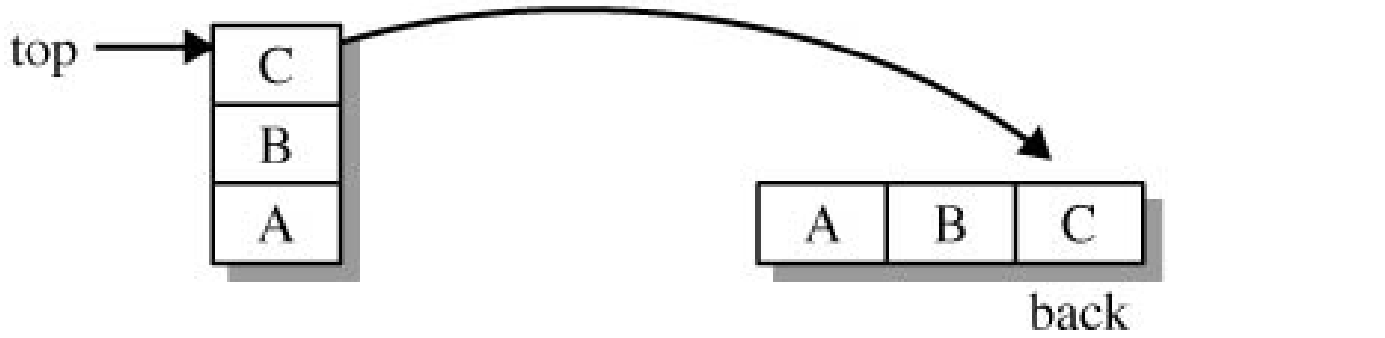
Check Point

- Draw a stack for the following :
  1. Push A
  2. Push B
  3. Push C
  4. Pop C
  5. Pop B
  6. Push D

Implementation

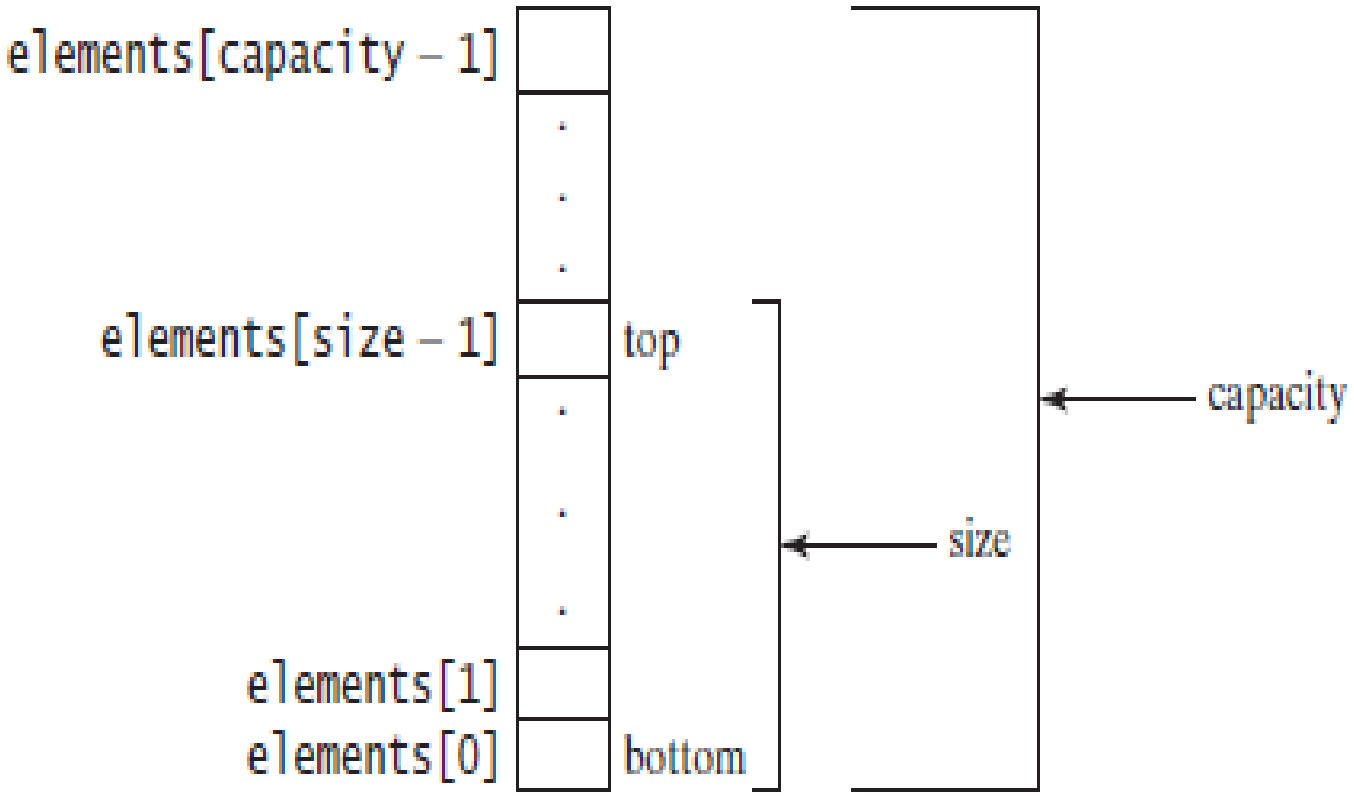
Implementing Stack

- Using an array list to implement a stack is more efficient than a linked list since the insertion and deletion operations on a stack are made only at the end of the stack.



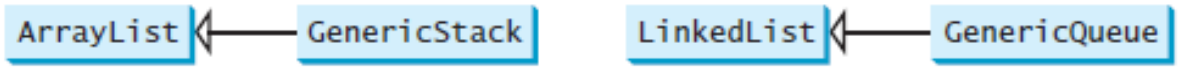
A stack conceptually. A stack implemented as an Array List.

Using Array list as the storage structure for Stack

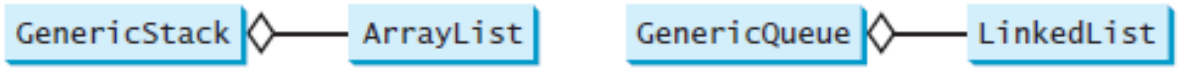

elements[capacity – 1]
elements[size – 1]
top
capacity
size
elements[1]
elements[0]
bottom

Implementing Stack

- 2 ways to design the stack class using ArrayList:
  - Using **inheritance**: You can define a stack (GenericStack) class by extending ArrayList class
  - Using **composition**: You can define an array list as a data field in the stack (GenericStack) class



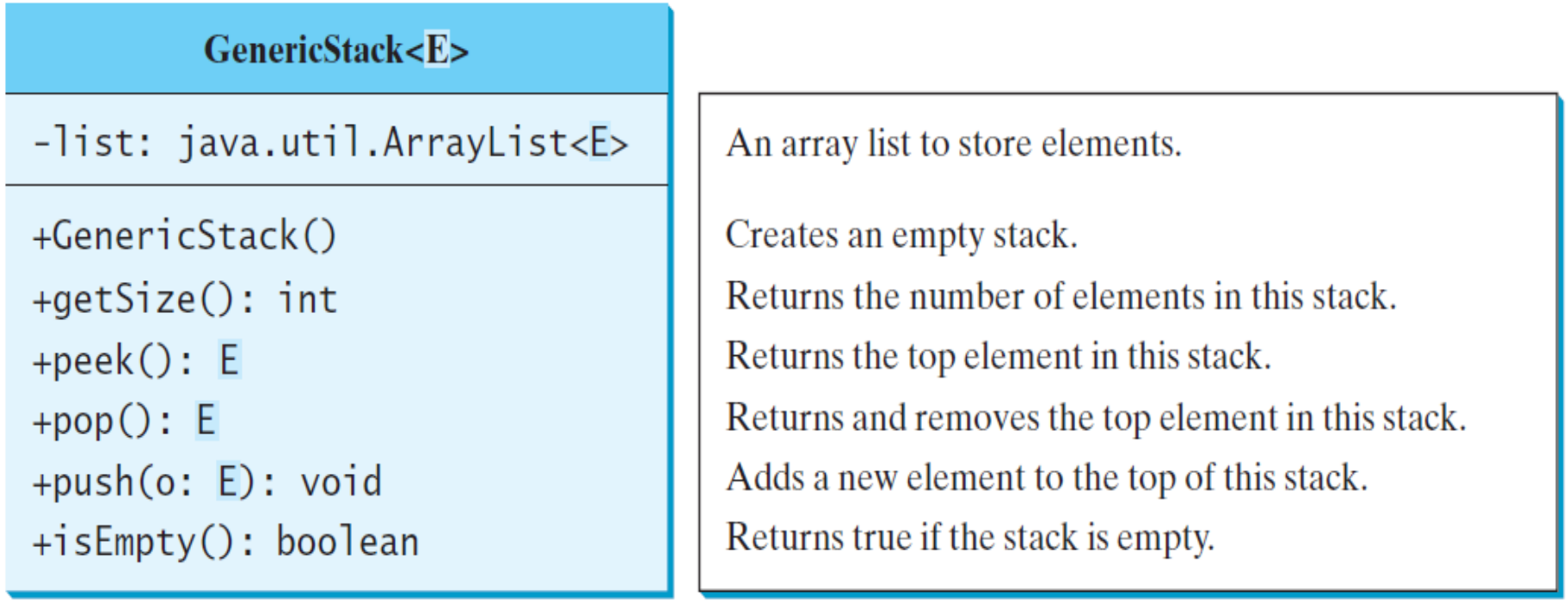
(a) Using inheritance



(b) Using composition

Implementing Stack

GenericStack class using an ArrayList & Composition approach.



* The peek() and pop() require that the stack contains at least one element.

Generic Stack

- The **peek()** and **pop()** require that the stack contains at least one element.
- If condition is not satisfied, the methods should throw an EmptyStackException.

```
if(isEmpty()) {
        throw new EmptyStackException();
}
```

GenericStack class

```
1  public class GenericStack<E> {
2    private java.util.ArrayList<E> list = new java.util.ArrayList<>();
3
4    public int getSize() {
5      return list.size();   }
6
7    public E peek() {
8      return list.get(getSize() - 1);   }
9
10   public void push(E o) {
11     list.add(o);   }
12
13   public E pop() {
14     E o = list.get(getSize() - 1);
15     list.remove(getSize() - 1);
16     return o;
17   }
18
19   public boolean isEmpty() {
20     return list.isEmpty();   }
21
22   @Override
23   public String toString() {
24     return "stack: " + list.toString();
25   }
26 }
```

Test GenericStack

```
(1) stack: [Tom]
(2) stack: [Tom, Susan]
(3) stack: [Tom, Susan, Kim, Michael]
(4) Michael
(5) Kim
(6) stack: [Tom, Susan]
```

```java
public class TestGenericStack {
  public static void main(String[] args) {
    // Create a stack
    GenericStack<String> stack = new GenericStack<>();

    // Add elements to the stack
    stack.push("Tom"); // Push it to the stack
    System.out.println("(1) " + stack);

    stack.push("Susan"); // Push it to the the stack
    System.out.println("(2) " + stack);

    stack.push("Kim"); // Push it to the stack
    stack.push("Michael"); // Push it to the stack
    System.out.println("(3) " + stack);

    // Remove elements from the stack
    System.out.println("(4) " + stack.pop());
    System.out.println("(5) " + stack.pop());
    System.out.println("(6) " + stack);
  }
}
```

Postfix Evaluation

Postfix Expressions

- Can be implemented using stacks
- Also called Reverse Polish Notation, is an alternative way of representing mathematics expressions
- In a postfix evaluation format for an arithmetic expression, an operator comes after its operands.
- Advantages:
  - Do not need precedence rules
  - Do not need parentheses
  - Reduce computer memory access

Postfix Expressions

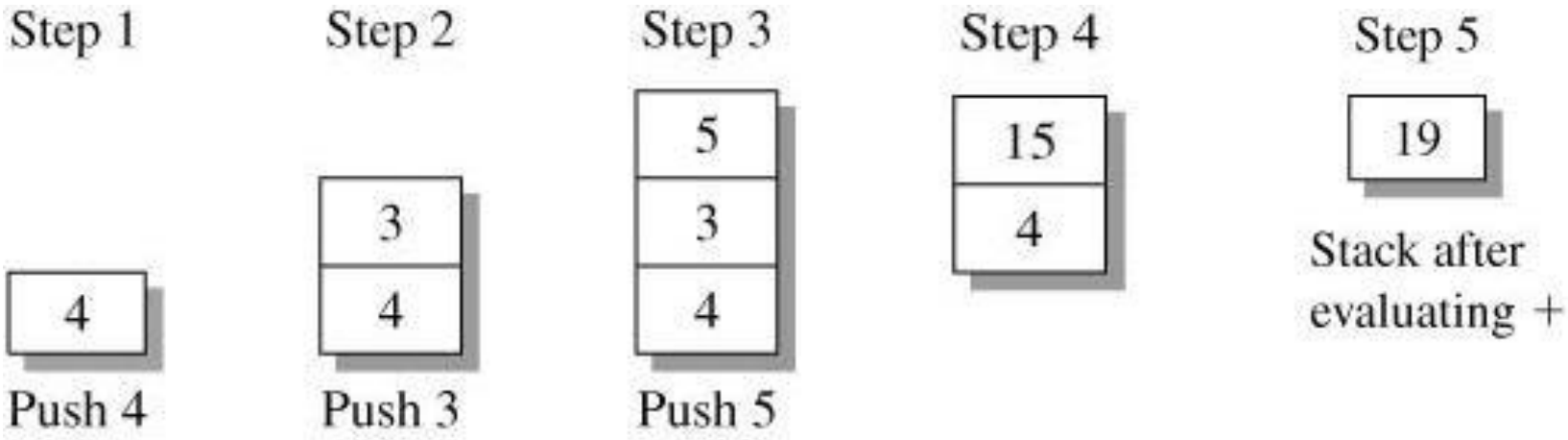
- If there are multiple operations, operators are given immediately after their second operands
- Examples:
- a + b * c RPN: a b c * +
  Operator * has higher precedence than +.
- (a + b) * c RPN: a b + c *
  The parenthesis creates subexpression a b +
- (a*b + c) / d + e RPN: a b * c + d/ e +
  The subexpression is a b * c +. Division is the next operator followed by addition.

Postfix Evaluation

- To evaluate a postfix expression, execute the following steps until the end of the expression.
  1. If recognize an operand, push it on the operand stack.
  2. If recognize an operator, perform the following:
     - pop an operand as x
     - pop another operand as y
     - Perform (x operator y)
  3. Repeat step 1 and 2 until the end. Pop the step for final result.

Postfix Evaluation

- Example: evaluate "4 3 5 * +"

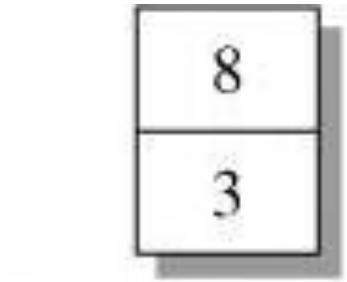


Postfix Evaluation

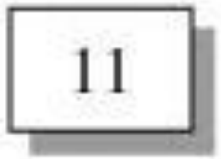
- At each step in the postfix evaluation algorithm, the state of the stack allows us to identify whether an error occurs and the cause of the error.

Postfix Evaluation - Too many operators

- In the expression 3 8 + * 9 the binary operator * is missing a second operand. Identify this error when reading * with the stack containing only one element.



After pushing operands 3 and 8

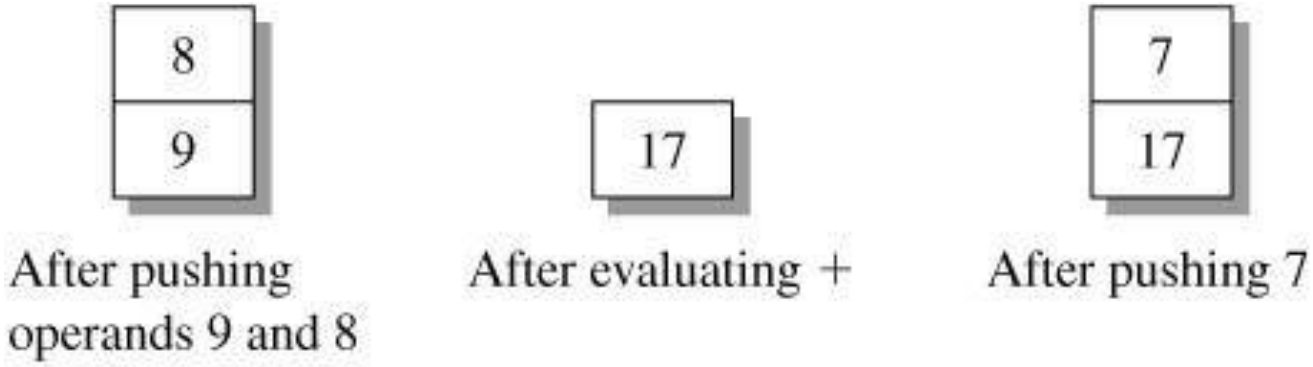


After evaluating +

Postfix Evaluation - Too many operands

- An expression may contain too many operands. Identify this error after processing the entire expression. At the conclusion of the process, the stack contains more than one element.

  Example: 9 8 + 7



PostfixEvaluation.java

```
 1  public class PostfixEvaluation
 2  {
 3      public static void main(String[] args)
 4      {
 5          System.out.println("Testing PostfixEvaluation:\n");
 6          System.out.println("2 3 + 4 * 5 - : "
 7              + evaluatePostfix("2 3 + 4 * 5 -") + "\n");
 8          System.out.println("2 3 * 4 2 - / 5 6 * + : "
 9              + PostfixEvaluation.evaluatePostfix("2 3 * 4 2 - / 5 6 * +")
10              + "\n");
11          System.out.println("2 4 - 3 ^ 5 + : "
12              + PostfixEvaluation.evaluatePostfix("2 4 - 3 ^ 5 +")
13              + "\n");
14          System.out.println("\n\nDone.");
15      }  // end main
16
```

```
Testing PostfixEvaluation:

2 3 + 4 * 5 - : 15.0

2 3 * 4 2 - / 5 6 * + : 33.0

2 4 - 3 ^ 5 + : -3.0



Done.
```

PostfixEvaluation.java

```
17      /** Evaluates a postfix expression.
18          @param postfix  a string that is a valid postfix expression.
19          @return  the value of the postfix expression. */
20      public static double evaluatePostfix(String postfix)   {
21         GenericStack<Double> valueStack = new GenericStack<>();
22         String[] tokens = postfix.split(" ");
23         for (String token: tokens)
24         {
25            if(isNumeric(token))
26            {
27               valueStack.push(new Double(token));
28            }
29            else if (token.equals("+") || token.equals("-") || token.equals("*")
30                    || token.equals("/") || token.equals("^"))
31            {
32                  Double operandTwo = valueStack.pop();
33                  Double operandOne = valueStack.pop();
34                  Double result = compute(operandOne, operandTwo, token);
35                  valueStack.push(result);
36            }
37         } // end for
38
39         return (valueStack.peek());
40      } // end evaluatePostfix
```

PostfixEvaluation.java

```
41
42      public static boolean isNumeric(String str)
43      {
44           try
45           {
46                double d = Double.parseDouble(str);
47           }
48           catch(NumberFormatException nfe)
49           {
50                return false;
51           }
52           return true;
53       }
54
```

PostfixEvaluation.java

```
55      private static Double compute(Double operandOne, Double operandTwo, String operator)
56      {
57         double result;
58
59         switch (operator)
60         {
61            case "+":
62               result = operandOne + operandTwo;
63               break;
64
65            case "-":
66               result = operandOne - operandTwo;
67               break;
68
69            case "*":
70               result = operandOne * operandTwo;
71                break;
72
73            case "/":
74               result = operandOne / operandTwo;
75               break;
76
77            case "^":
78               result = Math.pow(operandOne, operandTwo);
79               break;
80
81            default:    // Unexpected character
82               result = 0;
83               break;
84         } // end switch
85
86         return result;
87      } // end compute
```

References

- Chapter 19 and 24, Liang, Introduction to Java Programming, 10th Edition, Global Edition, Pearson, 2015
- Chapter 5, Carrano, Data Structures and Abstractions with Java, 3rd Edition, 2012