BACS2063 Data Structures and Algorithms

Applications of Abstract Data Types (ADTs)

Chapter 1

Learning Outcomes

At the end of this lecture, you should be able to

- Explain the benefits of *abstraction*.
- Apply the use of the *list*, *stack* and *queue* ADTs appropriately to solve problems.

Recap of Programming Paradigms

Procedural paradigm

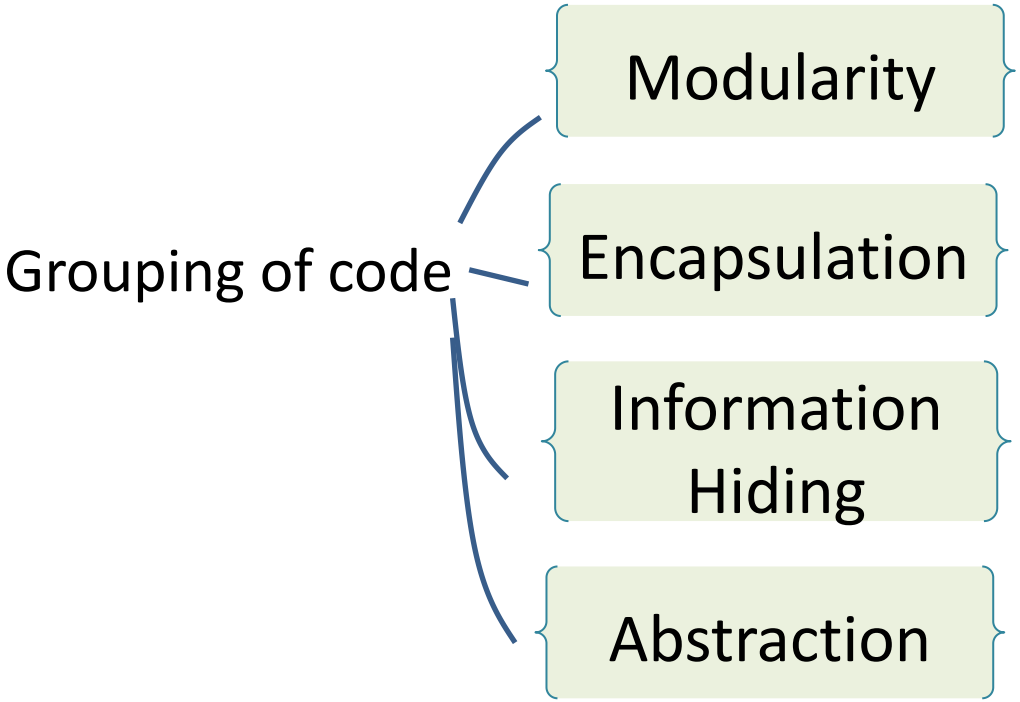
- sequence
- selection
- iteration

Object-oriented paradigm

- class
  - properties
  - methods
- inheritance
- encapsulation
- polymorphism.

- Which came first?
- What is the main difference?
- Why was the newer paradigm introduced?
  - reuse
  - maintainability

Terminology Recap

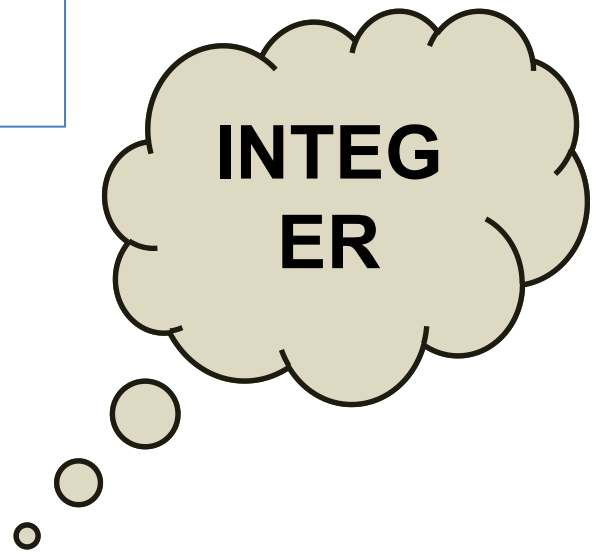


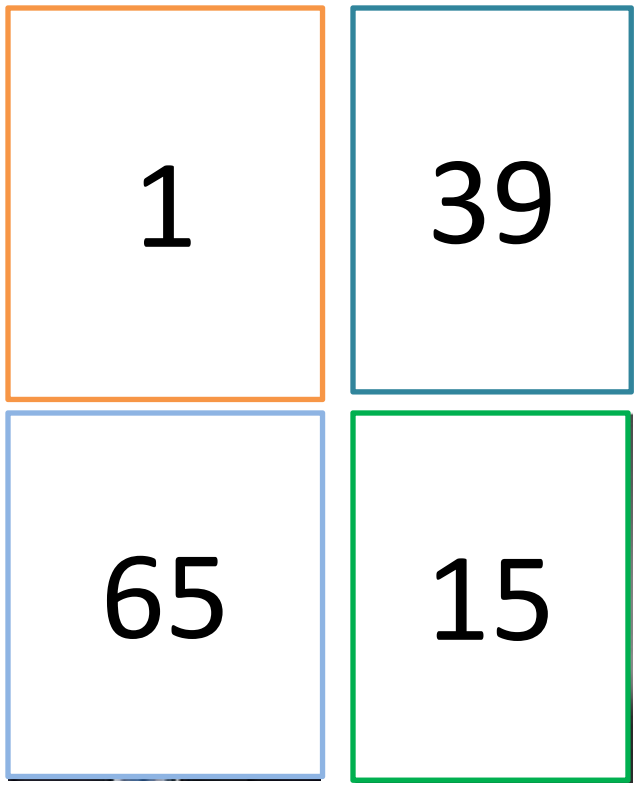
- What do these terms mean?
- Are they present in both procedural and OO paradigms?
- Why are they important?

What is Abstraction?

Abstract:
Exist in thought, but no concrete existence

… Oxford Dictionaries





Why Abstraction?

ABSTRACTION

Hiding the implementation details, by providing 1 layer over the basic functionality

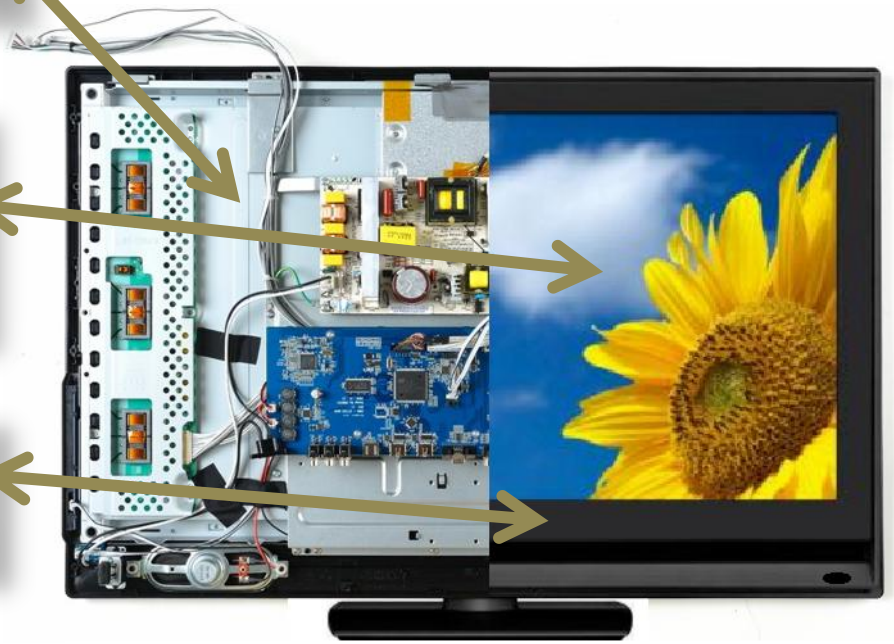
- ✓ User can focus on the "big picture", not the details.
- ✓ Developer can modify the class implementation without affecting the user.

Example:
Integer.parseInt()

Implementation

Abstraction (Interface and Operation)

Encapsulation (hiding details)



Abstraction vs Encapsulation

KEY DIFFERENCE

- Abstraction shows only useful data by providing the most necessary details whereas Encapsulation wraps code and data for necessary information.
- Abstraction is focused mainly on what should be done while Encapsulation is focused on how it should be done.
- Abstraction hides complexity by giving you a more abstract picture while Encapsulation hides internal working so that you can change it later.
- Abstraction helps you to partition the program into many independent portions whereas Encapsulation is easy to change with new requirements.
- Comparing Encapsulation vs Abstraction, Abstraction solves problem at design level while Encapsulation solves problem at implementation level.
- Abstraction hides the irrelevant details found in the code whereas Encapsulation helps developers to organize the entire code easily.

Source:
https://www.guru99.com/difference-between-abstraction-and-encapsulation.html

Design Principles: Encapsulation

- Different components of a software system should not reveal the internal details of their respective implementations (information hiding)
- Advantage
  - Programmer has freedom in implementing the details of a system
  - Programmer only needs to maintain the contract (interface) that outsiders see

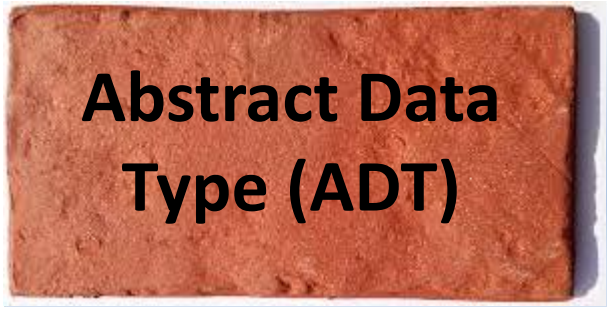
Design Principles: Modularity

- An organizational principle for code in which different components of a software system are **divided into separate functional units**.
- Rationale
  - Modern software systems consist of several **different components** that must interact correctly in order for the entire system to work properly.
  - Keeping these interactions straight requires that these different components be **well-organized**

Design Principles: Abstraction

- To distill a complicated system down to its most fundamental parts and
- describe these parts in a simple, precise language by naming the parts and explaining their functionality.

Abstract Data Type (ADT)



Abstract Data Type:
A TYPE that is conceived apart from concrete reality

An ADT consists of data that has certain characteristics + a set of operations that can be used to manipulate the data

Examples of ADTs

- Integer
- Double
- String
- Fraction
- Counter

Examples of Collection ADTs

- List
- Stack
- Queue
- Set

Data Structure

- The **implementation of ADT** is often referred to as **data structure**, using some constructs and primitive data types.

- A representation of data and the operations allowed on that data (Weiss, 2010)

Collection ADTs

- A general term for an ADT that contains a *group of objects*.
  - Some collections allow duplicate items, some do not.
    - *e.g. List, Stack, Queue allow duplicate items. Set does not allow duplicate items.*
  - Some collections arrange their contents in a certain order, while others do not.
    - *e.g. List is unordered. Sorted List is ordered.*

How are these collections different?

(A)

(B)

(C)

Algorithm

- A list of steps to solve a problem (perform as task).
- Different algorithms may be used to solve the same problem.

vs.

Resources?

Time?

The algorithm determines

Time Efficiency

- **Time** taken to complete a task

Space Efficiency

- The use of resources

Recall: Benefits of ADTs

- Reusability
- Maintainability

Introduction to Collections

- Recall: a collection is a general term for an ADT that contains a group of objects.
- A *linear collection* is a collection that stores its entries in a linear sequence.
- Examples of linear collections: List, Stack, Queue, etc.
  - They **differ** in the restrictions they place **on how these entries may be added, removed, or accessed**

What do you need to learn about List, Stack and Queue?

- Their basic characteristics.
- **Basic operations** they must have.
- ***Where*** to place an item during an **add** operation.
  - ***List*** → **Add/Insert** operation (Back or position specified).
  - ***Stack*** → **Push** operation (item is added to the top)
  - ***Queue*** → **Enqueue** operation (item is added to the back)
- What happens when an item is **removed**, specifically, which item in the collection is affected:
  - ***List*** → **Remove** operation (position of the item must be specified)
  - ***Stack*** → **Pop** operation (top item is removed)
  - ***Queue*** → **Dequeue** operation (front item is removed).

Lists

List: Definition

- A collection of items in which the items have a position (Venugopal, 2007)

- A linear collection of entries in which entries may be added, removed, and searched for without restriction (Weiss, 2010)

List: Basic Operations

add(*e*)	:	Insert element *e*, at the end of the list
remove(*i*)	:	Remove the element at position *i* from the list and return this element
isEmpty()	:	Return *true* if the list is empty
get(*i*)	:	Return the element at position *i* in the list, without removing it
clear()	:	Remove all the elements from the list so that the list is now empty

List: Applications

- Lists of things – task lists, list of names, playlists, etc

- High-Precision Arithmetics – to represent very long integer values

Note

- To illustrate the use of collection ADTs, we will use predefined Java interfaces and classes form the Java Collections Framework.

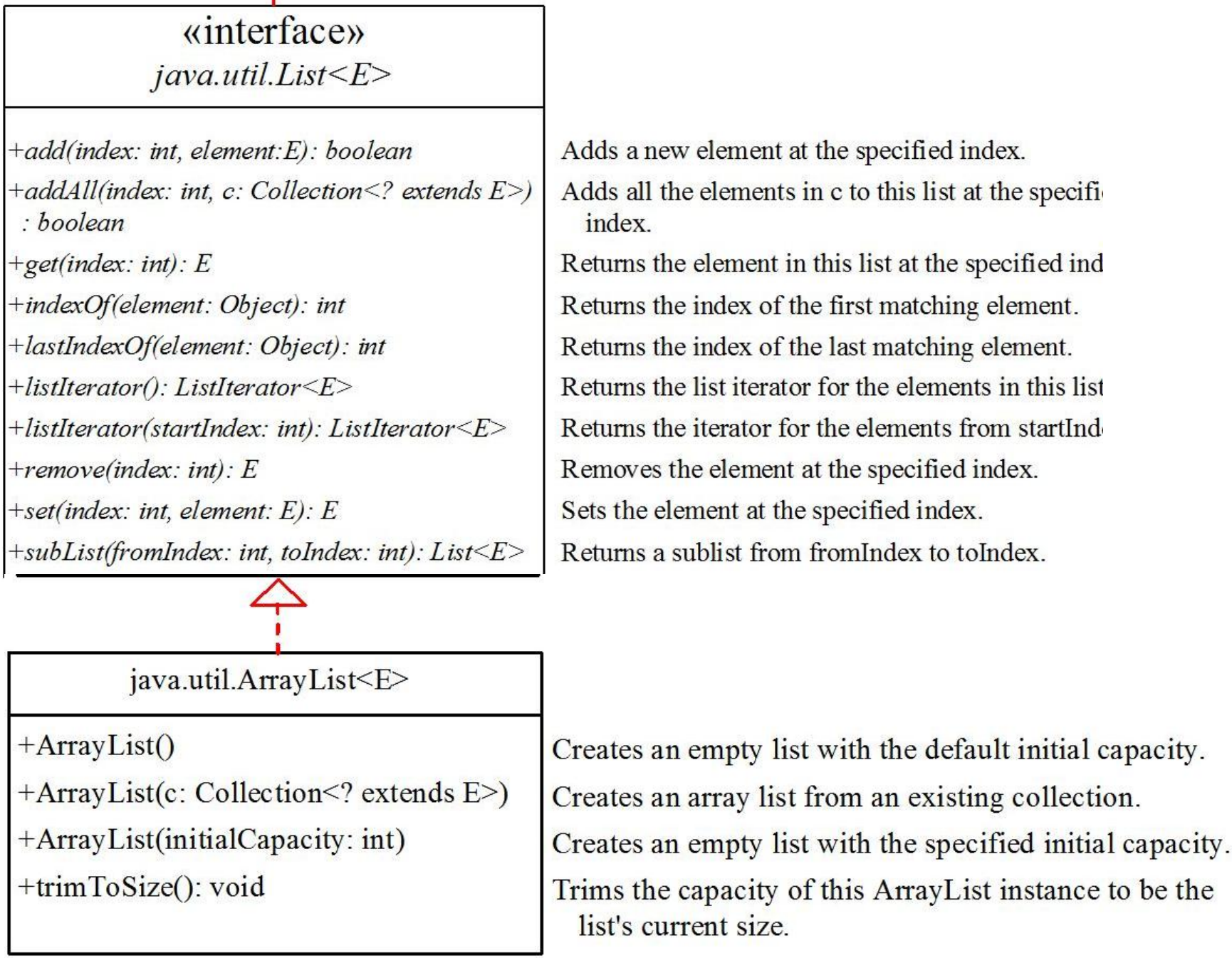
- In **Chapter 4 onwards**, you will learn how to create your own collection ADTs.

Java Collections Framework: `java.util.List` interface (1)

Return Type	Method and Description
`boolean`	`add(E e)` Appends the specified element to the end of this list.
`void`	`add(int index, E element)` Inserts the specified element at the specified position in this list.
`void`	`clear()` Removes all of the elements from this list.
`boolean`	`contains(Object o)` Returns `true` if this list contains the specified element.

Java Collections Framework: `java.util.List` interface (2)

Return Type	Method and Description
`E`	`get(int index)` Returns the element at the specified position in this list.
`boolean`	`isEmpty()` Returns `true` if this list contains no elements.
`E`	`remove(int index)` Removes the element at the specified position in this list.
`int`	`size()` Returns the number of elements in this list.


«interface»
java.util.List<E>
+add(index: int, element:E): boolean
+addAll(index: int, c: Collection<? extends E>) : boolean
+get(index: int): E
+indexOf(element: Object): int
+lastIndexOf(element: Object): int
+listIterator(): ListIterator<E>
+listIterator(startIndex: int): ListIterator<E>
+remove(index: int): E
+set(index: int, element: E): E
+subList(fromIndex: int, toIndex: int): List<E>
Adds a new element at the specified index.
Adds all the elements in c to this list at the specifi index.
Returns the element in this list at the specified ind
Returns the index of the first matching element.
Returns the index of the last matching element.
Returns the list iterator for the elements in this list
Returns the iterator for the elements from startInd
Removes the element at the specified index.
Sets the element at the specified index.
Returns a sublist from fromIndex to toIndex.
java.util.ArrayList<E>
+ArrayList()
+ArrayList(c: Collection<? extends E>)
+ArrayList(initialCapacity: int)
+trimToSize(): void
Creates an empty list with the default initial capacity.
Creates an array list from an existing collection.
Creates an empty list with the specified initial capacity.
Trims the capacity of this ArrayList instance to be the list's current size.

Sample Code: Shopping List App

In `Chapter1\examples\`

- **`ShoppingListApp.java`**

Problem: Registration of Runners in a Marathon

Sample Code: Registration of Marathon Runners

In `Chapter1\runningman\`

- **`Runner.java`**
- **`Registration.java`**

Note: Types of Lists

- ***Unordered* lists**
  - Entries are not arranged in any particular order
  - By default, "lists" refer to unordered lists

- ***Sorted* (*ordered*) lists**
  - Entries are maintained in sorted order
  - We will cover sorted lists in **Chapter 7**.

Stacks

Stack: Definition

- A collection of objects in which the objects are inserted and removed according to the last-in, first out (*LIFO*) principle (Goodrich, Tamassia and Goldwasser, 2015)
- A linear collection of entries in which entries may be only removed in the reverse order in which they are added, i.e. the last entry added is the first one to be removed (Venugopal, 2007)
- Typically, there is not provision to search for an entry in the stack

Stack: Basic Operations

push (*e*)	:	Insert element *e*, to be the top of the stack
pop()	:	Remove and return the element at the **top** of the stack
isEmpty()	:	Return ***true*** if the stack is **empty**
peek()	:	Return the **top** element in the stack, without removing it
clear()	:	Remove all the elements from the stack so that the stack is now empty

PYQ, October 2022 Q2ai

Question

Question 2

a) **Stack** is a collection of objects in which the objects are inserted and removed according to the last-in, first out (**LIFO**) principle.

(i) Discuss **THREE (3)** basic Stack operations, include appropriate keywords. **(6 marks)**

Answer:

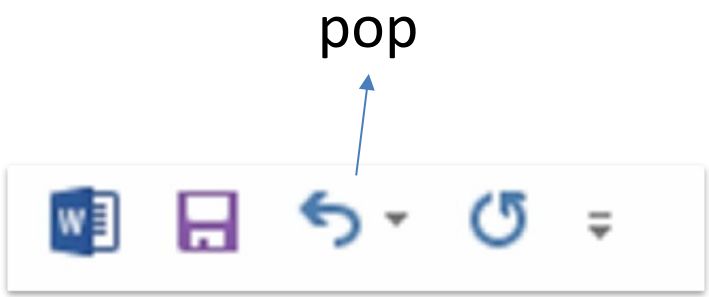
Operation	Description
push(e)	Insert element *e*, to be the top of the stack
pop()	Remove and return the element at the top of the stack
isEmpty()	Return *true* if the stack is empty
peek()	Return the top element in the stack, without removing it

Stack Trivia

Background Information

- The name "stack" is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser

Stack: Application



STRUCTURE|

First In

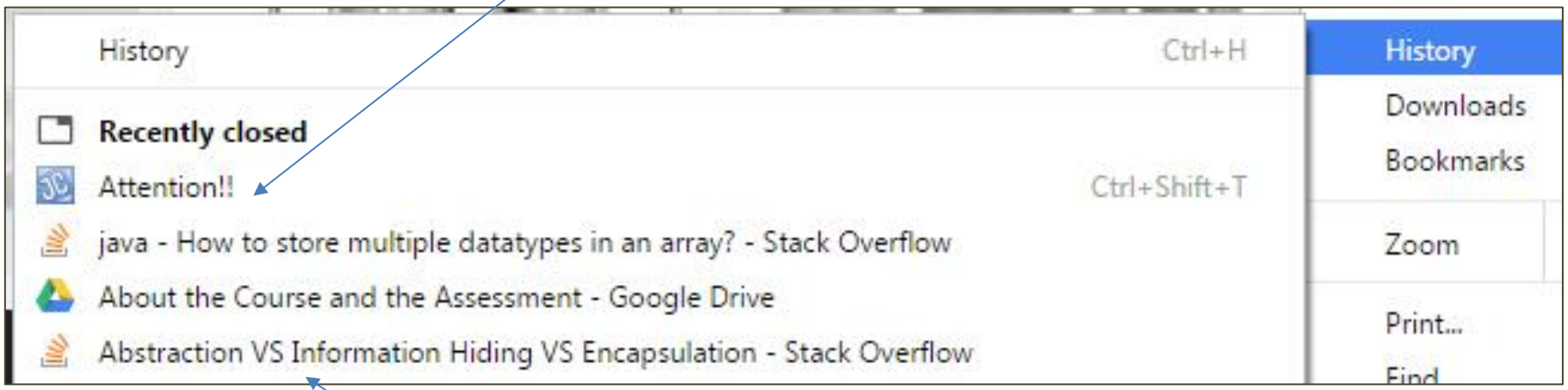
Last Out

Last In

First Out

Stack: Application

Last In



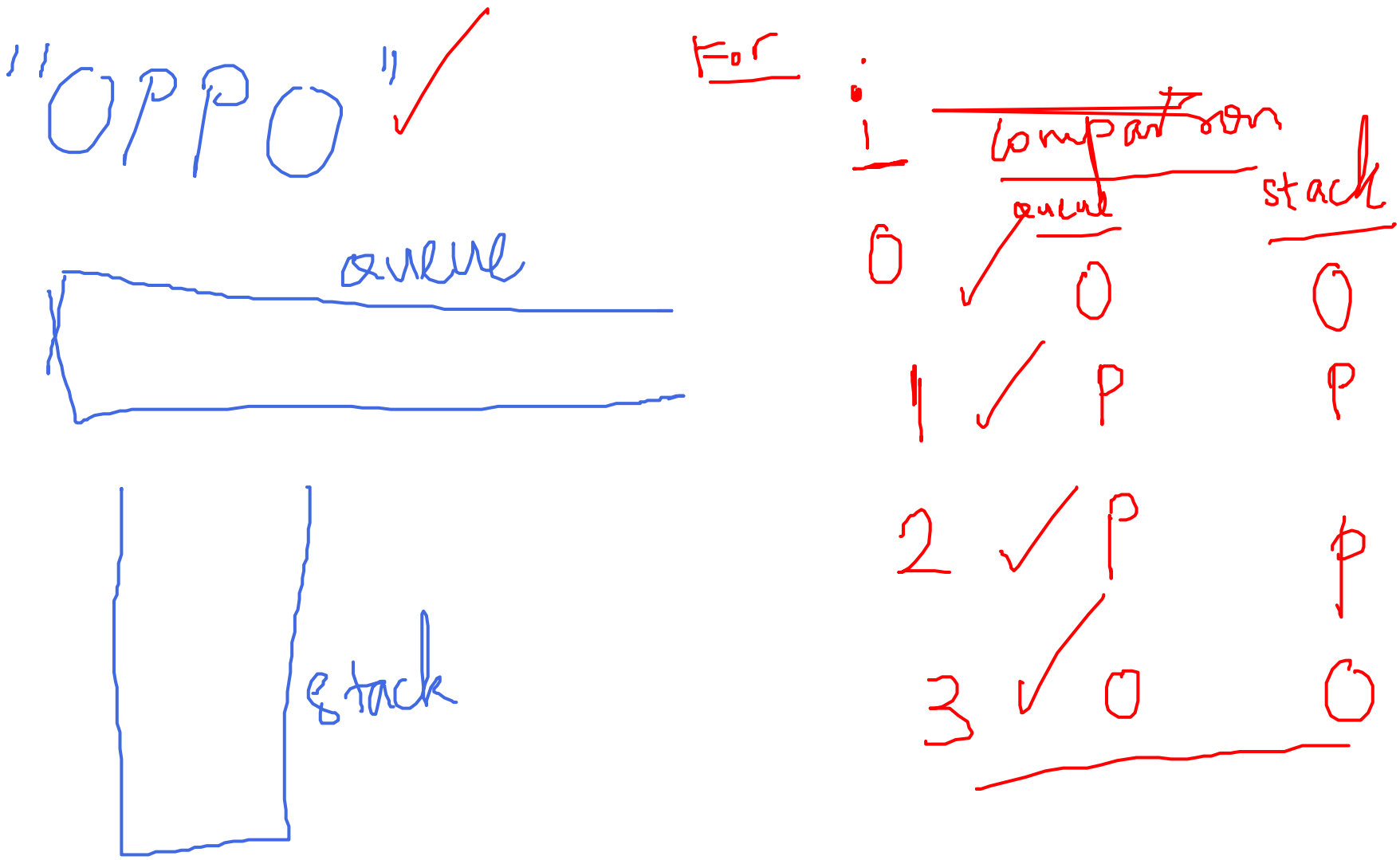
First In

Stack: Applications

- Any application that requires LIFO processing – checking for palindromes (e.g. “noon”, “kayak”), reversing a string, etc.

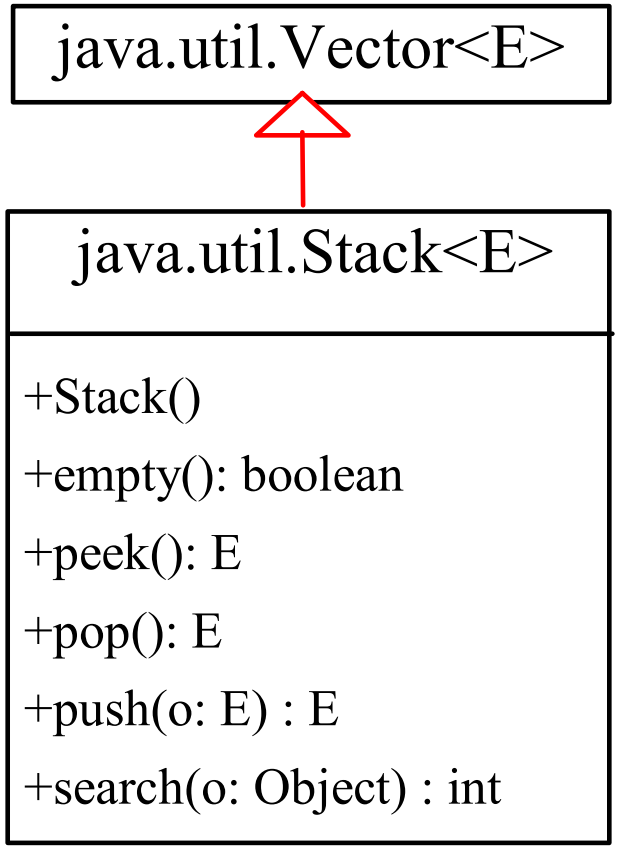
- Program stack – to keep track of method calls

Checking for Palindrome



Checking for Palindrome

Java Collections Framework: `java.util.Stack` class



Creates an empty stack.

Returns true if this stack is empty.

Returns the top element in this stack.

Returns and removes the top element in this stack.

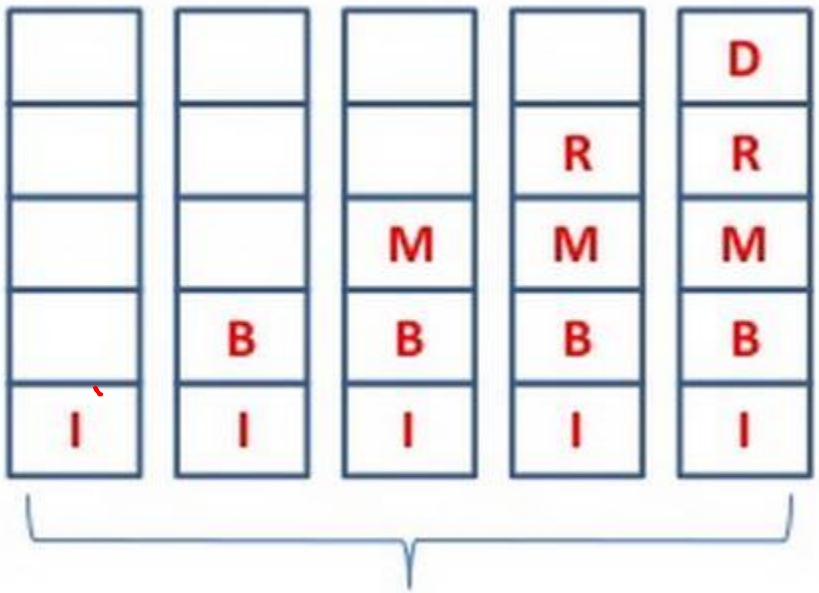
Adds a new element to the top of this stack.

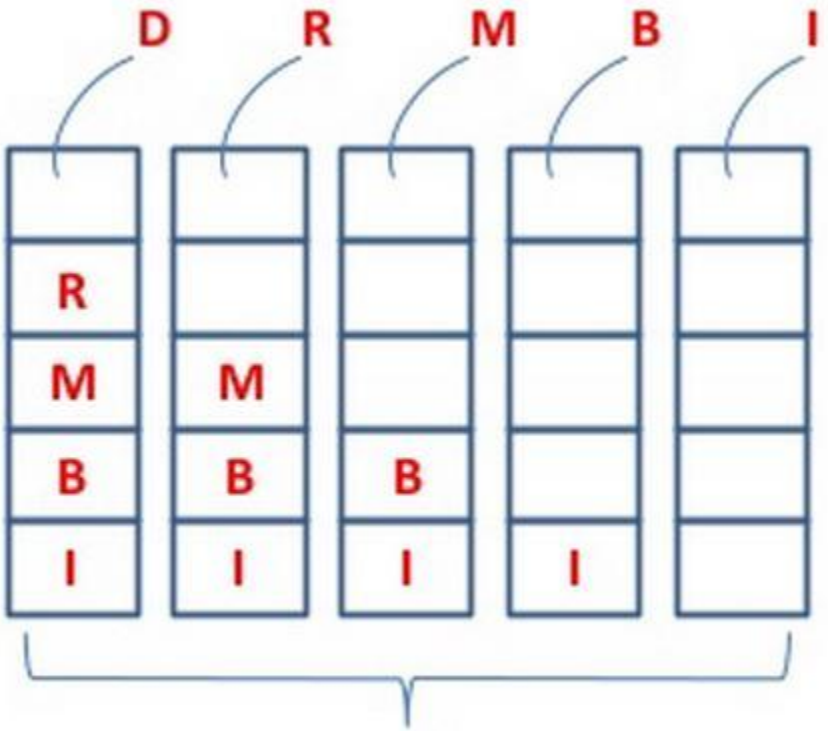
Returns the position of the specified element in this stack.

Problem: Reversing a String

E.g.

I/P String is I B M R D





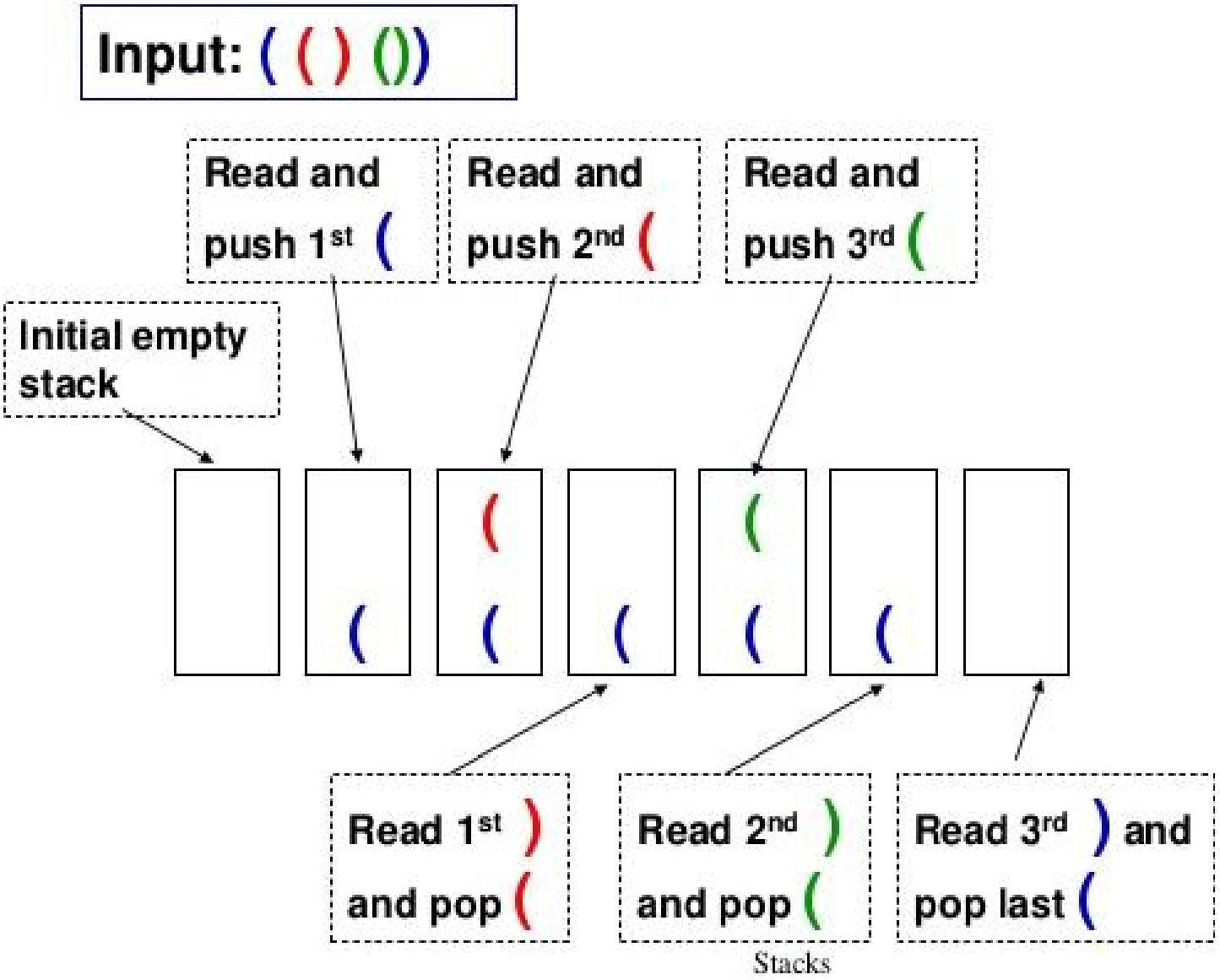
O/P String is D R M B I

Sample Code: Reversing a String

In **`Chapter1\examples\StringToolkit`**

- Method **`reverse()`** in **`StringToolkit.java`**

Problem: Matching Brackets



{(a+b)}

Explaining slides from 49 to 55

Checking for Balanced `()`, `[]`, `{}` (1/7)

- Use a stack to store the left brackets
- Brackets include
  - parentheses `()`
  - square brackets `[]` and
  - braces `{}`

Checking for Balanced `()`, `[]`, `{}` (2/7)

- Traverse the expression to process each character
  - If it is a left/open bracket, push it onto the stack.
  - If it is a right/close bracket, pop a bracket from the stack and compare whether the two bracket types match.

Checking for Balanced `()`, `[]`, `{}` (3/7)

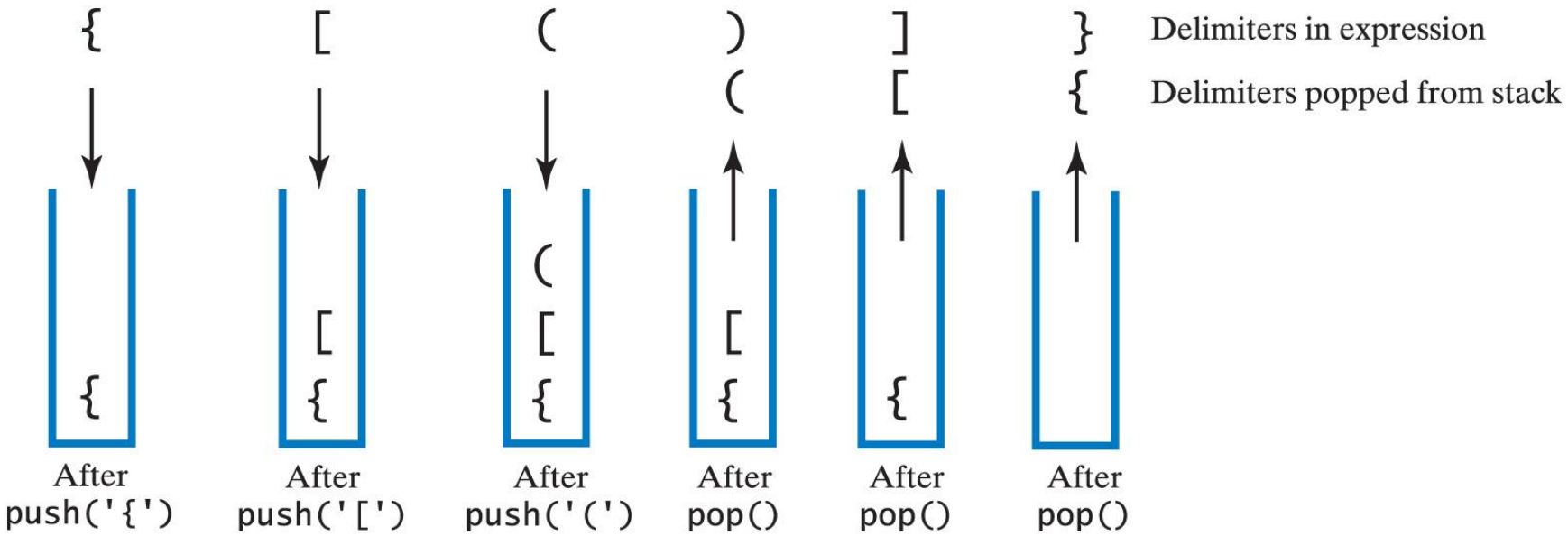


Fig. 21-3 The contents of a stack during the scan of an expression that contains the balanced delimiters **{ [( )] }**

Checking for Balanced `()`, `[]`, `{}` (4/7)

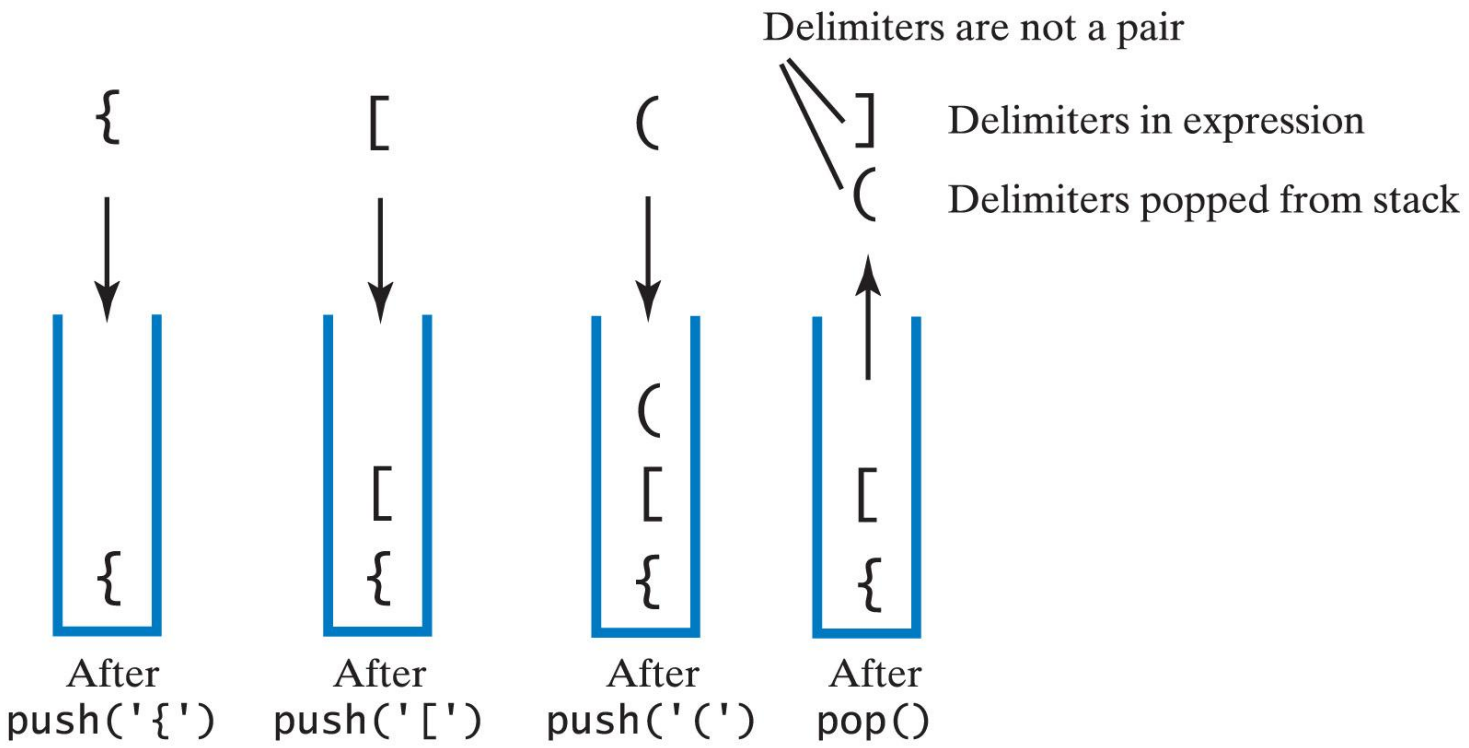


Fig. 21-4 The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[(])}`

Checking for Balanced `()`, `[]`, `{}` (5/7)

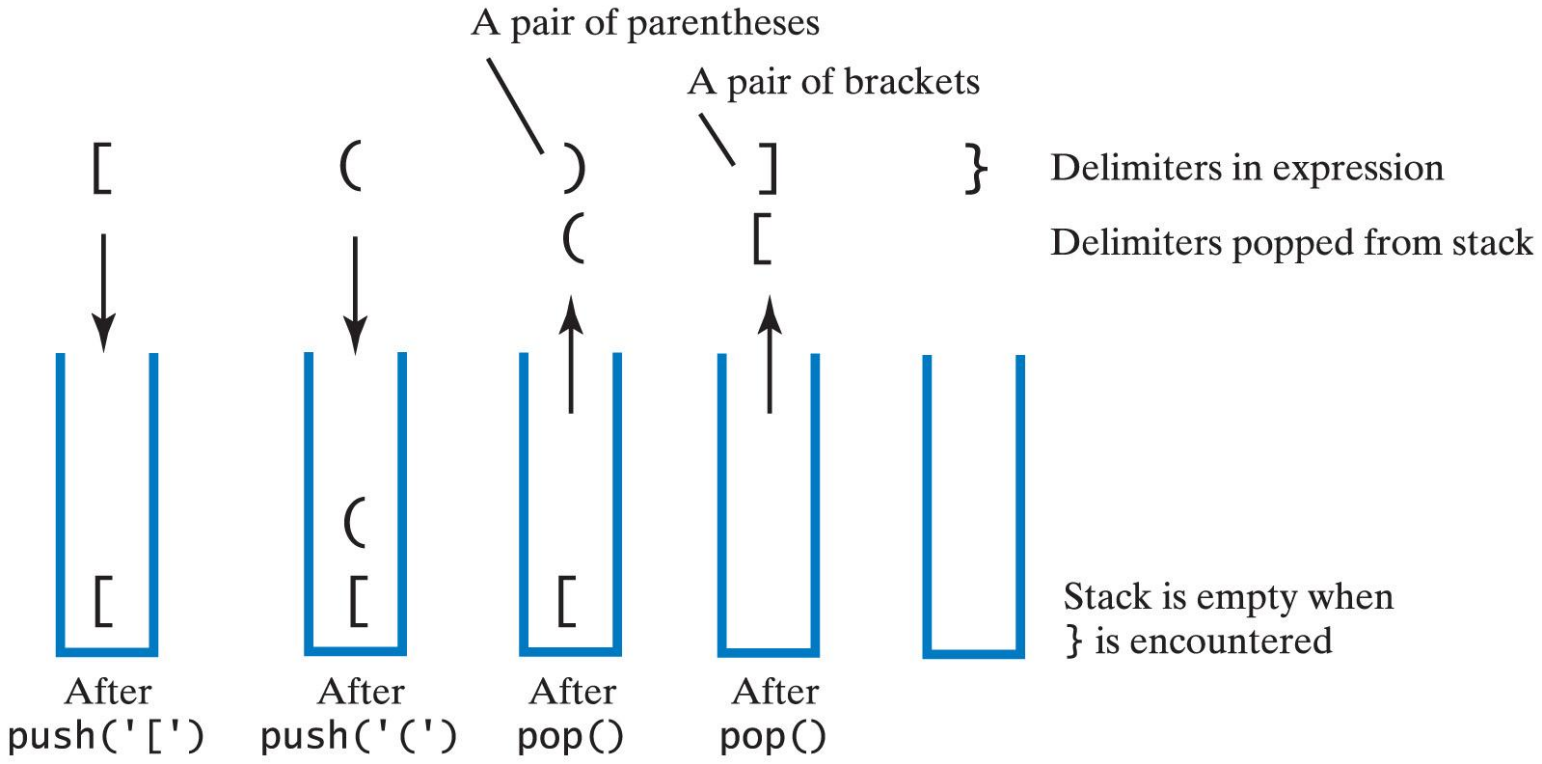


Fig. 21-5 The contents of a stack during the scan of an expression that contains the unbalanced delimiters **`[ ( ) ] }`**

Checking for Balanced `()`, `[]`, `{}` (6/7)

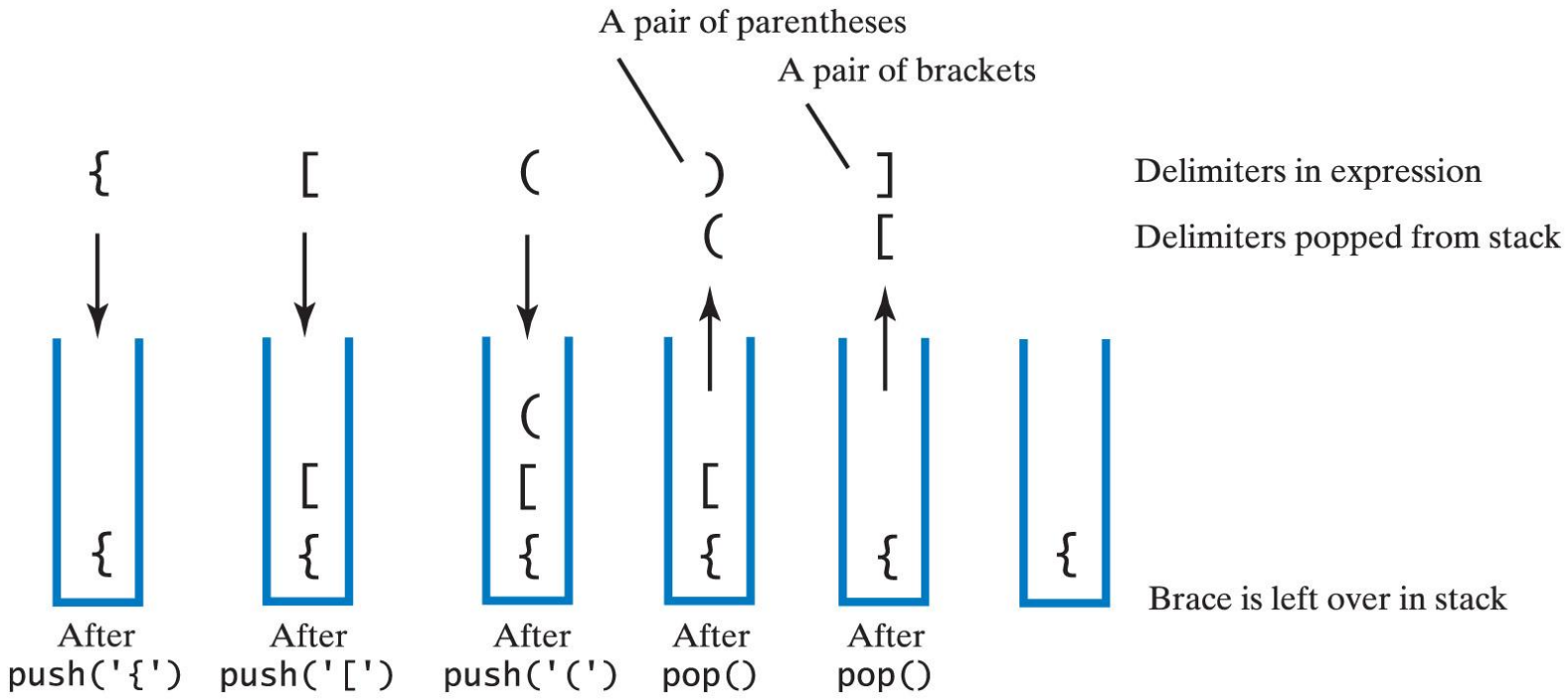


Fig. 21-6 The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[()]`

Checking for Balanced `()`, `[]`, `{}` (7/7)

- Algorithm 1.1 **`checkBalance`** - for checking for balanced parentheses in a string

PYQ: May 2022 Q2b

b) Describe how a stack may be used to check whether the parentheses in the following mathematical expression are balanced and matched.

[(3 + {7 - 6}) / {3 * (9 + 1)}] (7 marks)

PYQ: January 2023 Q2a

a) Stack is a collection of objects in which follows the last-in, first-out (LIFO) principle. Stack can be used for checking balanced brackets. Discuss how brackets are checked for balanced and match using stack based on Figure 2.

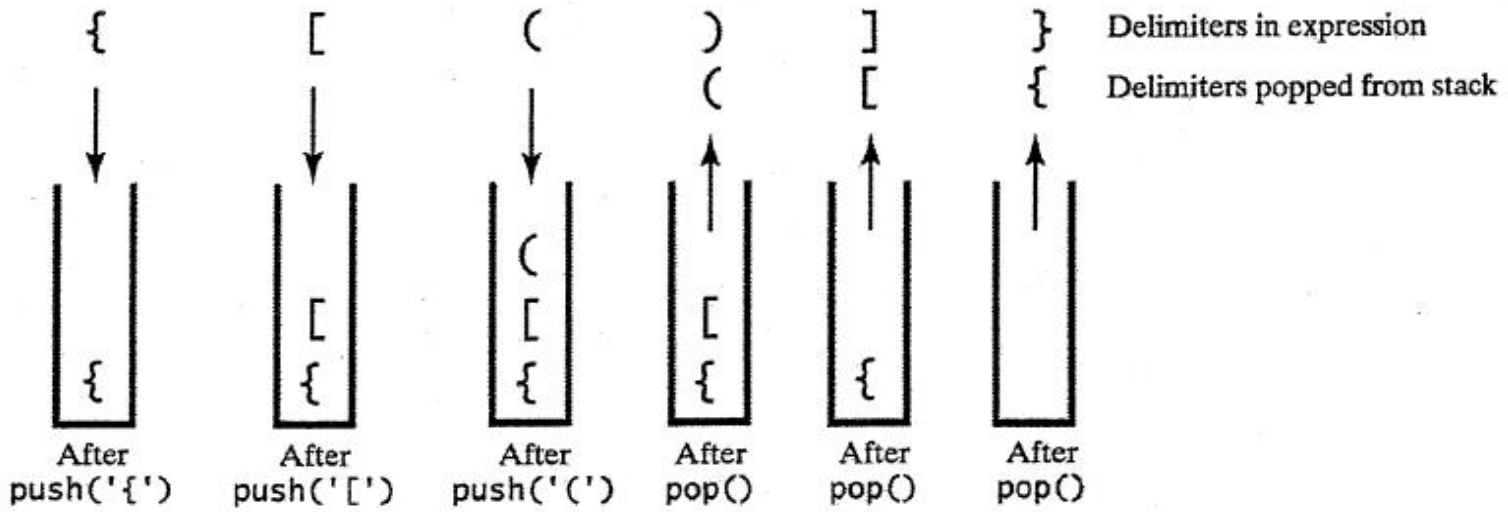


Figure 2: Checking for Balanced (), [], {}

(10 marks)

Infix, Prefix and Postfix 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

Exercise 1.1

Transform each of the following infix expressions to its postfix and prefix forms:

Infix	Postfix	Prefix
(a) a + b * c	a b c * +	+ a * b c
(b) a * b / (c – d)	a b * c d – /	/ * a b – c d
(c) a / b + (c – d)	a b / c d – +	+ / a b – c d
(d) a / b + c - d	a b / c + d -	- + / a b c d

Problem: Evaluating Postfix Expressions

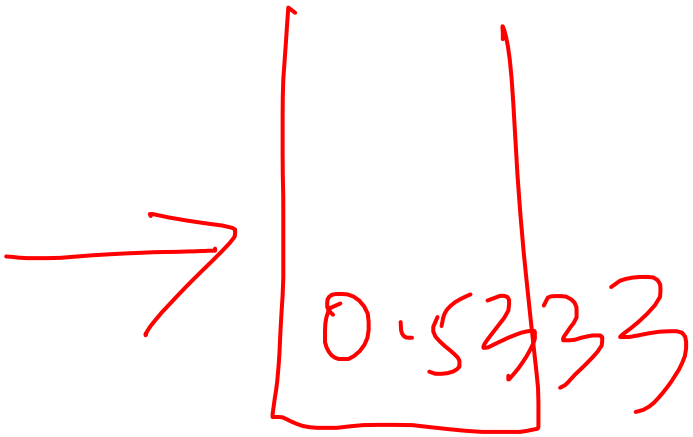
4	5	+	3	*	7	-
	=	9	3	*		
			=	27	7	-
					=	20

Evaluating Postfix Expression (1/3)

- Use a *stack of* operands
- Traverse the expression to process each character
  - If it is an **operand**, push it onto the stack
  - If it is an **operator**,
    - Pop the top two elements from the stack,

      (Note: the first pop execution returns the right operand while the second pop returns the left operand)
    - Perform the operation on the two elements, and
    - Push the result of the operation onto the stack

62+53*/


0.5233

Evaluating Postfix Expression (2/3)

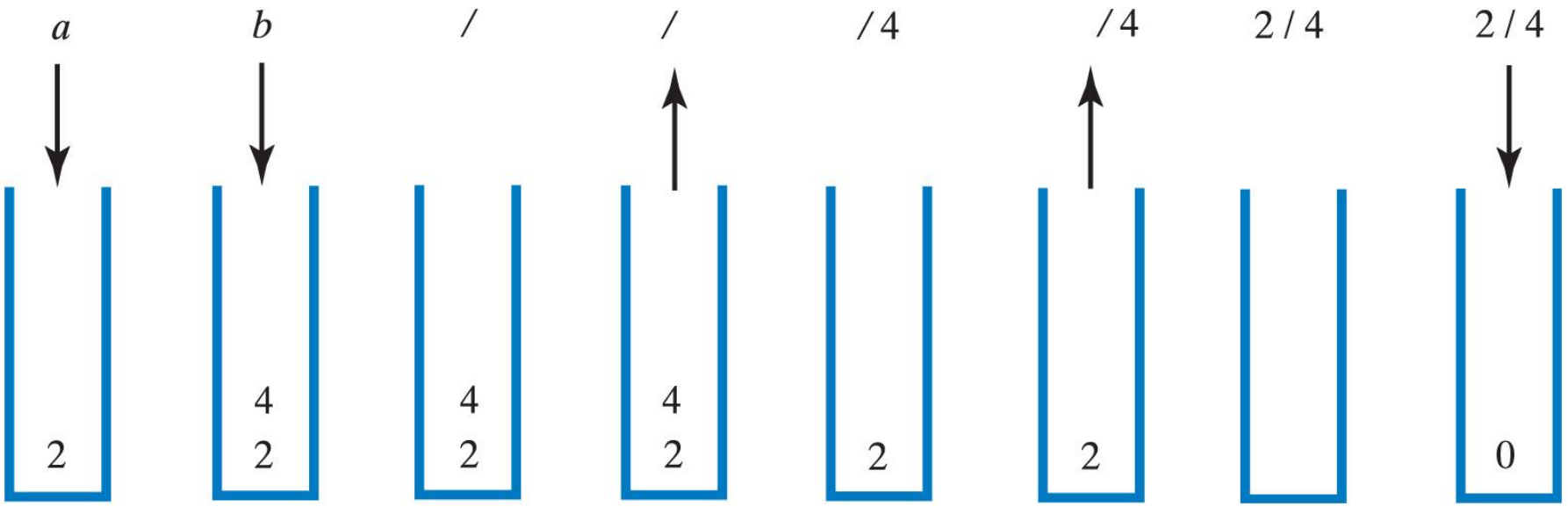


Fig. 21-10 The stack during the evaluation of the postfix expression **`a b /`** when **`a`** is 2 and **`b`** is 4

Evaluating Postfix Expression (3/3)

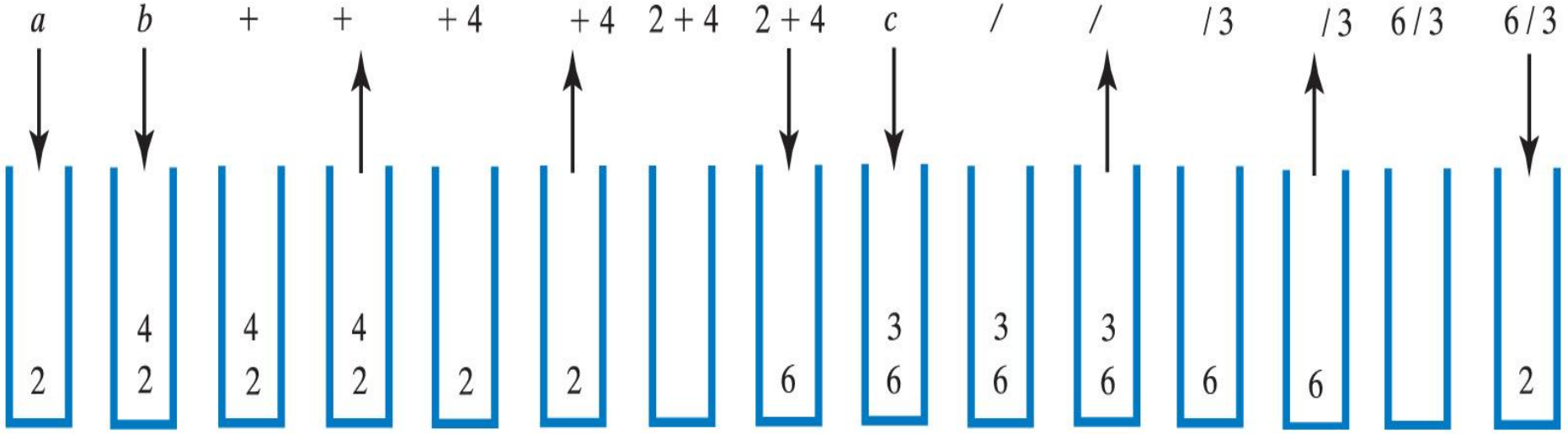


Fig. 21-11 The stack during the evaluation of the postfix expression **`a b + c /`** when **`a`** is 2, **`b`** is 4 and **`c`** is 3

PYQ: October 2022 Q2a

Question 2

a) **Stack** is a collection of objects in which the objects are inserted and removed according to the last-in, first out (**LIFO**) principle.

(ii) Describe using illustration how stack is used to evaluate postfix expression.

6 2 + 5 3 * /

(12 marks)

Why is postfix expression used?

- *Postfix notation is one of the few ways to represent algebraic expressions. It is used in computers because it is faster than other types of notations (such as infix notation) as parentheses are not required to represent them.*

- ***Source:***
- https://www.scaler.com/topics/postfix-evaluation/

The Program Stack (1/3)

- There is an area of memory in your computer known as the system/program stack which is used to allocate memory for method calls.
- An *activation record* is
  - A portion of memory on the program stack allocated to a method call
  - Used to store the called method's arguments, local variables and reference to the current instruction (program counter)

The Program Stack (2/3)

- When a method is called
  - Runtime environment creates activation record
  - Shows method's state during execution
- Activation record pushed onto the program stack
  - Top of stack belongs to currently executing method
  - Next method down is the one that called current method

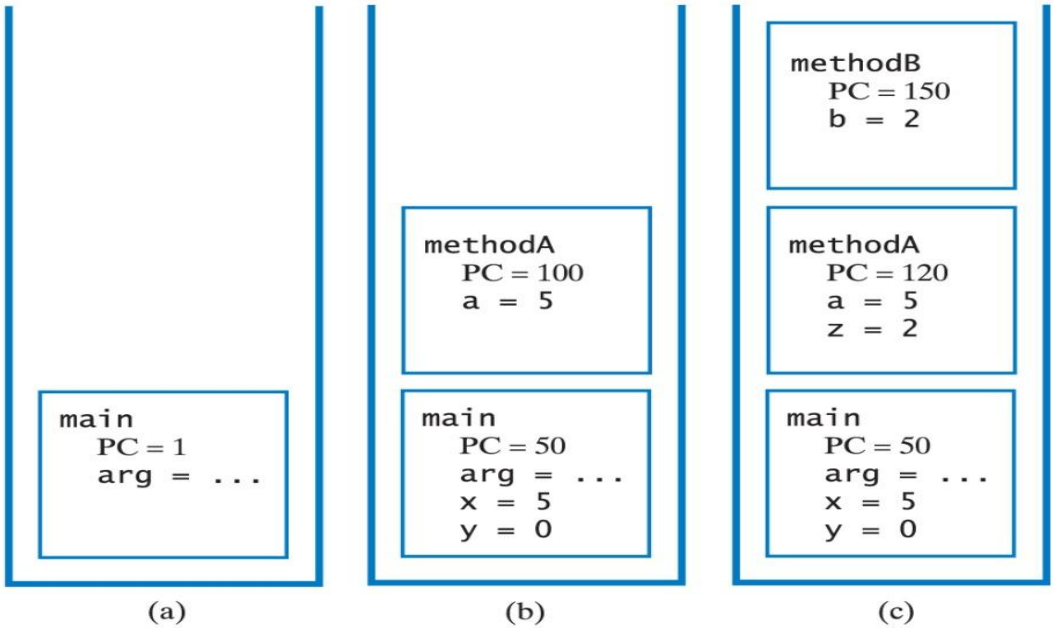
The Program Stack (3/3)

```
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. 21-13 The program stack at 3 points in time; (a) when **`main`** begins execution; (b) when **`methodA`** begins execution, (c) when **`methodB`** begins execution.

Queues

Queue: Definition

- A collection of objects that are inserted and removed according to the first-in, first out (***FIFO***) principle [2]. That is, the element that has been in the queue the longest will be the next one to be removed. Elements enter a queue at the ***rear*** or ***back***; elements are removed from the ***front***

- A linear collection of entries in which entries may be only removed in the order in which they are added. Typically, there is not provision to search for an entry in the queue. [4]

Queue: Basic Operations

Operation	Description
enqueue	Add an entry to the rear of the queue
dequeue	Remove the entry at the front of the queue
isEmpty	Check if the queue is empty
getFront	Retrieve (but do not remove) the front object in the queue
size	Return the number of entries in the queue

PYQ, June 2023 Q2a

Question

Question 2

a) Queue is a collection of objects which follows the first in, first out (FIFO) principle. Describe the characteristics and basic operations of a queue. (10 marks)

Answer

Operation	Description
enqueue	Add an entry to the rear of the queue
dequeue	Remove the entry at the front of the queue
isEmpty	Check if the queue is empty
getFront	Retrieve (but do not remove) the front object in the queue
size	Return the number of entries in the queue

Java Collections Framework: `java.util.Queue` interface

Modifier and Type	Method and Description
boolean	`add(E e)` Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.
E	`element()` Retrieves, but does not remove, the head of this queue.
boolean	`offer(E e)` Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
E	`peek()` Retrieves, but does not remove, the head of this queue, or returns `null` if this queue is empty.
E	`poll()` Retrieves and removes the head of this queue, or returns `null` if this queue is empty.
E	`remove()` Retrieves and removes the head of this queue.

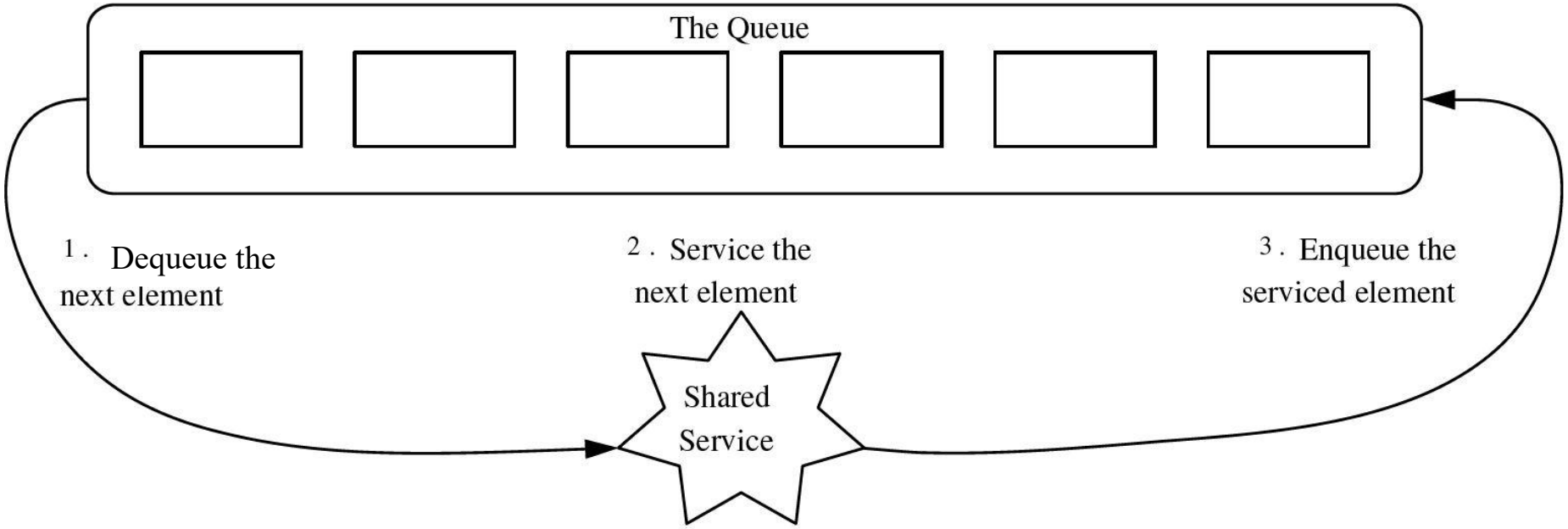
Java Collections Framework : **`ArrayBlockingQueue`** class

- **`java.util.concurrent.ArrayBlockingQueue`** is a concrete class that implements the `java.util.Queue` interface to support the FIFO principle.

Queue: Applications

- Processing of customer requests – *e.g.*, handling calls to the reservation center of a restaurant
- Print queues
- Round-robin schedulers – *e.g.*, to allocate a slice of CPU time to various applications running concurrently on a computer
- Simulation of queues

Problem: Round Robin Schedulers



Problem: Service Counters

- Single queue, multi-counters – *e.g.*, people at the Post Office take a number and wait for their number to be called

Problem: Using a Queue to Simulate a Waiting Line

Application:

- Businesses want to analyze the time that their customers must wait for service.
  - Waiting time ↓
  - Customer satisfaction ↑
  - No. of customers served ↑
  - Profits $$ ↑ 

Computer Simulation

- Computer simulation of a real-world situation is a common way to test various business scenarios.
- *E.g.* to simulate a waiting line with one serving agent (i.e. a line of people waiting for service at a single counter)
  - Customers arrive at different intervals and require various times to complete their transactions

81

Problem: Computing Profit in Sale of Shares

- Typically, people buy shares in a particular company over a period of time. E.g.,
  - Last year you bought 20 shares of Presto Pizza at RM45 per share;
  - Last month, you bought 20 additional shares at RM75 per share; and
  - Today you sold 30 shares at RM65 per share.

  What is your profit? Which of your 40 shares did you sell?
- When computing profit in the sale of shares, you must assume that you sell shares in the order in which you purchased them (i.e. FIFO).
- Design a way to record your investment transactions chronologically and to compute the profit of any sale of shares.

Queue of Shares of Stock

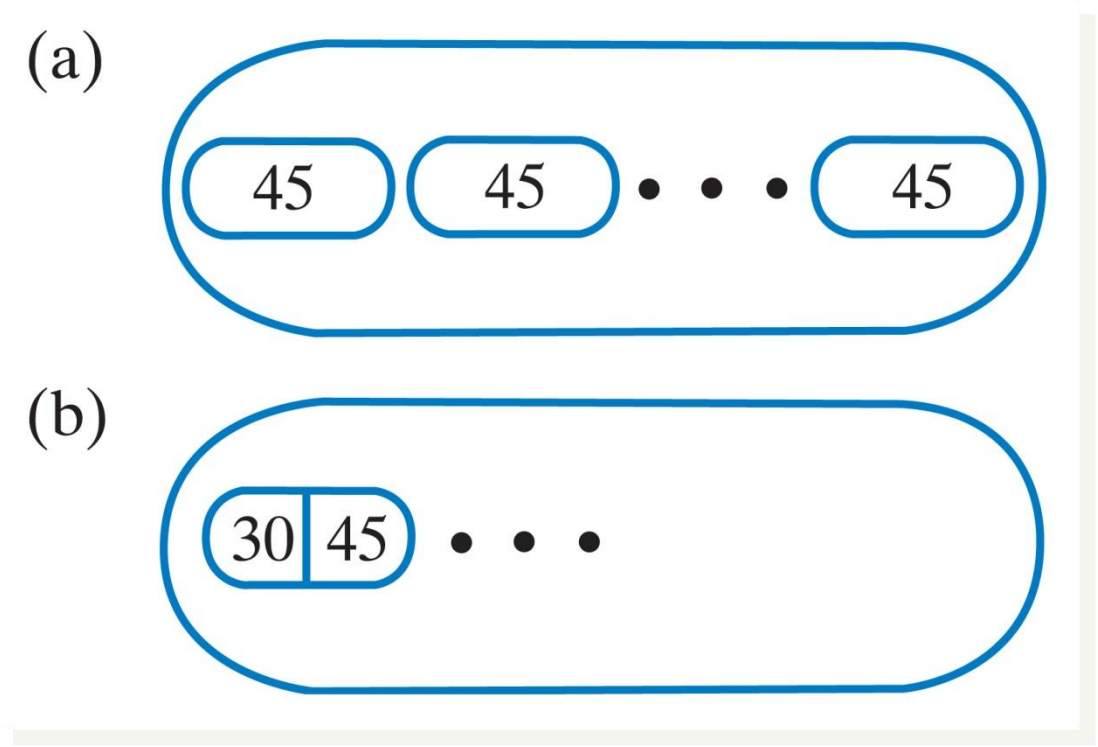


Fig. 23-9 A queue of (a) individual shares of stock; (b) grouped shares.

Sample Code: Computing Profit in Sale of Shares

In **`Chapter1\examples\`**

a) **`SharePurchase.java`**
   - A **`SharePurchase`** object stores the cost of a single share.

b) **`SharePortfolio.java`**
   - A **`SharePortfolio`** object records stock purchases in chronological order using a queue.
   - Note constructor and methods **`buy`** and **`sell`**

c) **`SharePortfolioDriver.java`**

Exercise 1.2

For each application below, identify if a stack or a queue would be the appropriate data structure. Provide justifications for your choices.

(a) page-visited history in a web browser **Stack**

(b) access to shared resources **Queue**

(c) undo-sequence in a text editor **Stack**

Other Past Year Questions on Chapter 1

2021 December Q1a

Question 1

a) Code reusability and maintainability can be achieved using the design principles such as Abstraction and Encapsulation. Distinguish the concept of Encapsulation and Abstraction by providing necessary examples to illustrate your answer. (10 marks)

2021 December Q1b

b) An Abstract Data Type (ADT) specification does not indicate how to store the data or how to implement the operations. It consists of data that has certain characteristics and a set of operations that can be used to manipulate the data. Provide the characteristics and operations for the following ADTs. Include relevant examples in your answers.

(i) List ADT
(ii) Stack ADT
(iii) Queue ADT

(9 marks)

2021 October, Q1a

Question 1

a) *Beautiful today* is a desktop application which allows the user to record their daily activities in an electronic journal or diary. When an entry is saved, the mood, date, and text will be recorded. The user can add more than one image to each entry. The main page of the application shows the latest 10 entries of the diary. Suggest a suitable Abstract Data Type (ADT) that can be applied in *Beautiful Today* application. Justify your answer and include explanation on what object would be stored in this ADT. You may make your own assumptions in your answer. (10 marks)

2021 April, Q1a

Question 1

a) Describe a real-life computer application or system in which a stack abstract data type (ADT) would be used. In your answer, explain what objects would be stored in the stack, why a stack is the most suitable ADT to use, and also include an example with diagrams to illustrate the use of the stack in your given application. (5 marks)

2020 December, Q1a

Question 1

a) Consider that you are developing a software for karaoke centers to host karaoke shows and playing karaoke songs. Besides displaying the latest new songs, the songs in the system are grouped by the gender of the singers, the name of the singer, language, region, musical genre, etc. The system can sort the songs according to the customer's selection. For example, when a customer is selecting the songs based on title, the titles are sorted ascendingly; when a customer is selecting the songs based on singer, the singers are sorted according to their surnames; and so on. The system allows multiple customers to add songs to their own playlists and then they have to wait for their turn. For every customer, it will allow each person to sing their chosen songs for around 10 minutes. After that, the system will select the next user in line to sing. This will repeat until all songs are played, or the time is up. Any customer can remove any of their selected songs from the playlist at any time, and they can move the song to the front of the playlist as well. The system is able to sync the first 10 songs in the playlist to the customers' smart phones in real time, so that they know when is their turn even they are outside the karaoke room.

Based on the case study above, analyze how list, stack and queue can be applied in the system. Explain with relevant examples.

(i) Propose **ONE** (1) application of list in the karaoke system with an example. (3 marks)

(ii) Propose **ONE** (1) application of stack in the karaoke system with an example. (3 marks)

(iii) Propose **ONE** (1) application of queue in the karaoke system with an example. (3 marks)

2020 October, Q1b & Q1c

b) When is it suitable to use a Queue ADT? Explain your answer by giving an example of a system or an application that would use a Queue ADT. Your answer should include what is the object in the queue and how this ADT is used. (10 marks)

c) Analyze **TWO (2)** benefits of using abstraction in the data structures. Justify your answer with relevant examples. (8 marks)

Learning Outcomes

You should be able to

- Explain the benefits of *abstraction*.
- Apply the use of the *list*, *stack* and *queue*. ADTs appropriately to solve problems.

References

Main references

1. Carrano, F. M., 2019, Data Structures and Abstractions with Java, 5th edn, Pearson
2. Liang, Y.D., 2018. Introduction to Java Programming and Data Structures.11th ed.United Kingdom:Pearson

Additional references

1. Dale, N, Joyce, DT. and Weems, C., 2018. Object-Oriented Data Structures Using Java. 4th ed. Burlington MA: Jones & Bartlett Learning