BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI CS F213 – Object Oriented Programming ***Lab 7: Collections***	
Time:	2 hours
Objective:	To learn different data sets present in Java
Concepts Covered:	Collections (List, Set, Queue, Map)
Prerequisites:	Basic knowledge of Java

1. Collections

A **Collection** in Java is like a **container** that holds a group of objects. It helps in **storing, managing, and processing** data efficiently.

Java provides a **Collection Framework**, which includes different types of collections, such as:

1. **List** – Stores elements in an **ordered** way and allows duplicates.
    - Example: `ArrayList`, `LinkedList`
    - Like a shopping list where you can have repeated items.
2. **Set** – Stores unique elements (no duplicates allowed).
    - Example: `HashSet`, `TreeSet`
    - Like a collection of student roll numbers (each number is unique).
3. **Queue** – Follows **FIFO (First In, First Out)** order.
    - Example: `PriorityQueue`, `LinkedList` (as `Queue`)
    - Like a queue in a bank where the first person in line is served first.
4. **Map** – Stores **key-value pairs** (not a part of Collection interface but part of the framework).
    - Example: `HashMap`, `TreeMap`
    - Like a dictionary, where each word (key) has a meaning (value).

These collections make it easier to **store, sort, search, and manipulate** data in Java.

In Java, the Collection interface (`java.util.Collection`) and Map interface (`java.util.Map`) are the two main “root” interfaces of Java collection classes.

Collection Interface

Interface	Description	Allows Duplicates?	Ordered?	Key Methods	Common Implementations
Collection	Root interface for working with groups of objects.	Yes	No	add(), remove(), size(), iterator()	N/A (parent interface)
List	Extends Collection to represent an ordered collection (sequence).	Yes	Yes (insertion order)	get(), set(), add(int, E), remove(int)	ArrayList, LinkedList
Set	Extends Collection to represent a collection that does not allow duplicates.	No	No (unordered, except for specific implementations)	add(), remove(), contains(), size()	HashSet, LinkedHashSet, TreeSet
Queue	Extends Collection to represent a FIFO (first-in, first-out) structure.	Yes	Yes (FIFO or based on priority, depending on implementation)	offer(), poll(), peek(), remove()	LinkedList, PriorityQueue
Deque	Extends Queue to allow elements to be added or removed from both ends (double-ended queue).	Yes	Yes (can be used as stack or queue)	addFirst(), addLast(), removeFirst(), removeLast()	ArrayDeque, LinkedList
Map	Represents a key-value mapping where each key is unique.	N/A	No (except for specific implementations)	put(), get(), remove(), keySet(), values()	HashMap, LinkedHashMap, TreeMap, EnumMap

2. List

Class	Description	Allows Duplicates?	Ordered?	Random Access?	Key Features
ArrayList	Resizable array implementation of the List interface.	Yes	Yes (insertion order)	Yes	Fast random access (index-based), resizing array dynamically, slow when adding/removing elements in the middle.
LinkedList	Doubly linked list implementation of the List and Deque interfaces.	Yes	Yes (insertion order)	No	Efficient at adding/removing elements at both ends of the list, can be used as a queue or stack.

There are 2 ways to define a list in java:-

1. `List<T> list=new ArrayList<>();`
2. `List<T> list=new LinkedList<>();`

`ArrayList` and `LinkedList` are both classes that implement List interface.

General List methods:-

1. Basic Operations
   a. `add(E e)` → Adds an element to the list.
   b. `add(int index, E element)` → Inserts an element at a specific position.
   c. `addAll(Collection<? extends E> c)` → Appends all elements from another collection.

    d. `addAll(int index, Collection<? extends E> c)` → Inserts elements at a specific index.
    e. `clear()` → Removes all elements from the list.
    f. `contains(Object o)` → Returns true if the list contains the specified element.
    g. `isEmpty()` → Returns true if the list has no elements.
    h. `size()` → Returns the number of elements in the list.
2. Access & Retrieval
    a. `get(int index)` → Retrieves the element at the specified index.
    b. `indexOf(Object o)` → Returns the first index of the specified element, or -1 if not found.
    c. `lastIndexOf(Object o)` → Returns the last index of the specified element, or -1 if not found.
3. Modification
    a. `set(int index, E element)` → Replaces the element at the specified index.
    b. `remove(int index)` → Removes the element at the given index.
    c. `remove(Object o)` → Removes the first occurrence of the specified element.
    d. `removeAll(Collection<?> c)` → Removes all matching elements from the list.
    e. `retainAll(Collection<?> c)` → Keeps only elements that are in the specified collection.
4. Iteration & Streams
    a. `iterator()` → Returns an iterator for sequential traversal.
    b. `listIterator()` → Returns a bidirectional iterator for the list.
    c. `listIterator(int index)` → Returns a list iterator starting at the given index.
    d. `forEach(Consumer<? super E> action)` → Performs the given action for each element.
    e. `stream()` → Returns a sequential stream from the list.
    f. `parallelStream()` → Returns a parallel stream for multi-threaded operations.
5. Sublist & Array Conversion
    a. `subList(int fromIndex, int toIndex)` → Returns a view of a portion of the list.
    b. `toArray()` → Converts the list into an array.
    c. `toArray(T[] a)` → Returns an array with the runtime type of the specified array.

2.1. ArrayList

An `ArrayList` in Java is a dynamic array that can grow and shrink in size automatically. Unlike a normal array, you don't need to specify its size when creating it.

Key Features of `ArrayList`:

- Resizable – It automatically expands when needed.
- Indexed – You can access elements using an index (like an array).
- Allows Duplicates – You can store duplicate values.

`ArrayList` specific Methods:-

1. Ensure Capacity & Trim

    a. `ensureCapacity(int minCapacity)` → Increases the capacity of the ArrayList to ensure it can hold at least the specified number of elements.
    b. `trimToSize()` → Reduces the ArrayList capacity to match the actual number of elements, freeing up unused memory.
2. Efficient Element Removal
    a. `removeRange(int fromIndex, int toIndex)` (Protected Method) → Removes elements between the specified indexes. (Accessible only via subclassing.)
3. Cloning & Conversion
    a. `clone()` → Creates a shallow copy of the ArrayList.
    b. `toArray()` → Converts the ArrayList into an array.
    c. `toArray(T[] a)` → Converts the ArrayList into an array of the specified type.
4. Bulk Addition
    a. `addAll(Collection<? extends E> c)` → Appends all elements from another collection.
    b. `addAll(int index, Collection<? extends E> c)` → Inserts elements at a specific index.

```java
public class ArrayListMethodsExample {
    public static void main(String[] args) {
        //  Creating an ArrayList
        ArrayList<String> fruits = new ArrayList<>();

        //  Adding elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        fruits.add("Mango");
        fruits.add("Banana"); // Duplicates allowed
        System.out.println("Original List: " + fruits);

        //  Getting an element by index
        System.out.println("Element at index 2: " + fruits.get(2));

        //  Modifying an element
        fruits.set(1, "Blueberry");
        System.out.println("After modification: " + fruits);

        //  Removing elements
        fruits.remove("Cherry"); // Remove by value
        fruits.remove(2); // Remove by index
        System.out.println("After removal: " + fruits);

        //  Checking if an element exists
        System.out.println("Contains Mango? " + fruits.contains("Mango"));

        //  Finding the size of the ArrayList
        System.out.println("Size of list: " + fruits.size());

        //  Sorting the ArrayList
        Collections.sort(fruits);
```

```java
        System.out.println("Sorted List: " + fruits);

        // Looping through the list (Using for-each loop)
        System.out.print("Using for-each loop: ");
        for (String fruit : fruits) {
            System.out.print(fruit + " ");
        }
        System.out.println();

        //  Looping using for loop with index
        System.out.print("Using for loop: ");
        for (int i = 0; i < fruits.size(); i++) {
            System.out.print(fruits.get(i) + " ");
        }
        System.out.println();

        //  Converting ArrayList to Array
        String[] fruitArray = fruits.toArray(new String[0]);
        System.out.println("Converted to Array: " + java.util.Arrays.toString(fruitArray));

        //  Clearing the ArrayList
        fruits.clear();
        System.out.println("After clearing: " + fruits);

        //  Checking if the list is empty
        System.out.println("Is list empty? " + fruits.isEmpty());
    }
}
```

2.2. LinkedList

A `LinkedList` in Java is a data structure where elements (nodes) are linked together using pointers. Unlike an `ArrayList`, which uses an internal dynamic array, a LinkedList consists of nodes that store:- data, reference to next node

Key Features of LinkedList:

- Dynamic Size – Grows and shrinks automatically.
- Fast Insert/Delete (O(1)) – Adding/removing elements in the middle is efficient.
- Slower Access (O(n)) – Fetching an element is slower than `ArrayList`.
- Implements List and Deque – Supports both List (like `ArrayList`) and `Deque` (double-ended queue) operations.

LinkedList specific Methods:-

1. First & Last Element Operations
    a. `getFirst()` → Returns the first element (throws `NoSuchElementException` if empty).
    b. `getLast()` → Returns the last element (throws `NoSuchElementException` if empty).
    c. `removeFirst()` → Removes and returns the first element.
    d. `removeLast()` → Removes and returns the last element.

e. `addFirst(E e)` → Inserts an element at the beginning.

f. `addLast(E e)` → Appends an element at the end.

```java
public class LinkedListMethodsExample {
    public static void main(String[] args) {
        // Creating a LinkedList
        LinkedList<String> names = new LinkedList<>();

        // Adding elements
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        names.add("David");
        System.out.println("Original List: " + names);

        // Adding elements at specific positions
        names.addFirst("Zara"); // Adds at the beginning
        names.addLast("Emma");  // Adds at the end
        names.add(2, "Mia");    // Inserts at index 2
        System.out.println("After additions: " + names);

        // Accessing elements
        System.out.println("First Element: " + names.getFirst());
        System.out.println("Last Element: " + names.getLast());
        System.out.println("Element at index 3: " + names.get(3));

        // Modifying elements
        names.set(2, "Sophia"); // Replace element at index 2
        System.out.println("After modification: " + names);
      // Removing elements
        names.removeFirst(); // Remove first element
        names.removeLast();  // Remove last element
        names.remove(2);     // Remove element at index 2
        System.out.println("After removals: " + names);

        // Checking properties
        System.out.println("Contains 'Charlie'? " + names.contains("Charlie"));
        System.out.println("Size of list: " + names.size());
        System.out.println("Is list empty? " + names.isEmpty());

        // Searching elements
        System.out.println("Index of 'Bob': " + names.indexOf("Bob"));
        System.out.println("Last index of 'Bob': " + names.lastIndexOf("Bob"));

        // Sorting & Reversing
        Collections.sort(names);
        System.out.println("Sorted List: " + names);
        Collections.reverse(names);
        System.out.println("Reversed List: " + names);

        // Converting to Array
        String[] namesArray = names.toArray(new String[0]);
        System.out.println("Converted to Array: " + java.util.Arrays.toString(namesArray));
```

```java
        // Iterating over elements using for-each loop
        System.out.print("Using for-each loop: ");
        for (String name : names) {
            System.out.print(name + " ");
        }
        System.out.println();

        // Iterating using iterator
        System.out.print("Using Iterator: ");
        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
        System.out.println();

        // Deque-Specific Methods (Queue Operations)
        names.offer("Oliver"); // Adds at the end
        names.offerFirst("Jack"); // Adds at the front
        names.offerLast("Liam");  // Adds at the end
        System.out.println("After offer methods: " + names);

        System.out.println("Peek (first element): " + names.peek());
        System.out.println("Poll (remove first element): " + names.poll());
        System.out.println("PollLast (remove last element): " + names.pollLast());

        // Clearing the list
        names.clear();
        System.out.println("After clearing: " + names);
    }
}
```

2.3. Exercise

Exercise 1:

Task:

- Create an `ArrayList<Integer>` and add random numbers.
- Convert it into a `LinkedList<Integer>`.
- Perform the following operations on the `LinkedList`:
- Add a new element at the beginning and end.
- Remove the second element.
- Reverse the list.
- Print both the `ArrayList` and modified `LinkedList`.

Find boiler-plate code below

```java
public class ConvertAndModifyLinkedList {
    public static void main(String[] args) {
        // Create an ArrayList
```

```
        // Add elements to ArrayList below


        // Convert ArrayList to LinkedList


        // Add an element at the beginning


        // Add an element at the end


        // Remove the second element


        // Reverse the LinkedList


        // Print both lists
        System.out.println("Original ArrayList: " + arrayList);
        System.out.println("Modified LinkedList: " + linkedList);
    }
}
```

Exercise 2:

Task:

- Create a `List<String>` and add several names.
- Iterate over the list using:
- A for-each loop
- A for loop with an index
- An Iterator
- A Lambda Expression (`forEach` method)
- Print the elements in each method.

```
public class ListIterationExercise {
    public static void main(String[] args) {
        // Create a List of names
        List<String> names = new ______<>();

        // Add elements to the list
        names.____("Alice");
        names.____("Bob");
        names.____("Charlie");
        names.____("David");
        names.____("Eve");

        // Iterate using a for-each loop
        System.out.println("Using for-each loop:");
```

```
        for (String name : ______) {
            System.out.println(name);
        }

        // Iterate using a for loop with an index
        System.out.println("\nUsing for loop with index:");
        for (int i = 0; i < names.____(); i++) {
            System.out.println(names.____(i));
        }

        // Iterate using an Iterator
        System.out.println("\nUsing Iterator:");
        Iterator<String> iterator = names.____();
        while (iterator.____()) {
            System.out.println(iterator.____());
        }

        // Iterate using a Lambda Expression (forEach)
        System.out.println("\nUsing Lambda Expression:");
        names.____(name -> System.out.println(name));
    }
}
```

3. Set

Class	Description	Allows Duplicates?	Ordered?	Sorting	Key Features
HashSet	Implements the Set interface backed by a hash table.	No	No	No	Fast access (O(1) time for add, remove, contains), no guaranteed order.
LinkedHash Set	Extends HashSet, maintains insertion order using a linked list.	No	Yes (insertion order)	No	Preserves insertion order, slightly slower than HashSet due to the extra cost of maintaining the order.
TreeSet	Implements NavigableSet, stores elements in a sorted tree.	No	Yes (sorted in natural or comparator order)	Yes (natural/comparat or order)	Provides O(log n) time for basic operations like add, remove, and contains, automatically sorted elements.

There are many ways to define a set:-

1. `Set<String> hashSet = new HashSet<>();`
2. `Set<String> linkedHashSet = new LinkedHashSet<>();`
3. `Set<Integer> treeSet = new TreeSet<>();`

Set is an interface whereas `hashSet`, `linkedHashSet`, `treeSet` are classes implementing the interface Set.

List of General Set Methods:

- `add(E e)` → Adds an element to the set (if not already present).

- `addAll(Collection<? extends E> c)` → Adds all elements from a given collection to the set.
- `clear()` → Removes all elements from the set.
- `contains(Object o)` → Returns true if the set contains the specified element.
- `containsAll(Collection<?> c)` → Returns true if the set contains all elements of the specified collection.
- `isEmpty()` → Returns true if the set is empty.
- `iterator()` → Returns an iterator over the elements in the set.
- `remove(Object o)` → Removes the specified element from the set.
- `removeAll(Collection<?> c)` → Removes all elements in the given collection from the set.
- `retainAll(Collection<?> c)` → Keeps only elements that are also in the specified collection.
- `size()` → Returns the number of elements in the set.
- `toArray()` → Returns an array containing all elements of the set.
- `toArray(T[] a)` → Returns an array of the set's elements in the runtime type of the specified array.
- `equals(Object o)` → Compares the set with another object for equality.
- `hashCode()` → Returns the hash code value of the set.
- `spliterator()` → Returns a `Spliterator` for the set, supporting parallel processing.

3.1. HashSet

`HashSet` is a collection in Java that implements the Set interface and is part of the `java.util` package. It is used to store unique elements and is based on a hash table (HashMap internally).

Key Features of `HashSet`

- No Duplicates – Stores only unique elements.
- Unordered Collection – Does not maintain insertion order.
- Allows null Values – Can store at most one null value.
- Fast Operations – Provides O(1) time complexity for add, remove, contains operations.
- Not Thread-Safe – Requires explicit synchronization in multithreading.

HashSet Specific Methods:-

1. Performance-Based Methods
    a. `HashSet()` → Creates an empty HashSet with default capacity (16) and load factor (0.75).
    b. `HashSet(int initialCapacity)` → Creates a HashSet with a specific initial capacity.

   c. `HashSet(int initialCapacity, float loadFactor)` → Creates a HashSet with a specific initial capacity and load factor.
2. Underlying Hash Table Mechanisms
   a. `hashCode()` → Returns the hash code of the HashSet, which is computed based on the elements.
   b. `clone()` → Creates a shallow copy of the HashSet.
3. Behavioral Differences
   a. Unlike `TreeSet`, `HashSet` does not maintain order of elements.
   b. Unlike `LinkedHashSet`, `HashSet` does not preserve insertion order.

```
public class SetInterfaceDemo {
    public static void main(String[] args) {
        // Create a HashSet (which implements the Set interface)
        Set<String> set = new HashSet<>();

        // Add elements to the set
        set.add("Apple");
        set.add("Banana");
        set.add("Cherry");
        set.add("Apple"); // Duplicate, will not be added

        // Display the set elements
        System.out.println("Set elements: " + set); // Output: [Apple, Banana, Cherry]

        boolean containsBanana = set.contains("Banana");// Check if the set contains
'Banana'
        System.out.println("Set contains 'Banana': " + containsBanana); // Output: true

        set.remove("Banana");// Remove an element from the set
        System.out.println("After removing 'Banana': " + set); // Output: [Apple, Cherry]

        int size = set.size();// Get the size of the set
        System.out.println("Size of the set: " + size); // Output: 2

        set.clear();// Clear the set
        System.out.println("Set after clear(): " + set); // Output: []
    }
}
```

3.2. LinkedHashSet

`LinkedHashSet` is a Set implementation in Java that maintains insertion order while ensuring that elements remain unique (just like HashSet).

It is part of `java.util` package and extends HashSet, which internally uses a `LinkedHashMap` to preserve order.

Key Features of `LinkedHashSet`

- Maintains Insertion Order → Unlike HashSet, which does not guarantee any order, `LinkedHashSet` keeps elements in the order they were added.
- No Duplicates → It does not allow duplicate elements, just like any other Set.
- Allows null Elements → Can store at most one null value.
- Performance (Slightly Slower than `HashSet`) → Provides O(1) time complexity for add, remove, and contains operations, but slightly slower than HashSet due to maintaining order.
- Not Thread-Safe → If multiple threads access it concurrently, synchronization is required.

`LinkedHashSet` specific Methods:-

1. Maintains Insertion Order
   a. Unlike HashSet, a `LinkedHashSet` preserves the order in which elements were inserted.
   b. Example: If you insert "A", "B", "C", it will always iterate in that order.
2. Constructors for Custom Behavior
   a. `LinkedHashSet()` → Creates an empty `LinkedHashSet` with a default capacity (16) and load factor (0.75).
   b. `LinkedHashSet(int initialCapacity)` → Creates a `LinkedHashSet` with a specific initial capacity.
   c. `LinkedHashSet(int initialCapacity, float loadFactor)` → Creates a `LinkedHashSet` with a specific capacity and load factor.
3. Same Methods as HashSet but Ordered
   a. `add(E e)` → Adds an element while preserving insertion order.
   b. `remove(Object o)` → Removes an element, maintaining the order of others.
   c. `iterator()` → Returns an iterator that iterates in insertion order.
   d. `clone()` → Creates a shallow copy of the `LinkedHashSet`, preserving order.
   e. `contains(Object o)` → Checks if the element exists without affecting order.

3.3. TreeSet

`TreeSet` is a Sorted Set implementation in Java that maintains elements in ascending order. It is part of `java.util` package and is implemented using a self-balancing Red-Black Tree.

Key Features of `TreeSet`

- Maintains Sorted Order → Elements are stored in ascending (natural) order by default.

- No Duplicates → Like any Set, it does not allow duplicate elements.
- Allows Custom Ordering → Can define custom sorting using Comparator.
- No null Allowed → Unlike `HashSet` or `LinkedHashSet`, `TreeSet` does not allow null values.
- Slower than `HashSet` → `TreeSet` operations like add, remove, and search take O(log n) time due to tree balancing.

TreeSet specific Methods:-

- `first()` → Returns the smallest (first) element in the set.
- `last()` → Returns the largest (last) element in the set.
- `ceiling(E e)` → Returns the smallest element greater than or equal to the given element.
- `floor(E e)` → Returns the largest element less than or equal to the given element.
- `higher(E e)` → Returns the smallest element strictly greater than the given element.
- `lower(E e)` → Returns the largest element strictly less than the given element.
- `pollFirst()` → Retrieves and removes the first (lowest) element.
- `pollLast()` → Retrieves and removes the last (highest) element.
- `headSet(E toElement, boolean inclusive)` → Returns a view of elements less than (or equal to if inclusive) the given element.
- `tailSet(E fromElement, boolean inclusive)` → Returns a view of elements greater than (or equal to if inclusive) the given element.
- `subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)` → Returns a subset within a range of elements.
- `descendingSet()` → Returns a reverse-ordered view of the set.

```java
public class SetInterfaceDemo {
    public static void main(String[] args) {
        // Create a HashSet (which implements the Set interface)
        Set<String> set = new HashSet<>();

        // Add elements to the set
        set.add("Apple");
        set.add("Banana");
        set.add("Cherry");
        set.add("Apple"); // Duplicate, will not be added

        // Display the set elements
        System.out.println("Set elements: " + set); // Output: [Apple, Banana, Cherry]

        boolean containsBanana = set.contains("Banana");// Check if the set contains
'Banana'
        System.out.println("Set contains 'Banana': " + containsBanana); // Output: true
```

```
        set.remove("Banana");// Remove an element from the set
        System.out.println("After removing 'Banana': " + set); // Output: [Apple, Cherry]

        int size = set.size();// Get the size of the set
        System.out.println("Size of the set: " + size); // Output: 2

        set.clear();// Clear the set
        System.out.println("Set after clear(): " + set); // Output: []
    }
}
```

```
public class TreeSetMethodsExample {
    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>();

        // Adding elements
        set.add(50);
        set.add(20);
        set.add(10);
        set.add(40);
        set.add(30);

        System.out.println("TreeSet: " + set);

        System.out.println("First Element: " + set.first());// Retrieving first and last
elements
        System.out.println("Last Element: " + set.last());

        System.out.println("Higher than 25: " + set.higher(25)); // Checking higher and
lower values
        System.out.println("Lower than 25: " + set.lower(25));

        System.out.println("Poll First: " + set.pollFirst());// Polling elements
        System.out.println("Poll Last: " + set.pollLast());

        System.out.println("HeadSet (less than 40): " + set.headSet(40)); // Subset
operations
        System.out.println("TailSet (greater than 20): " + set.tailSet(20));
        System.out.println("SubSet (between 20 and 40): " + set.subSet(20, 40));
    }
}
```

3.4. Exercises

Exercise 1:-

Write a Java program that reads a list of words from the user and stores them in a Set. The program should:

1. Ensure only unique words are stored (i.e., duplicate words should be ignored).
2. Print all unique words in sorted order.
3. Print the total count of unique words.

```
public class UniqueWordsSet {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Choose the correct Set implementation

        System.out.println("Enter words (type 'exit' to stop):");
        while (true) {
            String word = scanner.next();
            if (word.equalsIgnoreCase("exit")) break;
                        // Add the word to the set
        }

        System.out.println("Unique Words (Sorted): " + words);
        // Print set size

        scanner.close();
    }
}
```

Exercise 2:

Write a Java program that:

1. Takes two lists of numbers as input from the user.
2. Stores each list in a Set to remove duplicates.
3. Finds and prints the **common elements** between both sets.
4. Displays the number of common elements.

```
public class CommonElementsSet {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Creating sets to store unique numbers
        Set<Integer> set1 = new _______<>();  // Choose the correct Set implementation
        Set<Integer> set2 = new _______<>();  // Choose the correct Set implementation

        System.out.println("Enter numbers for first set (type -1 to stop):");
        while (true) {
            int num = scanner.nextInt();
            if (num == -1) break;
            set1.____(num);  // Add to first set
        }

        System.out.println("Enter numbers for second set (type -1 to stop):");
        while (true) {
```

```java
            int num = scanner.nextInt();
            if (num == -1) break;
            set2.____(num);  // Add to second set
        }

        // Finding common elements
        Set<Integer> commonElements = new ______<>(set1);  // Initialize with set1
        commonElements.____(set2);  // Retain only common elements

        System.out.println("Common Elements: " + commonElements);
        System.out.println("Total Common Elements: " + commonElements._____()); // Print
size

        scanner.close();
    }
}
```

4. Queue

Class	Description	Allows Duplicates?	Ordered?	FIFO/LIFO	Key Features
PriorityQueue	Implements Queue, orders elements based on priority (natural order or custom comparator).	Yes	Yes (based on priority, not insertion)	FIFO (priority-based)	Elements are ordered based on their natural ordering or a comparator; best for scenarios where priority matters.
ArrayDeque	Implements Deque, resizable array supporting FIFO and LIFO operations.	Yes	Yes (insertion order)	FIFO (Queue) / LIFO (Stack)	Efficient for both queue (FIFO) and stack (LIFO) operations

Different ways to define queue:-

1. `Queue<Integer> queue = new PriorityQueue<>();`
2. `Queue<Integer> queue = new ArrayDeque<>();`
3. `Queue<Integer> queue=new LinkedList<>();`

Queue is an interface whereas `PriorityQueue` and `ArrayDeque` are instantiable classes which extend Queue interface.

General Queue Methods:-

1. Basic Queue Operations
   a. `add(E e)` → Inserts an element into the queue. Throws an exception if the queue is full.
   b. `offer(E e)` → Inserts an element into the queue. Returns false if the queue is full instead of throwing an exception.
   c. `remove()` → Removes and returns the head (first) element. Throws `NoSuchElementException` if empty.
   d. `poll()` → Removes and returns the head element, or null if the queue is empty.

e. `element()` → Retrieves, but does not remove, the head element. Throws `NoSuchElementException` if empty.
f. `peek()` → Retrieves, but does not remove, the head element, or returns null if empty.

2. Bulk Operations (Inherited from Collection)
   a. `size()` → Returns the number of elements in the queue.
   b. `isEmpty()` → Returns true if the queue is empty.
   c. `contains(Object o)` → Returns true if the queue contains the given element.
   d. `toArray()` → Converts the queue into an array.
   e. `clear()` → Removes all elements from the queue.

4.1. PriorityQueue

A **`PriorityQueue`** in Java is a special type of queue where elements are ordered based on priority rather than insertion order (FIFO).

- By default, it follows natural ordering (ascending order for numbers, alphabetical order for strings).
- You can define a custom comparator to prioritize elements in a different way (e.g., max-heap behavior).
- Implemented as a binary heap (a complete binary tree stored as an array), ensuring efficient O(log n) time for insertion and deletion.

`PriorityQueue` specific Methods:-

1. Ordering & Priority Management
   a. `comparator()` → Returns the Comparator used for ordering, or null if natural ordering is used.
2. Element Retrieval & Removal
   a. `poll()` → Retrieves and removes the highest-priority element (smallest by default), returns null if empty.
   b. `remove(Object o)` → Removes a specific element from the queue.
   c. `peek()` → Retrieves, but does not remove, the highest-priority element (smallest by default), returns null if empty.
3. Bulk Operations
   a. `offer(E e)` → Inserts an element while maintaining the priority order.
   b. `addAll(Collection<? extends E> c)` → Adds all elements from another collection while maintaining priority order.
4. Utility & Conversion
   a. `toArray()` → Converts the priority queue to an Object array.

```
public class PriorityQueueMethodsExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        // Adding elements
        pq.add(30);
```

```java
        pq.offer(10);
        pq.offer(20);

        System.out.println("Priority Queue: " + pq); // Output: [10, 30, 20] (Heap order)

        // Peek and Poll
        System.out.println("Peek (Highest Priority): " + pq.peek());  // Output: 10
        System.out.println("Poll (Removing Highest Priority): " + pq.poll()); // Output: 10
        System.out.println("After Poll: " + pq); // Output: [20, 30]

        // Checking size and containment
        System.out.println("Size: " + pq.size()); // Output: 2
        System.out.println("Contains 20? " + pq.contains(20)); // Output: true

        // Convert to array
        Object[] arr = pq.toArray();
        System.out.println("Queue as Array: " + Arrays.toString(arr)); // Output: [20, 30]

        // Removing elements
        pq.remove(20);
        System.out.println("After Removing 20: " + pq); // Output: [30]

        // Clearing the queue
        pq.clear();
        System.out.println("Is Empty? " + pq.isEmpty()); // Output: true
    }
}
```

4.2. ArrayDeque

`ArrayDeque` (Array Double-Ended Queue) is a resizable, array-based implementation of the Deque interface in Java. It allows fast insertion and removal from both ends of the queue. It is often preferred over LinkedList due to its better performance.

Key Features of `ArrayDeque`

- Faster than **`LinkedList`** for queue operations.
- Supports both FIFO (Queue) and LIFO (Stack) behavior.
- No capacity restrictions (dynamically resizable).
- Does not allow null values.
- Not thread-safe (Use **`ConcurrentLinkedDeque`** for multi-threading).

`ArrayDeque` specific Methods:-

1. First & Last Element Operations
    a. **`addFirst(E e)`** → Inserts an element at the front of the deque (throws exception if full).
    b. **`addLast(E e)`** → Inserts an element at the end of the deque (throws exception if full).
    c. **`offerFirst(E e)`** → Inserts an element at the front (returns false if full instead of throwing an exception).

d. `offerLast(E e)` → Inserts an element at the end (returns false if full instead of throwing an exception).
e. `getFirst()` → Retrieves the first element without removing it (throws `NoSuchElementException` if empty).
f. `getLast()` → Retrieves the last element without removing it (throws `NoSuchElementException` if empty).
g. `peekFirst()` → Retrieves the first element without removing it (returns null if empty).
h. `peekLast()` → Retrieves the last element without removing it (returns null if empty).
i. `removeFirst()` → Removes and returns the first element (throws `NoSuchElementException` if empty).
j. `removeLast()` → Removes and returns the last element (throws `NoSuchElementException` if empty).
k. `pollFirst()` → Removes and returns the first element (returns null if empty).`pollLast()` → Removes and returns the last element (returns null if empty).

```java
public class ArrayDequeDemo {
    public static void main(String[] args) {
        // Create an ArrayDeque
        Deque<String> arrayDeque = new ArrayDeque<>();

        // Add elements to both ends of the ArrayDeque
        arrayDeque.addFirst("Front");
        arrayDeque.addLast("Back");

        // Display the ArrayDeque
        System.out.println("ArrayDeque: " + arrayDeque); // Output: [Front, Back]

        // Add more elements
        arrayDeque.push("Stack Top"); // Pushes to the front (as in stack)
        System.out.println("After push: " + arrayDeque); // Output: [Stack Top, Front, Back]

        // Peek at the first and last elements
        System.out.println("First element (peekFirst): " + arrayDeque.peekFirst()); //
Output: Stack Top
        System.out.println("Last element (peekLast): " + arrayDeque.peekLast()); // Output:
Back

        // Pop an element from the deque (LIFO operation)
        String popped = arrayDeque.pop(); // Pops from the front
        System.out.println("Popped element: " + popped); // Output: Stack Top
        System.out.println("ArrayDeque after pop: " + arrayDeque); // Output: [Front, Back]

        // Remove the last element
        arrayDeque.removeLast();
        System.out.println("ArrayDeque after removing last element: " + arrayDeque); //
Output: [Front]
    }
}
```

4.3. Exercises

Exercise 1:

You are designing a simple customer support ticketing system where incoming support requests (tickets) are processed in a first-come, first-served manner.

Task:

- Create a `Queue<String>` using `LinkedList` to store customer requests.
- Add at least five support requests to the queue.
- Process (remove) requests one by one and print them.
- Check if the queue is empty after processing all requests.

```java
public class SupportTicketQueue {
    public static void main(String[] args) {
        // Step 1: Create a Queue to store support tickets
        Queue<String> supportQueue = new LinkedList<>();

        // Step 2: Add 3 more support tickets to the queue
        supportQueue.offer("Issue with login");
        supportQueue.offer("Payment not processed");


        // Step 3: Process each ticket one by one


        // Step 4: Check if the queue is empty

    }
}
```

Exercise 2:

```java
public class GroceryStoreQueue {
    public static void main(String[] args) {
        // Step 1: Create a queue for customers
        Queue<String> checkoutQueue = new LinkedList<>();

        // Step 2: Add 2 customers to the queue
        checkoutQueue.offer("Alice");
        checkoutQueue.offer("Bob");

        // Step 3: Process each customer in the queue
        System.out.println("Serving customers at checkout...");
        while (!checkoutQueue.isEmpty()) {
            String customer = __________ ; // Serve the first customer
            System.out.println("Checking out: " + customer);
        }

        // Step 4: All customers checked out
```

```java
        System.out.println("All customers have been checked out.");
    }
}
```

Problem Statement:

A grocery store has a checkout line where customers wait to be served in a First-In-First-Out (FIFO) order. Your task is to simulate a simple checkout queue using a `Queue<String>`.

Each customer:

- Joins the queue in order.
- Gets served one by one.
- Leaves the queue after being served.
- When all customers are served, print "All customers have been checked out."

---

5. Map

Class	Description	Ordered?	Sorting	Key Features
HashMap	Unordered map implementation backed by a hash table.	No	No	Fast access (O(1) on average) for key-value pairs, no ordering of elements, allows one null key.
LinkedHashMap	Extends HashMap, maintains insertion or access order.	Yes (insertion or access order)	No	Preserves the order in which entries were inserted or accessed
TreeMap	Implements NavigableMap, stores key-value pairs in a sorted tree.	Yes (sorted by natural or custom comparator)	Yes (natural/comparator order)	Entries sorted by key in natural order or via a comparator, O(log n) time for most operations.

Different ways to define map:-

1. `Map<Integer, String> map = new HashMap<>();`
2. `Map<Integer, String> map = new LinkedHashMap<>();`
3. `Map<Integer, String> map = new TreeMap<>();`

Map is an interface whereas `HashMap`, `LinkedHashMap` and `TreeMap` are classes which implement map interface.

Generic Methods for Map:-

1. Basic Operations
    a. `put(K key, V value)` → Inserts a key-value pair into the map.
    b. `get(Object key)` → Retrieves the value associated with the given key.
    c. `remove(Object key)` → Removes the mapping for the given key.
    d. `containsKey(Object key)` → Checks if the map contains the given key.
    e. `containsValue(Object value)` → Checks if the map contains the given value.

   f. `size()` → Returns the number of key-value pairs in the map.
   g. `isEmpty()` → Returns true if the map is empty.
   h. `clear()` → Removes all key-value pairs from the map.
2. **Iteration & Views**
   a. `keySet()` → Returns a `Set<K>` of all keys.
   b. `values()` → Returns a `Collection<V>` of all values.
   c. `entrySet()` → Returns a `Set<Map.Entry<K, V>>` of all key-value pairs.
3. **Default Values & Compute Methods**
   a. `getOrDefault(Object key, V defaultValue)` → Returns the value for the key or the default value if not found.
   b. `compute(K key, BiFunction<K, V, V> remappingFunction)` → Computes a new value for the key.
   c. `computeIfAbsent(K key, Function<K, V> mappingFunction)` → Computes value only if the key is absent.
   d. `computeIfPresent(K key, BiFunction<K, V, V> remappingFunction)` → Computes value only if the key is present.
4. **Bulk Operations**
   a. `putAll(Map<? extends K, ? extends V> m)` → Copies all key-value pairs from another map.
   b. `putIfAbsent(K key, V value)` → Inserts the key-value pair if the key is not already present.
   c. `replace(K key, V value)` → Replaces the value for a key if it exists.
   d. `replace(K key, V oldValue, V newValue)` → Replaces the value only if it matches the old value.
   e. `replaceAll(BiFunction<K, V, V> function)` → Applies a function to all key-value pairs.
5. **Removal & Cleanup**
   a. `remove(Object key, Object value)` → Removes the mapping only if the key is mapped to the specified value.
   b. `merge(K key, V value, BiFunction<V, V, V> remappingFunction)` → Merges a value with an existing one based on a function.

5.1. HashMap

A HashMap is a key-value pair data structure in Java that allows fast access to values using keys. It is part of the `java.util` package and is widely used because of its efficiency in searching, inserting, and deleting elements.

Key Features of `HashMap`

- Unordered → Does not maintain the order of elements.
- Allows null key and multiple null values.

- Fast retrieval (O(1) average time complexity) due to hashing.
- Duplicate keys are not allowed (if a duplicate key is inserted, it replaces the old value).
- Not thread-safe (use `ConcurrentHashMap` for multi-threading).

```
public class MapInterfaceDemo {
    public static void main(String[] args) {
        // Create a HashMap (which implements the Map interface)
        HashMap<String, Integer> map = new HashMap<>();

        // Add key-value pairs to the map
        map.put("Apple", 50);
        map.put("Banana", 30);
        map.put("Cherry", 20);
        map.put("Cherry", 40);
        System.out.println(map.capacity());

        // Display the map
        System.out.println("Initial Map: " + map); // Output: {Apple=50, Banana=30,
Cherry=20}

        // Get the value for a specific key
        int applePrice = map.get("Apple");
        System.out.println("Price of Apple: " + applePrice); // Output: 50

        // Remove a key-value pair
        map.remove("Banana");
        System.out.println("Map after removing Banana: " + map); // Output: {Apple=50,
Cherry=20}

        // Check if a key is present in the map
        boolean containsCherry = map.containsKey("Cherry");
        System.out.println("Contains Cherry: " + containsCherry); // Output: true

        // Display all keys and values
        System.out.println("Keys: " + map.keySet()); // Output: [Apple, Cherry]
        System.out.println("Values: " + map.values()); // Output: [50, 20]
    }
}
```

5.2. LinkedHashMap

A `LinkedHashMap` is a subclass of `HashMap` that maintains the insertion order of key-value pairs. Unlike HashMap, which does not guarantee any order, `LinkedHashMap` preserves the order in which elements are inserted.

It is part of the `java.util` package and is useful when you need fast lookups (like HashMap) but with predictable iteration order.

Key Features of `LinkedHashMap`

- Maintains insertion order (or access order if enabled).

- Allows one null key and multiple null values.
- Faster than `TreeMap` but slightly slower than HashMap due to ordering overhead.
- Not thread-safe (Use `Collections.synchronizedMap()` for synchronization).
- Can be configured as an LRU [1](Least Recently Used) cache by enabling access order.

`LinkedHashMap`-Specific Methods

- protected boolean `removeEldestEntry(Map.Entry<K, V> eldest)`
  → Used to **remove the eldest entry** when a new one is added. Useful for **implementing LRU caching**.

```java
public class LinkedHashMapDemo {
    public static void main(String[] args) {
        // Create a LinkedHashMap
        Map<String, Integer> linkedHashMap = new LinkedHashMap<>();

        // Add key-value pairs to the LinkedHashMap
        linkedHashMap.put("Apple", 50);
        linkedHashMap.put("Banana", 30);
        linkedHashMap.put("Cherry", 20);

        // Display the LinkedHashMap (in insertion order)
        System.out.println("LinkedHashMap: " + linkedHashMap);

        // Access the 'Banana' key to demonstrate order (default insertion order)
        linkedHashMap.get("Banana");

        // Add another entry to demonstrate order persistence
        linkedHashMap.put("Date", 40);
        System.out.println("LinkedHashMap after adding Date: " + linkedHashMap);

        // Remove an entry
        linkedHashMap.remove("Apple");
        System.out.println("LinkedHashMap after removing Apple: " + linkedHashMap);

        // Iterating through the LinkedHashMap
        System.out.println("Iterating over LinkedHashMap:");
        for (Map.Entry<String, Integer> entry : linkedHashMap.entrySet()) {
            System.out.println(entry.getKey() + " = " + entry.getValue());
        }
    }
}
```

5.3. TreeMap

A `TreeMap` in Java is a part of the `java.util` package and implements the `NavigableMap` interface, which extends `SortedMap`. Unlike HashMap and `LinkedHashMap`, which do not guarantee any ordering,

[1] LRU (Least Recently Used) removes the item that hasn't been accessed for the longest time. It keeps only the most recently used items in memory or cache. LRU cache makes systems faster and more efficient by keeping recent/frequent stuff handy.

`TreeMap` stores key-value pairs in sorted order based on the natural ordering of keys or using a custom comparator.

Key Features of `TreeMap`

- Stores elements in a sorted order based on keys (Ascending order by default).
- Implements **`NavigableMap`**, **`SortedMap`**, and extends **`AbstractMap`**.
- Faster search compared to **`LinkedHashMap`**, but slightly slower than HashMap due to sorting overhead.
- Not thread-safe (Use **`Collections.synchronizedSortedMap()`** for synchronization).
- Does NOT allow null keys, but multiple null values are allowed

TreeMap specific methods:-

1. Navigation Methods
   a. **`firstKey()`** – Returns the first (lowest) key.
   b. **`lastKey()`** – Returns the last (highest) key.
   c. **`pollFirstEntry()`** – Removes and returns the first (lowest) key-value pair.
   d. **`pollLastEntry()`** – Removes and returns the last (highest) key-value pair.
   e. **`higherKey(K key)`** – Returns the smallest key greater than the given key.
   f. **`lowerKey(K key)`** – Returns the largest key smaller than the given key.
   g. **`ceilingKey(K key) -`** Returns the smallest key greater than or equal to the given key.
   h. **`floorKey(K key)`** – Returns the largest key less than or equal to the given key.
   i. **`higherEntry(K key)`** – Returns the key-value pair with the smallest key greater than the given key.
   j. **`lowerEntry(K key)`** – Returns the key-value pair with the largest key smaller than the given key.
   k. **`ceilingEntry(K key)`** – Returns the key-value pair with the smallest key greater than or equal to the given key.
   l. **`floorEntry(K key)`** – Returns the key-value pair with the largest key less than or equal to the given key.
2. Range Query Methods
   a. **`headMap(K toKey, boolean inclusive)`** – Returns a view of the map with entries less than the given key.
   b. **`tailMap(K fromKey, boolean inclusive)`** – Returns a view of the map with entries greater than or equal to the given key.
   c. `subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)` – Returns a view of the map within the given range.

```java
public class TreeMapExample {
    public static void main(String[] args) {
        // Creating a TreeMap
        TreeMap<Integer, String> treeMap = new TreeMap<>();
```

```java
        // Adding elements (TreeMap sorts keys automatically)
        treeMap.put(10, "Apple");
        treeMap.put(30, "Mango");
        treeMap.put(20, "Banana");
        treeMap.put(40, "Orange");
        treeMap.put(50, "Grapes");

        // Displaying the TreeMap (sorted order)
        System.out.println("TreeMap: " + treeMap);

        // Navigation Methods
        System.out.println("First Key: " + treeMap.firstKey()); // 10
        System.out.println("Last Key: " + treeMap.lastKey());   // 50
        System.out.println("Higher Key (20): " + treeMap.higherKey(20)); // 30
        System.out.println("Lower Key (30): " + treeMap.lowerKey(30));   // 20
        System.out.println("Ceiling Key (25): " + treeMap.ceilingKey(25)); // 30
        System.out.println("Floor Key (25): " + treeMap.floorKey(25));   // 20

        // Entry navigation
        System.out.println("First Entry: " + treeMap.firstEntry());
        System.out.println("Last Entry: " + treeMap.lastEntry());

        // Removing first and last entries
        System.out.println("Poll First Entry: " + treeMap.pollFirstEntry());
        System.out.println("Poll Last Entry: " + treeMap.pollLastEntry());

        // Display TreeMap after polling
        System.out.println("TreeMap after polling: " + treeMap);

        // Submaps and Range Queries
        System.out.println("HeadMap (less than 30): " + treeMap.headMap(30));
        System.out.println("TailMap (from 30 onwards): " + treeMap.tailMap(30));
        System.out.println("SubMap (20 to 40): " + treeMap.subMap(20, 40));
    }
}
```

5.4. Exercise

Exercise 1:

Problem Statement:

Create a `HashMap` that stores the names of students as keys and their marks as values. Then, perform the following tasks:

- Insert at least five students with their marks.
- Print all students and their marks.
- Find and print the marks of a student named "Alice".
- Remove a student named "Bob" from the map.
- Check if a student named "David" exists in the map.
- Display the total number of students in the map.

```java
public class StudentMarksMap {
    public static void main(String[] args) {
        // Step 1: Create a HashMap to store student names and marks
        Map<String, Integer> studentMarks = new HashMap<>();

        // Step 2: Add 3 more students and their marks
        studentMarks.put("Alice", 85);
        studentMarks.put("Bob", 78);

        // Step 3: Print all students and their marks

        // Step 4: Find and print Alice's marks

        // Step 5: Remove Bob from the map

        // Step 6: Check if David exists in the map

        // Step 7: Display the total number of students
    }
}
```

Exercise 2:

Problem Statement:

You are given a list of employee names and their salaries. Perform the following operations using a `TreeMap`:

- Add employees and their salaries to the map.
- Print all employees in sorted order (since `TreeMap` maintains keys in sorted order).
- Update the salary of a specific employee (e.g., increase John's salary by 5000).
- Find the employee with the highest and lowest salary.
- Check if an employee named "Michael" exists in the map.
- Remove an employee from the map.
- Display the final list of employees and their salaries

```java
public class EmployeeSalaryMap {
    public static void main(String[] args) {
        // Step 1: Create a TreeMap to store employee names and salaries

        //map entries
        employeeSalaries.put("John", 50000);
        employeeSalaries.put("Alice", 60000);
        employeeSalaries.put("Bob", 55000);
        employeeSalaries.put("David", 65000);
        employeeSalaries.put("Eve", 70000);

        // Step 2: Print all employees in sorted order

        // Step 3: Increase John's salary by 5000

        // Step 4: Find the employee with the highest and lowest salary
```

```
        // Step 5: Check if "Michael" exists in the map;

        // Step 6: Remove an employee (e.g., "Bob")
    }
}
```

6. Exercise

Exercise 1:

Problem Statement:

You are developing a Library Management System that uses different Java Collection types (List, Set, Queue, and Map) to manage books and users. Implement the following functionalities:

Task Breakdown:

- Use an `ArrayList` to store book names.
- Add at least 5 books to the list.
- Print all books.
- Remove a book from the list.
- Use a `HashSet` to store unique users who borrowed books.
- Add at least 3 users.
- Attempt to add a duplicate user and observe the result.
- Use a `HashMap` to track books and their borrowers.
- Store book names as keys and borrower names as values.
- Find out who borrowed a specific book.
- Remove a book entry when it's returned.

```java
public class LibraryManagementSystem {
    public static void main(String[] args) {
        // 1 Using List (ArrayList) to store books (A,B,C,D,E)

        System.out.println("Books Available: " + books);
         // Remove book C
        System.out.println("Books After Removal: " + books);

        // 2 Using Set (HashSet) to store unique users(Alice,Bob,Charlie)

        users.add("Alice"); // Attempt to add duplicate user
        System.out.println("Registered Users: " + users); // HashSet prevents duplicates



        // 3 Using Map (HashMap) to track borrowed books {(A,Alice),(B,Bob),(D,Charlie)}
```

```java
        System.out.println("Borrowed Books: " + borrowedBooks);
        System.out.println("Who borrowed 'Book A'? " + borrowedBooks.get("Book A"));

        // Remove an entry when book is returned

        System.out.println("Updated Borrowed Books: " + borrowedBooks);
    }
}
```

Expected Output:

```
Books Available: [Book A, Book B, Book C, Book D, Book E]
Books After Removal: [Book A, Book B, Book D, Book E]
Registered Users: [Alice, Bob, Charlie]   // Alice is not added twice
```

```
Borrowed Books: {Book A=Alice, Book B=Bob, Book D=Charlie}
Who borrowed 'Book A'? Alice
Updated Borrowed Books: {Book B=Bob, Book D=Charlie}
```

Exercise 2:

Problem Statement:

You need to develop a Task Scheduler that manages and executes tasks in the order they are received.

- Tasks are stored in an array before they are processed.
- The tasks are then added to a `Queue` (FIFO order) for execution.
- The system should process the tasks one by one and remove them from the queue.

```java
public class TaskScheduler {
    public static void main(String[] args) {
        // Step 1: Define an array of tasks {A,B,C,D,E}

        // Step 2: Initialize a Queue (LinkedList) to store tasks

        // Step 3: Add tasks from array to queue

        // Step 4: Process tasks in FIFO order
        System.out.println("Processing Tasks...");

        System.out.println("All tasks completed!");
    }
}
```

Task Breakdown:-

- Store a list of tasks (as Strings) in an array (e.g., "Task A", "Task B", etc.).
- Add all tasks to a `Queue` (using `LinkedList`) to ensure FIFO order.
- Process and remove each task from the queue (print a message when processing each task).

Expected Output:

```
Processing Tasks...
```

```
Executing: Task A
Executing: Task B
Executing: Task C
Executing: Task D
Executing: Task E
All tasks completed!
```

References:-

https://github.com/kudhru/m24-oop-examples by Dhruv Kumar (https://kudhru.github.io/)