BACS2063 Data Structures and Algorithms

Applications of Abstract Data Types (ADTs)

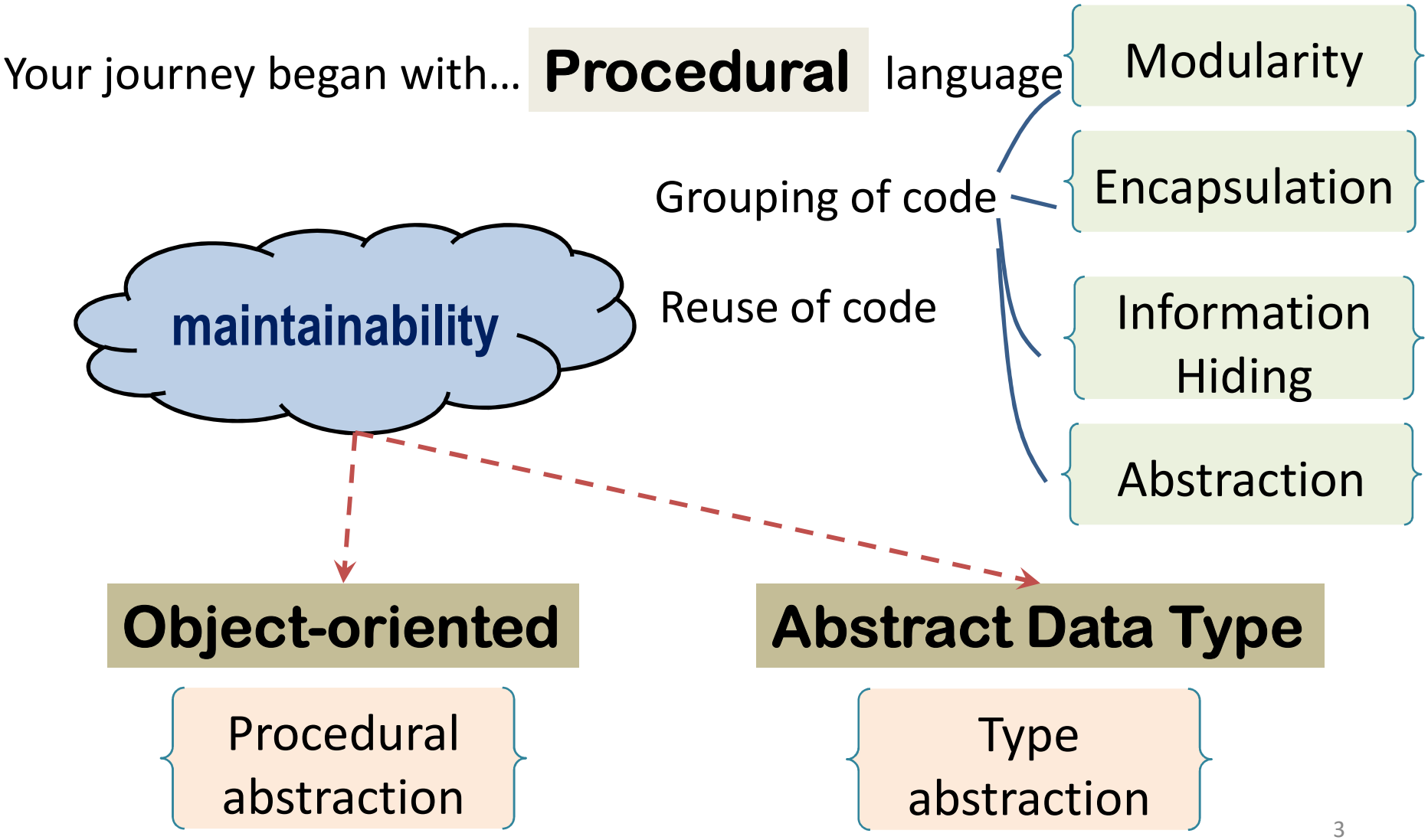
Chapter 1

Learning Outcomes

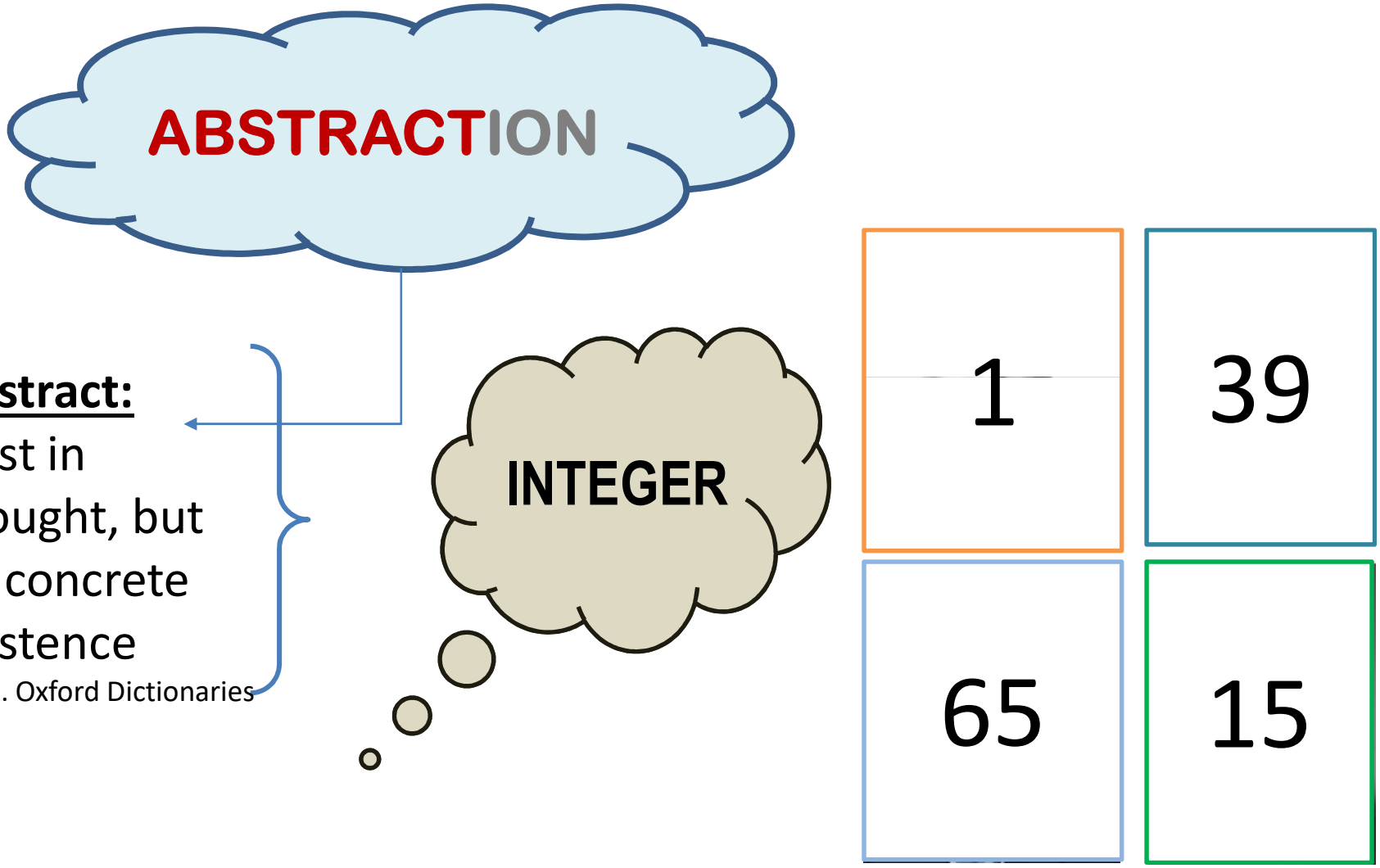
At the end of this lecture, you should be able to

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

Why Data Structures



Why Data Structures



Why Data Structures

ABSTRACTION

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

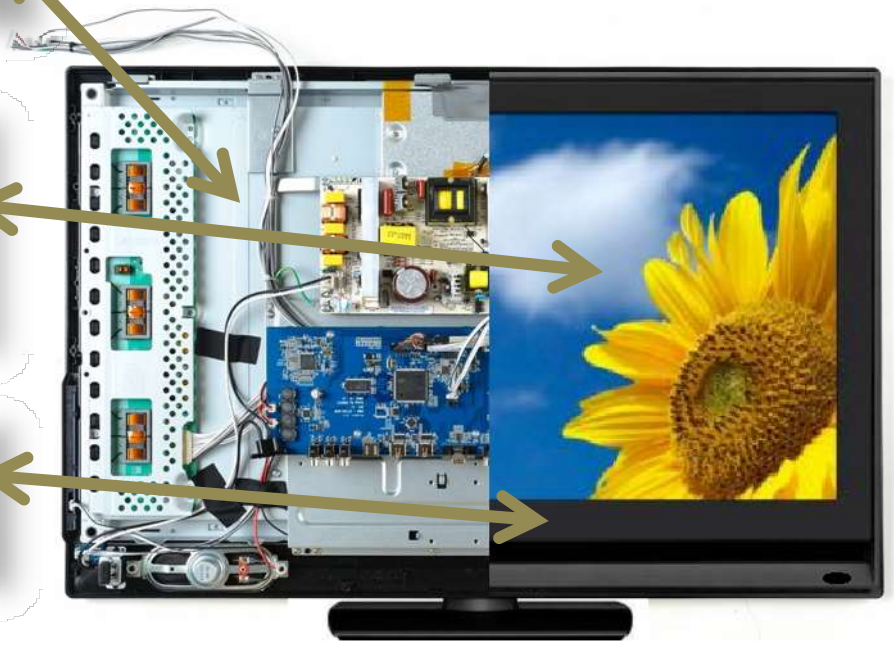
✓User can focus on the "big picture" but not the details.
✓Developer can switch implementation details but user won't be aware

Example:
Integer.parseInt()

Implementation

Abstraction (Interface and Operation)

Encapsulation (hiding details)



Why Data Structures

User-defined Type

Type abstraction – can use to declare variables directly but the representation cannot be inspected

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

- LIST
- STACK
- QUEUE

The implementation of ADT often referred as **DATA STRUCTURE**, using some **collection** of constructs and primitive data types

List, Stack, Queue

Which structure should be used?

(A)

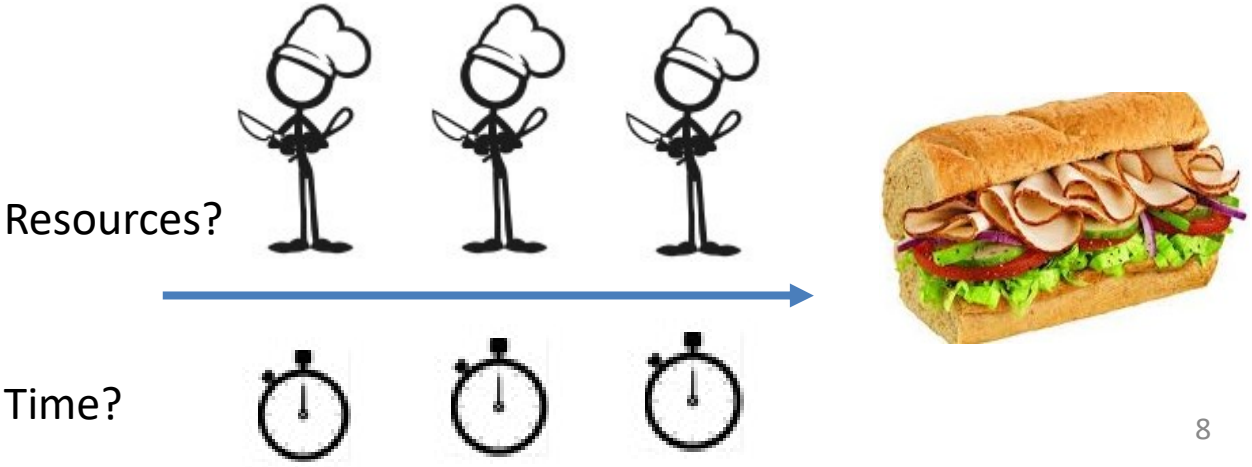
(B)

(C)

Algorithm: Definition

- A step-by-step procedure for performing some task in a finite amount of time [2]

vs.



Why Algorithms?

Algorithm determines

Time Efficiency

- **Time** taken to complete a task

Space Efficiency

- The use of resources

Data Structure: Definition

- A systematic way of organizing and accessing data [2]
- A representation of data and the operations allowed on that data [4]

Data Structure: Benefits

- Reusability
- Maintainability

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.

Linear Structures

- The list, stack and queue are referred to as linear structures [3]
- Each of them is a collection that stores its entries in a linear sequence
- They differ in the restrictions they place on how these entries may be added, removed, or accessed

Lists

List: Definition

- A **collection** of items in which the **items have a position** [4]
- A linear collection of entries in which entries may be **added**, **removed**, and **searched** for without restriction [3]

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

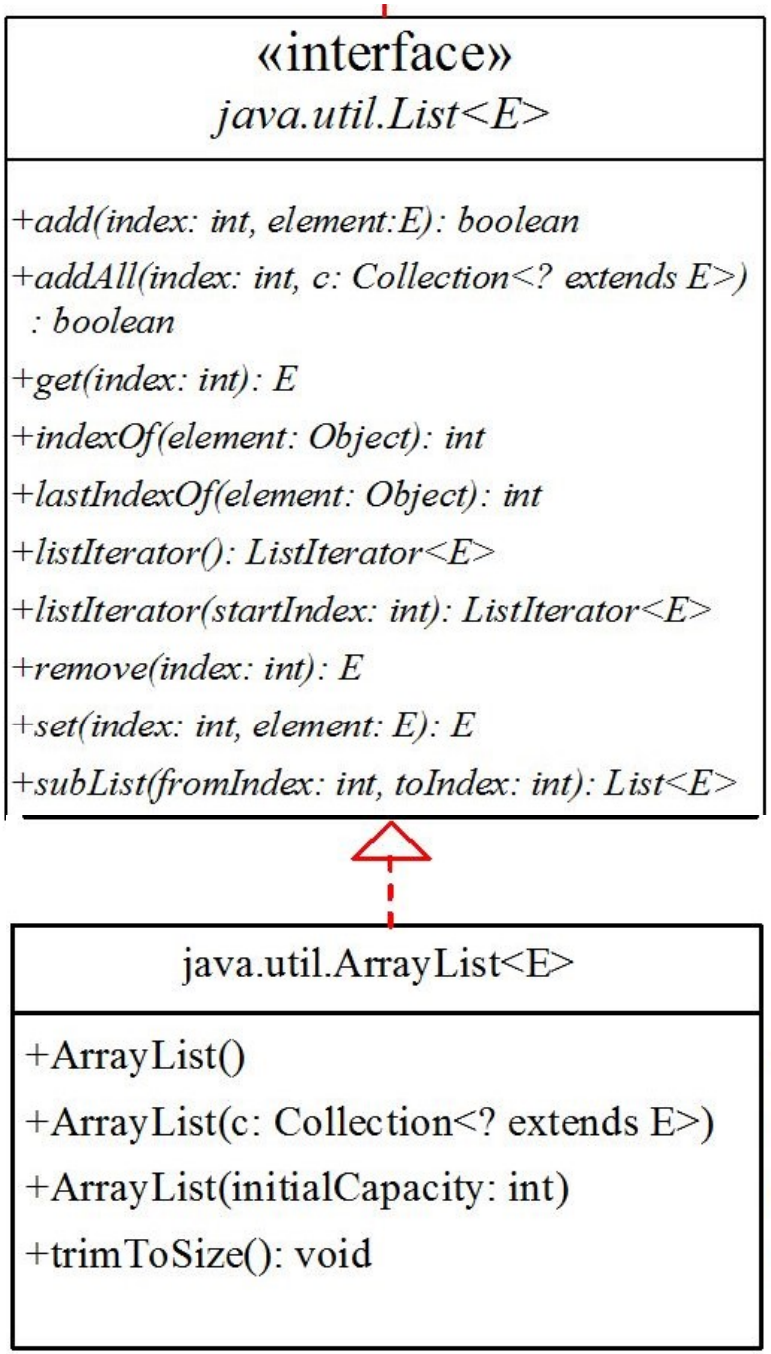
```
import java.util.ArrayList;
import java.util.List;

private List<String> nameList = new ArrayList<>();
// add (e)
nameList.add("Lee");
nameList.add("Kim");
//remove(i)
nameList.remove("Kim"); or nameList.remove(1);
//get(i)
nameList.get(1);
//clear()
nameList.clear();
```

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.



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.

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.

List: Applications

- **Lists of things** – **task lists, list of names, playlists**, etc
- **High-Precision Arithmetics** – to represent very long integer values

Note: Types of Lists

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

Problem: Registration of Runners in a Marathon

Sample Code:
Registration of Marathon Runners

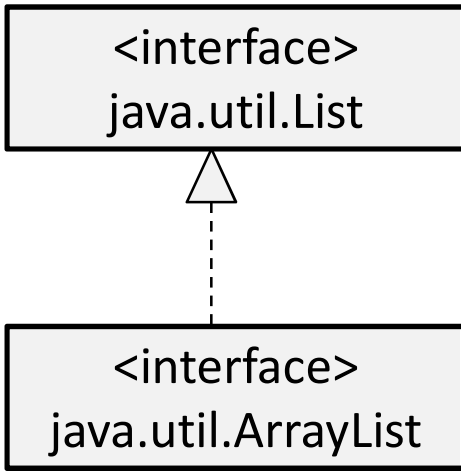
In **`Chapter1\runningman\`**

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

List: Discussion

a. Is it legal to declare a reference of type `List` and assign it with the address of an `ArrayList` object?

b. What is the difference between (i) and (ii) in terms of the methods available to `listRef`?




```
(i)List<Runner> listRef = new ArrayList<>();

(ii)ArrayList<Runner> listRef = new ArrayList<>();
```

BASIS FOR COMPARISON	LIST	ARRAYLIST
Basic	List is an Interface	ArrayList is a standard Collection Class.
Syntax	interface List	class ArrayList
Extend/Implement	List interface extends Collection Framework.	ArrayList extends AbstractList and implements List Interface.
Namespace	System.Collections.Generic.	System.Collections.
Work	It is used to create a list of elements (objects) which are associated with their index numbers.	ArrayList is used to create a dynamic array that contains objects.

Stacks

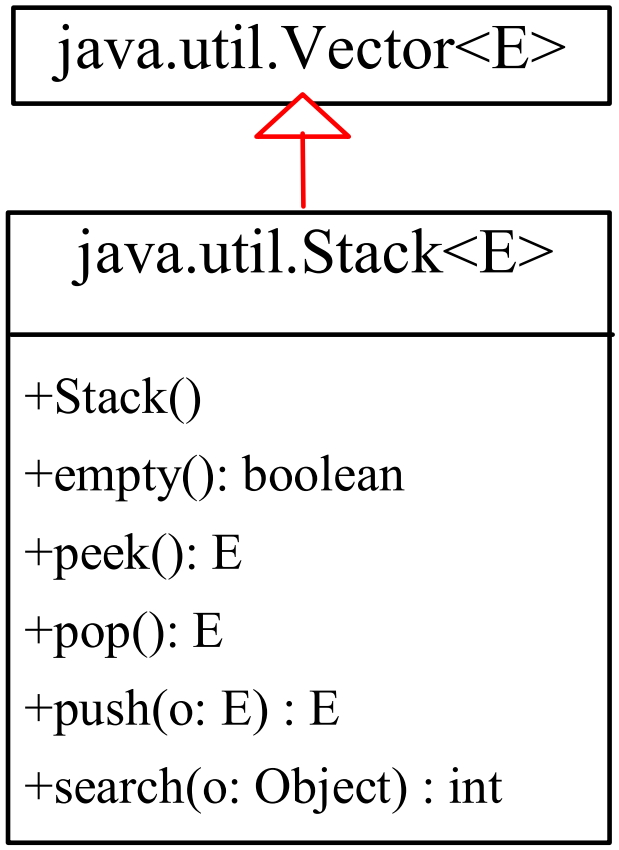
Stack: Definition

- A collection of objects in which the objects are **inserted and removed according to the last-in, first out (*LIFO*) principle** [2]
- 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** [3]
- 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**

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.

```
import java.util.Stack;
public class StackDemo {
   public static void main(String a[]){
      Stack<Integer> stack = new Stack<>();
      System.out.println("Empty stack : "  + stack);
      System.out.println("Empty stack : "  + stack.isEmpty());

      stack.push(100);
      stack.push(102);
      stack.push(103);
      stack.push(104);
      System.out.println("Stack elements are : "  + stack);
      System.out.println("Stack: Pop Operation : "  + stack.pop());
      System.out.println("Stack: After Pop Operation : "  + stack);
      System.out.println("Stack : search() Operation : "  +
stack.search(1002));
      System.out.println("If Stack is empty : "  + stack.isEmpty());
   }      }
```

```
run:
Empty stack : []
Empty stack : true
Stack elements are : [100, 102, 103, 104]
Stack: Pop Operation : 104
Stack: After Pop Operation : [100, 102, 103]
Stack : search() Operation : -1
If Stack is empty : false
BUILD SUCCESSFUL (total time: 1 second)
```

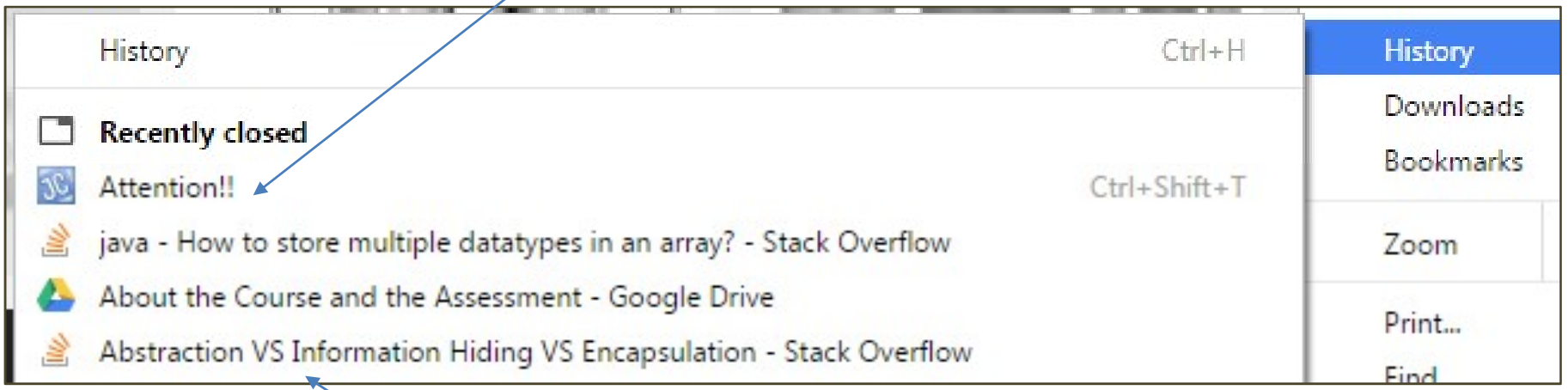
Background Information

Stack Trivia

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

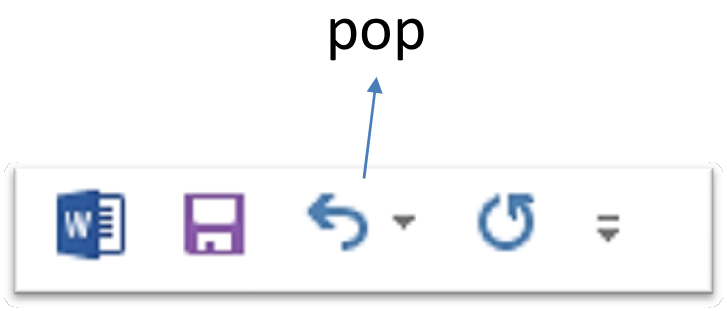
Stack: Application

Last In



First In

Stack: Application



STRUCTURE

First In

Last Out

Last In

First Out

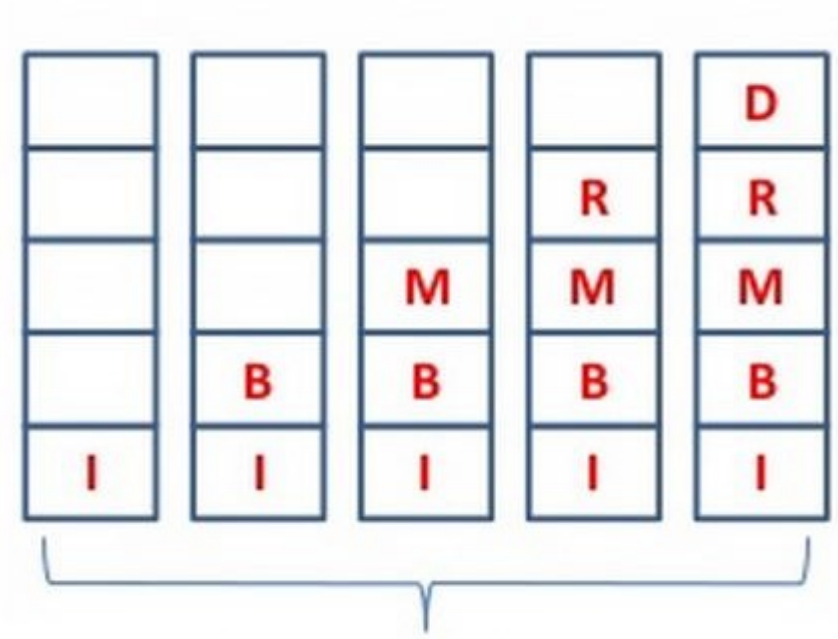
Stack: Applications

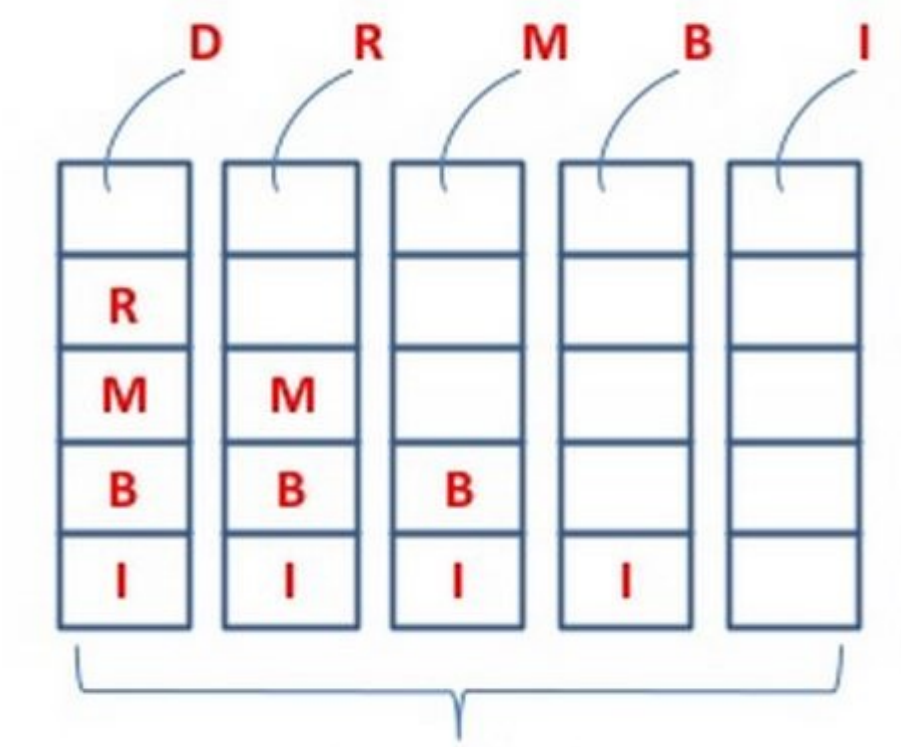
- Any application that requires LIFO processing – **checking for palindromes** (e.g. "noon", "kayak"), reversing a string, etc.
- Program stack – **to keep track of method calls**

Problem: Reversing a String

E.g.

I/P String is I B M R D





O/P String is D R M B I

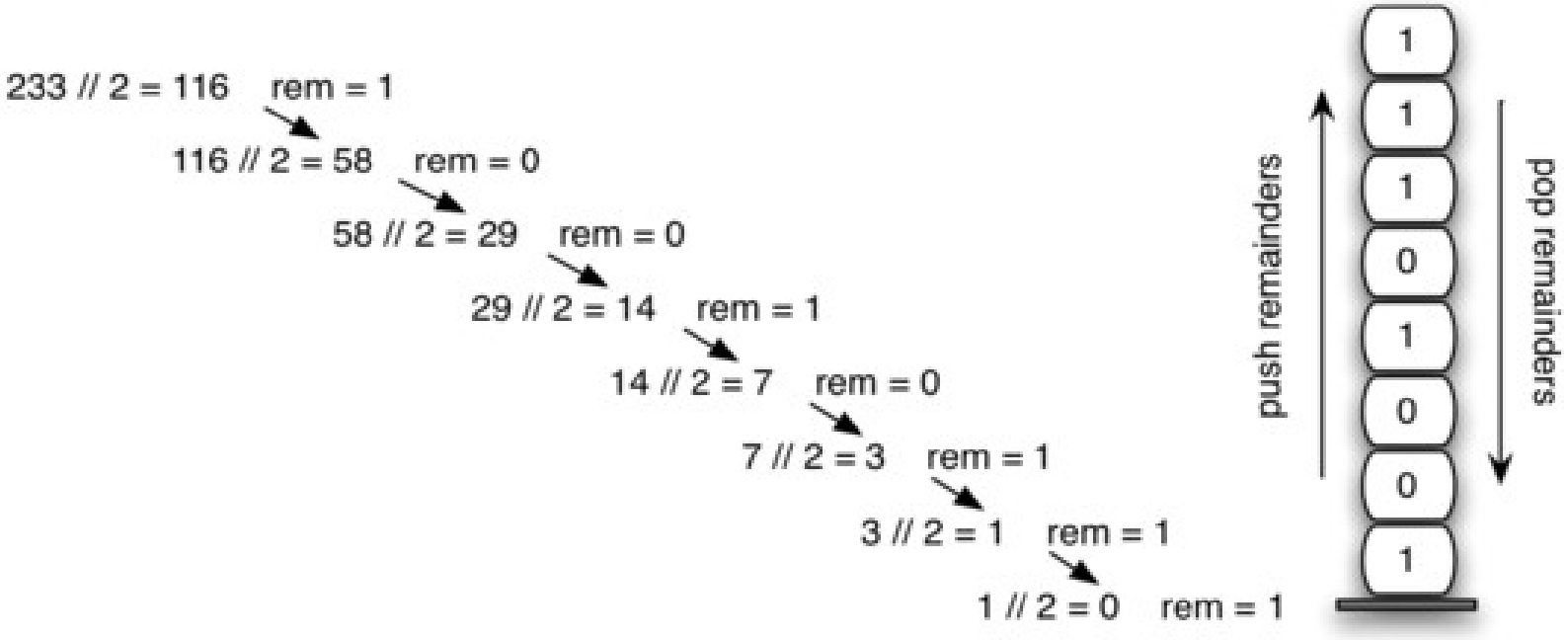
Sample Code: Reversing a String

In **`Chapter1\samplecode\`**

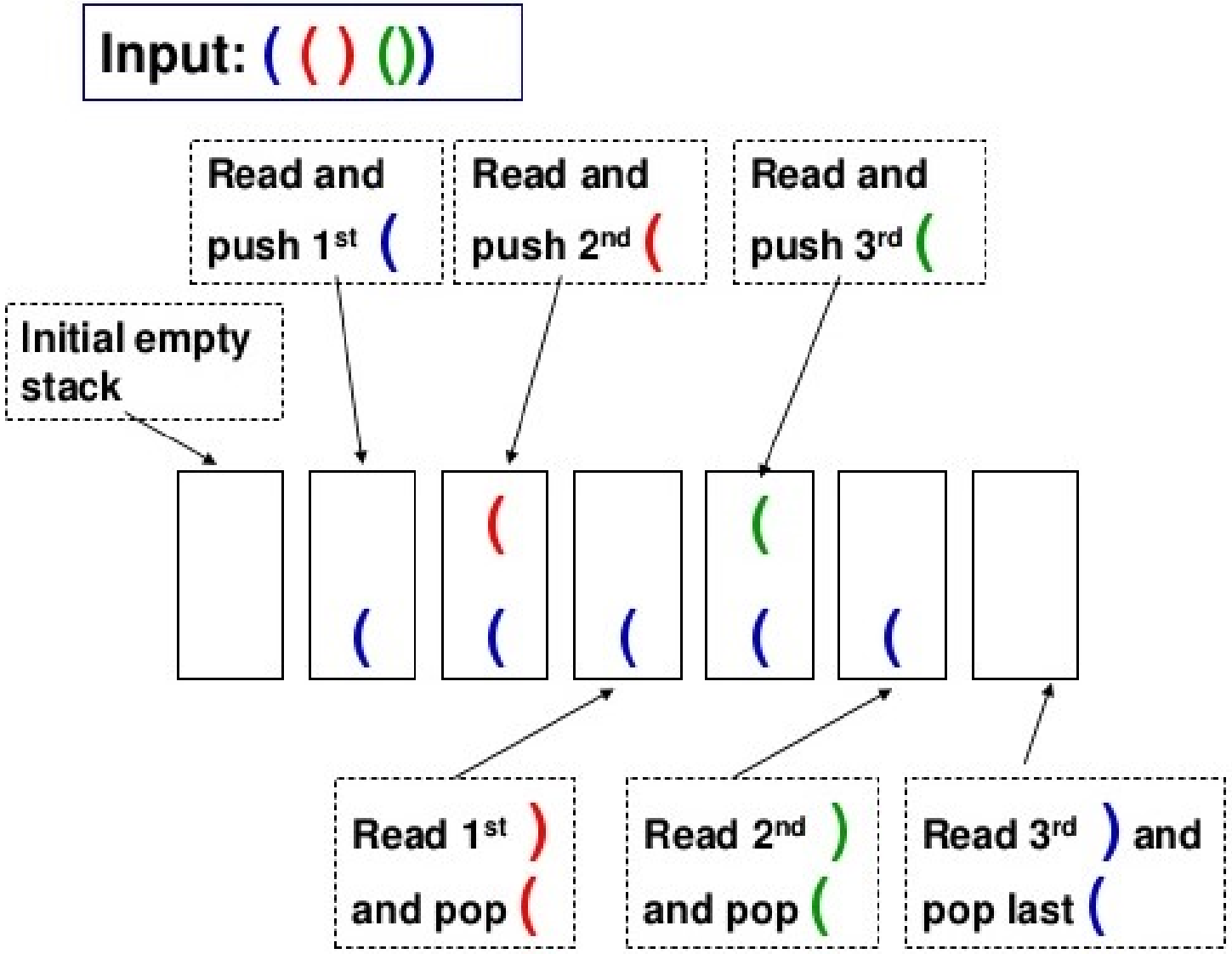
- **`StringReversal.java`**

Stack: Discussion

- Converts a number to its equivalent binary (base-2) number using a stack, and then returns the binary number as a string value.



Problem: Matching Brackets



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)

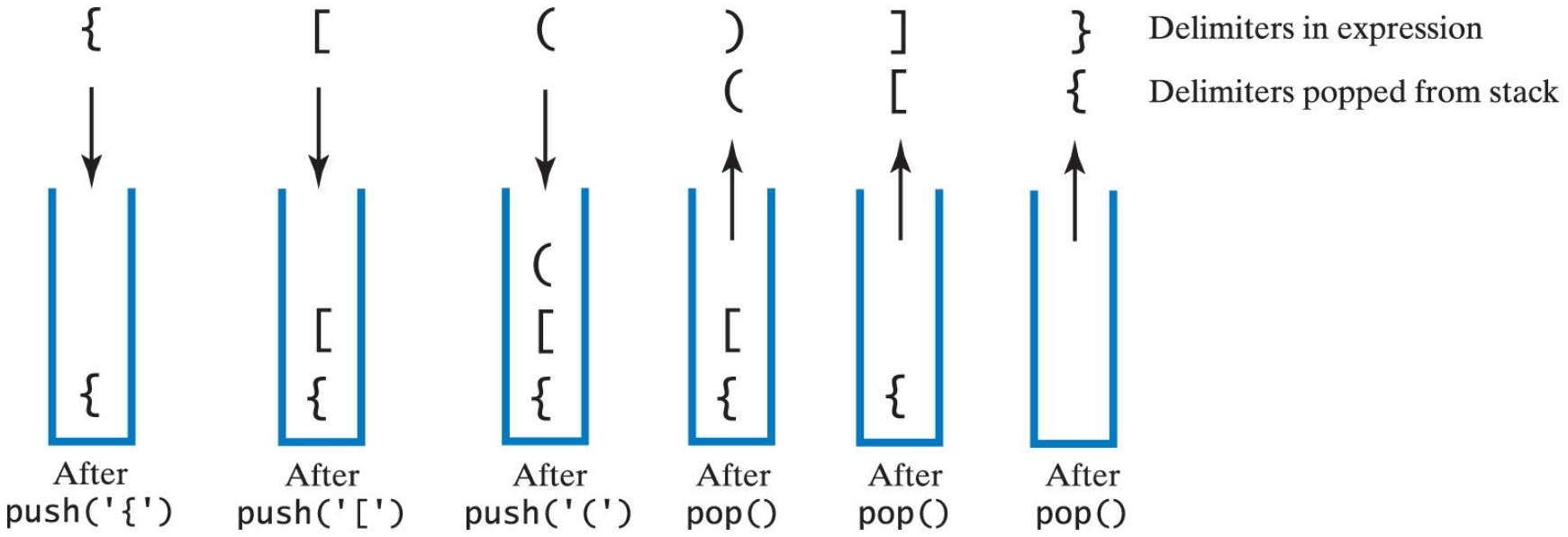


Fig. 21-3 The contents of a stack during the scan of an expression that contains the balanced delimiters **{ [( )] }**

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

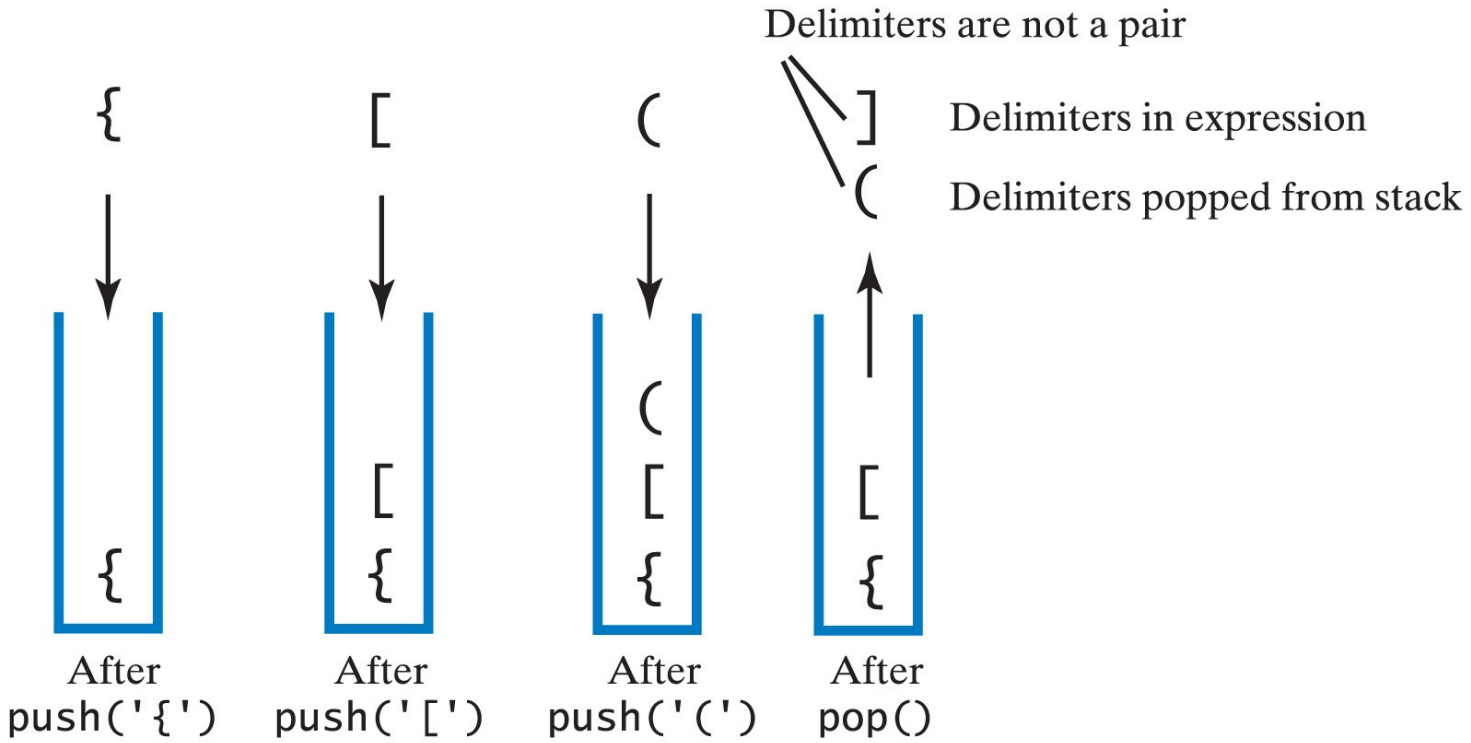


Fig. 21-4 The contents of a stack during the scan of an expression that contains the unbalanced delimiters **{ [( ]) }**

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

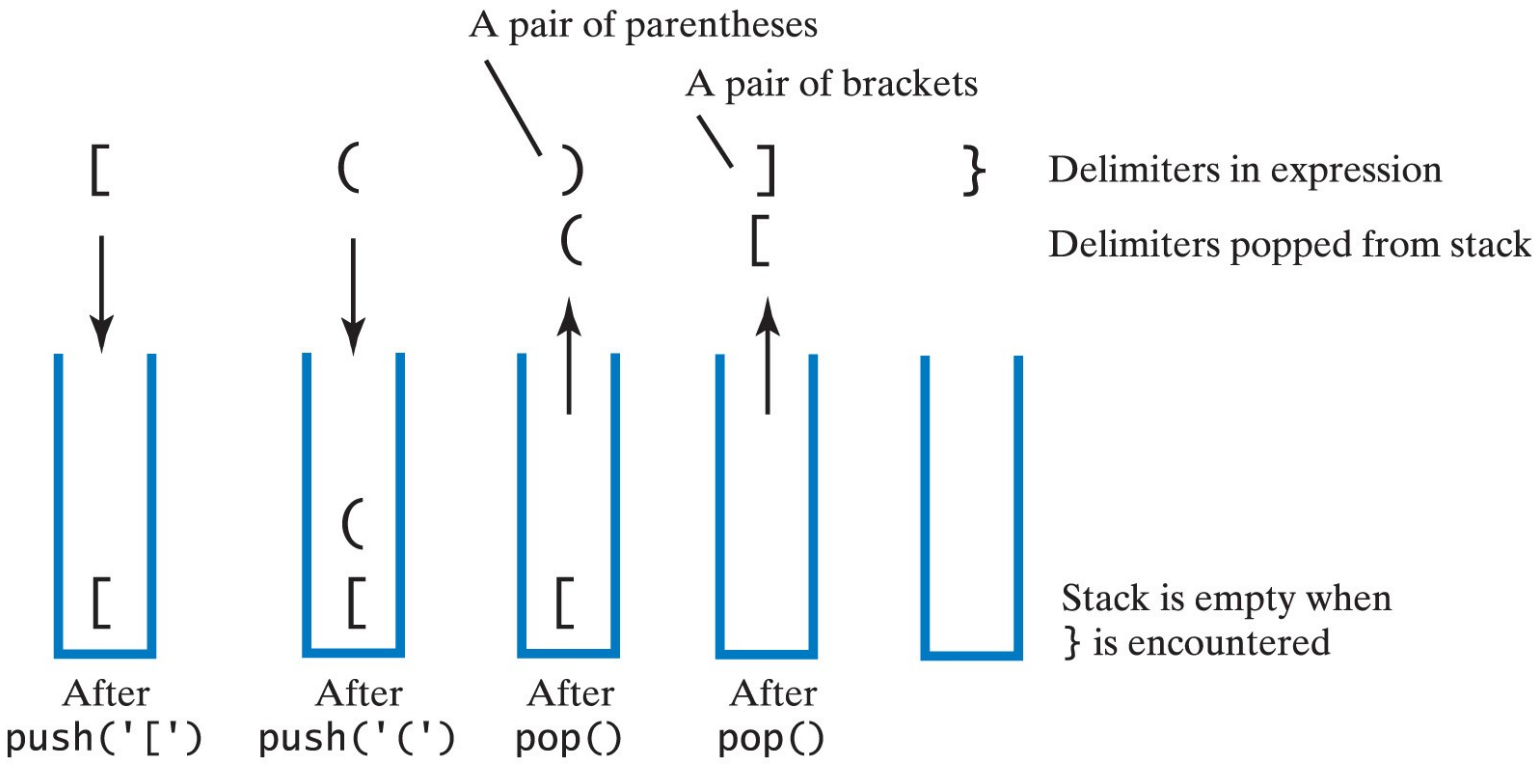


Fig. 21-5 The contents of a stack during the scan of an expression that contains the unbalanced delimiters `[()]}`

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

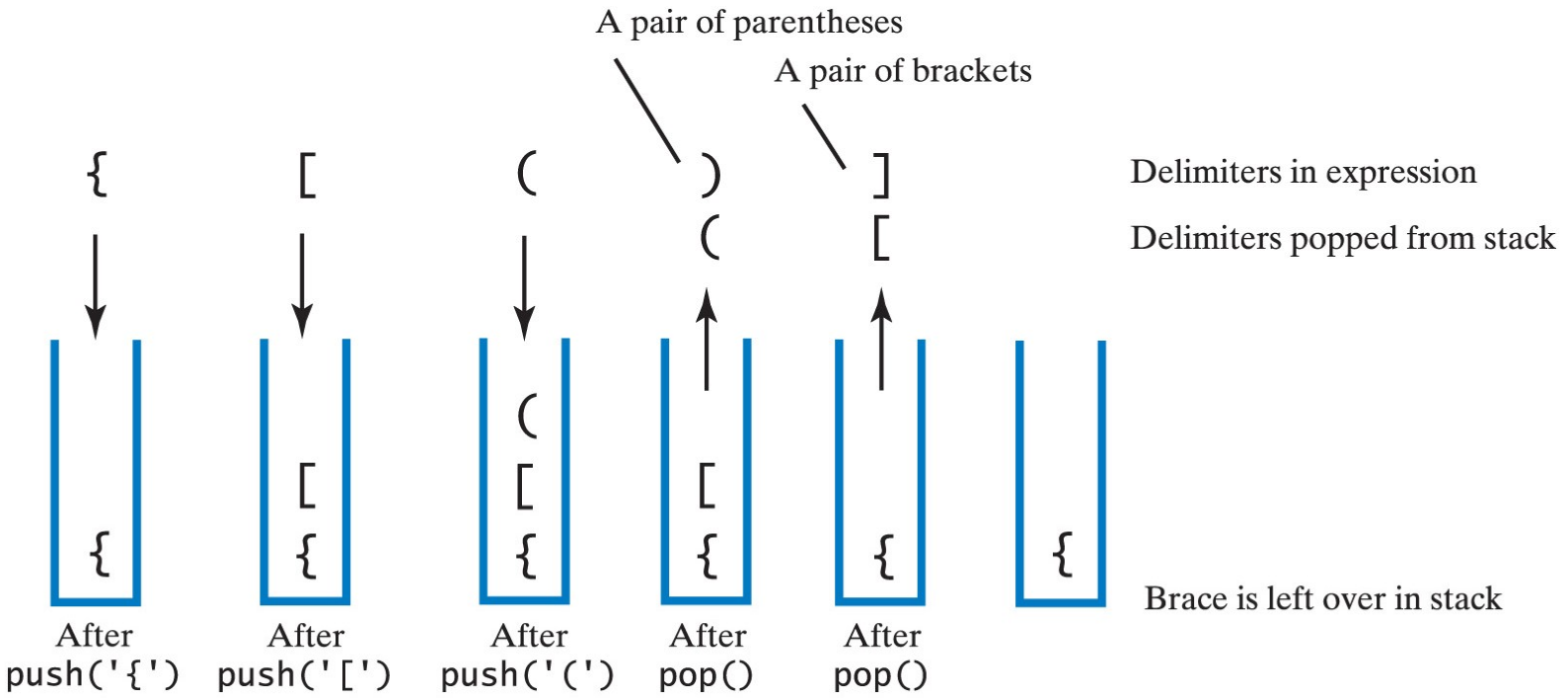


Fig. 21-6 The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[()]`

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

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

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

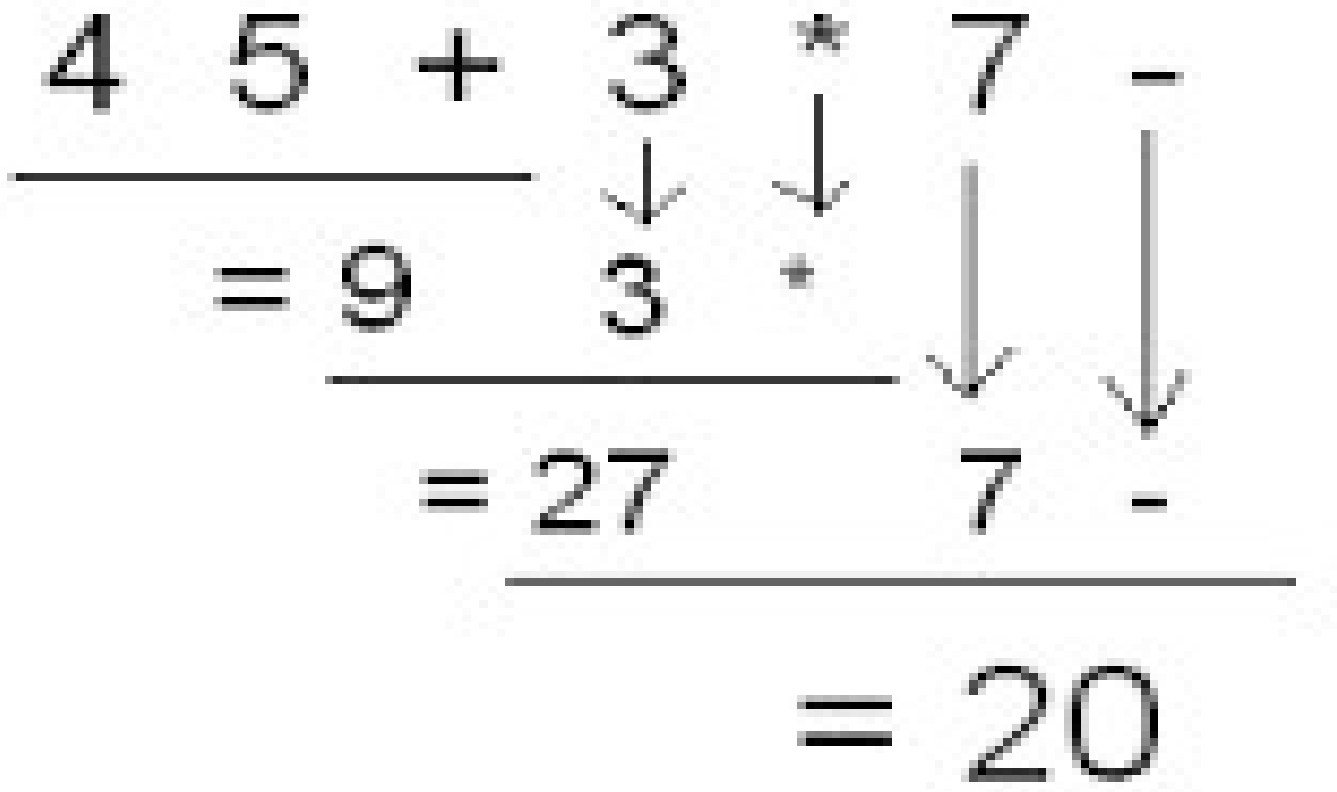
Exercise: Prefix & Postfix

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

Alg



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

Evaluating Postfix Expression (2/3)

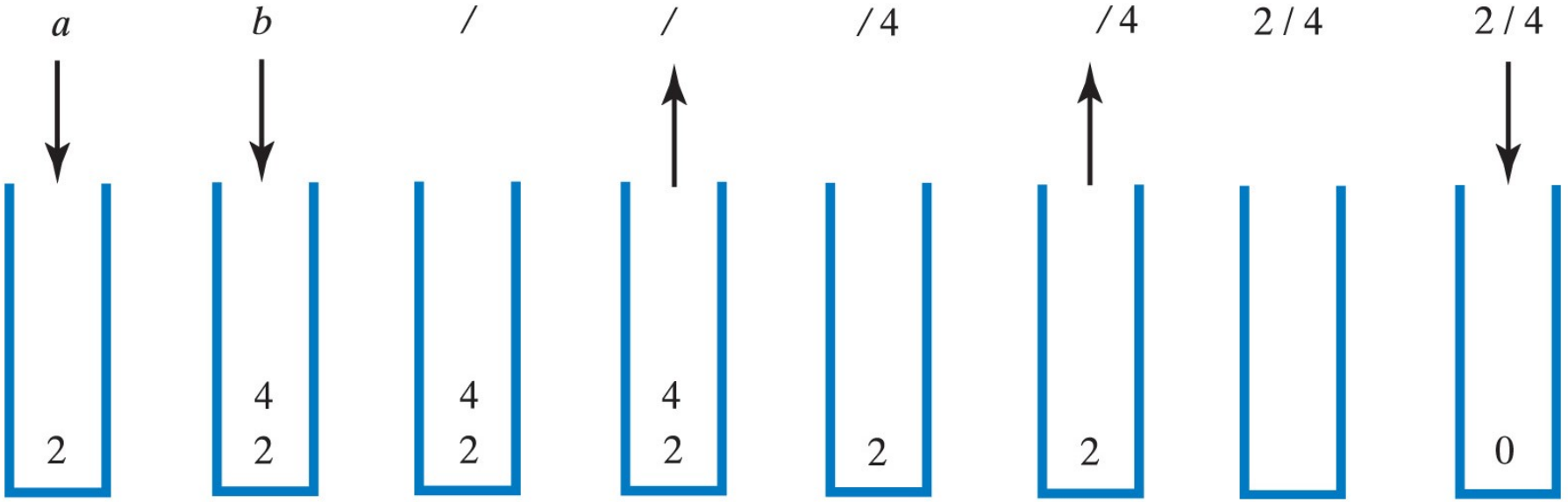


Fig. 21-10 The stack during the evaluation of the postfix expression **`a b /`** when **`a`** is 2 and **`b`** is 4

Evaluating Postfix Expression (3/3)

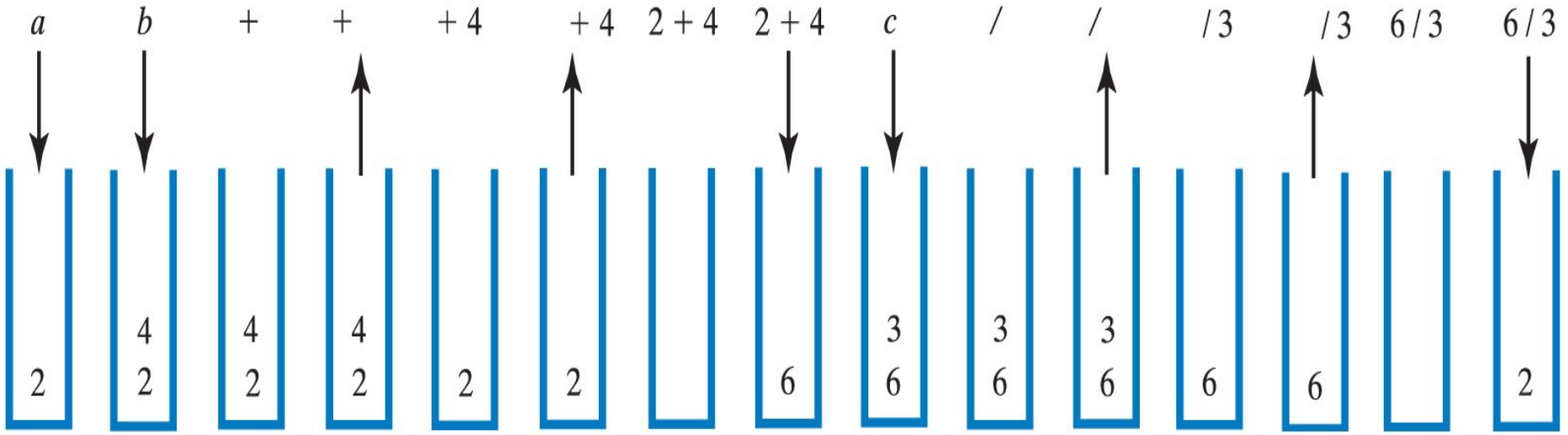


Fig. 21-11 The stack during the evaluation of the postfix expression **`a b + c /`** when **`a`** is 2, **`b`** is 4 and **`c`** is 3

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)

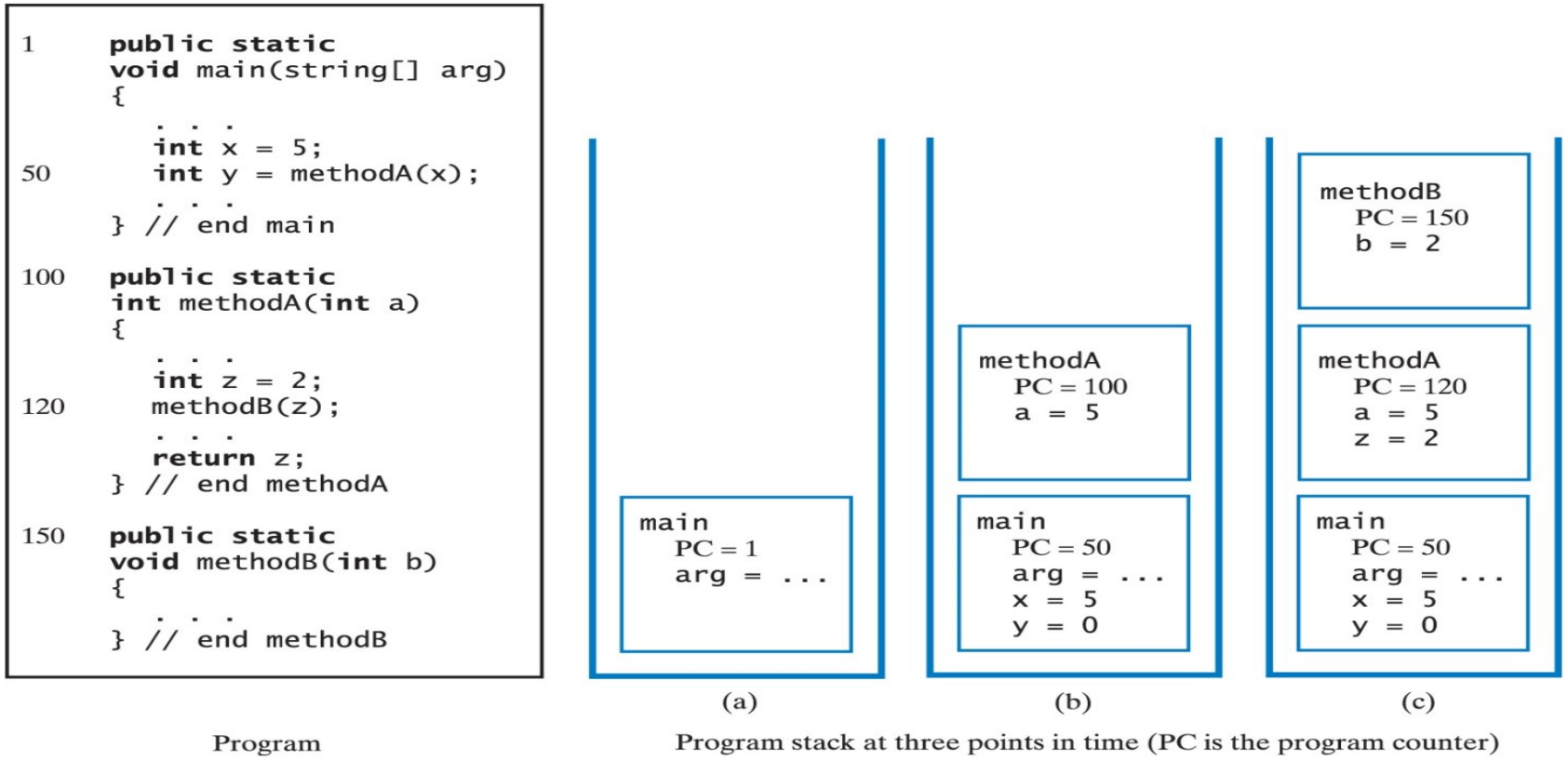


Fig. 21-13 The program stack at 3 points in time; (a) when **`main`** begins execution; (b) when **`methodA`** begins execution, (c) when **`methodB`** begins execution.

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

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.

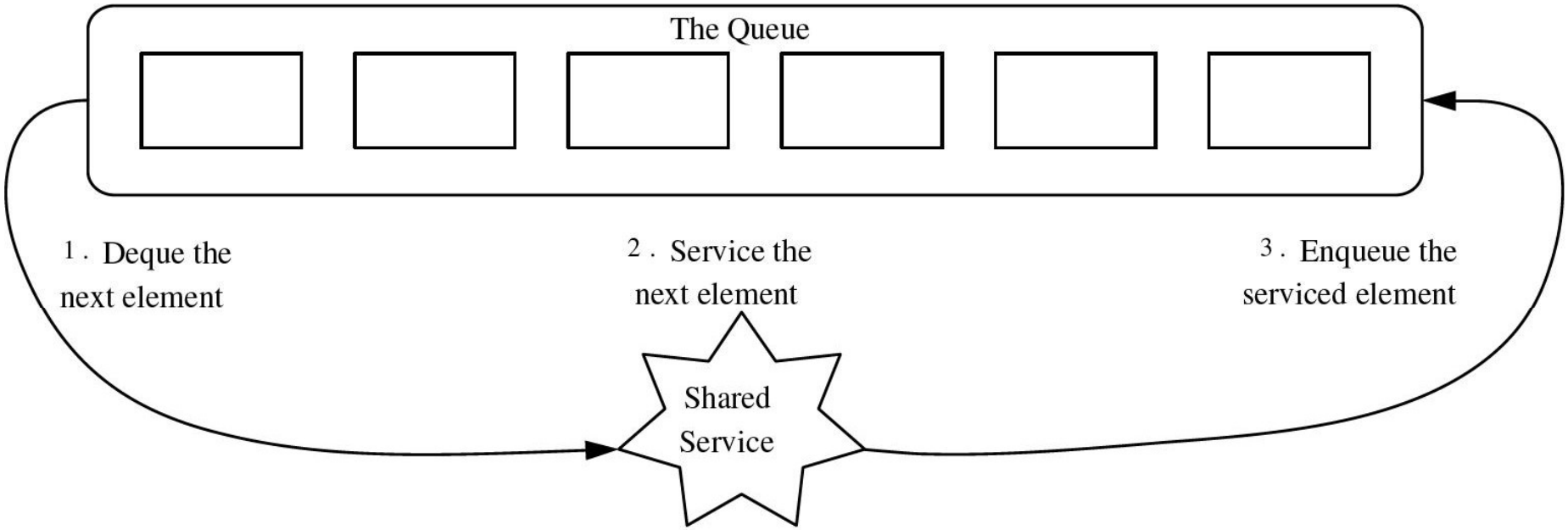
Java Collections Framework : **`ArrayBlockingQueue`** class

- **`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

Time-driven Simulation

- A counter enumerates simulated units of time (e.g. in minutes).
- Customers arrive at random times during the simulation and enter the queue.
- Each customer is assigned a random transaction time.
- During simulation, the waiting time for each customer is recorded
- At the end, summary statistics are generated, *e.g.*
  - Number of customers served
  - Average waiting time for a customer

Classes `WaitLine` and `Customer` (1/4)

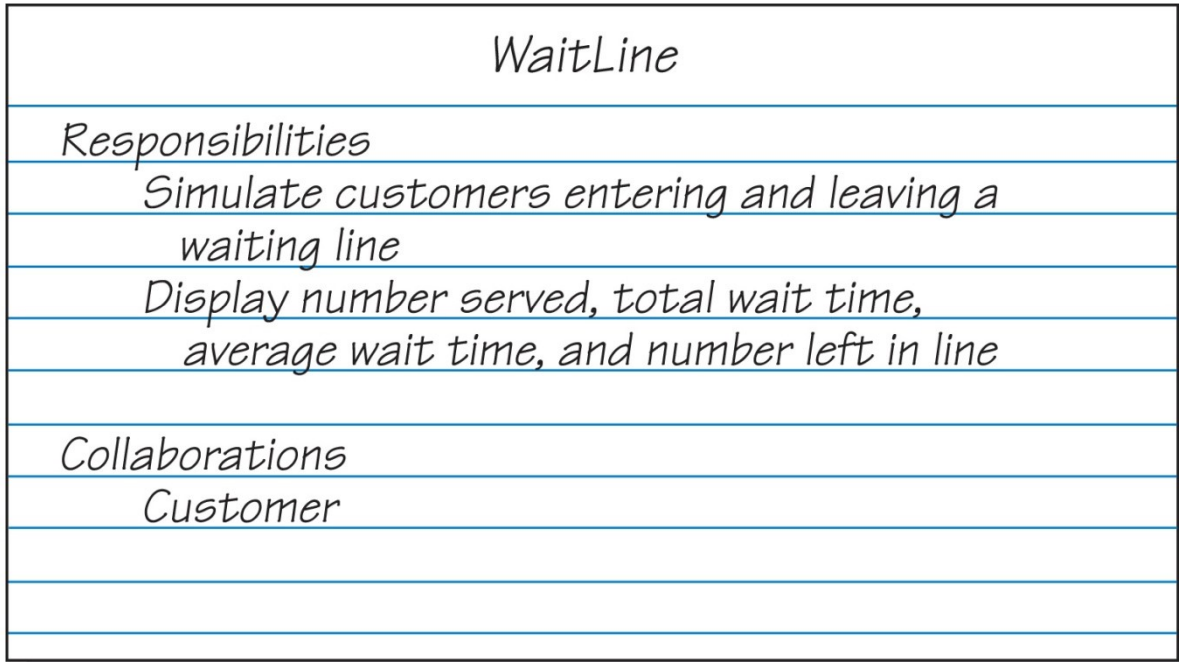


Fig. 23-4 A CRC card for the class **`WaitLine`**.

Classes `WaitLine` and `Customer` (2/4)

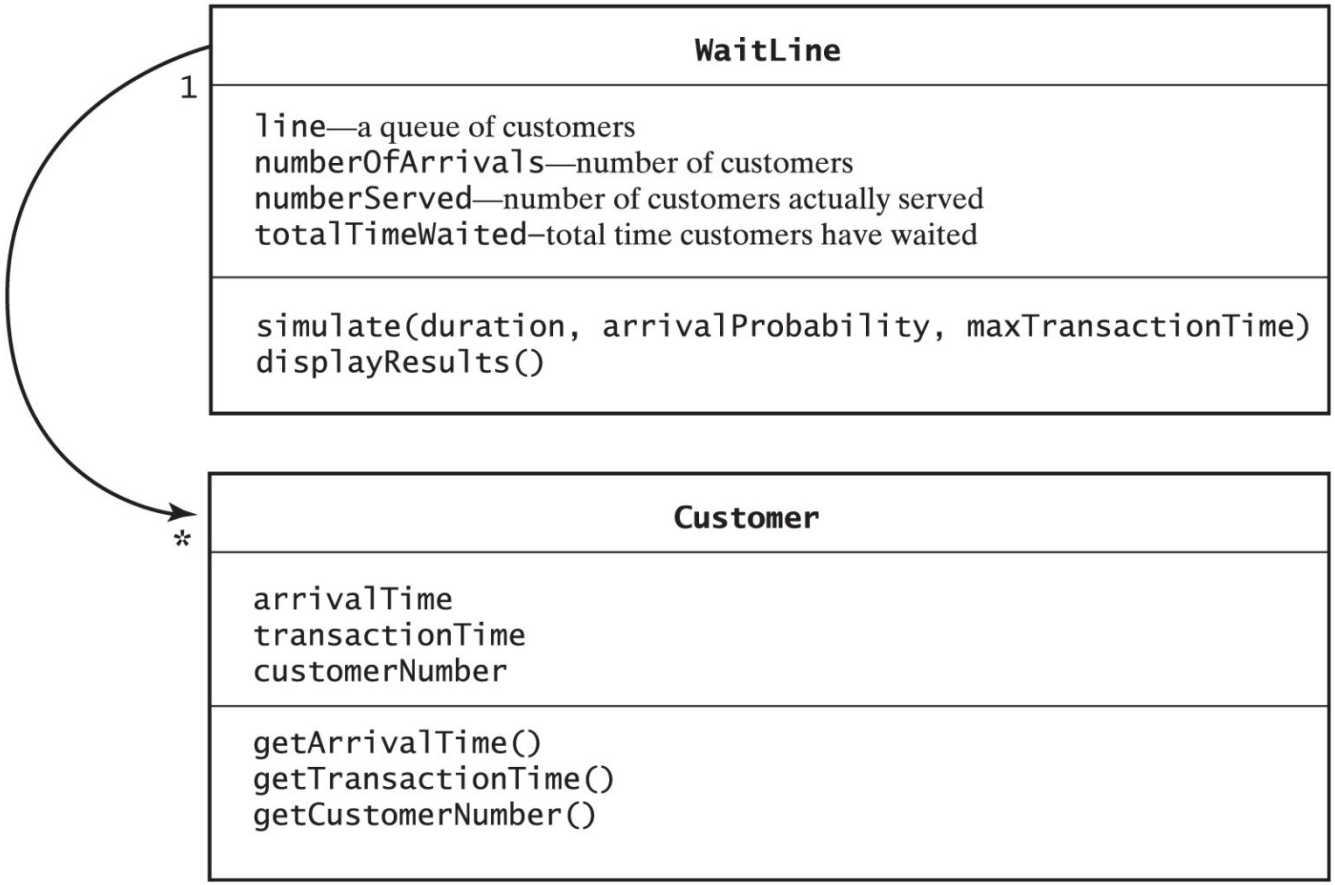


Fig. 23-5 A diagram of the classes **`WaitLine`** and **`Customer`**.

Classes `WaitLine` and `Customer` (3/4)

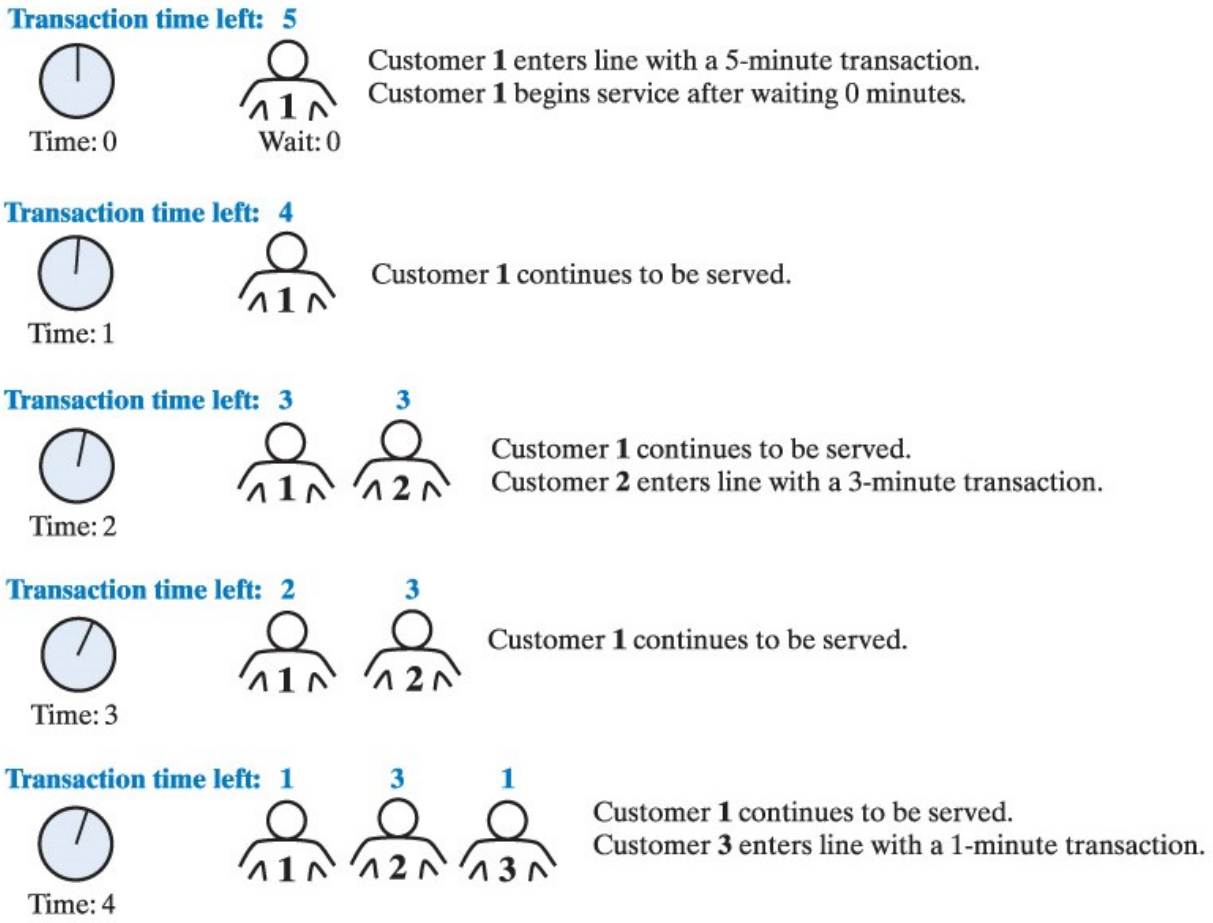


Fig. 23-6 A simulated waiting line … continued →

Classes `WaitLine` and `Customer` (4/4)

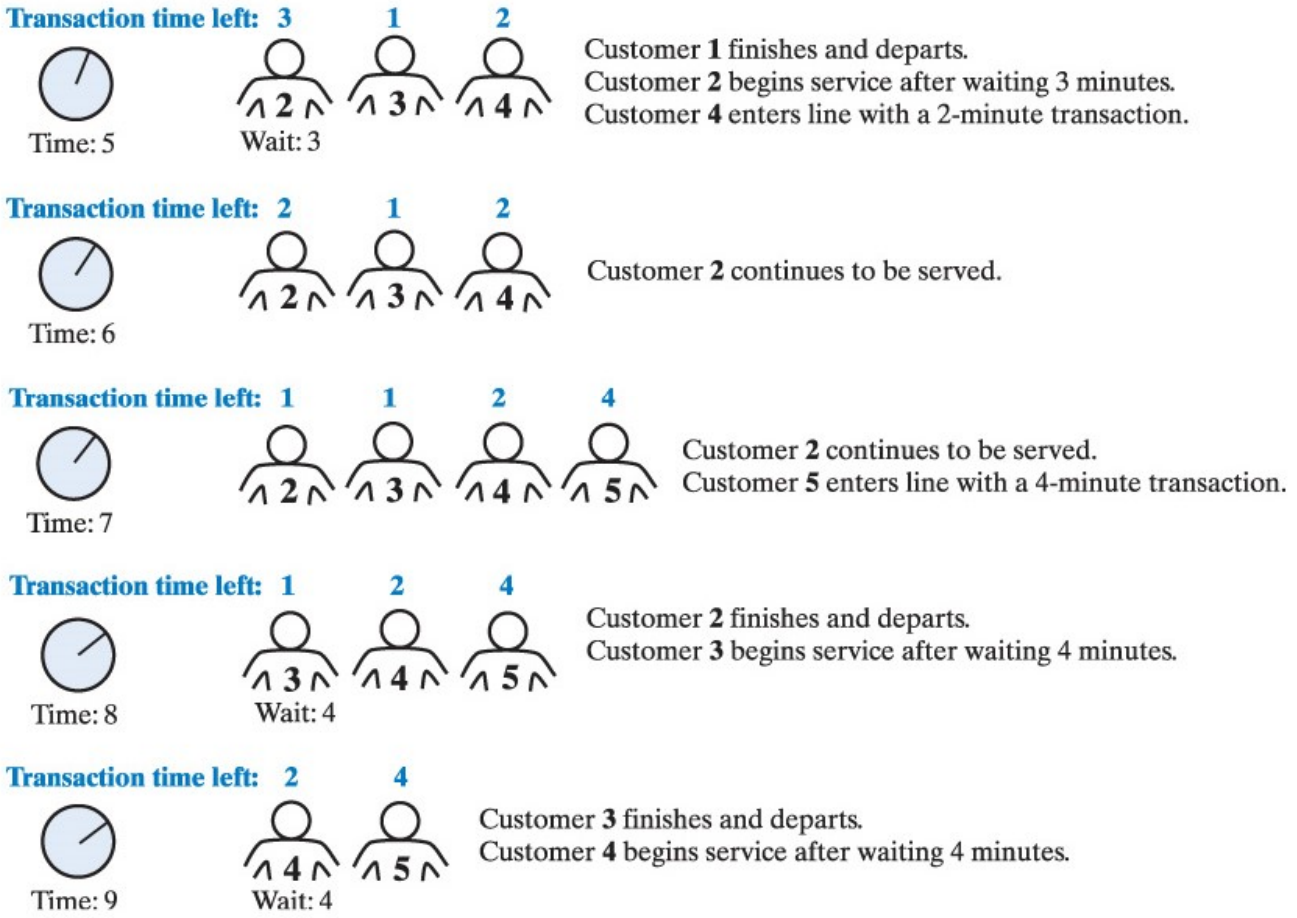


Fig. 23-6 (ctd) A simulated waiting line.

Sample Code: Time-Driven Simulation

In **`Chapter1\samplecode\`**

- **`Customer.java`**
- **`WaitLine.java`**
- **`SimulationDriver.java`**

The Class `Customer`

Note data fields:

`arrivalTime`	-	The time the customer arrives in the queue
`transactionTime`	-	The amount of time required for the customer's transaction
`customerNumber`	-	The sequence number assigned to the customer

The Class `Waitline`

Note methods

- Constructor
- **`simulate`** - Uses *time-driven simulation*
- **`displayResults`**
- **`reset`**

Problem: Using a Queue to Compute Capital Gain in a Sale of Stock

- Must sell stocks in same order they were purchased
  - Must use the difference in selling price and purchase price to calculate capital gain
- We use a queue to
  - Record investment transactions chronologically
  - Compute capital gain of the stock

The Class `StockLedger`

StockLedger
Responsibilities
Record the shares of a stock purchased, in chronological order
Remove the shares of a stock sold, beginning with the ones held the longest
Compute the capital gain (loss) on shares of a stock sold
Collaborations
Share of stock

Fig. 23-7 A CRC card for the class **`StockLedger`**

Classes `StockLedger` and `StockPurchase`

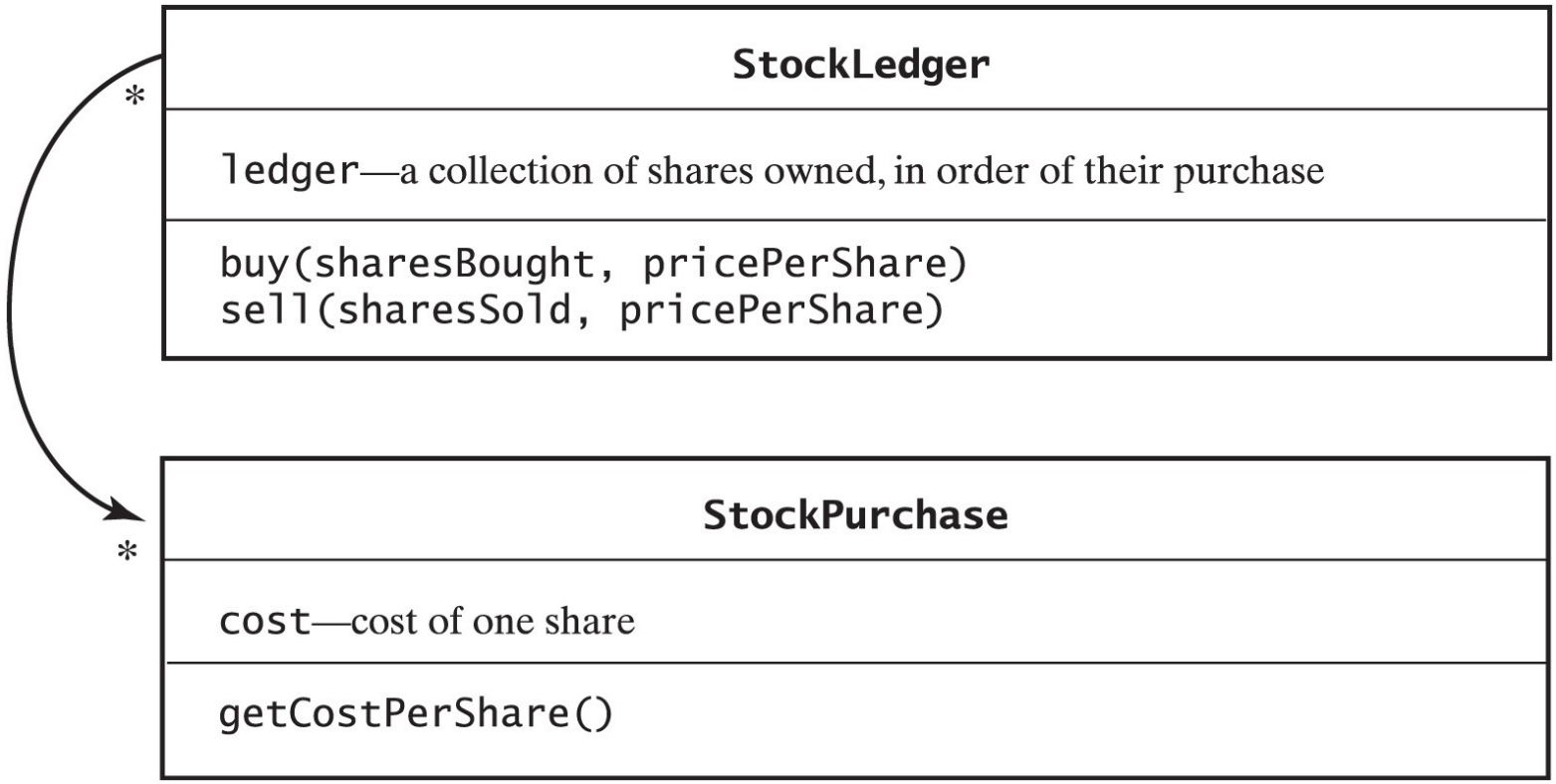


Fig. 23-8 A diagram of the class **`StockLedger`** and **`StockPurchase`**.

Sample Code: Computing Capital Gain in a Sale of Stock

In **`Chapter1\samplecode\`**

a) `StockPurchase.java`

- A **`StockPurchase`** object stores the cost of a single share of stock.

b) `StockLedger.java`

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

c) `StockLedgerDriver.java`

Queue of Shares of Stock

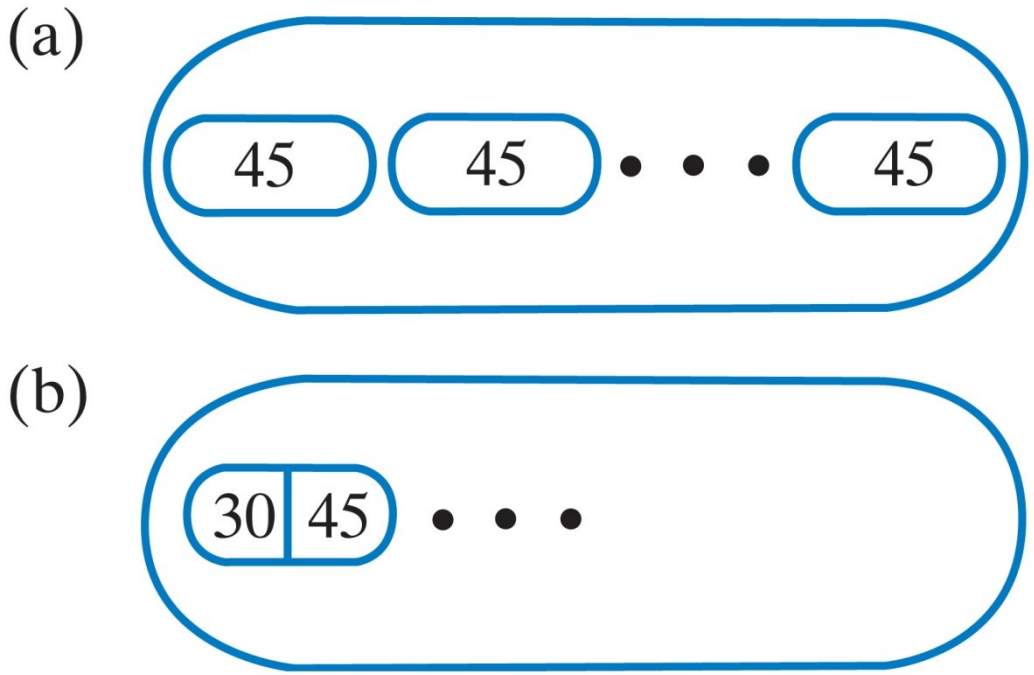
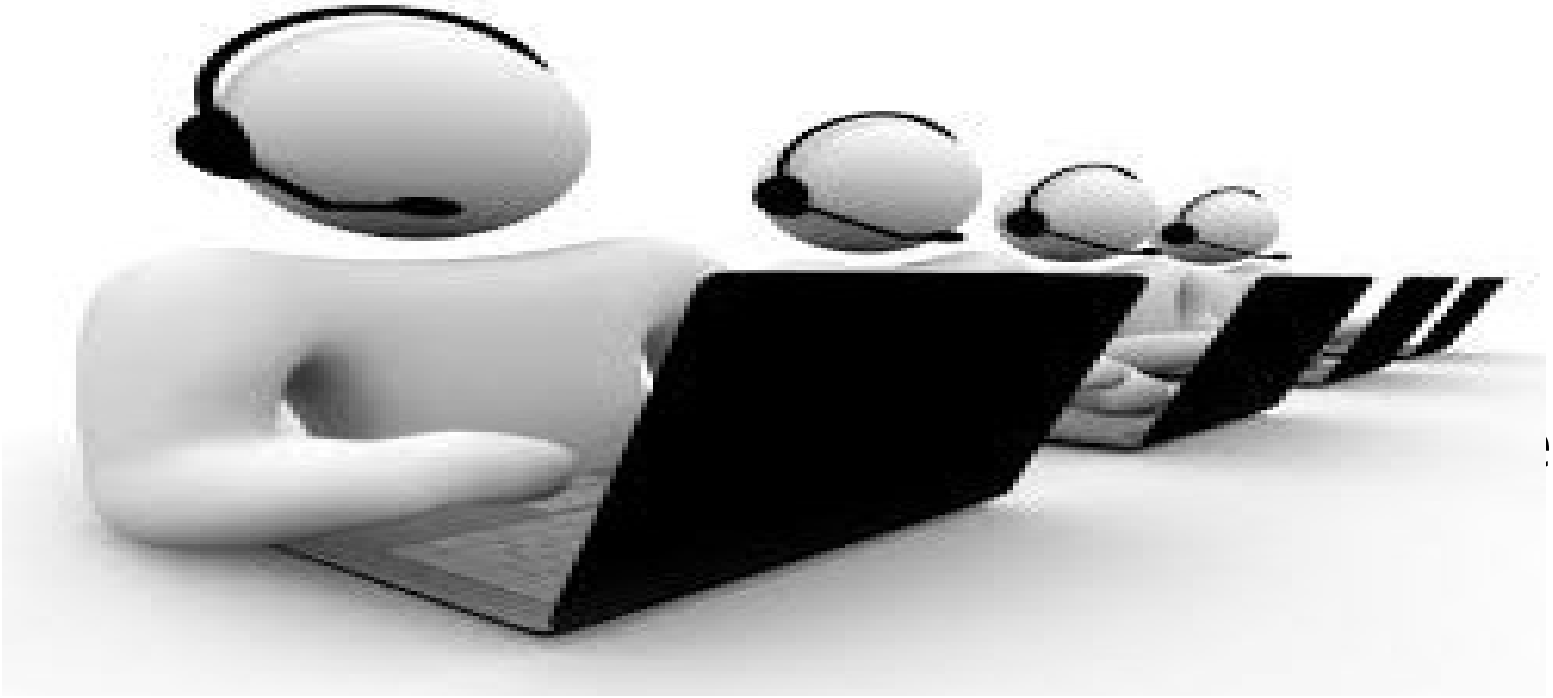


Fig. 23-9 A queue of (a) individual shares of stock; (b) grouped shares.

Queue: Discussion

- Call Center phone systems will use Queue, to hold people calling them in an order, until a service representative is free.



Exercise

What are the differences between lists, stacks and queues?

Exercise

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

(b) access to shared resources

(c) undo-sequence in a text editor

Learning Outcomes

You should be able to

- Explain the benefits of **encapsulation**, **modularity** and **data 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