SOHAMGLOBAL

JAVA COLLECTIONS
FRAMEWORK

DSA INTERVIEW SERIES

73919 66656

Training & Projects

AI READY COMPANIES

sohamglobal.com
spiderprojectsone.com

Microsoft Partner aws partner network

SHARAYU-PRAFFULL
RESEARCH & TRAINING
20+ Years of Java Experience

Covers topics & questions asked in the interviews for a Java Developer profile from freshers to experienced.

LIVE online on Microsoft Teams

Collections Framework & Data Structures in Java

2025 Edition

{ SohamGlobal }

Java: Powering Innovation for 30 Years!
The Technology Behind our Digital Lifestyle!

Research: Sharayu | Presentation : Praffull | Projects : Megha

Instagram: sohamglobal.praffull

Technology is Power. Technology is Future.

What are Data Structures?

A **data structure** is a specialized way of organizing, storing, and managing data in a computer to perform operations efficiently. It provides a systematic way to handle and process data, making it easier to access, modify, and retrieve when needed.

Types of Data Structures

1. Linear Data Structures

- Data is arranged in a sequential manner.
- Examples:
  - **Array** – Collection of elements stored in contiguous memory locations.
  - **Linked List** – Collection of nodes, where each node contains data and a reference to the next node.
  - **Stack** – Follows the LIFO (Last In, First Out) principle.
  - **Queue** – Follows the FIFO (First In, First Out) principle.

2. Non-Linear Data Structures

- Data is arranged in a hierarchical or interconnected manner.
- Examples:
  - **Tree** – Hierarchical structure with a root node, parent-child relationships. (e.g., Binary Tree, Binary Search Tree)
  - **Graph** – A collection of nodes (vertices) connected by edges. Used in networks, social media connections, etc.

3. Hash-Based Data Structures

- Uses a **hash function** to map keys to values.
- Example: **Hash Table** – Stores data in key-value pairs for fast lookups.

Importance of Data Structures

✔ Efficient data management
✔ Faster algorithm execution
✔ Reduces time and space complexity
✔ Used in real-world applications like databases, networking, AI, and operating systems

What is the collections framework in Java?

The **Java Collections Framework (JCF)** is a unified architecture for storing, managing, and processing groups of objects efficiently. It provides **ready-to-use data structures and algorithms**, making it easier to handle data in Java programs.

Key Components of the Java Collections Framework

1. Interfaces (Abstract Data Types)

The core interfaces define the types of collections in Java:

- `Collection` – The root interface for all collection classes.
- `List` – Ordered collection allowing duplicate elements. *(e.g., ArrayList, LinkedList)*
- `Set` – Unordered collection that doesn't allow duplicates. *(e.g., HashSet, TreeSet)*
- `Queue` – Follows FIFO (First In, First Out). *(e.g., PriorityQueue, LinkedList)*
- `Map` – Stores key-value pairs. *(e.g., HashMap, TreeMap, LinkedHashMap)*

---

2. Implementations (Classes)

The framework provides concrete implementations of the interfaces:

Interface	Implementation Classes	Description
List	`ArrayList`, `LinkedList`, `Vector`, `Stack`	Ordered, allows duplicates
Set	`HashSet`, `LinkedHashSet`, `TreeSet`	Unique elements only
Queue	`PriorityQueue`, `ArrayDeque`	FIFO order, priority-based retrieval
Map	`HashMap`, `TreeMap`, `LinkedHashMap`, `Hashtable`	Stores key-value pairs

3. Utility Classes

The `Collections` and `Arrays` classes provide utility methods to perform operations like sorting, searching, and shuffling.

Why is the collections framework so important in java for someone who wants to build a Java developer profile?

The **Collections Framework** is a **crucial** part of Java programming, especially for anyone aiming to build a **strong Java developer profile**. Here's why:

1️⃣ Essential for Data Handling

- In real-world applications, you need to **store, retrieve, and manipulate data efficiently**.
- The Collections Framework provides **ready-to-use, optimized** data structures like **List, Set, Queue, and Map**.

2️⃣ Improves Code Efficiency

- Collections like **ArrayList, HashMap, HashSet, and LinkedList** offer efficient ways to **store and process large datasets**.
- These implementations are optimized for **speed, memory, and performance**.

3️⃣ Reduces Boilerplate Code

- Without collections, developers would need to **write custom data structures**, making code lengthy and error-prone.
- With collections, complex operations like **sorting, searching, and iteration** are simplified.

4️⃣ Enhances Problem-Solving Skills

- Many Java **interview questions** revolve around collections (e.g., **difference between HashMap and HashTable**).
- Understanding **how different collections work** helps in writing optimized code for **data-heavy applications**.

5️⃣ Supports Multithreading & Concurrency

- Java provides **Concurrent Collections** like **ConcurrentHashMap, CopyOnWriteArrayList**, which help in **multithreaded environments**.

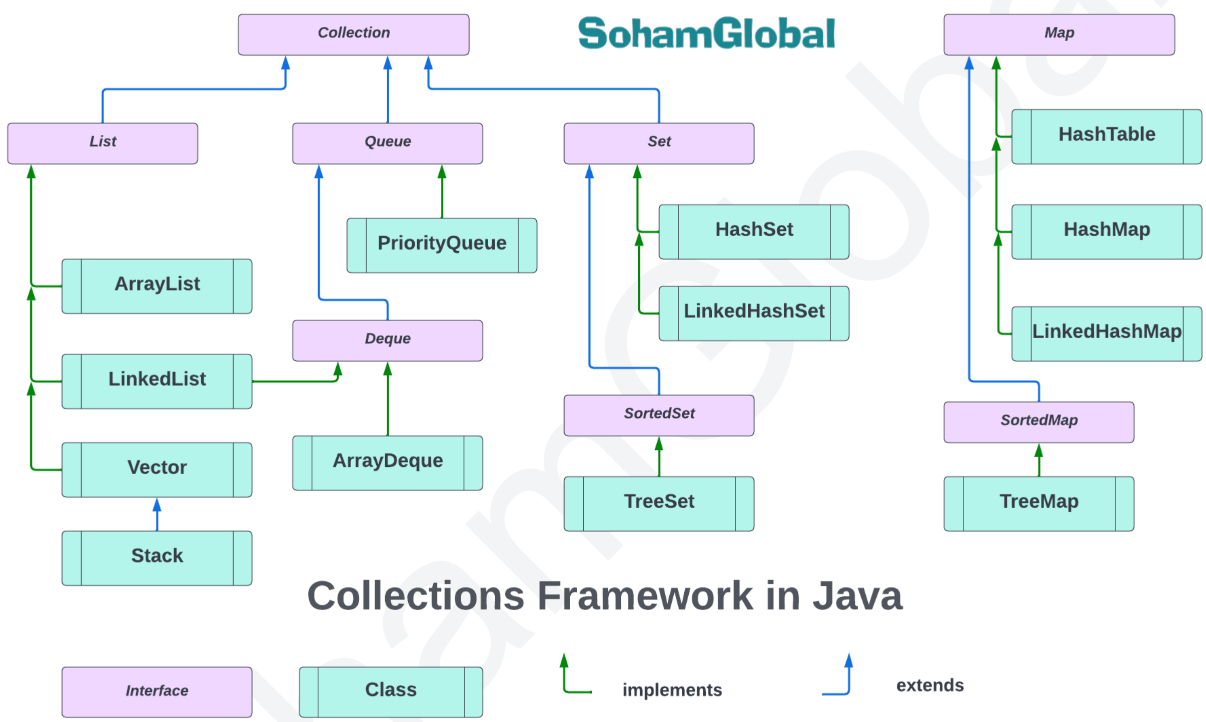
6️⃣ Widely Used in Frameworks & APIs

- Collections are used in **Spring, Hibernate, REST APIs, and Microservices**.
- Knowledge of collections is required to work with **real-world projects**.

7️⃣ Backbone for Competitive Coding & System Design

- Strong knowledge of **Lists, Maps, and Queues** helps in **coding competitions** and **system design interviews**.
- **Algorithmic Knowledge:** Collections tie into algorithms and data structures, so a strong grasp of them demonstrates proficiency in core computer science concepts.

Collections Framework Structure



What are the key components of the Collections Framework?

The key components of the Collections Framework include:

- **Interfaces:** The framework defines a hierarchy of interfaces, such as `Collection`, `List`, `Set`, `Map`, etc. These interfaces provide a common set of methods that any collection class must implement. For example, the `List` interface defines methods for working with ordered collections of elements, and the `Set` interface defines methods for working with collections that do not allow duplicate elements.

- **Classes:** Java provides several concrete classes that implement the collection interfaces. Some common classes include `ArrayList`, `LinkedList`, `HashSet`, `HashMap`, and many more. These classes provide different implementations of collections, allowing you to choose the one that best suits your needs.
- **Algorithms:** The Collections Framework includes a wide range of utility methods and algorithms for common operations on collections, such as sorting, searching, and filtering. These algorithms are provided as static methods in the `Collections` and `Arrays` classes.
- **Iterators:** Iterators are used to traverse the elements of a collection one by one. The framework defines the `Iterator` interface, which is implemented by collection classes to provide a way to iterate over their elements.
- **Comparator and Comparable:** These interfaces allow you to specify custom ordering for elements in a collection. `Comparator` is used to define an external comparison logic, while `Comparable` is implemented by elements to define their natural ordering.

By providing a standard framework for working with collections, Java makes it easier for developers to write code that is more efficient, maintainable, and reusable. It also ensures that collections are consistent and reliable across different parts of the Java ecosystem, which is especially useful in large and complex software projects.

What does it mean by synchronized and non synchronized?

Synchronized vs. Non-Synchronized in Java

In Java, **synchronization** refers to the mechanism that controls access to shared resources in a **multi-threaded** environment to prevent race conditions.

---

1. Synchronized (Thread-Safe)

- **Only one thread can access a resource at a time.**
- Used to avoid **data inconsistency and race conditions**.
- Slower performance due to thread locking.

✔ Ensures thread safety but **reduces performance** due to synchronization overhead.

---

2. Non-Synchronized (Not Thread-Safe)

- Multiple threads can access the resource **simultaneously**.
- Faster performance but **prone to data corruption** if multiple threads modify data.

✔ Faster but **unsafe in multi-threaded environments**.

Key Differences

Feature	Synchronized	Non-Synchronized
Thread Safety	Yes	No
Performance	Slower	Faster
Example Classes	`Vector`, `Hashtable`, `Collections.synchronizedList()`	`ArrayList`, `HashMap`, `LinkedList`

Which One to Use?

- **Single-threaded applications?** → Use **non-synchronized** collections for better performance.
- **Multi-threaded applications?** → Use **synchronized** collections to avoid concurrency issues.

What is Thread Safety in Java?

Thread safety means that a piece of code or a data structure **can be safely accessed and modified by multiple threads without leading to race conditions or inconsistent results**.

When multiple threads execute **concurrently**, they may try to read and write shared resources simultaneously, leading to **unexpected behavior** or **data corruption**. A thread-safe implementation ensures that only **one thread** modifies a shared resource at a time, preventing conflicts.

How to Achieve Thread Safety in Java?

There are several ways to make a program thread-safe:

1. Synchronization (`synchronized` keyword)

- **Locks a resource** so that only one thread can access it at a time.
- Used in methods or blocks.

2. Using Thread-Safe Classes (Collections & Utilities)

- Java provides built-in **thread-safe** collections and utilities.

Thread-Safe	Not Thread-Safe
`Vector`	`ArrayList`
`Hashtable`	`HashMap`
`ConcurrentHashMap`	`HashMap`
`CopyOnWriteArrayList`	`ArrayList`

3. Using `volatile` Keyword

- Ensures that a variable's **latest value is always read from memory**, preventing caching issues.

4. Using `Lock` API (More Control)

- More flexible than `synchronized`, supports **tryLock()**, **fairness policies**, etc.

Why is Thread Safety Important?

✔ Prevents **race conditions**
✔ Ensures **data consistency**
✔ Avoids **unexpected behavior** in concurrent applications

Thread-safe code ensures proper handling of shared resources.
Non-thread-safe code can lead to **race conditions** and **unexpected behavior** in multi-threaded programs.

Choose **thread safety methods** based on performance needs.

What is fail-fast and fail-safe?

In Java, **fail-fast** and **fail-safe** refer to the behavior of iterators when a collection is modified **during iteration**.

1. Fail-Fast Iterators

- **Throw `ConcurrentModificationException` if a collection is modified while iterating.**
- Designed to **detect modifications early** to prevent inconsistent behavior.
- Uses **modCount** (modification count) to track changes.
- Examples:
    - `ArrayList`
    - `HashMap`
    - `HashSet`

Issue: The iterator detects modification and **fails immediately**.

2. Fail-Safe Iterators

- **Do not throw `ConcurrentModificationException` when the collection is modified.**
- Work on a **copy of the collection** instead of the original one.
- Changes made to the collection **won't be reflected** in the iterator.
- Examples:
    - `CopyOnWriteArrayList`
    - `ConcurrentHashMap`

No Exception! The iterator works on a separate copy of the list.

Key Differences Between Fail-Fast and Fail-Safe

Feature	Fail-Fast	Fail-Safe
Behavior	Throws `ConcurrentModificationException` if modified	Allows modifications during iteration
Works On	Original collection	A cloned copy of the collection
Thread Safety	Not thread-safe	Thread-safe
Performance	Faster (direct access)	Slower (copy overhead)
Examples	`ArrayList`, `HashMap`, `HashSet`	`CopyOnWriteArrayList`, `ConcurrentHashMap`

Which One to Use?

- **Use Fail-Fast** for single-threaded applications where consistency is critical.
- **Use Fail-Safe** for multi-threaded environments where modifications must be allowed during iteration.

List Interface

Tell me brief information, behavior and functions of the List interface.

List Interface in Java

1. Brief Information

The **`List`** interface in Java is a part of the **Java Collections Framework (JCF)** and extends the `Collection` interface. It represents an **ordered collection (sequence) of elements**, where **duplicates are allowed**, and elements can be accessed by their **index**.

---

2. Behavior of List Interface

✔ **Ordered Collection** – Maintains the order in which elements are inserted.
✔ **Allows Duplicates** – Unlike `Set`, `List` allows multiple occurrences of the same element.
✔ **Indexed Access** – Elements can be retrieved, updated, or removed using an index.
✔ **Can Contain Null Values** – Depending on the implementation.

3. Functions (Methods) of List Interface

The `List` interface provides various methods to manipulate elements:

Method	Description
`void add(E e)`	Adds an element at the end of the list.
`void add(int index, E e)`	Inserts an element at a specific position.
`E get(int index)`	Returns the element at the specified index.
`E set(int index, E element)`	Replaces the element at a given index.
`E remove(int index)`	Removes and returns the element at a specified index.
`boolean remove(Object o)`	Removes the first occurrence of the specified element.
`int size()`	Returns the number of elements in the list.
`int indexOf(Object o)`	Returns the index of the first occurrence of an element.
`int lastIndexOf(Object o)`	Returns the last occurrence index of an element.
`boolean contains(Object o)`	Checks if the list contains a specified element.
List<E> subList(int from, int to)	Return[illegible]ublist between the specified indices.

4. Implementations of List Interface

The `List` interface is implemented by the following classes:

Implementation	Description
`ArrayList`	Dynamic array, fast read access, slow insert/delete.
`LinkedList`	Doubly linked list, efficient insert/delete, slower random access.
`Vector`	Synchronized version of `ArrayList`, thread-safe.
`Stack`	LIFO (Last-In, First-Out) structure, extends `Vector`.

When to Use the List Interface?

✔ When you need an **ordered collection** with **duplicate elements**.
✔ When **random access or indexed operations** are required.
✔ When frequent **insertions or deletions** are needed (use `LinkedList`).

1. ArrayList

What is an ArrayList?

Introduction to `ArrayList`

`ArrayList` is a part of the **Java Collections Framework (JCF)** and is present in the `java.util` package. It provides a **dynamic array** that can grow and shrink as needed. Unlike a regular array, `ArrayList` automatically **resizes itself** when elements are added or removed.

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable

What are the characteristics of ArrayList?

Characteristics and Behavior of `ArrayList`

✔ **Dynamic Resizing:** It expands automatically when more elements are added.
✔ **Ordered Collection:** Maintains the **insertion order** of elements.
✔ **Allows Duplicates:** Duplicate elements can be stored.
✔ **Indexed Access:** Elements can be accessed directly using an **index**.

✔ **Allows `null` Values:** `null` elements can be stored in an `ArrayList`.
✔ **Not Thread-Safe:** It is **not synchronized** by default, meaning multiple threads modifying it concurrently can lead to issues.
✔ **Performance Considerations:**

- **Fast read access** (O(1) time complexity for `get(int index)`)
- **Slow insertions/removals** (O(n) time complexity for `add(index, element)` and `remove(index)` if elements need to be shifted)

How is the internal working of an ArrayList?

How Elements are Stored?

Internally, `ArrayList` uses an **array** (`Object[] elementData`) to store elements.

- **Initial Capacity:** Default capacity is **10** if created without specifying a size.
- **Resizing Mechanism:** When the array is full, `ArrayList` increases its capacity by **1.5 times** the current size.

Growth Mechanism Example

Initial Capacity	Add 11th Element	New Capacity (1.5x Rule)
10	Add new element	`10 + (10/2) = 15`
15	Add new element	`15 + (15/2) = 22`
22	Add new element	`22 + (22/2) = 33`

How ArrayList Handles Resizing?

Step-by-Step Working of `add(E e)`

1. Checks if there is enough space in `elementData[]`.
2. If the array is full, **a new array is created** with `1.5 * old size`.
3. The existing elements are **copied** to the new array.
4. The new element is added at the end.

How to Make ArrayList Thread-Safe?

Since `ArrayList` is **not thread-safe**, we can synchronize it using **`CopyOnWriteArrayList`** for better performance.

When to Use ArrayList?

✔ **Fast random access is needed (O(1))**
✔ **Data is mostly read-heavy (less insert/delete)**
✔ **Memory-efficient structure is required**

🚫 **Avoid `ArrayList` when:**

- Frequent **insertions/removals** are needed → Use `LinkedList`
- **Thread safety** is required → Use `Vector` or `CopyOnWriteArrayList`

✅ `ArrayList` is a great choice for **dynamic storage** with **fast retrieval**.
✅ Understand its **resizing mechanism** to optimize performance.
✅ Use **thread-safe alternatives** when working in multi-threaded environments.

ArrayList: Points to remember

- It is found in the java.util package
- Uses a dynamic array for storing the elements
- There is no size limit
- Dynamic- We can add, insert or remove elements anytime
- The ArrayList maintains the insertion order internally
- The ArrayList in Java can have the duplicate elements
- NULL values are allowed
- ArrayList class is non synchronized
- Heterogeneous objects are allowed.
- ArrayList is initialized by size. However, the size is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
- It is much more flexible than the traditional array

- It inherits the AbstractList class and implements List interface
- It allows random access because the array works on an index basis
- Manipulation is slower due to shifting of elements
- ArrayList of the primitive types can't be created, Wrapper classes are required with ArrayList in such cases
- Iterator and ListIterators available.
- Only List collections support ListIterator (next and previous access).

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;

public class ArrayListOperations {
    public static void main(String[] args) {
        // 1. Creating an ArrayList and adding elements
        ArrayList<String> names = new ArrayList<>();
        names.add("Soham");
        names.add("Praffull");
        names.add("Sharayu");
        names.add("Shailaja");
        names.add("Megha");
        names.add("Owee");

        System.out.println("Original List: " + names);

        // 2. Inserting an element at a specific index
        names.add(2, "Aarya");
        System.out.println("After inserting 'Aarya' at index 2: " + names);

        // 3. Accessing an element
        System.out.println("Element at index 3: " + names.get(3));

        // 4. Modifying an element
        names.set(4, "Shabana");
```

```
        System.out.println("After replacing index 4 with 'Shabana': " +
names);

        // 5. Removing an element by index
        names.remove(5);
        System.out.println("After removing element at index 5: " + names);

        // 6. Removing an element by value
        names.remove("Aarya");
        System.out.println("After removing 'Aarya': " + names);

        // 7. Checking if an element exists
        System.out.println("Does the list contain 'Megha'? " +
names.contains("Megha"));

        // 8. Finding index of an element
        System.out.println("Index of 'Sharayu': " +
names.indexOf("Sharayu"));

        // 9. Iterating using a for loop
        System.out.println("\nIterating using for loop:");
        for (int i = 0; i < names.size(); i++) {
            System.out.println(names.get(i));
        }

        // 10. Iterating using for-each loop
        System.out.println("\nIterating using for-each loop:");
        for (String name : names) {
            System.out.println(name);
        }

        // 11. Iterating using an Iterator
        System.out.println("\nIterating using Iterator:");
        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
```

```java
        // 12. Sorting the ArrayList
        Collections.sort(names);
        System.out.println("\nSorted List: " + names);

        // 13. Reversing the ArrayList
        Collections.reverse(names);
        System.out.println("Reversed List: " + names);

        // 14. Checking size of the ArrayList
        System.out.println("Size of the list: " + names.size());

        // 15. Converting ArrayList to an array
        String[] array = names.toArray(new String[0]);
        System.out.println("Array elements: ");
        for (String s : array) {
            System.out.print(s + " ");
        }
        System.out.println();

        // 16. Clearing the list
        names.clear();
        System.out.println("After clearing the list: " + names);
    }
}
```

{SohamGlobal & Spider Projects One}

2. CopyOnWriteArrayList

What is CopyOnWriteArrayList?

`CopyOnWriteArrayList` is a **thread-safe** variant of `ArrayList`, introduced in **Java 5** (`java.util.concurrent` package). It allows **safe concurrent reads and writes** by creating a **new copy** of the underlying array for every write operation.

What are the features of CopyOnWriteArrayList?

✔ Thread-safe for concurrent access
✔ Iterators do not throw `ConcurrentModificationException`
✔ Good for read-heavy operations
✔ Slower write operations (as it creates a copy on modification)

Feature	Description
Thread-Safety	✅ Uses a **copy-on-write** mechanism for modifications.
Performance	⚡ **Fast Reads** (O(1)), **Slow Writes** (O(n)) due to copying.
Modification Strategy	📌 On every write operation (`add()` , `set()` , `remove()`), a new copy of the array is created.
Null Values Allowed?	✅ Yes.
Fail-Safe Iterators?	✅ Yes, as iterators work on a snapshot of the list and **don't throw** `ConcurrentModificationException` .

Explain Internal Working of CopyOnWriteArrayList

- **Read Operations (`get()`)** → **No Locking**, fast access (`O(1)`).
- **Write Operations (`add()`, `set()`, `remove()`)** → **Creates a new copy** of the list (`O(n)`).
- **Iterators operate on a snapshot** → No risk of `ConcurrentModificationException`.

🔹 How CopyOnWrite Works?

- When a **modification occurs**, a **new array is created** with the updated data.
- The old array remains **unchanged** for ongoing **read operations**.
- Once the new copy is ready, the reference is **updated atomically**.

🚀 **This ensures thread-safety without explicit synchronization!**

When to Use CopyOnWriteArrayList?

✔ **Best for Read-Mostly Operations** → Many reads, few writes.
✔ **Ideal for Multi-threaded Scenarios** → Prevents `ConcurrentModificationException`.
❌ **Not Suitable for Frequent Updates** → High memory & time cost (`O(n)`).

```
package com.sharayu.programs;

import java.util.Iterator;
import java.util.concurrent.CopyOnWriteArrayList;

public class FailSafeCopyOnWriteArrayList {
    public static void main(String[] args) {
        CopyOnWriteArrayList<String> list=new CopyOnWriteArrayList<String>();
        list.add("chelsea");
        list.add("liverpool");
        list.add("tottenham");
        list.add("arsenal");
        list.add("manchester city");
        list.add("crystal palace");

        Iterator<String> iterator=list.iterator();
        while(iterator.hasNext())
        {
            System.out.println(iterator.next());

            if(!list.contains("newcastle"))
            list.add("newcastle");
            //no error
            //fail safe
        }

        System.out.println("\nupdated list - ");
        System.out.println(list);

    }
}
```

3. LinkedList

What is LinkedList?

`LinkedList` is a part of the **Java Collections Framework (JCF)** that implements the **List** and **Deque** interfaces. It is a **doubly linked list**, meaning each node has a **reference to both its previous and next node**.

public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable

What are the characteristics of LinkedList?

✔ **Dynamically Sized** – Unlike `ArrayList`, it doesn't need resizing.
✔ **Efficient Insertions/Deletions** – No need to shift elements like in `ArrayList`.
✔ **Slower Random Access** – Accessing an element requires **traversing nodes** (O(n)).
✔ **Maintains Order** – Stores elements in **insertion order**.
✔ **Allows Duplicates & Nulls** – Can store duplicate elements and `null` values.
✔ **Not Thread-Safe** – Requires external synchronization for multi-threading.

How is the internal representation of a LinkedList?

`LinkedList` uses **nodes** (objects of `Node<E>` class) to store elements.
Each **node** contains:

- **Data (E item)** – The actual element.
- **Next reference (Node<E> next)** – Pointer to the next node.
- **Previous reference (Node<E> prev)** – Pointer to the previous node.

Internal Representation

HEAD → [prev | Data: 10 | next] → [prev | Data: 20 | next] → [prev | Data: 30 | next] → NULL

- `prev` of the first node is `null` (head).
- `next` of the last node is `null` (tail).
- Traversal happens in both directions (forward & backward).

Explain the behavior of LinkedList.

Behavior of `LinkedList`

✔ **Faster insertions & deletions** – O(1) time complexity if at head/tail.
✔ **Slower search operations** – O(n) complexity (linear traversal required).
✔ **More memory usage** – Each node stores two references (prev & next).
✔ **Supports Queue Operations** – Implements **Deque**, so it can be used as **Queue/Stack**.

How LinkedList Works Internally?

Insertion (`add(E e)`)

1. If empty, a new **head node** is created.
2. If adding at the end:
   - New node is linked to the **previous last node**.
   - **Tail pointer is updated**.

Deletion (`remove(E e)`)

1. If an element is in the **middle**, `prev.next` and `next.prev` are updated.
2. If removing **first/last**, `head` or `tail` is updated.

Explain the difference between ArrayList and LinkedList.

Feature	`ArrayList`	`LinkedList`
Storage	Dynamic **resizable array**	**Doubly linked list**
Access Time (`get(index)`)	**O(1)** (Direct access)	**O(n)** (Sequential search)
Insertion/Deletion	**O(n)** (Shifting required)	**O(1)** at head/tail
Memory Usage	Less (stores only data)	More (stores `prev` & `next` references)
Best for	**Read-heavy operations**	**Insert/delete-heavy operations**

{SohamGlobal & Spider Projects One}

How to Make LinkedList Thread-Safe?

Since `LinkedList` is **not synchronized**, use `ConcurrentLinkedDeque` for High Performance.

When to Use LinkedList?

✔ **Frequent insertions/deletions at head/tail** (O(1) performance).
✔ **Memory is not a constraint** (Extra space for pointers).
✔ **Queue/Deque operations are required** (`LinkedList` implements `Deque`).

🚫 **Avoid `LinkedList` when:**

- **Frequent random access (`get(index)`) is needed** → Use `ArrayList`
- **Memory efficiency is a priority** → Uses more memory than `ArrayList`

✅ `LinkedList` is **best suited** for **dynamic data structures** with **frequent insertions/removals**.
✅ It **trades off fast access speed** for **better insert/delete efficiency**.
✅ Use **thread-safe alternatives** in **multi-threaded environments**.

LinkedList: Points to remember

- LinkedList class uses a doubly linked list to store the elements
- It provides a linked-list data structure
- Each element is known as a node.
- The elements are not stored in a continuous fashion therefore, there is no need to increase the size.
- It inherits the AbstractList class and implements List and Deque interfaces.
- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In LinkedList, manipulation is fast because no element shifting is required
- LinkedList class can be used as a list, stack or queue.
- In the case of a doubly linked list, we can add or remove elements from both sides.

```
import java.util.LinkedList;
import java.util.Collections;
import java.util.Iterator;

public class LinkedListOperations {
    public static void main(String[] args) {
        // 1. Creating a LinkedList and adding elements
        LinkedList<String> names = new LinkedList<>();
        names.add("Soham");
        names.add("Praffull");
        names.add("Sharayu");
        names.add("Shailaja");
        names.add("Megha");
        names.add("Owee");

        System.out.println("Original List: " + names);

        // 2. Adding elements at first and last position
        names.addFirst("FirstName");
        names.addLast("LastName");
        System.out.println("After adding First and Last: " + names);

        // 3. Inserting at a specific index
        names.add(3, "Aarya");
        System.out.println("After inserting 'Aarya' at index 3: " + names);

        // 4. Accessing elements
        System.out.println("First element: " + names.getFirst());
        System.out.println("Last element: " + names.getLast());
        System.out.println("Element at index 4: " + names.get(4));

        // 5. Modifying elements
        names.set(3, "Shabana");
        System.out.println("After replacing index 3 with 'Shabana': " +
names);

        // 6. Removing elements
```

```
        names.removeFirst();
        names.removeLast();
        System.out.println("After removing first and last element: " +
names);

        // 7. Removing by index and value
        names.remove(4);
        System.out.println("After removing element at index 4: " + names);
        names.remove("Aarya"); // No effect as 'Aarya' was replaced earlier
        System.out.println("After removing 'Aarya': " + names);

        // 8. Checking if an element exists
        System.out.println("Does the list contain 'Megha'? " +
names.contains("Megha"));

        // 9. Finding index of an element
        System.out.println("Index of 'Sharayu': " +
names.indexOf("Sharayu"));

        // 10. Iterating using a for loop
        System.out.println("\nIterating using for loop:");
        for (int i = 0; i < names.size(); i++) {
            System.out.println(names.get(i));
        }

        // 11. Iterating using for-each loop
        System.out.println("\nIterating using for-each loop:");
        for (String name : names) {
            System.out.println(name);
        }

        // 12. Iterating using an Iterator
        System.out.println("\nIterating using Iterator:");
        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
```

```java
        // 13. Sorting the LinkedList
        Collections.sort(names);
        System.out.println("\nSorted List: " + names);

        // 14. Reversing the LinkedList
        Collections.reverse(names);
        System.out.println("Reversed List: " + names);

        // 15. Checking the size of the LinkedList
        System.out.println("Size of the list: " + names.size());

        // 16. Converting LinkedList to an array
        String[] array = names.toArray(new String[0]);
        System.out.println("Array elements: ");
        for (String s : array) {
            System.out.print(s + " ");
        }
        System.out.println();

        // 17. Clearing the LinkedList
        names.clear();
        System.out.println("After clearing the list: " + names);
    }
}
```

4. Vector

What is a Vector?

`Vector` is a **resizable array-based** collection in **Java**, found in `java.util` package. It implements the **List interface**, allowing **ordered storage** of elements.

public class Vector<E> extends Object<E> implements List<E>, Cloneable, Serializable

What are the key characteristics of a Vector?

✔ **Synchronized (Thread-Safe)** – Unlike `ArrayList`, `Vector` is **synchronized** for thread safety.
✔ **Allows Duplicates** – Stores duplicate elements.
✔ **Maintains Insertion Order** – Elements remain in the order they were added.
✔ **Allows Null Values** – Can store `null` elements.
✔ **Slower than `ArrayList`** – Due to synchronization overhead.

How Vector Works Internally?

`Vector` is similar to `ArrayList` but **thread-safe**. Internally, it uses a **dynamic array** (**`Object[] elementData`**) that increases in size when needed.

Growth Mechanism (`Vector` vs. `ArrayList`)

Feature	`ArrayList`	`Vector`
Default Capacity	10	10
Resizing	Grows by **1.5x**	Grows by **2x**

{SohamGlobal & Spider Projects One}

What is the difference between Vector and ArrayList?

Feature	`ArrayList`	`Vector`
Synchronization	❌ **Not synchronized** (Not thread-safe)	✅ **Synchronized** (Thread-safe)
Performance	⚡ **Faster** (No synchronization overhead)	🐢 **Slower** (Synchronization overhead)
Growth Mechanism	Increases by **50% (1.5x)** when full	Increases by **100% (2x)** when full
Usage Scenario	✅ **Single-threaded applications**	✅ **Multi-threaded applications**
Iteration	❌ **Not thread-safe** without external synchronization	✅ **Thread-safe** iteration
Legacy Support	Part of **Java 1.2** (Preferred for modern applications)	Introduced in **Java 1.0** (Legacy)
Performance in Multi-threading	🚀 **Better with** `Collections.synchronizedList()`	🐌 **Built-in synchronization (slower)**

Performance Comparison

Adding Elements (`add(E e)`)

- `ArrayList` is **faster** since it doesn't have synchronization overhead.
- `Vector` synchronizes each method, making it **slower**.

Retrieving Elements (`get(int index)`)

- **Both are O(1)** (direct array access).
- But `ArrayList` is slightly faster due to **no synchronization overhead**.

Removing Elements (`remove(int index)`)

- **O(n) in both cases**, but `ArrayList` performs better **due to no locking**.

When to Use a Vector?

✔ **Multi-threaded environments (Thread Safety)**
✔ **Large dynamic collections needing fast random access (`get(index)`)**
✔ **When an array-based structure is preferred over linked lists**

🚫 **Avoid `Vector` when:**

- **Single-threaded application** → Use `ArrayList` (better performance)
- **Frequent insertions/removals** → Use `LinkedList`

✅ `Vector` is a **thread-safe alternative** to `ArrayList`.
✅ It is **slower** due to **synchronization overhead**.
✅ Use **`ArrayList`** for **single-threaded** applications and **`Vector`** only when thread safety is required.

Vector: Points to remember

- Vector is like the dynamic array which can grow or shrink its size
- We can store number of elements in it as there is no size limit
- Old class from Java 1.2 (Legacy class)
- implements the List interface
- It is recommended to use the Vector class in the thread-safe implementation only
- The Iterators returned by the Vector class are fail-fast
- Vector is synchronized.
- Java Vector contains many legacy methods that are not part of a collections framework.
- Iterators are not used instead Enumerations are used

```java
import java.util.Vector;
import java.util.Collections;
import java.util.Iterator;
import java.util.Enumeration;
```

```java
public class VectorOperations {
    public static void main(String[] args) {
        // 1. Creating a Vector and adding elements
        Vector<String> names = new Vector<>();
        names.add("Soham");
        names.add("Praffull");
        names.add("Sharayu");
        names.add("Shailaja");
        names.add("Megha");
        names.add("Owee");

        System.out.println("Original Vector: " + names);

        // 2. Adding elements at specific positions
        names.add(3, "Aarya");
        System.out.println("After inserting 'Aarya' at index 3: " + names);

        // 3. Accessing elements
        System.out.println("First element: " + names.firstElement());
        System.out.println("Last element: " + names.lastElement());
        System.out.println("Element at index 4: " + names.get(4));

        // 4. Modifying elements
        names.set(3, "Shabana");
        System.out.println("After replacing index 3 with 'Shabana': " +
names);

        // 5. Removing elements
        names.remove(4);
        System.out.println("After removing element at index 4: " + names);
        names.remove("Megha");
        System.out.println("After removing 'Megha': " + names);

        // 6. Checking if an element exists
        System.out.println("Does the vector contain 'Sharayu'? " +
names.contains("Sharayu"));
```

```
// 7. Finding index of an element
System.out.println("Index of 'Soham': " + names.indexOf("Soham"));

// 8. Checking size and capacity
System.out.println("Size of vector: " + names.size());
System.out.println("Capacity of vector: " + names.capacity());

// 9. Iterating using for loop
System.out.println("\nIterating using for loop:");
for (int i = 0; i < names.size(); i++) {
    System.out.println(names.get(i));
}

// 10. Iterating using for-each loop
System.out.println("\nIterating using for-each loop:");
for (String name : names) {
    System.out.println(name);
}

// 11. Iterating using Iterator
System.out.println("\nIterating using Iterator:");
Iterator<String> iterator = names.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}

// 12. Iterating using Enumeration
System.out.println("\nIterating using Enumeration:");
Enumeration<String> enumeration = names.elements();
while (enumeration.hasMoreElements()) {
    System.out.println(enumeration.nextElement());
}

// 13. Sorting the Vector
Collections.sort(names);
System.out.println("\nSorted Vector: " + names);
```

```java
        // 14. Reversing the Vector
        Collections.reverse(names);
        System.out.println("Reversed Vector: " + names);

        // 15. Converting Vector to an array
        String[] array = names.toArray(new String[0]);
        System.out.println("Array elements: ");
        for (String s : array) {
            System.out.print(s + " ");
        }
        System.out.println();

        // 16. Clearing the Vector
        names.clear();
        System.out.println("After clearing the vector: " + names);
    }
}
```

{SohamGlobal & Spider Projects One}

5. Stack

What is a Stack?

A **Stack** is a **Last In, First Out (LIFO)** data structure, meaning the **last element added is the first to be removed**.

What are the key features of a Stack?

✔ **Follows LIFO (Last In, First Out)**
✔ **Synchronized (Thread-Safe) in Java**
✔ **Allows null elements**
✔ **Uses `push()`, `pop()`, and `peek()` methods**

How Stack Works Internally?

Internally, `Stack` **extends `Vector`**, meaning:

- It uses a **dynamic array (`Object[] elementData`)** for storage.
- It grows dynamically (default capacity **10, then doubles when full**).
- It is **synchronized**, but slower than alternatives.

When to Use Stack?

✔ **Expression evaluation (e.g., postfix, infix, prefix)**
✔ **Undo/Redo functionality**
✔ **Backtracking algorithms (e.g., DFS in graphs)**
✔ **Call Stack in recursion**

✅ `Stack` is a **thread-safe LIFO structure** but is **slow** due to synchronization.
✅ `Deque (ArrayDeque)` is **preferred for modern applications** (faster).
✅ Use `Stack` when **legacy compatibility** or **synchronization** is required.

Stack: Points to remember

- The stack is a linear data structure that is used to store the collection of objects.
- It is based on Last-In-First-Out (LIFO).
- provides different operations such as push, pop, search, etc.
- The push operation inserts an element into the stack
- pop operation removes an element from the top of the stack.

```
import java.util.Stack;
import java.util.Collections;
import java.util.Iterator;

public class StackOperations {
    public static void main(String[] args) {
        // 1. Creating a Stack and pushing elements
        Stack<String> names = new Stack<>();
```

```
        names.push("Soham");
        names.push("Praffull");
        names.push("Sharayu");
        names.push("Shailaja");
        names.push("Megha");
        names.push("Owee");

        System.out.println("Original Stack: " + names);

        // 2. Peeking the top element
        System.out.println("Top element (peek): " + names.peek());

        // 3. Popping elements from the stack
        System.out.println("Popped element: " + names.pop());
        System.out.println("Stack after pop: " + names);

        // 4. Searching for an element (1-based index)
        System.out.println("Position of 'Soham' in Stack: " +
names.search("Soham"));
        System.out.println("Position of 'Megha' in Stack: " +
names.search("Megha"));

        // 5. Checking if stack is empty
        System.out.println("Is stack empty? " + names.isEmpty());

        // 6. Checking stack size
        System.out.println("Stack size: " + names.size());

        // 7. Iterating using for loop
        System.out.println("\nIterating using for loop:");
        for (int i = 0; i < names.size(); i++) {
            System.out.println(names.get(i));
        }

        // 8. Iterating using for-each loop
        System.out.println("\nIterating using for-each loop:");
        for (String name : names) {
```

```
            System.out.println(name);
        }

        // 9. Iterating using Iterator
        System.out.println("\nIterating using Iterator:");
        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // 10. Sorting the Stack
        Collections.sort(names);
        System.out.println("\nSorted Stack: " + names);

        // 11. Reversing the Stack
        Collections.reverse(names);
        System.out.println("Reversed Stack: " + names);

        // 12. Clearing the Stack
        names.clear();
        System.out.println("After clearing the stack: " + names);
    }
}
```

Queue Interface

What is a Queue in Java?

A **Queue** is a collection that follows the **First In, First Out (FIFO)** principle, meaning the **element added first is removed first**. It is commonly used for **task scheduling, buffering, and inter-thread communication**.

✔ **FIFO (First In, First Out) Order**
✔ **Supports insertion (`offer()`) and removal (`poll()`) operations**
✔ **Can have different implementations (`LinkedList`, `PriorityQueue`, `ArrayDeque`)**
✔ **Can be bounded (fixed size) or unbounded (dynamic growth)**

{SohamGlobal & Spider Projects One}

What are the important implementations of Queue interface?

Implementation	Description
`LinkedList<E>`	Implements `Queue` as a **doubly linked list** (not thread-safe).
`PriorityQueue<E>`	Elements are **sorted based on priority** instead of FIFO.
`ArrayDeque<E>`	A **faster alternative to** `Stack` **and** `LinkedList`, used as a queue or stack.
`BlockingQueue<E>`	**Thread-safe queues** used in multi-threaded applications (e.g., `LinkedBlockingQueue`, `ArrayBlockingQueue`).

What are the important methods of the Queue interface?

Method	Description	Behavior
`add(E e)`	Inserts element, throws exception if full	**Throws** `IllegalStateException` if queue is full
`offer(E e)`	Inserts element, returns `false` if full	**No exception, returns** `false` if full
`remove()`	Removes and returns head element	**Throws** `NoSuchElementException` if empty
`poll()`	Removes and returns head element	**Returns** `null` if empty
`element()`	Retrieves (but does not remove) the head	**Throws** `NoSuchElementException` if empty
`peek()`	Retrieves head without removal	**Returns** `null` if empty

Explain Working & Behavior of Queue

Insertion (`offer()` vs. `add()`)

- `offer(E e)`: Inserts the element **without exception** if full (returns `false`).
- `add(E e)`: Inserts the element but **throws `IllegalStateException`** if full.

Retrieval (`peek()` vs. `element()`)

- `peek()`: Returns head element **without removing** it. Returns `null` if empty.
- `element()`: Returns head element **without removing** it, **throws exception** if empty.

Removal (`poll()` vs. `remove()`)

- `poll()`: Removes the **head** element, returns `null` if empty.
- `remove()`: Removes the **head** element, **throws exception** if empty.

When to Use Queue?

✔ Task Scheduling (e.g., Printer Queue, OS Job Queue)
✔ Producer-Consumer Pattern (Multi-threading)
✔ Event Handling (e.g., Messaging Systems, BFS Traversal)
✔ Processing Requests in Order (e.g., Load Balancers, Web Servers)

🚫 **When NOT to use `Queue`**

- If **random access** is required (use `List` instead).
- If **LIFO behavior** is needed (use `Stack` or `Deque`).

✅ `Queue` is an **ordered collection following FIFO**.
✅ Use `PriorityQueue` for **priority-based processing**.
✅ Use `BlockingQueue` for **thread-safe queuing**.
✅ `Deque (ArrayDeque)` is **faster** than `LinkedList` for queue operations.

6. PriorityQueue

What is a PriorityQueue?

A `PriorityQueue` in Java is a **special type of queue** that orders elements based on **natural ordering** (for numbers, smallest comes first) or a **custom comparator**. It does **not** follow FIFO (First In, First Out); instead, elements with **higher priority are dequeued first**.

public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable

✅ **Elements are stored in sorted order**
✅ **Uses a Binary Heap internally**
✅ **By default, elements are ordered in ascending order (min-heap)**
✅ **Can use custom comparator for different sorting orders**

What are the Key Characteristics of PriorityQueue?

- **Unbounded Queue** – Grows dynamically, but can have an initial capacity.
- **Not Thread-Safe** – Use `PriorityBlockingQueue` for thread safety.
- **No Null Values** – Throws `NullPointerException` if `null` is inserted.
- **Duplicates Allowed** – Can store duplicate elements.

Explain Internal Working of PriorityQueue.

- **Uses a Binary Heap** (Min-Heap by default) for sorting.
- The **smallest element is always at the root**.
- **Heapify operation (`O(log n)`)** maintains order when inserting/removing elements.
- **Insertion (`offer()`) and deletion (`poll()`) take `O(log n)` time**.

Heap Representation of `PriorityQueue`

For elements: `{30, 20, 10, 50, 40}`

```
     10
    / \
  20   30
 / \
50   40
```

Min-Heap Property ensures **smallest element is at the top**.
When **polling**, `10` is removed, and heap adjusts (`O(log n)`).

How can we change the default Min-heap storage to Max-heap?

By using a Comparator in the constructor -
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(**Comparator.reverseOrder()**);

✅ Now the **largest element** is removed first (Max-Heap).

When to Use PriorityQueue?

✔ Task Scheduling (CPU scheduling, OS process queue)
✔ Dijkstra's Algorithm (Shortest Path Finding)
✔ Event Handling (Message Queue in Java)
✔ Load Balancing (Handling Priority-Based Requests)

What are the Limitations of PriorityQueue

🚫 **Not Thread-Safe** – Use PriorityBlockingQueue for multi-threading.
🚫 **No Direct Index Access** – Unlike ArrayList, you cannot access elements by index.
🚫 **Only Head Element is Sorted** – The full queue is **not** always fully sorted.

✅ PriorityQueue uses a **Binary Heap** for efficient priority-based processing.
✅ **By default, it orders elements in ascending order (Min-Heap)**.
✅ **Use a Comparator for custom sorting (e.g., Max-Heap, Object Sorting)**.
✅ **Not thread-safe** – Use PriorityBlockingQueue if needed.

Use Case: Job Scheduling in an Operating System

In modern operating systems, tasks (processes) are scheduled based on **priority**. Higher-priority tasks get executed first, while lower-priority tasks wait in the queue.

A **Priority Queue** is used in **CPU scheduling algorithms** such as **Shortest Job First (SJF)** or **Priority Scheduling**.

Scenario: CPU Task Scheduler

- Each task (process) has a **priority level**.
- The **highest priority task executes first**.
- If two tasks have the same priority, they execute in **arrival order**.

Where is PriorityQueue Used in Real Life?

✅ **Operating System Process Scheduling** – OS prioritizes system-critical processes.
✅ **Networking (Packet Scheduling)** – Internet traffic prioritization (VoIP calls vs. regular browsing).
✅ **Dijkstra's Algorithm** – Finding the **shortest path** in Google Maps & GPS.
✅ **Hospital Emergency System** – Critical patients get treated first.
✅ **Stock Market Order Processing** – Higher-priority trades execute first.

```java
import java.util.PriorityQueue;
import java.util.Iterator;
import java.util.Collections;
import java.util.ArrayList;

public class PriorityQueueOperations {
    public static void main(String[] args) {
        // 1. Creating a PriorityQueue and adding elements
        PriorityQueue<String> names = new PriorityQueue<>();
        names.add("Soham");
        names.add("Praffull");
        names.add("Sharayu");
        names.add("Shailaja");
        names.add("Megha");
        names.add("Owee");

        System.out.println("Original PriorityQueue: " + names); // The order
may not be insertion order

        // 2. Adding elements using offer()
        names.offer("Shabana");
        System.out.println("After adding 'Shabana' using offer(): " + names);

        // 3. Peeking the top (smallest) element
        System.out.println("Top element (peek): " + names.peek());

        // 4. Polling (removing the smallest element)
        System.out.println("Polled element: " + names.poll());
```

```
        System.out.println("PriorityQueue after poll: " + names);

        // 5. Removing a specific element
        names.remove("Megha");
        System.out.println("After removing 'Megha': " + names);

        // 6. Checking if an element exists
        System.out.println("Does the queue contain 'Sharayu'? " +
names.contains("Sharayu"));

        // 7. Checking the size of the queue
        System.out.println("Size of PriorityQueue: " + names.size());

        // 8. Iterating using for-each loop
        System.out.println("\nIterating using for-each loop:");
        for (String name : names) {
            System.out.println(name);
        }

        // 9. Iterating using Iterator
        System.out.println("\nIterating using Iterator:");
        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // 10. Sorting the PriorityQueue manually (since it does not allow
direct sorting)
        ArrayList<String> sortedList = new ArrayList<>(names);
        Collections.sort(sortedList);
        System.out.println("\nSorted PriorityQueue elements: " + sortedList);

        // 11. Clearing the PriorityQueue
        names.clear();
        System.out.println("After clearing the PriorityQueue: " + names);
    }
}
```

7. ArrayDeque

What is ArrayDeque?

`ArrayDeque` (Array Double-Ended Queue) is a **resizable** and **efficient** data structure in Java that allows elements to be added or removed from **both ends** (front and rear). It is implemented as a **growable array** and does not have the capacity restrictions of `ArrayList` or `LinkedList`.

public class ArrayDeque<E> extends AbstractCollection<E> implements Deque<E>, Cloneable, Serializable

✅ **Fast insertions/removals from both ends (`O(1)`)**
✅ **Better than `Stack` & `LinkedList` for Deque operations**
✅ **No capacity restriction** – grows dynamically
✅ **No thread-safety** – not synchronized

Can you compare the performance of using ArrayDeque and using LinkedList as a deque?

Both `ArrayDeque` and `LinkedList` implement the `Deque` interface, but they have different performance characteristics. `ArrayDeque` is usually **faster** than `LinkedList` due to **cache locality** and **less memory overhead**.

Why ArrayDeque is faster than LinkedList?

✅ `ArrayDeque` – Backed by a Circular Array

- Uses a **growable array** (doubles in size when full).
- Maintains **two pointers** (front and rear).
- Cache-friendly: data is stored **contiguously** in memory.

🚫 `LinkedList` – Doubly Linked List Implementation

- Each node stores **data + two pointers** (next & prev).
- **No resizing cost**, but **higher memory usage** (extra pointers).

When to use an ArrayDeque and LinkedList?

✅ **Use `ArrayDeque`** when:

- You need **fast insertion/removal from both ends**.
- You care about **low memory overhead**.
- You need **fast iteration**.

🚀 **Use `LinkedList` only if:**

- You frequently **insert/remove elements from the middle**.
- You **don't care about memory overhead**.

🔹 **Overall, `ArrayDeque` is the better choice in most scenarios.**

ArrayDeque: Points to remember

- Deque is an acronym for "double ended queue".
- It grows and shrinks as per usage.
- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.
- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

```
import java.util.ArrayDeque;
import java.util.Iterator;

public class ArrayDequeOperations {
```

```
    public static void main(String[] args) {
        // 1. Creating an ArrayDeque and adding elements
        ArrayDeque<String> names = new ArrayDeque<>();
        names.add("Soham");
        names.add("Praffull");
        names.add("Sharayu");
        names.add("Shailaja");
        names.add("Megha");
        names.add("Owee");

        System.out.println("Original ArrayDeque: " + names);

        // 2. Adding elements at the front and rear
        names.addFirst("First");
        names.addLast("Last");
        System.out.println("After addFirst and addLast: " + names);

        names.offerFirst("OfferFirst");
        names.offerLast("OfferLast");
        System.out.println("After offerFirst and offerLast: " + names);

        // 3. Retrieving first and last elements
        System.out.println("First element (getFirst): " + names.getFirst());
        System.out.println("Last element (getLast): " + names.getLast());

        System.out.println("First element (peekFirst): " +
names.peekFirst());
        System.out.println("Last element (peekLast): " + names.peekLast());

        // 4. Removing elements from the front and rear
        System.out.println("Removed first element (removeFirst): " +
names.removeFirst());
        System.out.println("Removed last element (removeLast): " +
names.removeLast());

        System.out.println("Deque after removeFirst and removeLast: " +
names);
```

```
        System.out.println("Polled first element (pollFirst): " +
names.pollFirst());
        System.out.println("Polled last element (pollLast): " +
names.pollLast());

        System.out.println("Deque after pollFirst and pollLast: " + names);

        // 5. Checking if an element exists
        System.out.println("Does the deque contain 'Sharayu'? " +
names.contains("Sharayu"));

        // 6. Checking if deque is empty
        System.out.println("Is deque empty? " + names.isEmpty());

        // 7. Checking the size of the deque
        System.out.println("Size of deque: " + names.size());

        // 8. Iterating using for-each loop
        System.out.println("\nIterating using for-each loop:");
        for (String name : names) {
            System.out.println(name);
        }

        // 9. Iterating using Iterator
        System.out.println("\nIterating using Iterator:");
        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // 10. Clearing the deque
        names.clear();
        System.out.println("After clearing the deque: " + names);
    }
}
```

Set Interface

What is the Set Interface?

The `Set` interface in Java is part of the **Java Collections Framework (JCF)** and represents an **unordered collection of unique elements**. Unlike `List`, `Set` does **not allow duplicate values**.

What are the key features of Set interface?

✅ **No duplicates allowed** – Ensures uniqueness.
✅ **Unordered collection** – No guarantee of element order.
✅ **Implements `Collection<E>` interface** – Supports basic collection operations.
✅ **Three main implementations**:

- `HashSet` (Unordered, best performance)
- `LinkedHashSet` (Maintains insertion order)
- `TreeSet` (Sorted order)

`HashSet` → Uses **hashing**, best performance (`O(1)` for basic operations).
`LinkedHashSet` → Uses **linked list + hashing**, maintains insertion order.
`TreeSet` → Uses **Red-Black Tree**, maintains **sorted order (`O(log n)`)**.

{SohamGlobal & Spider Projects One}

What are the important methods of the Set interface?

Method	Description	Time Complexity
`add(E e)`	Adds an element to the set (if not already present)	`O(1)` (HashSet), `O(log n)` (TreeSet)
`remove(Object o)`	Removes the specified element	`O(1)` (HashSet), `O(log n)` (TreeSet)
`contains(Object o)`	Checks if element is in the set	`O(1)` (HashSet), `O(log n)` (TreeSet)
`size()`	Returns the number of elements	`O(1)`
`isEmpty()`	Checks if the set is empty	`O(1)`
`clear()`	Removes all elements	`O(1)`
`iterator()`	Returns an iterator for traversal	`O(n)`

8. HashSet

What is HashSet?

`HashSet` is a **collection in Java** that implements the `Set` interface and is part of the **Java Collections Framework**. It is used to store **unique elements** in an **unordered manner**, making it highly efficient for operations like **searching, insertion, and deletion**.

What are the key features of HashSet?

✅ **No Duplicate Elements** – Ensures data uniqueness.
✅ **Uses Hashing Mechanism** – Provides fast lookups.
✅ **Unordered Collection** – No guarantee of insertion order.
✅ **Allows `null` Values** – But only one `null` is permitted.
✅ **Not Thread-Safe** – Needs explicit synchronization in multi-threaded environments.

✅ **Fast operations** – `O(1)` for `add()`, `remove()`, `contains()`

How Does HashSet Work Internally?

Uses `HashMap<K, V>` internally

- When an element is added, it is stored as a **key** in the HashMap.
- The value is always a **constant dummy object** (`PRESENT`).

Computes Hash Code of Element

- When an element is inserted, its **hash code** is calculated using `hashCode()`.

Finds the Bucket (Index) in the Hash Table

- The hash code is mapped to an **index** in the internal array (buckets).

Handles Collisions Using Linked List (Chaining) or Tree (After Java 8)

- If multiple elements have the **same hash index**, they are stored using **Linked List (before Java 8)**.
- **After Java 8**, if there are **more than 8 elements in the same bucket**, it **converts to a Red-Black tree** for faster lookup (`O(log n)`).

Ensures Uniqueness Using `equals()`

- If two objects have the **same hash**, `equals()` is checked to prevent duplicates.

Resizes the Hash Table When Full

- When the load factor (`0.75` default) is exceeded, the HashSet **doubles its size**.

🔹 **Note:** The order of elements is **not fixed** because `HashSet` does not maintain insertion order.

When to use HashSet?

✅ **Fast Membership Checking:** Quickly check if an element exists.
✅ **Remove Duplicates:** Store only unique elements.
✅ **Unordered Data Storage:** When order doesn't matter.
✅ **High-Performance Data Structure:** Ideal for large datasets.

What are the limitations of HashSet?

❌ **Unordered Collection** – Cannot maintain insertion order (use `LinkedHashSet` instead).
❌ **Not Thread-Safe** – Requires `Collections.synchronizedSet()` for multi-threading.
❌ **High Memory Usage** – Stores elements inside a `HashMap`, which has additional memory overhead.

What is the load factor of HashSet?

Default **initial capacity** = `16`
Load factor = `0.75` (Triggers resizing when **75% full**).
When resizing occurs:

- **Capacity Doubles** (`newCapacity = oldCapacity * 2`).
- **Rehashing Happens** (Recomputes indexes for all elements).

{SohamGlobal & Spider Projects One}

HashSet: Points to remember

- HashSet class is used to create a collection that uses a hash table for storage
- HashSet stores the elements by using a mechanism called hashing.
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order.
- elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.
- A list can contain duplicate elements whereas Set contains unique elements only.

```java
import java.util.HashSet;
import java.util.Iterator;

public class HashSetOperations {
    public static void main(String[] args) {
        // 1. Creating a HashSet and adding elements
        HashSet<String> names = new HashSet<>();
        names.add("Soham");
        names.add("Praffull");
        names.add("Sharayu");
        names.add("Shailaja");
        names.add("Megha");
        names.add("Owee");

        System.out.println("Original HashSet: " + names);

        // 2. Adding a duplicate element (HashSet does not allow duplicates)
        names.add("Soham");
        System.out.println("After adding duplicate 'Soham': " + names);

        // 3. Checking if an element exists
        System.out.println("Does the set contain 'Sharayu'? " +
names.contains("Sharayu"));

        // 4. Removing an element
        names.remove("Megha");
        System.out.println("After removing 'Megha': " + names);

        // 5. Checking if the set is empty
        System.out.println("Is the set empty? " + names.isEmpty());

        // 6. Checking the size of the set
        System.out.println("Size of HashSet: " + names.size());

        // 7. Iterating using for-each loop
        System.out.println("\nIterating using for-each loop:");
        for (String name : names) {
```

```
            System.out.println(name);
        }

        // 8. Iterating using Iterator
        System.out.println("\nIterating using Iterator:");
        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // 9. Converting HashSet to an array
        String[] array = names.toArray(new String[0]);
        System.out.println("\nHashSet converted to array:");
        for (String name : array) {
            System.out.println(name);
        }
        // 10. Clearing the HashSet
        names.clear();
        System.out.println("After clearing the HashSet: " + names);
    }
}
```

9. LinkedHashSet

What is LinkedHashSet?

`LinkedHashSet` is a part of the **Java Collections Framework** that implements the `Set` interface. It extends `HashSet` and maintains the **insertion order** of elements while ensuring **unique values**.

public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, Serializable

What are the features of LinkedHashSet?

✅ **No Duplicate Elements** – Ensures data uniqueness.
✅ **Maintains Insertion Order** – Unlike `HashSet`, it keeps elements in the order they were inserted.

✅ **Uses Hashing and Linked List** – Provides fast operations and maintains order.
✅ **Allows `null` Values** – But only one `null` is permitted.
✅ **Not Thread-Safe** – Needs explicit synchronization in multi-threaded environments.

Explain the internal working of LinkedHashSet.

How Does `LinkedHashSet` Store Elements?

1. **Uses a `LinkedHashMap` Internally**
   - Every element is stored as a **key** inside a `LinkedHashMap`, with a dummy constant value.
   - Ensures **uniqueness** (no duplicate keys in `LinkedHashMap`).
2. **Maintains Order Using a Doubly Linked List**
   - Each entry in `LinkedHashMap` maintains a **before** and **after** pointer.
   - These pointers create a **doubly linked list** connecting all elements in insertion order.
3. **Uses `hashCode()` for Fast Lookup**
   - When an element is added, `hashCode()` determines its **bucket (index)**.
4. **Handles Collisions Using Linked List (Java 7) or Balanced Tree (Java 8+)**
   - If multiple elements have the **same hash code**, Java handles them using **linked lists** or **balanced trees**.

When to Use LinkedHashSet?

✅ **Need for Fast Lookups (`O(1)`)** – Similar to `HashSet`.
✅ **Preserving Insertion Order** – Unlike `HashSet`, maintains the order of elements.
✅ **Removing Duplicates While Keeping Order** – Ideal for ordered unique collections.

What are the Limitations of LinkedHashSet

❌ **Slightly Slower Than `HashSet`** – Due to the extra linked list overhead.
❌ **Not Thread-Safe** – Requires `Collections.synchronizedSet()` for multi-threading.
❌ **Higher Memory Consumption** – Stores extra pointers for maintaining order.

Give me the summary of features of LinkedHashSet.

Feature	Description
Implements	`Set<E>` Interface
Underlying Data Structure	`LinkedHashMap<E, Object>`
Duplicates Allowed?	❌ No
Null Elements?	✅ Yes (only one)
Order of Elements?	✅ Maintains Insertion Order
Performance	🚀 `O(1)` (Average), `O(n)` (Worst Case)
Thread-Safe?	❌ No (Use `Collections.synchronizedSet()` if needed)
Best For	Fast lookup, unique elements, and maintaining order

LinkedHashSet: Points to remember

- LinkedHashSet class is a Hashtable and double Linked list implementation of the Set interface.
- It inherits the HashSet class and implements the Set interface.
- LinkedHashSet class contains unique elements only like HashSet.
- LinkedHashSet class provides all optional set operations and permits null elements.
- The LinkedHashSet class is non-synchronized.
- LinkedHashSet class maintains insertion order.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

```
import java.util.LinkedHashSet;
import java.util.Iterator;

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

```
        // 1. Creating a LinkedHashSet and adding elements
        LinkedHashSet<String> names = new LinkedHashSet<>();
        names.add("Soham");
        names.add("Praffull");
        names.add("Sharayu");
        names.add("Shailaja");
        names.add("Megha");
        names.add("Owee");

        System.out.println("Original LinkedHashSet: " + names);

        // 2. Adding a duplicate element (LinkedHashSet does not allow
duplicates)
        names.add("Soham");
        System.out.println("After adding duplicate 'Soham': " + names);

        // 3. Checking if an element exists
        System.out.println("Does the set contain 'Sharayu'? " +
names.contains("Sharayu"));

        // 4. Removing an element
        names.remove("Megha");
        System.out.println("After removing 'Megha': " + names);

        // 5. Checking if the set is empty
        System.out.println("Is the set empty? " + names.isEmpty());

        // 6. Checking the size of the set
        System.out.println("Size of LinkedHashSet: " + names.size());

        // 7. Iterating using for-each loop
        System.out.println("\nIterating using for-each loop:");
        for (String name : names) {
            System.out.println(name);
        }

        // 8. Iterating using Iterator
```

```
        System.out.println("\nIterating using Iterator:");
        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // 9. Converting LinkedHashSet to an array
        String[] array = names.toArray(new String[0]);
        System.out.println("\nLinkedHashSet converted to array:");
        for (String name : array) {
            System.out.println(name);
        }

        // 10. Clearing the LinkedHashSet
        names.clear();
        System.out.println("After clearing the LinkedHashSet: " + names);
    }
}
```

10. TreeSet

What is TreeSet?

`TreeSet` is a class in Java that implements the `NavigableSet` interface and extends `AbstractSet`. It **stores unique elements in sorted order** and is based on a **self-balancing Red-Black Tree**.

public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable

What are the key features of TreeSet?

✅ **Unique elements only** – No duplicates allowed.
✅ **Sorted order** – Maintains elements in **ascending order** (natural ordering).
✅ **Implements `NavigableSet`** – Provides additional methods for navigation.
✅ **Tree-based structure** – Uses a **Red-Black Tree** for balancing.
✅ **Logarithmic time complexity (`O(log n)`)** – Efficient for insertions, deletions, and lookups.

✅ **Thread-safe?** – ❌ No, but can be synchronized manually.
✅ **Allows `null`?** – ❌ No (throws `NullPointerException` in Java 8+).

How Does TreeSet Work Internally?

Uses `TreeMap` Internally

- `TreeSet` is backed by a `TreeMap`, where:
    - The **elements of the set** are stored as **keys** in `TreeMap`.
    - The **values** in `TreeMap` are **dummy objects**.

Elements Are Stored in a Red-Black Tree

- The Red-Black Tree is a **self-balancing Binary Search Tree (BST)**.
- Ensures that operations (insert, delete, search) take **O(log n)** time.

Sorting Mechanism

- By default, `TreeSet` follows **natural ordering** (`Comparable` interface).
- A custom **Comparator** can be provided for different sorting orders.

Insertion in `O(log n)` Time

- When adding an element:
    - The **Red-Black Tree finds the correct position** using binary search.
    - It **rebalances itself** if needed to maintain `O(log n)` height.

No Duplicates Allowed

- Before insertion, `TreeSet` checks if the element **already exists**.
- If `compareTo()` returns `0` (meaning equal elements), the new element is **ignored**.

How can we change the sorting order in TreeSet?

By default, `TreeSet` sorts elements **in ascending order**. We can define **custom sorting** using a `Comparator`.

```
TreeSet<Integer> treeSet = new TreeSet<>(Collections.reverseOrder());
```

Tell me the detailed difference between HashSet, LinkedHashSet and TreeSet.

`HashSet`, `LinkedHashSet`, and `TreeSet` are implementations of the `Set` interface in Java. While they all store **unique elements**, they differ in ordering, performance, and underlying data structures.

Feature	`HashSet`	`LinkedHashSet`	`TreeSet`
Ordering	❌ No Order	✅ Insertion Order	✅ Sorted Order
Duplicates Allowed?	❌ No	❌ No	❌ No
Null Allowed?	✅ Yes	✅ Yes	❌ No
Underlying Structure	`HashMap`	`HashMap + LinkedList`	`TreeMap (Red-Black Tree)`
Performance (`add()`, `remove()`, `contains()`)	⚡ `O(1)`	⚡ `O(1)`	🐢 `O(log n)`
Thread Safety	❌ No	❌ No	❌ No
Best Use Case	Fastest lookups	Preserve insertion order	Sorted unique elements

Scenario	Best Choice	Reason
Fastest operations, no order needed	`HashSet`	`O(1)` performance
Maintain insertion order	`LinkedHashSet`	Uses a linked list
Maintain sorted order	`TreeSet`	Uses a Red-Black Tree
When memory is a concern	`HashSet`	Uses less memory
Finding min/max or range queries	`TreeSet`	Provides `first()`, `last()`, `ceiling()`, `floor()`

🚀 **Use `HashSet` for best performance (`O(1)`) when ordering doesn't matter.**
📌 **Use `LinkedHashSet` when insertion order must be maintained.**
🔍 **Use `TreeSet` for automatically sorted unique elements (`O(log n)`).**

TreeSet: Points to remember

- TreeSet class implements the Set interface that uses a binary tree for storage.
- The objects of the TreeSet class are stored in ascending order.
- Performs sorting when an object is added to the collection.
- Java TreeSet class contains unique elements only like HashSet.
- Faster access and retrieval
- TreeSet class doesn't allow null elements, only homogeneous values allowed (else ClassCastException).
- The TreeSet class is non synchronized.
- TreeSet is being implemented using a binary search tree, which is self-balancing just like a Red-Black Tree.

```java
import java.util.TreeSet;
import java.util.Iterator;

public class TreeSetOperations {
    public static void main(String[] args) {
        // 1. Creating a TreeSet and adding elements
        TreeSet<String> names = new TreeSet<>();
        names.add("Soham");
        names.add("Praffull");
        names.add("Sharayu");
        names.add("Shailaja");
        names.add("Megha");
        names.add("Owee");

        System.out.println("Original TreeSet: " + names); // Sorted order

        // 2. Adding a duplicate element (TreeSet does not allow duplicates)
        names.add("Soham");
        System.out.println("After adding duplicate 'Soham': " + names);
```

```
        // 3. Checking if an element exists
        System.out.println("Does the set contain 'Sharayu'? " +
names.contains("Sharayu"));

        // 4. Removing an element
        names.remove("Megha");
        System.out.println("After removing 'Megha': " + names);

        // 5. Checking if the set is empty
        System.out.println("Is the set empty? " + names.isEmpty());

        // 6. Checking the size of the set
        System.out.println("Size of TreeSet: " + names.size());

        // 7. Iterating using for-each loop
        System.out.println("\nIterating using for-each loop:");
        for (String name : names) {
            System.out.println(name);
        }

        // 8. Iterating using Iterator
        System.out.println("\nIterating using Iterator:");
        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // 9. Fetching first and last elements
        System.out.println("First element: " + names.first());
        System.out.println("Last element: " + names.last());

        // 10. Fetching headSet (elements less than "Sharayu")
        System.out.println("Elements before 'Sharayu': " +
names.headSet("Sharayu"));

        // 11. Fetching tailSet (elements greater than or equal to "Sharayu")
```

```
        System.out.println("Elements from 'Sharayu' onwards: " +
names.tailSet("Sharayu"));

        // 12. Fetching subSet (elements between "Praffull" and "Soham")
        System.out.println("Subset between 'Praffull' and 'Soham': " +
names.subSet("Praffull", "Soham"));

        // 13. Polling first and last elements
        System.out.println("Polling first element: " + names.pollFirst());
        System.out.println("Polling last element: " + names.pollLast());
        System.out.println("After polling, TreeSet: " + names);

        // 14. Converting TreeSet to an array
        String[] array = names.toArray(new String[0]);
        System.out.println("\nTreeSet converted to array:");
        for (String name : array) {
            System.out.println(name);
        }

        // 15. Clearing the TreeSet
        names.clear();
        System.out.println("After clearing the TreeSet: " + names);
    }
}
```

Map interface

What is a Map interface?

The `Map` interface in Java is a part of the **Java Collections Framework** and is used for storing key-value pairs. Unlike other collections such as `List` or `Set`, which store only values, `Map` allows you to **associate a unique key with each value**.

What are the key features of Map interface?

✅ **Stores Key-Value Pairs** – Each key is mapped to a single value.
✅ **Unique Keys** – A `Map` **does not allow duplicate keys**, but values can be duplicated.
✅ **Efficient Lookups** – You can retrieve values in **O(1) (HashMap)** or **O(log n) (TreeMap)** time.
✅ **Implements No Direct Collection Interface** – `Map` is separate from `Collection` but part of the framework.

Which are the most popular methods of Map interface?

Method	Description
`V put(K key, V value)`	Inserts a key-value pair. If the key exists, it updates the value.
`V get(Object key)`	Returns the value associated with the key.
`boolean containsKey(Object key)`	Checks if a key exists.
`boolean containsValue(Object value)`	Checks if a value exists.
`V remove(Object key)`	Removes the mapping for the specified key.
`int size()`	Returns the number of key-value pairs.
`boolean isEmpty()`	Returns `true` if the map is empty.
`Set<K> keySet()`	Returns a `Set` of all keys.
`Collection<V> values()`	Returns a `Collection` of all values.
`Set<Map.Entry<K,V>> entrySet()`	Returns a `Set` of key-value pairs (`Map.Entry`).

11. HashMap

What is a HashMap?

A `HashMap` is a **part of the Java Collections Framework** that implements the `Map` interface. It is used to store **key-value pairs**, where **keys are unique** and values can be duplicated. The data inside a `HashMap` is stored using a **hashing mechanism**, making retrieval operations extremely fast.

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable

What are the key features of HashMap?

✅ **Unordered Storage** – Does not maintain any specific order.
✅ **Allows One `null` Key** – But multiple `null` values are allowed.
✅ **Fast Performance (`O(1)`)** – Provides **constant-time lookup and insertion** in most cases.
✅ **Not Thread-Safe** – Multiple threads can cause concurrency issues unless synchronized.

Explain the internal working of HashMap.

A `HashMap` stores data using a **hash table** and a **bucket system**. When you add a key-value pair:

1. **The key's hashCode() is computed**.
2. **The hash value determines the bucket** in which the entry is stored.
3. **If two keys have the same hash (collision occurs)**, the elements are stored in a **linked list** (or **balanced tree** in Java 8+ if more than 8 entries exist in a bucket).
4. **On retrieval**, the hash value is calculated again to locate the bucket.

Example of Bucket Storage

If keys `"A"`, `"B"`, and `"C"` have similar hash values:

Index	Data (Linked List / Tree)
0	(empty)
1	("A", 100) -> ("B", 200) -> ("C", 300) (Linked List)
2	(empty)
3	("D", 400)

{SohamGlobal & Spider Projects One}

What are the best Use Cases for HashMap?

✔ **Fast lookup and insert operations are needed**
✔ **Ordering of elements is not required**
✔ **You need to store key-value pairs with unique keys**

HashMap: Points to remember

- HashMap class implements the Map interface which allows us to store key and value pair, where keys should be unique.
- If you try to insert the duplicate key, it will replace the element of the corresponding key.
- It is easy to perform operations using the key index like updation, deletion, etc.
- HashMap in Java is like the legacy Hashtable class, but it is not synchronized.
- HashMap contains values based on the key.
- HashMap contains only unique keys.
- HashMap may have one null key and multiple null values.
- HashMap is non synchronized.
- HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

```
import java.util.HashMap;
import java.util.Map;
import java.util.Iterator;

public class HashMapOperations {
    public static void main(String[] args) {
        // 1. Creating a HashMap and adding key-value pairs
        HashMap<Integer, String> names = new HashMap<>();
        names.put(101, "Soham");
        names.put(102, "Praffull");
        names.put(103, "Sharayu");
        names.put(104, "Shailaja");
        names.put(105, "Megha");
        names.put(106, "Owee");

        System.out.println("Original HashMap: " + names);

        // 2. Fetching value by key
        System.out.println("Value associated with key 103: " +
names.get(103));
```

```
        // 3. Checking if a key exists
        System.out.println("Does key 104 exist? " + names.containsKey(104));

        // 4. Checking if a value exists
        System.out.println("Does value 'Megha' exist? " +
names.containsValue("Megha"));


        // 5. Removing an entry by key
        names.remove(105);
        System.out.println("After removing key 105: " + names);

        // 6. Checking if the HashMap is empty
        System.out.println("Is HashMap empty? " + names.isEmpty());

        // 7. Checking the size of the HashMap
        System.out.println("Size of HashMap: " + names.size());

        // 8. Iterating over keys using keySet()
        System.out.println("\nIterating over keys:");
        for (Integer key : names.keySet()) {
            System.out.println("Key: " + key);
        }

        // 9. Iterating over values using values()
        System.out.println("\nIterating over values:");
        for (String value : names.values()) {
            System.out.println("Value: " + value);
        }

        // 10. Iterating over key-value pairs using entrySet()
        System.out.println("\nIterating over key-value pairs:");
        for (Map.Entry<Integer, String> entry : names.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
        }
```

```
        // 11. Iterating using Iterator
        System.out.println("\nIterating using Iterator:");
        Iterator<Map.Entry<Integer, String>> iterator =
names.entrySet().iterator();
        while (iterator.hasNext()) {
            Map.Entry<Integer, String> entry = iterator.next();
            System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
        }

        // 12. Replacing a value for a key
        names.replace(102, "Praffull Updated");
        System.out.println("After replacing value of key 102: " + names);

        // 13. Fetching a default value if key is absent
        System.out.println("Fetching key 107 (not present): " +
names.getOrDefault(107, "Not Found"));

        // 14. Clearing the HashMap
        names.clear();
        System.out.println("After clearing the HashMap: " + names);
    }
}
```

12. Hashtable

What is a Hashtable?

A `Hashtable` is a part of **Java's legacy collection framework** that implements the `Map` interface. It is used to **store key-value pairs**, similar to `HashMap`, but with an important difference:

✔ **It is synchronized and thread-safe.**
❌ **It does not allow `null` keys or values.**

public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable, Serializable

What are the Key Characteristics of Hashtable?

✅ **Thread-Safe & Synchronized** – Can be shared between multiple threads without external synchronization.
✅ **No `null` Keys or Values** – Unlike `HashMap`, `Hashtable` does not accept `null` keys or `null` values.
✅ **Unordered Storage** – Does not maintain insertion order or sorting.
✅ **Performance Slower than `HashMap`** – Synchronization adds overhead, making `Hashtable` slower than `HashMap`.

Explain internal working of Hashtable.

Internal Working of Hashtable

- Similar to `HashMap`, `Hashtable` uses **hashing** to store key-value pairs.
- It **computes the hash of a key** and places the entry in a **bucket** based on this hash value.
- If multiple keys produce the **same hash (collision)**, it stores them in a **linked list** within the same bucket.
- Since `Hashtable` is **synchronized**, multiple threads can access it safely, but this adds overhead.

What are the differences between HashMap and Hashtable?

Feature	HashMap	Hashtable
Thread-Safe	❌ No	✅ Yes
Synchronized	❌ No	✅ Yes
Performance	⚡ Fast	🐢 Slower (due to synchronization)
Allows `null` Key?	✅ Yes (One)	❌ No
Allows `null` Values?	✅ Yes (Multiple)	❌ No
Ordering	❌ No ordering	❌ No ordering
Introduced in	Java 1.2 (part of Collections Framework)	Java 1.0 (Legacy class)

When to Use a Hashtable?

✔ **When multiple threads need to access a map safely.**
✔ **When synchronization is required and you don't want to use `ConcurrentHashMap`.**
❌ **Avoid if you need better performance** – Use `ConcurrentHashMap` instead.

Hashtable: Points to remember

- Hashtable class implements a hashtable, which maps keys to values.
- It inherits Dictionary class and implements the Map interface.
- A Hashtable is an array of a list. Each list is known as a bucket.
- The position of the bucket is identified by calling the hashcode() method.
- A Hashtable contains values based on the key.
- Java Hashtable class contains unique elements.
- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.
- Legacy class
- If thread safety is not required better to use HashMap
- If thread safety is required use ConcurrentHashMap

```java
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Map;

public class HashtableOperations {
    public static void main(String[] args) {
        // 1. Creating a Hashtable and adding key-value pairs
        Hashtable<Integer, String> names = new Hashtable<>();
        names.put(101, "Soham");
        names.put(102, "Praffull");
        names.put(103, "Sharayu");
        names.put(104, "Shailaja");
        names.put(105, "Megha");
        names.put(106, "Owee");

        System.out.println("Original Hashtable: " + names);

        // 2. Fetching value by key
        System.out.println("Value associated with key 103: " +
names.get(103));

        // 3. Checking if a key exists
        System.out.println("Does key 104 exist? " + names.containsKey(104));

        // 4. Checking if a value exists
        System.out.println("Does value 'Megha' exist? " +
names.containsValue("Megha"));

        // 5. Removing an entry by key
        names.remove(105);
        System.out.println("After removing key 105: " + names);

        // 6. Checking if the Hashtable is empty
        System.out.println("Is Hashtable empty? " + names.isEmpty());

        // 7. Checking the size of the Hashtable
        System.out.println("Size of Hashtable: " + names.size());

        // 8. Iterating over keys using keys()
        System.out.println("\nIterating over keys:");
        Enumeration<Integer> keys = names.keys();
        while (keys.hasMoreElements()) {
            System.out.println("Key: " + keys.nextElement());
        }

        // 9. Iterating over values using elements()
        System.out.println("\nIterating over values:");
        Enumeration<String> values = names.elements();
        while (values.hasMoreElements()) {
            System.out.println("Value: " + values.nextElement());
```

```
        }

        // 10. Iterating over key-value pairs using entrySet()
        System.out.println("\nIterating over key-value pairs:");
        for (Map.Entry<Integer, String> entry : names.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
        }

        // 11. Replacing a value for a key
        names.replace(102, "Praffull Updated");
        System.out.println("After replacing value of key 102: " + names);

        // 12. Fetching a default value if key is absent
        System.out.println("Fetching key 107 (not present): " +
names.getOrDefault(107, "Not Found"));

        // 13. Clearing the Hashtable
        names.clear();
        System.out.println("After clearing the Hashtable: " + names);
    }
}
```

13. LinkedHashMap

What is a LinkedHashMap?

`LinkedHashMap` is a part of the **Java Collections Framework** that extends `HashMap` while maintaining **insertion order**. It stores key-value pairs like `HashMap`, but it also maintains a **linked list of entries** to preserve the order in which keys are inserted.

What are the Key Characteristics of LinkedHashMap?

✅ **Maintains Insertion Order** – Unlike `HashMap`, it remembers the order in which elements were added.
✅ **Faster Access (`O(1)`)** – Similar performance to `HashMap`.
✅ **Allows `null` Keys and Values** – Just like `HashMap`, it allows one `null` key and multiple `null` values.
✅ **Not Thread-Safe** – Needs external synchronization for multi-threaded access.
✅ **Provides Access Order Mode** – Can be configured to maintain access order (useful for LRU caches).

How is the Internal Working of LinkedHashMap?

- Uses a **combination of a Hash Table and a Doubly Linked List**.
- Each entry contains pointers to **previous** and **next** elements, forming a **linked list**.
- When an entry is added, it is linked to the **previous entry** in insertion order.
- **Access Order Mode (`accessOrder = true`)** allows the most recently accessed elements to move to the end (used for implementing LRU caches).

When to Use LinkedHashMap?

✔ **When insertion order matters**
✔ **When you need an LRU cache implementation**
❌ **Avoid if ordering is not needed (use `HashMap` instead for better performance).**

What is LRU?

LRU (Least Recently Used) Cache is a **caching algorithm** that removes the **least recently used** items when the cache reaches its capacity.

✔ **Efficiently manages memory usage**
✔ **Ensures frequently used items stay in cache**
✔ **Used in databases, operating systems, and web caching**

How Does LRU Work?

- The cache has a **fixed size**.
- When a new item is accessed, it is moved to the **most recently used position**.
- If an item is accessed again, it moves to the front.
- When the cache is **full**, the **least recently used item** (at the back) is removed to make space for a new entry.

🔹 **Think of it like a queue where the most recently used elements stay in front!**

What are the Real-World Examples of LRU?

🔹 **Web Browsers (Chrome, Firefox, Edge, etc.)**

- The browser caches recently visited web pages.
- If cache memory is full, **older pages (least accessed)** are removed first.

🔹 **Operating Systems**

- OS manages memory using LRU in **page replacement algorithms**.
- When RAM is full, the **least used pages** are swapped out.

🔹 **Database Systems**

- Databases use LRU for **query caching** to optimize repeated queries.

```java
import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashMapOperations {
    public static void main(String[] args) {
        // 1. Creating a LinkedHashMap and adding key-value pairs
        LinkedHashMap<Integer, String> names = new LinkedHashMap<>();
        names.put(101, "Soham");
        names.put(102, "Praffull");
        names.put(103, "Sharayu");
        names.put(104, "Shailaja");
        names.put(105, "Megha");
        names.put(106, "Owee");

        System.out.println("Original LinkedHashMap: " + names);

        // 2. Fetching value by key
        System.out.println("Value associated with key 103: " +
names.get(103));

        // 3. Checking if a key exists
        System.out.println("Does key 104 exist? " + names.containsKey(104));

        // 4. Checking if a value exists
        System.out.println("Does value 'Megha' exist? " +
names.containsValue("Megha"));

        // 5. Removing an entry by key
        names.remove(105);
        System.out.println("After removing key 105: " + names);
```

```
        // 6. Checking if the LinkedHashMap is empty
        System.out.println("Is LinkedHashMap empty? " + names.isEmpty());

        // 7. Checking the size of the LinkedHashMap
        System.out.println("Size of LinkedHashMap: " + names.size());

        // 8. Iterating over keys using keySet()
        System.out.println("\nIterating over keys:");
        for (Integer key : names.keySet()) {
            System.out.println("Key: " + key);
        }

        // 9. Iterating over values using values()
        System.out.println("\nIterating over values:");
        for (String value : names.values()) {
            System.out.println("Value: " + value);
        }

        // 10. Iterating over key-value pairs using entrySet()
        System.out.println("\nIterating over key-value pairs:");
        for (Map.Entry<Integer, String> entry : names.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
        }

        // 11. Replacing a value for a key
        names.replace(102, "Praffull Updated");
        System.out.println("After replacing value of key 102: " + names);

        // 12. Fetching a default value if key is absent
        System.out.println("Fetching key 107 (not present): " +
names.getOrDefault(107, "Not Found"));

        // 13. Clearing the LinkedHashMap
        names.clear();
        System.out.println("After clearing the LinkedHashMap: " + names);
```

```
    }
}
```

14. TreeMap

What is a TreeMap?

`TreeMap` is a part of Java's **Collections Framework** that implements the `NavigableMap` interface and extends `AbstractMap`. It **stores key-value pairs** in a **sorted order** based on the natural ordering of keys or a custom comparator.

What are the features of TreeMap?

✔ **Sorted Order** → Maintains keys in ascending order (by default).
✔ **Efficient Search** → Uses a **Red-Black Tree** for operations.
✔ **No `null` Keys** → Unlike `HashMap`, `TreeMap` does not allow `null` keys.
✔ **Fast Lookup & Update** → `O(log n)` time complexity for `put()`, `get()`, and `remove()`.

Feature	TreeMap
Sorting	✅ Sorted in **ascending order** (Natural order or Custom Comparator).
Time Complexity	⚡ `O(log n)` for insertion, deletion, and retrieval (Red-Black Tree).
Allows `null` Keys?	❌ No (`NullPointerException` for null key).
Allows `null` Values?	✅ Yes.
Thread-Safe?	❌ No (Needs external synchronization).

Explain Internal Working of TreeMap

TreeMap is implemented using a **Self-Balancing Red-Black Tree**, where:

- Keys are **sorted** as per **natural order** or a **custom comparator**.
- Operations like `put()`, `get()`, and `remove()` take **O(log n)** time.
- The tree is **rebalanced automatically** to maintain performance.

Give some Real-World Applications of TreeMap

✅ **Maintaining Ordered Data** → Storing user data in sorted order (e.g., IDs, timestamps).
✅ **Range Queries** → Used in financial applications (e.g., stock price tracking).
✅ **NavigableMap Operations** → Useful when we need access to nearest higher/lower keys.

When to Use TreeMap?

✔ **When you need sorted data retrieval.**
✔ **When fast range queries (subMap, tailMap, etc.) are required.**
❌ **Avoid if ordering is not required (use `HashMap` for better performance).**

```java
import java.util.NavigableMap;
import java.util.TreeMap;
import java.util.Map;

public class TreeMapOperations {
    public static void main(String[] args) {
        // 1. Creating a TreeMap and adding key-value pairs
        TreeMap<Integer, String> names = new TreeMap<>();
        names.put(101, "Soham");
        names.put(102, "Praffull");
        names.put(103, "Sharayu");
        names.put(104, "Shailaja");
        names.put(105, "Megha");
        names.put(106, "Owee");

        System.out.println("Original TreeMap: " + names);

        // 2. Fetching value by key
        System.out.println("Value associated with key 103: " +
names.get(103));

        // 3. Checking if a key exists
        System.out.println("Does key 104 exist? " + names.containsKey(104));

        // 4. Checking if a value exists
```

```
        System.out.println("Does value 'Megha' exist? " +
names.containsValue("Megha"));

        // 5. Removing an entry by key
        names.remove(105);
        System.out.println("After removing key 105: " + names);

        // 6. Checking if the TreeMap is empty
        System.out.println("Is TreeMap empty? " + names.isEmpty());

        // 7. Checking the size of the TreeMap
        System.out.println("Size of TreeMap: " + names.size());

        // 8. Iterating over keys using keySet()
        System.out.println("\nIterating over keys:");
        for (Integer key : names.keySet()) {
            System.out.println("Key: " + key);
        }

        // 9. Iterating over values using values()
        System.out.println("\nIterating over values:");
        for (String value : names.values()) {
            System.out.println("Value: " + value);
        }

        // 10. Iterating over key-value pairs using entrySet()
        System.out.println("\nIterating over key-value pairs:");
        for (Map.Entry<Integer, String> entry : names.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
        }

        // 11. Replacing a value for a key
        names.replace(102, "Praffull Updated");
        System.out.println("After replacing value of key 102: " + names);

        // 12. Fetching a default value if key is absent
```

```
        System.out.println("Fetching key 107 (not present): " +
names.getOrDefault(107, "Not Found"));

        // 13. Getting first and last key
        System.out.println("First key: " + names.firstKey());
        System.out.println("Last key: " + names.lastKey());

        // 14. Getting first and last entry
        System.out.println("First entry: " + names.firstEntry());
        System.out.println("Last entry: " + names.lastEntry());

        // 15. Getting lower and higher keys
        System.out.println("Lower key than 103: " + names.lowerKey(103));
        System.out.println("Higher key than 103: " + names.higherKey(103));

        // 16. Getting lower and higher entries
        System.out.println("Lower entry than 103: " + names.lowerEntry(103));
        System.out.println("Higher entry than 103: " +
names.higherEntry(103));

        // 17. Sub-map operations
        System.out.println("Sub-map from 102 to 104: " + names.subMap(102,
104));
        System.out.println("Head-map (keys less than 104): " +
names.headMap(104));
        System.out.println("Tail-map (keys greater than or equal to 103): " +
names.tailMap(103));

        // 18. Clearing the TreeMap
        names.clear();
        System.out.println("After clearing the TreeMap: " + names);
    }
}
```

15. ConcurrentHashMap

What is ConcurrentHashMap?

`ConcurrentHashMap` is a **thread-safe** implementation of the `Map` interface, introduced in **Java 1.5** as part of the **java.util.concurrent** package. It is designed to allow multiple threads to read and write simultaneously **without blocking the entire map**.

What are the features of ConcurrentHashMap?

✔ **Efficient for multi-threaded applications**
✔ **Thread-safe without using `synchronized` on the entire map**
✔ **Faster than `Hashtable` and `Collections.synchronizedMap()`**

Feature	Details
Thread-Safety	✅ Thread-safe for concurrent access.
Performance	⚡ Faster than `Hashtable`, avoids global locking.
Null Keys & Values?	❌ No `null` keys or `null` values allowed.
Time Complexity	🔥 `O(1)` for `get()`, `put()`, `remove()` (similar to `HashMap`).
Internal Data Structure	Uses **segmented buckets** (unlike `HashTable`).

Explain Internal Working of ConcurrentHashMap

Unlike `HashMap`, which uses a single array of buckets, `ConcurrentHashMap` divides the map into **segments (buckets)** to allow concurrent access.

1. **Segmented Locking (Bucket-Level Locking)**
   - Instead of **locking the entire map**, it locks only the **bucket** that a key belongs to.
   - This allows **multiple threads** to perform operations on **different keys** without interference.
2. **How It Works**

- Read operations (`get()`) → **Non-blocking**, very fast (`O(1)`).
- **Write operations (`put()`, `remove()`) → Bucket-level locking**, so multiple threads can modify different keys **concurrently**.
- **Combines lock-free reads with controlled writes** to maximize performance.

How is ConcurrentHashMap Better than Other Maps?

Feature	ConcurrentHashMap	HashMap	Hashtable
Thread-Safety	✅ Yes (High performance)	❌ No (Not thread-safe)	✅ Yes (Global lock)
Locking Mechanism	✅ Bucket-Level Locking	❌ No Locking	🚫 Whole Map Locking
Null Keys Allowed?	❌ No	✅ Yes	❌ No
Performance in Multi-Threading	🚀 Excellent	🐢 Poor (Needs external sync)	🐌 Slow (Global locking)

When to Use ConcurrentHashMap?

✔ When you need a **thread-safe** map in **multi-threaded environments**.
✔ When you require **high performance with minimal locking**.
❌ Avoid if you need to store **`null` keys or values** (use `HashMap` instead).

```
package com.sharayu.programs;

import java.util.Iterator;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapDemo {
    public static void main(String[] args) {
        ConcurrentHashMap<String, String> users=new ConcurrentHashMap<String, String>();
        users.put("sharayu", "spider");
        users.put("praffull", "chelsea");
        users.put("soham", "liverpool");
```

```
        users.put("megha", "projects");

        Iterator<String> iterator=users.keySet().iterator();
        while(iterator.hasNext())
        {
            System.out.println(iterator.next());
            users.put("buttler", "england");
        }

        System.out.println(users);
    }


}
```

Summary of Comparison

{SohamGlobal & Spider Projects One}

Collection Class	Internal Data Structure	Insertion Order	Allows `null`	Allows Duplicates	Synchronized	Initial Capacity	Load Factor	Sorting	Fail-Safe
ArrayList	**Dynamic Array**	✅ Preserved	✅ Yes (1 `null`)	✅ Yes	❌ No	**10**	N/A	❌ No (Manual Sorting)	❌ No (Fail-Fast)
LinkedList	**Doubly Linked List**	✅ Preserved	✅ Yes (Multiple)	✅ Yes	❌ No	N/A	N/A	❌ No (Manual Sorting)	❌ No (Fail-Fast)

Vector	**Dynamic Array**	✅ Preserved	✅ Yes (Multiple)	✅ Yes	✅ Yes	**10**	**2.0**	❌ No (Manual Sorting)	❌ No (Fail-Fast)
Stack	**Dynamic Array (LIFO)**	✅ Preserved	✅ Yes (Multiple)	✅ Yes	✅ Yes	**10**	**2.0**	❌ No	❌ No (Fail-Fast)
HashSet	**Hash Table**	❌ Unordered	✅ Yes (1 `null`)	❌ No	❌ No	**16**	**0.75**	❌ No	❌ No (Fail-Fast)
Collection Class	**Internal Data Structure**	**Insertion Order**	**Allows `null`**	**Allows Duplicates**	**Synchronized**	**Initial Capacity**	**Load Factor**	**Sorting**	**Fail-Safe**
LinkedHashSet	**Hash Table + Linked List**	✅ Preserved	✅ Yes (1 `null`)	❌ No	❌ No	**16**	**0.75**	❌ No	❌ No (Fail-Fast)
TreeSet	**Red-Black Tree**	✅ Sorted (Natural)	❌ No	❌ No	❌ No	N/A	N/A	✅ Yes (Natural/Custom Comparator)	❌ No (Fail-Fast)

HashMap	**Hash Table + Linked List (after threshold)**	❌ Unordered	✅ Yes (1 `null` key, many `null` values)	✅ Yes (Keys Unique)	❌ No	**16**	**0.75**	❌ No	❌ No (Fail-Fast)
LinkedHashMap	**Hash Table + Linked List**	✅ Preserved	✅ Yes (1 `null` key)	✅ Yes (Keys Unique)	❌ No	**16**	**0.75**	❌ No	❌ No (Fail-Fast)
TreeMap	**Red-Black Tree**	✅ Sorted (Natural Order)	❌ No	✅ Yes (Keys Unique)	❌ No	N/A	N/A	✅ Yes (Natural/Custom Comparator)	❌ No (Fail-Fast)
Collection Class	**Internal Data Structure**	**Insertion Order**	**Allows `null`**	**Allows Duplicates**	**Synchronized**	**Initial Capacity**	**Load Factor**	**Sorting**	**Fail-Safe**
Hashtable	**Hash Table**	❌ Unordered	❌ No (`null` keys/values not allowed)	✅ Yes (Keys Unique)	✅ Yes	**11**	**0.75**	❌ No	❌ No (Fail-Fast)

ConcurrentHashMap	**Segmented Hash Table**	❌ Unordered	❌ No	✅ Yes (Keys Unique)	✅ Yes (Thread-Safe)	**16**	**0.75**	❌ No	✅ Yes (Fail-Safe)
CopyOnWriteArrayList	**Array (Copy-On-Write)**	✅ Preserved	✅ Yes	✅ Yes	✅ Yes	N/A	N/A	❌ No	✅ Yes (Fail-Safe)
PriorityQueue	**Heap (Binary Heap by default, Min-Heap)**	❌ Unordered	❌ No	✅ Yes	❌ No	**11**	N/A	✅ Yes (Natural/Custom Comparator)	❌ No (Fail-Fast)
Collection Class	**Internal Data Structure**	**Insertion Order**	**Allows `null`**	**Allows Duplicates**	**Synchronized**	**Initial Capacity**	**Load Factor**	**Sorting**	**Fail-Safe**
ArrayDeque	**Resizable Array (Double-Ended Queue)**	✅ Preserved	❌ No	✅ Yes	❌ No	**16**	N/A	❌ No	❌ No (Fail-Fast)

Functional Interfaces & Lambda Expressions

What are functional interfaces?

Functional Interfaces in Java 🎯

A **functional interface** in Java is an interface that has **exactly one abstract method** but can have **multiple default or static methods**. It is used primarily in **lambda expressions** and **method references**, making Java code more concise and readable.

✅ Key Features of Functional Interfaces

1. **Single Abstract Method (SAM)** → Can have only **one** abstract method.
2. **Can Have Default and Static Methods** → But only **one abstract method** is allowed.
3. **Used with Lambda Expressions** → Enables cleaner, functional-style programming.
4. **Automatically Annotated (@FunctionalInterface)** → This annotation is optional but ensures the interface follows the functional interface rule.
5. **Part of Java 8 Features** → Introduced in Java 8 for better functional programming support.

🚀 Built-in Functional Interfaces in Java (java.util.function Package)

Interface	Abstract Method	Description
Runnable	`void run()`	Represents a task to run in a thread.
Supplier<T>	`T get()`	Returns a value but takes no input.
Consumer<T>	`void accept(T t)`	Takes input but returns nothing.
Function<T, R>	`R apply(T t)`	Takes input of type `T` and returns type `R`.
Predicate<T>	`boolean test(T t)`	Tests a condition and returns `true` or `false`.

Why Use Functional Interfaces?

✅ Makes code **concise** and **readable**.
✅ Improves **code reusability** with lambda expressions.
✅ Essential for **streams, multithreading, and event handling**.
✅ Provides better **performance** in large-scale applications.

✅ Key Features of `Supplier<T>`

1. **No Input Parameter** → Unlike `Function<T, R>` or `Consumer<T>`, `Supplier<T>` **does not take** any arguments.
2. **Returns a Value** → It **produces** or **supplies** a result of type `T`.
3. **Used in Lazy Evaluation** → Value is computed **only when needed**.
4. **Common Use Cases** → Object factory, generating random numbers, fetching configuration values, database connections, etc.
5. **Functional Interface** → Can be implemented using **lambda expressions** or **method references**.

Supplier Interface with get()

```java
package com.soham.programs;

import java.util.Calendar;
import java.util.function.Supplier;

public class SupplierDemo {
    public static void main(String[] args) {
        Supplier<String> obj=()->{
            Calendar cal=Calendar.getInstance();
            return cal.getTime().toString();
        };

        System.out.println(obj.get());
    }

}
```

✅ Key Features of `Consumer<T>`

1. **Takes an Input, Returns Nothing** → The `accept(T t)` method processes an input but does **not return any value**.
2. **Belongs to `java.util.function` Package** → It is a built-in Java functional interface.
3. **Used for Side Effects** → Commonly used for logging, printing, modifying objects, etc.
4. **Supports Method Chaining** → The `andThen(Consumer<T> after)` method allows chaining multiple `Consumer` operations.
5. **Compatible with Lambda Expressions** → It can be implemented using **lambda expressions** for concise and readable code.

Consumer Interface with accept()

```java
package com.soham.programs;

import java.util.function.Consumer;

public class ConsumerDemo {
    public static void main(String[] args) {
        Consumer<Integer> obj=(Integer num)->{
            int sq=num*num;
            System.out.println("Square is "+sq);
        };

        obj.accept(13);
    }

}
```

✅ Key Features of `Function<T, R>`

1. **Takes One Input (T) and Returns One Output (R)** → Used for transforming data.
2. **Belongs to `java.util.function` Package** → It is a built-in functional interface.
3. **Supports Method Chaining** → The `andThen()` and `compose()` methods allow combining multiple functions.
4. **Compatible with Lambda Expressions** → Can be used with **lambda expressions** for concise coding.
5. **Commonly Used in Streams API** → Used in `map()`, `collect()`, and other transformation operations.

Function Interface with apply()

```
package com.soham.programs;

import java.util.function.Function;

public class FunctionDemo {
    public static void main(String[] args) {
        Function<Integer, String> obj=(Integer n)->{
            int sq=n*n;
            return "square is "+sq;
        };

        System.out.println(obj.apply(9));
    }

}
```

✅ Key Features of `Predicate<T>`

1. **Takes an Input, Returns a Boolean** → It evaluates a condition and returns **`true` or `false`**.
2. **Belongs to `java.util.function` Package** → It is a built-in Java functional interface.
3. **Can Be Used in Streams and Filtering** → Works well with **`filter()`** in streams.
4. **Supports Method Chaining** → Has methods like `and()`, `or()`, and `negate()` for combining multiple conditions.
5. **Compatible with Lambda Expressions** → Simplifies conditional logic using **lambda expressions**.

Predicate Interface with test()

```java
package com.soham.programs;

import java.util.function.Predicate;

public class PredicateDemo {
    public static void main(String[] args) {
        Predicate<String> obj=(String password)->{
            if(password.equals("chelsea"))
                return true;
            else
                return false;
        };

        System.out.println(obj.test("spider"));
    }

}
```

What are Lambda Expressions?

Lambda Expressions were introduced in **Java 8** to provide a **concise way to write anonymous functions**. They allow you to write **functional-style code** by **eliminating boilerplate code** for simple method implementations.

✅ Key Features of Lambda Expressions

1. **Eliminates Anonymous Classes** → Reduces unnecessary boilerplate code.
2. **Functional Programming Style** → Works well with functional interfaces.
3. **More Readable & Concise** → Shortens the code compared to traditional method definitions.
4. **Improves Performance** → Uses **lazy evaluation** and **functional programming paradigms**.
5. **Supports Stream API** → Used extensively in **Streams, Collections, and Multithreading**.

🎯 Types of Lambda Expressions

Type	Example
No Parameters	`() -> System.out.println("Hello World");`
Single Parameter	`(a) -> a * a;`
Multiple Parameters	`(a, b) -> a + b;`
Multiple Statements	`(a, b) -> { int sum = a + b; return sum; };`

What are Streams?

Streams in Java 🚀

Java **Streams** are a powerful feature introduced in **Java 8** as part of the `java.util.stream` package. They provide a **functional-style** way to process collections (like lists, sets, and maps) using a **pipeline of operations**. Streams help in writing **concise, readable, and efficient** code for data processing.

✅ Key Features of Streams

1. **Streams do not store data** → They only process elements from a source (e.g., List, Set, Array, etc.).
2. **Functional Programming** → Supports **lambda expressions** and **method references**.
3. **Internal Iteration** → Unlike traditional loops, streams manage iteration internally.
4. **Lazy Evaluation** → Streams do not execute intermediate operations until a **terminal operation** is invoked.
5. **Parallel Processing** → Supports **parallel streams** for faster execution on multi-core processors.
6. **Immutable & Non-Modifying** → Streams **do not modify the original data**; instead, they return a new stream.

🛠️ Stream Pipeline: How Streams Work?

A **Stream pipeline** consists of three parts:

1️⃣ **Source** → Collection, Array, File, etc.
2️⃣ **Intermediate Operations** (Transformations) → `filter()`, `map()`, `sorted()`, etc. (Lazy evaluation)
3️⃣ **Terminal Operation** (Final result) → `collect()`, `forEach()`, `count()`, etc.

1️⃣ Creating Streams

```
Stream<Integer> stream1 = Stream.of(1, 2, 3, 4, 5); // From values
Stream<String> stream2 = Arrays.stream(new String[]{"A", "B", "C"}); // From array
List<Integer> list = List.of(10, 20, 30);
Stream<Integer> stream3 = list.stream(); // From collection
```

2 Intermediate Operations (Lazy - Modify Stream)

Method	Description
`filter(Predicate<T>)`	Filters elements based on a condition.
`map(Function<T,R>)`	Transforms each element (e.g., convert to uppercase).
`sorted()`	Sorts elements in natural order.
`distinct()`	Removes duplicates.
`limit(n)`	Limits the number of elements.
`skip(n)`	Skips the first `n` elements.

3 Terminal Operations (Eager - Consume Stream)

Method	Description
`forEach(Consumer<T>)`	Performs an action on each element.
`collect(Collector<T>)`	Converts stream into List, Set, or Map.
`count()`	Returns the number of elements.
`reduce(BinaryOperator<T>)`	Reduces elements into a single value.
`allMatch(Predicate<T>)`	Returns `true` if all elements match a condition.
`anyMatch(Predicate<T>)`	Returns `true` if any element matches a condition.

🎯 Benefits of Streams

✅ **Less Code** → Functional-style reduces boilerplate.
✅ **Faster Processing** → Parallel streams utilize multi-core CPUs.
✅ **More Readable** → Chainable operations improve readability.
✅ **Immutable Operations** → Original data remains unchanged.

```
package com.sharayu.programs;

import java.util.ArrayList;
import java.util.Iterator;

public class BasicStreamOperation {
    public static void main(String[] args) {
        ArrayList<String> list=new ArrayList<String>();
        list.add("bombay");
        list.add("london");
        list.add("tokyo");
        list.add("berlin");
        list.add("dubai");

        /*
        for(int i=0;i<list.size();i++)
            System.out.println(list.get(i));


        Iterator<String> itr=list.iterator();
        while(itr.hasNext())
            System.out.println(itr.next());

            */

        //list.stream().forEach(nm->System.out.println(nm));


list.stream().filter(nm->nm.startsWith("b")).forEach(nm->System.out.println(nm));
        //filter is an intermediate operation
        //forEach is a terminal operation
    }

}
```

Intermediate Operations

```java
package com.sharayu.programs;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class IntermediateOperations {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();

        // Adding names, some with the same starting letter
        names.add("Sharayu");
        names.add("Praffull");
        names.add("Shailaja");
        names.add("Soham");
        names.add("Alice");
        names.add("Diana");
        names.add("Charles");
        names.add("Andrew");
        names.add("Bella");
        names.add("Catherine");
        names.add("Benjamin");
        names.add("Daniel");
        names.add("Ethan");
        names.add("Adam");


        //filter - retrieve elements on the basis of a condition

names.stream().filter(nm->nm.startsWith("S")).forEach(nm->System.out.println(
nm));
        System.out.println("---------");

names.stream().filter(nm->nm.length()>7).forEach(nm->System.out.println(nm));
```

```
        List<String>
Anm=names.stream().filter(nm->nm.startsWith("A")).collect(Collectors.toList()
);
        System.out.println(Anm);

        //map - transform all elements

names.stream().map(nm->nm.toUpperCase()).forEach(nm->System.out.println(nm));

        ArrayList<Integer> nums = new ArrayList<>();
        nums.add(9);
        nums.add(13);
        nums.add(9);
        nums.add(26);
        nums.add(13);
        nums.add(9);
        nums.add(1);
        nums.add(9);

        List<Integer> sqrs= nums.stream()
        .map(n->n*n)
        .collect(Collectors.toList());

        System.out.println(sqrs);
        System.out.println("after squares : "+nums);

        //sorted
        names.stream().sorted().forEach(nm->System.out.println(nm));

        //distinct
        nums.stream().distinct().forEach(n->System.out.println(n));
        //Set<Integer> uniques=nums.stream().collect(Collectors.toSet());
        //System.out.println(uniques);


        //limit - limit the number of elements
```

```
        System.out.println("-----limit--------");
        names.stream().limit(3).forEach(nm->System.out.println(nm));
        System.out.println("----sorted limit----");
        names.stream().sorted().limit(3).forEach(nm->System.out.println(nm));

        //skip - skip first N elements
        System.out.println("----skip----------");
        names.stream().skip(2).forEach(nm->System.out.println(nm));
        System.out.println("----limit skip -----------");
        names.stream().skip(2).limit(1).forEach(nm->System.out.println(nm));
        System.out.println("Rank second");

names.stream().sorted().skip(1).limit(1).forEach(nm->System.out.println(nm));
    }
}
```

Terminal Operations

```
package com.sharayu.programs;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class TerminalOperations {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();

        // Adding names, some with the same starting letter
        names.add("Sharayu");
        names.add("Praffull");
        names.add("Shailaja");
        names.add("Soham");
        names.add("Alice");
        names.add("Diana");
        names.add("Charles");
```

```java
        names.add("Andrew");
        names.add("Bella");
        names.add("Catherine");
        names.add("Benjamin");
        names.add("Daniel");
        names.add("Ethan");
        names.add("Adam");

        //forEach
        names.stream().forEach(nm->System.out.println(nm));
        //System.out.println("----------");
        //names.stream().forEach(System.out::println);

        //collect
        List<String> res=names.stream().collect(Collectors.toList());
        Set<String> set=names.stream().collect(Collectors.toSet());

        //count
        //long cnt=names.stream().count();
        long cnt=names.stream().filter(nm->nm.length()>7).count();
        System.out.println("number of elements : "+cnt);

        //anyMatch()
        boolean stat=names.stream().anyMatch(nm->nm.startsWith("P"));
        System.out.println(stat);

        //reduce
        List<Integer> numbers=List.of(10,20,30,40,50);
        int result=numbers.stream().reduce(0, Integer::sum);
        System.out.println("Sum of elements : "+result);

        result=numbers.stream().reduce(0, Integer::max);
        System.out.println("largest value : "+result);

    }

}
```

Sorting of collections

What is a Comparator in Java?

In Java, **`Comparator<T>`** is an interface in the `java.util` package used to define custom sorting logic for objects. It provides a way to **compare two objects of a specific type** and determine their order.

Why Use Comparator?

- When you need **custom sorting logic** (e.g., sorting employees by salary instead of name).
- When the **natural ordering (Comparable)** of a class is not suitable.
- When sorting **third-party classes** where you can't modify their source code.

{SohamGlobal & Spider Projects One}

How Does It Work?

The **`Comparator<T>`** interface has a single method:

```
int compare(T o1, T o2);
```

- **Returns `-1` (or any negative number)** if `o1` should come before `o2`.
- **Returns `1` (or any positive number)** if `o1` should come after `o2`.
- **Returns `0`** if `o1` and `o2` are considered equal.

Film.java

```java
package com.sharayu.classes;

public class Film {

    private String filmName;
    private int releaseYear;
    private String language;
    private String genre;
    private double rating;
```

```
    public Film(String filmName, int releaseYear, String language, String
genre, double rating) {
        super();
        this.filmName = filmName;
        this.releaseYear = releaseYear;
        this.language = language;
        this.genre = genre;
        this.rating = rating;
    }

    @Override
    public String toString() {
        return "Film [filmName=" + filmName + ", releaseYear=" + releaseYear
+ ", language=" + language + ", genre="
                + genre + ", rating=" + rating + "]";
    }

    public String getFilmName() {
        return filmName;
    }

    public int getReleaseYear() {
        return releaseYear;
    }

    public String getLanguage() {
        return language;
    }

    public String getGenre() {
        return genre;
    }

    public double getRating() {
        return rating;
    }
}
```

NameComparator.java

```
package com.sharayu.classes;

import java.util.Comparator;

public class NameComparator implements Comparator<Film>{

    @Override
    public int compare(Film o1, Film o2) {
        // TODO Auto-generated method stub
        return o1.getFilmName().compareTo(o2.getFilmName());
    }

}
```

YearComparator.java

```
package com.sharayu.classes;

import java.util.Comparator;

public class YearComparator implements Comparator<Film> {

    @Override
    public int compare(Film o1, Film o2) {
        // TODO Auto-generated method stub
        return o1.getReleaseYear()-o2.getReleaseYear();
    }

}
```

RatingComparator.java

```
package com.sharayu.classes;

import java.util.Comparator;

public class RatingComparator implements Comparator<Film> {

    @Override
    public int compare(Film o1, Film o2) {
        // TODO Auto-generated method stub
        //return o1.getRating()-o2.getRating();
        return Double.compare(o1.getRating(), o2.getRating());
    }

}
```

ArrayListOfFilms.java

```
package com.praffull.programs;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import com.sharayu.classes.*;

public class ArrayListOfFilms {
    public static void main(String[] args) {
        List<Film> filmlist=new ArrayList<Film>();

        Film film1 = new Film("Sholay", 1975, "Hindi", "Action", 9.2);
        Film film2 = new Film("Inception", 2010, "English", "Sci-Fi", 8.8);
        Film film3 = new Film("3 Idiots", 2009, "Hindi", "Comedy-Drama",
8.4);
        Film film4 = new Film("Titanic", 1997, "English", "Romance", 7.8);
```

```
        Film film5 = new Film("Dangal", 2016, "Hindi", "Biographical", 8.5);

        filmlist.add(film1);
        filmlist.add(film2);
        filmlist.add(film3);
        filmlist.add(film4);
        filmlist.add(film5);

        System.out.println(filmlist);

        Collections.sort(filmlist, new YearComparator());
        System.out.println(filmlist);

        Collections.sort(filmlist,new NameComparator());
        System.out.println(filmlist);

        Collections.sort(filmlist,new RatingComparator());
        System.out.println(filmlist);
    }

}
```

THANK YOU
FOR YOUR SUPPORT

Java is world's #1 development platform and is at the heart of our digital lifestyle.

At SohamGlobal, we have been using Java for **more than 20** years.

CONTACT US FOR TRAINING, WORKSHOPS AND PROJECTS DEVELOPMENT

Instagram: sohamglobal.praffull