⭐ Collection Framework

The Collection Framework in Java is a standardized architecture for storing and manipulating groups of objects.
It provides:

- Interfaces (Collection, List, Set, Queue, Deque, Map)
- Classes (ArrayList, LinkedList, Vector, Stack, HashSet, LinkedHashSet, TreeSet, PriorityQueue, etc.)
- Utility Classes (Collections, Arrays)
- Algorithms (Sorting, Searching, Shuffling, Reversing)

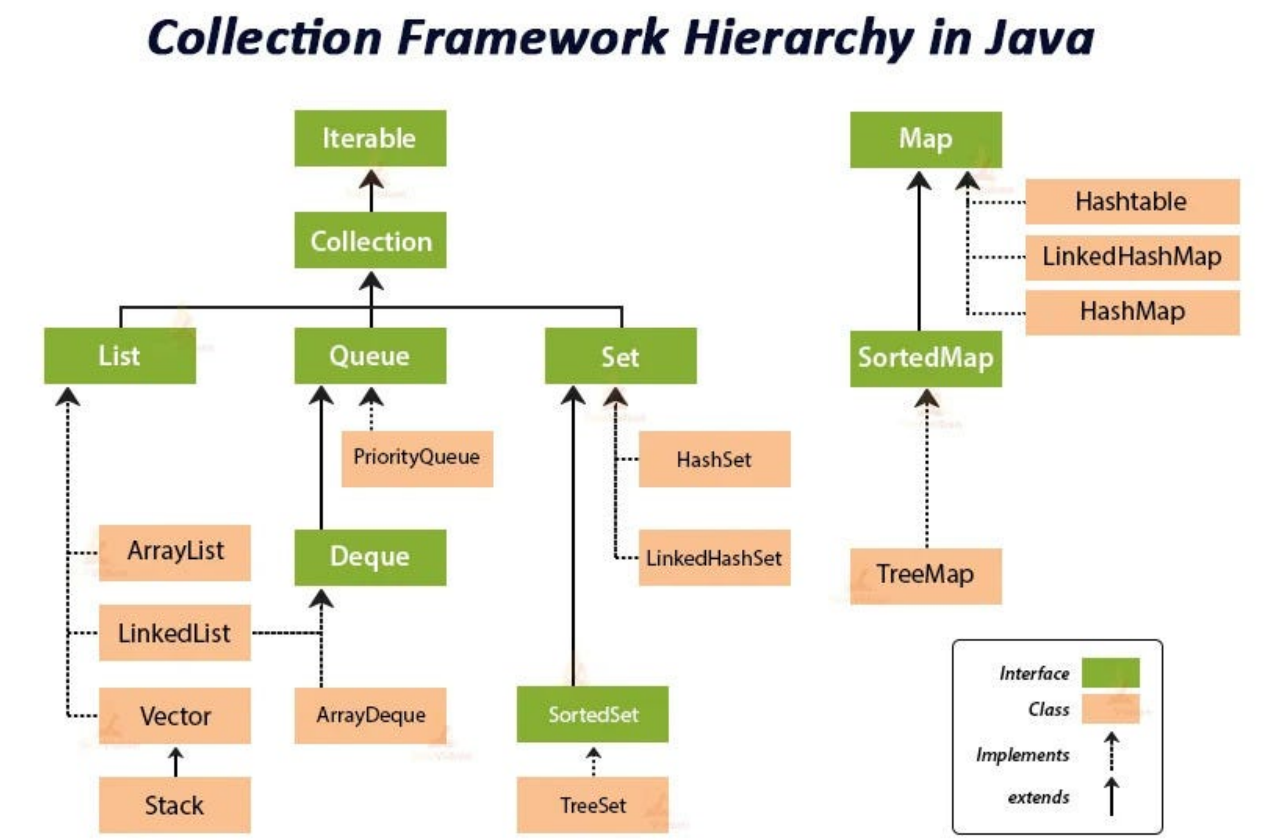
The framework supports operations like insertion, deletion, searching, sorting, updating, iteration, and manipulation of data.

⭐ Difference between Array and ArrayList

Array	ArrayList / Collection
Static size (fixed-length).	Dynamic size (grows/shrinks automatically).
Can store **primitive types** and objects.	Stores **objects only** (wrapper classes for primitives).
Fast and memory efficient for fixed-size data.	More flexible but slightly higher overhead.
Uses direct indexing with a fixed structure.	Built on top of a **dynamic internal array** (`Object[]`).
No built-in methods for manipulation.	Provides many built-in methods (add, remove, contains, etc.).
Supports multi-dimensional arrays.	One-dimensional; can nest lists to simulate multi-dimension.

⭐ Difference between Collection and Collections

Collection	Collections
It is a **interface** in the Collection Framework.	It is a **utility class** in `java.util`.
Represents a group of objects as a single unit.	Provides **static methods** like `sort()`, `reverse()`, `min()`, `max()`, `synchronizedList()`, etc.
Parent of List, Set, Queue.	Works *on* Collection objects.



⭐ List Interface

Definition:
List is a child interface of Collection.
It represents an ordered collection that preserves insertion order and allows duplicate elements.

Key Features of List

1. Maintains Order: Insertion order is preserved.
2. Duplicates Allowed: Multiple identical elements permitted.
3. Index-Based Access: Supports add(index), get(index), set(index), remove(index).
4. Null Allowed: List implementations allow null values.
5. Supports Iterators:
   - Iterator (forward)
   - ListIterator (forward + backward)

Implementations of List

1. ArrayList
   - Backed by dynamic array.
   - Fast random access (O(1)).
   - Slower for insert/delete in the middle (O(n)).
   - Not synchronized.

2. LinkedList
   - Backed by doubly linked list.
   - Fast insert/delete operations (O(1) at ends).
   - Slow random access (O(n)).
   - Also implements Queue/Deque.

3. Vector (Legacy)
   - Similar to ArrayList but synchronized.
   - Slower due to synchronization.
   - Rarely used today.

4. Stack (Legacy)
   - Extends Vector.
   - LIFO structure (push, pop, peek).
   - Modern replacement: ArrayDeque.

1. ArrayList

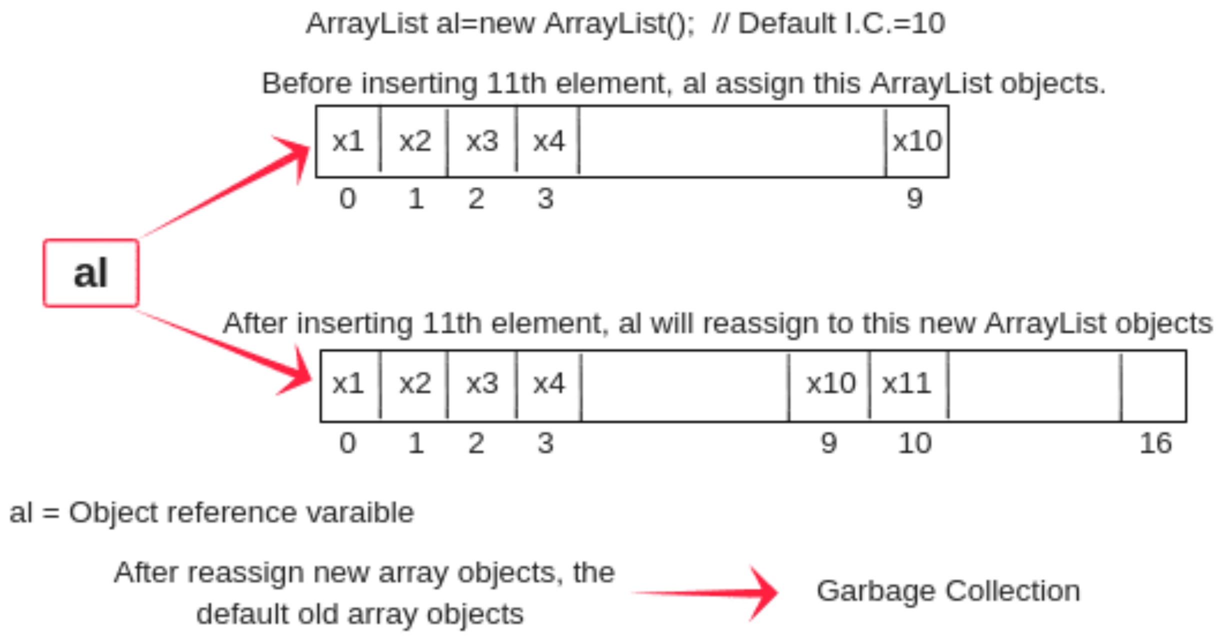
ArrayList is a dynamic array implementation of the List interface.
It can grow or shrink automatically as elements are added or removed.

Key Features of ArrayList

1. Dynamic Array: Automatically resizes.
2. Maintains Insertion Order: Index-based access.
3. Allows Duplicates: Same values can be inserted.
4. Allows Null Values: Can store one or more nulls.
5. Fast Random Access: get(index) is O(1).
6. Default Capacity: 10 (auto-expands by 1.5×).
7. Resizable Underlying Array: Uses Object[] internally.
8. Not Synchronized: Not thread-safe by default.
   Thread-safe options:
      - Collections.synchronizedList(list)
      - CopyOnWriteArrayList

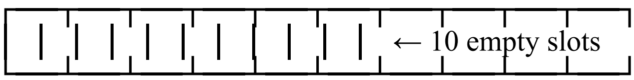
Common Methods in ArrayList

- add(E e)
- add(int index, E e)
- get(int index)
- set(int index, E e)
- remove(int index) / remove(Object o)
- contains(Object o)
- size()
- clear()
- indexOf() / lastIndexOf()
- iterator() / listIterator()
- addAll(Collection c)



Initially, when we declare ArrayList<Integer> al = new ArrayList<>();

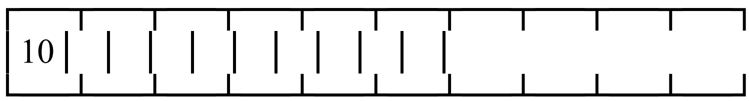
al



Index: 0 1 2 3 4 5 6 7 8 9

After first add → al.add(10);

al



Index: 0 1 2 3 4 5 6 7 8 9

After adding all initial elements (10 elements total)

al.add(10);
al.add(20);
al.add(30);
al.add(40);
al.add(50);
al.add(60);
al.add(70);
al.add(80);

al.add(90);
al.add(100);

al (Initial capacity full)

10	20	30	40	50	60	70	80	90	100		

Now add a new element → al.add(110);
➡ Capacity is full → ArrayList **resizes by 50%**
➡ New capacity = **15**
✔ Default Capacity = 10

```
al     (Resized by 50%)

┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
|10   |20   |30   |40   |50   |60   |70   |80   | 90  |100  |110  |     |     |     |     |
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘

 Capacity: 15
 Size: 11
```

Resize happens by 1.5× (50%)
Array grows only when full
After removing elements, capacity does NOT shrink automatically

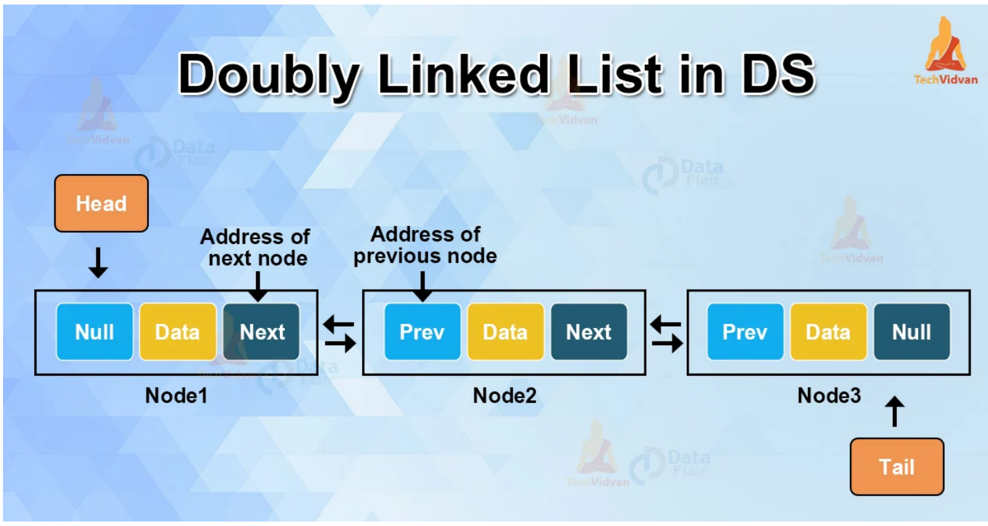
2. LinkedList

The LinkedList class in Java is part of the Java Collection Framework and implements the List, Deque, and Queue interfaces.
It represents a doubly-linked list internally and provides efficient insertion and deletion operations compared to ArrayList.

Key Features of LinkedList

1. **Doubly Linked List:**
   Each element (node) contains references to both its **previous** and **next** nodes.
2. **Efficient Insertions and Deletions:**
   Adding or removing elements—especially at the **beginning** or **middle**—is faster than ArrayList
   because no shifting of elements is required.
3. **Maintains Insertion Order:**
   Preserves the order in which elements are added.
4. **Allows Duplicates and Nulls:**
   Supports duplicate elements and can store null values.

5. Implements Queue and Deque Interfaces:
   Can be used as a Queue (FIFO) or as a Deque (Double-Ended Queue) using methods like



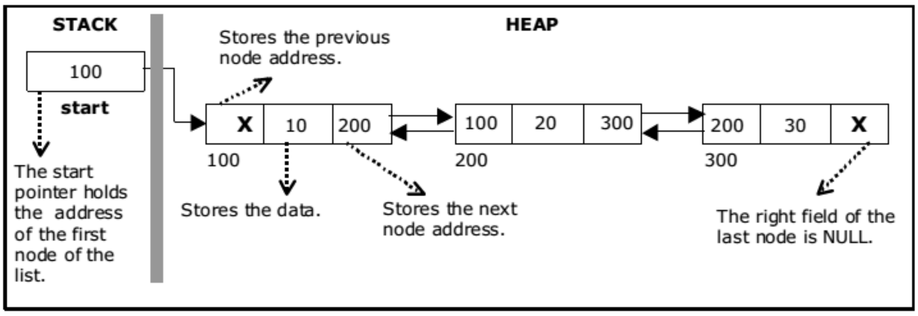


Figure 3.3.1. Double Linked List

When to Use LinkedList?

✔ You perform **frequent insertion/deletion**
✔ You need **Queue/Deque** implementation
✔ Order of elements matters
✔ You don't need fast random access

When NOT to Use LinkedList?

Avoid LinkedList when:
You frequently access elements by index
You need fast data reading
Memory is constrained (because nodes take more memory)

Why LinkedList is Faster for Insert/Delete?

Because only **pointers** change — no shifting of elements.

Diff b/w arrayList And LinkedList

Aspect	ArrayList	LinkedList
Data Structure	Dynamic Array (contiguous memory)	Doubly Linked List (nodes: data + next + prev)
Random Access Performance	**Fast (O(1))** → direct index access	**Slow (O(n))** → must traverse nodes
Insertion/Deletion Performance	**Slow (O(n))** → shifting required	**Fast (O(1))** at beginning/end
Memory Usage	Less memory	More memory (extra pointers)
Thread-Safety	Not synchronized	Not synchronized
Growth	Increases capacity by **50%** (old * 1.5)	Adds nodes dynamically (**no fixed capacity**)
Best Use Case	Frequent read/search	Frequent add/delete operations
Allows Null	Yes	Yes
Maintains Order	Yes	Yes
Duplicates Allowed	Yes	Yes
Access by Index	Fast (O(1))	Slow (O(n))
Resizing	Automatic	Automatic

3. Vector

The **Vector** class in Java is part of the **Java Collection Framework** and is located in the java.util package. It implements the **List** interface and stores elements in a **dynamic, resizable array**.
Unlike ArrayList, **Vector is synchronized**, which makes it **thread-safe** for multi-threaded environments.

Key Features of Vector

1. **Resizable Array:**
   Automatically grows or shrinks as elements are added or removed.
2. **Synchronized:**
   All methods in Vector are synchronized, making it thread-safe but slightly slower than ArrayList.
3. **Maintains Insertion Order:**
   Elements remain in the order they are added.
4. **Allows Duplicates:**
   Duplicate elements are permitted.
5. Allows Null Elements:
   Can store one or more null values.
6. **Legacy Class:**
   Introduced in **JDK 1.0** and later retrofitted to implement the **List** interface.

Difference b/w arrayList and vector

Aspect	ArrayList	Vector
Thread-Safety	Not synchronized	**Synchronized (thread-safe)**
Performance	Faster	Slower (synchronization overhead)
Synchronization	Must be done manually	All methods synchronized
Growth	Grows by **50%** (old * 1.5)	Grows by **100%** (doubles capacity)
Memory Overhead	Less	More (sync + internal overhead)
Access Time	Fast	Slower
Resize Behavior	Automatic	Automatic (double size)
Null Elements Allowed	Yes	Yes
Insertion/Deletion Performance	Slower (shifting)	Slower than LinkedList but OK
Legacy Class	JDK 1.2	**JDK 1.0 (Legacy)**
Initial Capacity	10	10
Duplicates Allowed	Yes	Yes

⭐ 2. Set Interface

Set interface represents a collection of unique elements where duplicates are not allowed.
It extends the Collection interface and is implemented by HashSet, LinkedHashSet, and TreeSet.
Set does not maintain index, and ordering depends on the implementation class.

✔ Key Features of the Set Interface

1. **No Duplicate Elements:**
   A Set does not allow duplicate values. If you try to add a duplicate element, the Set simply ignores it.
2. **Unordered Collection:**
   A Set does not guarantee any specific order of elements.
   (However, specific implementations like **LinkedHashSet** maintain insertion order, and **TreeSet** maintains sorted order.)
3. **Extends Collection Interface:**
   Since Set extends the Collection interface, it inherits all common methods such as add(), remove(), contains(), size(), etc.

⭐IMPLEMENTATION CLASSES OF SET

The Set Interface has 3 main implementation classes:

1. HashSet
2. LinkedHashSet
3. TreeSet

WHY SET CANNOT STORE DUPLICATES?

Set does NOT allow duplicate values because:

✔ Set uses hashing
Every element is converted into a **hash code** and stored in **buckets**.
✔ If a duplicate element is added:

- Same hashCode
- Same bucket
- Set checks using equals() method
- If values are equal → element is **NOT added**

Set checks equality using:

1. hashCode()
2. equals()

⭐ HashSet in Java

The **HashSet** class (in java.util) implements the Set interface.
It stores elements in a **hash table** and ensures that the collection contains **only unique elements**.

✔ Key Features of HashSet

1. **No Duplicates:**
   HashSet does not allow duplicate values.
   When a duplicate is added, HashSet simply ignores it.
2. **Unordered:**
   HashSet does **not** maintain any specific order of elements.
   The output order depends on the **hash value** of each element.
3. Allows Null:
   HashSet allows only one null value.
4. **Not Synchronized:**
   HashSet is **not thread-safe**.
   If multiple threads access it simultaneously, external synchronization is required.

How HashSet Stores an Element?

✔ Step 1 → hashCode() of the element is generated
"Java".hashCode() → 23123

Step 2 → Bucket location calculated
bucketIndex = hashCode % capacity

✔ Step 3 → Check if bucket is empty

- If empty → store element
- If not empty (collision) → LinkedList/Tree structure used

✔ Step 4 → equals() check
If hashCode matches → equals() confirms if same element.
✔ Step 5 → If equals() = true → duplicate ignored
✔ If equals() = false → stored in same bucket (collision chain)

4. Default Capacity of HashSet

HashSet hs = new HashSet();

✔ Default Capacity = 16 buckets
(Because HashSet uses internal HashMap → HashMap capacity = 16)

5. Load Factor (VERY IMPORTANT)

Load Factor = 0.75 (Default)
Load Factor decides when to increase capacity.
✔ Load Factor Meaning:
When 75% of HashSet buckets are filled → it grows (rehashing).

6. Capacity Growth Formula
New Capacity = Old Capacity × 2

When 12 elements are added → HashSet grows its size to **32**

⭐ LinkedHashSet

LinkedHashSet stores unique elements while maintaining insertion order.It uses a combination of HashTable and LinkedList internally. Fast search, preserves order, and allows one null value.

LinkedHashSet is a **Set** implementation that:
✔ **Does NOT allow duplicates**
✔ **Maintains insertion order**
✔ Uses **LinkedList + HashTable** internally
✔ Allows **only one null**
✔ Faster than TreeSet
✔ Slightly slower than HashSet

2. Internal Structure (Very Important)
LinkedHashSet uses 2 data structures together:

1. **HashTable** → For hashing + fast search
2. **Doubly Linked List** → To maintain insertion order

. How LinkedHashSet Works Internally?
Step 1 → hashCode() generates bucket index
Step 2 → equals() checks duplicates
Step 3 → Node stored in bucket + a LinkedList chain
Step 4 → LinkedList preserves insertion order

3. TREESET

TreeSet is a **sorted Set** implementation in Java that:
✔ Stores **unique values only**
✔ Maintains **ascending sorted order**
✔ Does **NOT** allow null
✔ Uses **Red-Black Tree** internally
✔ Is slower than HashSet & LinkedHashSet
✔ Allows **range operations** (higher, lower, ceiling, floor)

3. Sorting Behavior
TreeSet maintains data in **ascending order** by default:

Input : 30, 10, 40, 20
Output: [10, 20, 30, 40]

4. Why TreeSet Does NOT Allow null?
Because null cannot be compared using comparison logic (compareTo()).

⭐ HashSet vs LinkedHashSet vs TreeSet — Comparison Table

Feature	HashSet	LinkedHashSet	TreeSet
Order	No order (random)	✔Maintains **insertion order**	✔Maintains **sorted** ✔**(ascending) order**
Internal Structure	Hash Table	Hash Table + LinkedList	Red-Black Tree (Self-balancing BST)
Duplicates Allowed?	No	No	No
Null Allowed?	One null allowed	✔ One null allowed	No null allowed
Performance (Speed)	Fastest	Fast	Slowest
Time Complexity	O(1) search, add, delete	O(1) but slightly slower	O(log n)
When to Use?	When you only need unique elements, no order	When you want unique + maintain insertion order	When you want unique + automatically sorted values
Sorting	Not supported	Not supported	Always sorted
Memory Usage	Least	Medium	High
Best Use Case	Fast lookup	Keep order + uniqueness	Sorted unique data (numbers, names)
Implement	Set	Set	NavigableSet (SortedSet)
Suitable for	Searching operations	Caching, predictable iteration	Range operations (higher(), lower(), ceiling())
Thread-safe?	No	No	No

Difference between List vs Set

Aspect	List	Set
Order	Maintains **ordered elements** (insertion order).	Does **not guarantee any order** (unordered or sorted, depends on implementation).
Duplicates	Allows **duplicate elements**.	Does **not allow duplicate elements**.
Index-Based Access	Supports **index-based access** (like arrays).	No index-based access; elements are stored without positions.
Insertion Order	Always preserves **insertion order**.	May or may not preserve insertion order (depends on implementation).

Structure	Internally based on **dynamic array** (ArrayList) or **linked list** (LinkedList).	Internally based on **hash table** (HashSet, LinkedHashSet) or **tree structure** (TreeSet).
Synchronized	Some implementations like **Vector** are synchronized. Others are not.	Generally **not synchronized**, unless manually synchronized.
Performance	Slower for frequent insert/delete operations (due to shifting or linked traversal). Searching is fast in ArrayList but slower in LinkedList.	Faster for insertions, deletions, and lookups because of **hash-based** or **tree-based** structure.
Method Support	Supports methods like `get()`, `set()`, `add()`, `remove()`, `size()`, `contains()`.	Supports methods like `add()`, `remove()`, `size()`, `contains()`, but **no index-based methods**.
Use Case	Best when you need **ordered data**, **duplicates**, and **index-based access**.	Best when you need **unique values**, **faster lookup**, and order does not matter (or sorted order using TreeSet).

3. Queue :

Queue in Java is an interface in the **java.util** package.It follows the **FIFO (First In First Out)** principle. First element added → first removed

Queue is implemented by:

- LinkedList
- PriorityQueue
- ArrayDeque

Queue is used when order of processing matters.

Example real-world queues:

- People standing in a line
- Printer jobs
- Tasks scheduling

Operations in Queue (Text Format)

1. Insert Operations
   - **add(element)** → Inserts element; throws exception if the queue is full.
   - **offer(element)** → Inserts element; returns false if the queue is full (safe method).
2. Remove Operations
   - **remove()** → Removes head element; throws exception if queue is empty.
   - **poll()** → Removes head element; returns null if queue is empty.
3. Read/Access Operations
   - **element()** → Returns head element; throws exception if queue is empty.
   - **peek()** → Returns head element; returns null if queue is empty.

Difference Between add() and offer()

add()

- Throws exception if insertion fails
- Strict operation
- Used in unbounded queues (like LinkedList)

offer()

- Returns false if insertion fails
- Safe operation
- Recommended for bounded queues (capacity-restricted)

Difference Between remove() and poll()

remove()

- Removes and returns head element
- Throws NoSuchElementException if queue is empty
- Strict method

poll()

- Removes and returns head element
- Returns null if queue is empty
- Safe method

Difference Between element() and peek()

element()

- Returns head element
- Throws NoSuchElementException if queue is empty
- Strict operation

peek()

- Returns head element
- Returns null if queue is empty
- Safe operation

Queue Characteristics

- Ordered collection
- Allows duplicates
- No index-based access
- Fast insertion/removal from ends
- Mostly used in scheduling, task handling, messaging

2. PriorityQueue – Deep Explanation

What is PriorityQueue?

PriorityQueue is a special type of queue where:Elements are ordered based on priority, not insertion order.

Default priority → **Natural ordering**

- Numbers → ascending
- Strings → alphabetical

Stores elements using a **Min Heap** internally (smallest element at head).

Features of PriorityQueue

- No null values
- Allows duplicates
- Not synchronized
- Automatically sorts elements
- Head always contains **minimum** value

What is ArrayDeque?

ArrayDeque (Double-ended queue) can behave like:

- Queue → FIFO
- Stack → LIFO

Faster than **Stack class** (Legacy).

4. Queue vs Deque vs Stack – Comparison

Feature	Queue	Deque	Stack
Full Form	First-In-First-Out	Double-Ended Queue	Last-In-First-Out
Insert	Rear	Both ends	Top
Remove	Front	Both ends	Top
Order	FIFO	FIFO + LIFO	LIFO
Methods	add, offer, remove, poll, peek	addFirst, addLast, pollFirst, pollLast	push, pop, peek
Null	allowed sometimes	No (ArrayDeque)	No
Example Class	LinkedList, PriorityQueue	ArrayDeque, LinkedList	Stack
Use Case	Scheduling, queues	Double-ended operations	Backtracking, Undo-redo

Map Interface

The Map Interface (java.util.Map) represents a data structure that stores data in Key–Value pairs.
Key - value
Roll No → Student Name
101 → Akash
102 → Rohan
103 → Neha

KEY points about Map:

✔ A map maps keys to values
✔ Keys must be unique
✔ Values can be duplicate
✔ A key can map to only one value
✔ No indexing (Cannot access by position)
✔ Not part of Collection hierarchy (but still under java.util)

Why do we need a Map?

✔ Storing user credentials
email → password
✔ Storing config (Automation Testing)
"URL" → "https://test.com"
"browser" → "chrome"
✔ API Response (JSON)
JSON is nothing but a Map internally

```
{
  "id": 101,
  "name": "Laptop",
  "price": 55000
}
```

✔ Database records
EmployeeID → EmployeeObject

Map Architecture (Internal Representation)
A Map entry is stored as:
Entry = (Key, Value)

Internally represented as objects of **Map.Entry interface**.

101 = "Java"
102 = "Python"
103 = "DevOps"

Important Rule of Map

If same key is inserted again:

map.put(101, "Java");
map.put(101, "Selenium");

✔ Allows multiple null values

map.put(1, null);
map.put(2, null);

Type	Order	Sort	Null Key	Null Value	Speed	Internal Structure
HashMap	No	No	1 allowed	Many	Fastest	HashTable + LinkedList
LinkedHashMap	Insertion order	No	1 allowed	Many	Fast	HashTable + LinkedList
TreeMap	Yes	Sorted (ASC)	No	No	Slowest	Red-Black Tree
Hashtable	No	No	No	No	Slow	Synchronized

HASHMAP IN JAVA

What is HashMap?
HashMap is a class in java.util package that stores data in the form of key–value pairs.

HashMap is a class in java.util package that stores data in the form of key–value pairs.

101 → "Java"
102 → "Python"
103 → "Selenium"

Key Points

✔ Key must be unique
✔ Value can be duplicate
✔ Stores data using hashing
✔ Allows one null key
✔ Allows multiple null values
✔ Does NOT maintain order
✔ Very fast for search, insert, delete

2. Why do we need HashMap? (Real Life Usage)

HashMap is used when you want FAST search based on key.
Real-time examples:
✔ Store user details → username → password
✔ Store configuration → URL → value
✔ API JSON response (internally Map)

✔ Cache data → productId → productData
✔ Browser cookies → key → value
✔ Database record mapping

LINKEDHASHMAP IN JAVA

LinkedHashMap is a subclass of HashMap that maintains **insertion order** of elements.
It combines the advantages of **HashMap's fast lookup** and a **doubly linked list's predictable ordering**.

✔ Stores data in **key–value** pairs
✔ Maintains **insertion order**
✔ Allows **one null key**
✔ Allows multiple null values
✔ Does **NOT** allow duplicate keys
✔ Provides **fast access** (almost same speed as HashMap)

It is basically:
HashMap + LinkedList (doubly-linked list)

HashMap is fast but **unordered** → output unpredictable.
LinkedHashMap solves this by maintaining the order in which elements are inserted.

```
LinkedHashMap<String, Integer> map = new LinkedHashMap<>();
map.put("B", 20);
map.put("A", 10);
map.put("C", 30);
```

Output → {B=20, A=10, C=30}

Internal Structure of LinkedHashMap
LinkedHashMap uses:

1. **HashTable** → For key hashing & fast lookup
2. **Doubly LinkedList** → For maintaining insertion order

2. Key Characteristics

✔ Maintains insertion order
(Unlike HashMap which is completely unordered)
✔ Uses Hash Table + Doubly Linked List
✔ Allows one null key, multiple null values
✔ Not synchronized (NOT thread-safe)
✔ Faster iteration than HashMap due to predictable ordering
✔Load factor default: 0.75

4. Order Types in LinkedHashMap
1. Insertion Order (default)

new LinkedHashMap<>(16, 0.75f, true);
Whenever a key is accessed → it moves to end of list.
Used in LRU Cache implementations.

5. Important Methods

Inherited from HashMap:

- put()
- get()
- remove()
- containsKey()
- containsValue()
- size()
- clear()
- keySet()
- values()
- entrySet()

6. Advantages
✔ Maintains stable, predictable ordering
✔ Fast lookup (like HashMap)
✔ Faster iteration
✔ Can implement LRU cache
✔ Good for JSON-like ordered data

HASHTABLE

1. Introduction
Hashtable is a **legacy**, **thread-safe** implementation of the Map interface that stores key–value pairs using **hashing**.
It was introduced in **Java 1.0** and existed even before Java Collections Framework.

2. Key Characteristics
✔ Thread-safe (all methods synchronized)
✔ Does NOT allow null keys or null values
✔ Slower due to synchronization
✔ Uses hash table internally
✔ Legacy class (older than HashMap)
✔ Keys must be unique
✔ Values can be duplicates
✔ Iteration order is NOT guaranteed
✔ Used rarely in modern code
✔ Fail-fast iterator

4. Why Hashtable Does NOT Allow Null?
Because null cannot be used for:

- null.hashCode()
- equals()
- Synchronization

Thus, null key/value leads to confusion & thread safety issues, so Java disallowed it.

5. Hashtable vs HashMap vs LinkedHashMap

Feature	Hashtable	HashMap	LinkedHashMap
Thread-safe	Yes	No	No
Synchronized	Synchronized	Not synchronized	Not synchronized
Null key/ value	No	Yes	Yes
Order	None	None	Insertion order
Performance	Slow (due to locking)	Fast	Slightly slower than HashMap
Use Case	Old multi-threaded apps	Modern apps	Ordered apps
Introduced	Java 1.0	Java 1.2	Java 1.4

Difference Table: List vs Set vs Map

Feature	List	Set	Map
Type of Collection	Ordered collection of elements	Unordered (or ordered depending on implementation) collection of **unique elements**	Key–value pairs
Duplicates Allowed?	Yes	No	Keys: NoValues: Yes
Null Allowed?	Yes (multiple)	Mostly (e.g., HashSet allows one null)	Depends: HashMap , Hashtable
Order Maintained ?	Yes (insertion order)	No (HashSet), Yes (LinkedHashSet), Sorted (TreeSet)	No (HashMap), Yes (LinkedHashMap), Sorted (TreeMap)
Index-based Access?	Yes (list.get(index))	No	No
Underlying Structure	Dynamic array / LinkedList	HashTable / Tree / LinkedList	HashTable / Tree / LinkedList
Use Case	Ordered data, duplicates allowed	Unique items, checking duplicates	Fast lookup using keys
Example Implementations	ArrayList, LinkedList, Vector	HashSet, LinkedHashSet, TreeSet	HashMap, LinkedHashMap, TreeMap, Hashtable

WHAT IS A THREAD?

A thread is the smallest unit of execution in a program.
In simple words:
✔ A thread is like a lightweight sub-program
✔ It runs independently inside a program
✔ A single program can have multiple threads running at the same time

VERY SIMPLE REAL-LIFE EXAMPLE
Imagine you are using your mobile:

- You are listening to music → **Thread 1**
- Downloading a file → **Thread 2**
- Chatting on WhatsApp → **Thread 3**

All are happening **together** → This is **multithreading**.

DEFINITION
A **thread** is a lightweight process that executes a part of a program.It shares the same memory and resources of the main program (process), but executes independently.

SYNCHRONIZED vs NON-SYNCHRONIZED

1. What is Synchronization?

Synchronization means locking.
Only **one thread** can access a shared resource at a time.
✔ Prevents data corruption
✔ Ensures thread safety
✔ Used in multi-threaded applications
✔ Creates a "queue" — one thread enters, others wait

2. What is Non-Synchronized?

Non-synchronized means:
✔ Multiple threads can access a resource at the **same time**
✔ No locking
✔ Faster performance
Risk of data corruption
Used in single-threaded apps or where speed is more important.

3. Real-Life Example

SYNCHRONIZED = ATM Queue

- Only **one person** can use ATM at a time
- Others **wait**
- Safe
- Slow

This is how Hashtable, Vector, StringBuffer behave.

NON-SYNCHRONIZED = Restaurant Buffet

- Many people take food **together**
- No waiting
- Fast
- But things can get messy (like spilled food)

This is how HashMap, ArrayList, StringBuilder behave.

4. Why Synchronized is Slow?

Because:

- Every method has **lock**
- Only one thread works at a time
- Other threads **wait** → queue is formed

This increases **waiting time** → slow performance.

5. Why Non-Synchronized is Fast?

- No locking
- Multiple threads work at same time
- No waiting queue

But risk of **race condition** (two threads changing data together).

6. Which One Should We Use?

✔ Use Synchronized (Thread-safe) when:

- Multiple threads modify same data
- Banking systems
- Server logs
- Shared counters
- Multi-thread automation frameworks

✔ Use Non-Synchronized when:

- Only single thread
- Speed is more important
- Data does not need locking
- Lists used only inside one test case

Difference Between Synchronized and Non-Synchronized

Feature	Synchronized	Non-Synchronized
Definition	Only **one thread** can access a resource at a time (locking applied)	**Multiple threads** can access the resource at the same time
Thread Safety	✔ Thread-safe	Not thread-safe
Speed	Slow (due to locking)	Fast (no locking)
Risk of Data Corruption	No (safe)	✔ Yes (unsafe in multi-threading)
Suitable for	Multi-threaded environments	Single-threaded environments
Memory Usage	Slightly high (locking overhead)	Low

Examples (Java Classes)	Hashtable, Vector, StringBuffer	HashMap, ArrayList, StringBuilder
Locking Mechanism	Uses "lock" or "monitor"	No locking
Performance	Low performance	High performance
Use Case	Banking, ticket booking, financial transactions	Web apps, automation scripts, general programming
Failure When Multi-Threaded?	No	Yes (race condition)

StringBuffer and StringBuilder

What is String in Java (Quick Reminder) String is **immutable** → cannot be changed once created. So Java gives two mutable (changeable) classes:

StringBuffer
StringBuilder

STRINGBUFFER
✔ Mutable (can be changed)
✔ Thread-safe (all methods synchronized)
✔ Slower
✔ Safe in multithreading
✔ Uses char[] array internally
✔ Default capacity = 16

StringBuffer is a **mutable**, **synchronized**, **thread-safe** class used to create and modify strings without creating new objects.

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb); // Hello World
```

Why Slow?
Because every method is **synchronized** → LOCKING.

STRINGBUILDER

Mutable
Not synchronized
Not thread-safe
Fastest
Best for single-threaded applications
Uses char[] array internally
Default capacity = 16

StringBuilder is a **mutable**, **non-synchronized**, **fast** class used to build or modify strings in single-threaded environments.

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" Java");
System.out.println(sb); // Hello Java
```

Real-Time Use Cases

StringBuffer (Thread-safe)
Used in:

- Banking systems
- Transaction logs

- Multi-threaded applications
- When multiple threads modify same string

StringBuilder

Used in:

- Automation frameworks
- API response building
- Dynamic SQL building
- Creating large strings
- JSON creation
- String concatenation in loops

```
public class Demo {
    public static void main(String[] args) {

        StringBuffer sb1 = new StringBuffer("Hello");
        sb1.append(" World");
        System.out.println(sb1);

        StringBuilder sb2 = new StringBuilder("Java");
        sb2.append(" Programming");
        System.out.println(sb2);
    }
}
```

Feature	StringBuffer	StringBuilder
Mutability	Mutable	Mutable
Synchronized	Yes	No
Thread-safe	Yes	No
Performance	Slow	Fast
Best Use Case	Multi-threaded apps	Single-threaded apps
Introduced	Java 1.0	Java 1.5
Methods	Same	Same
Internal Structure	char[]	char[]

Q: Why String is immutable but StringBuilder/StringBuffer are mutable?

String:

- Stores values in **String Pool**
- Immutable for security + caching + thread safety

StringBuilder / StringBuffer:

- Use expandable arrays
- Designed for modification → so mutable

String vs StringBuffer vs StringBuilder

Feature	String	StringBuffer	StringBuilder
Mutability	Immutable	Mutable	Mutable
Thread-safe	Yes (immutable)	Yes (synchronized)	No
Synchronized	No need	All methods synchronized	Not synchronized
Performance	Slow (new object on change)	Moderate (locking overhead)	Fast (no locking)
Memory Usage	High (new object every modification)	Low (same object modified)	Low (same object modified)
Use Case	Constant text, secure data	Multi-threaded environment	Single-threaded, high-performance
Null Allowed	Yes	Yes	Yes
Methods	substring, concat, replace	append, insert, delete, reverse	Same as StringBuffer
Package	java.lang	java.lang	java.lang
Introduced In	Java 1.0	Java 1.0	Java 1.5
Default Capacity	Not applicable	16	16
Suitable For	Fixed data, keys, messages	Banking apps, loggers	String concatenation, fast apps