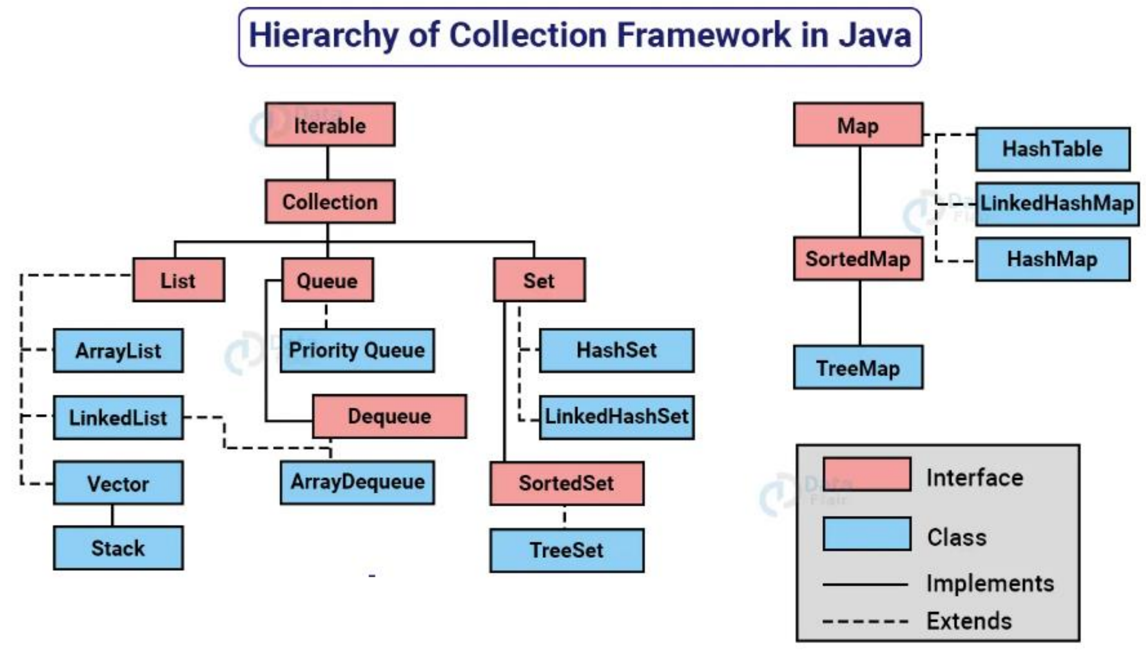
Collection



What is collection

In Java, a **Collection** is a **framework** that provides a **standardized way to store, access, and manipulate groups of objects**. A **Collection** is an **object** that represents a group of elements, such as a list of names, a set of unique IDs, or a queue of tasks.

List

Maintains **insertion order**
Allows **null elements**
Supports **positional access** (get, set, remove by index)
Allows **duplicate elements**

ArrayList

Maintains **insertion order**
Allows **duplicate and null** elements
Supports **random access** using index (get(int index))
Not synchronized (not thread-safe by default)
ArrayList can **grow or shrink dynamically** as elements are added or removed.

Internal Working of ArrayList

Under the hood, ArrayList uses a **dynamic array** (i.e., an internal array like Object[] elementData) to store elements.

Basic Steps:

1. When you create an ArrayList, it creates an **empty array** or a **default-sized array** (default capacity is 10 if using new ArrayList<>()).

2. When elements are added and the internal array becomes **full**, it **automatically resizes** (usually increases by **50%** of the current size).
3. When you remove elements, it doesn't shrink immediately but leaves space to avoid frequent resizing.

Initially:

- Capacity = 10
- When 11th item is added → capacity increases to 15 (10 + 10/2)

🔶 What is LinkedList in Java?

LinkedList is a class in Java's **Collections Framework** that implements both the **List** and **Deque** interfaces. It is a **doubly linked list** which allows insertion and removal from both ends (head and tail).

✅ Key Features:

- Maintains **insertion order**
- Allows **null elements**
- Allows **duplicate elements**
- Faster at **insertion/deletion** compared to ArrayList (especially in the middle)
- Slower at **accessing elements by index** (no random access)

🔧 Internal Working of LinkedList:

Unlike ArrayList (which uses an internal array), LinkedList uses a **doubly linked list** structure:

- Each **node** contains:
  - data
  - a reference to the **next node**
  - a reference to the **previous node**

The LinkedList maintains two pointers:

- first – head of the list
- last – tail of the list

▯ Structure:

null <- [10] <-> [20] <-> [30] -> null

🔁 Internal Operations:

- **Add element**:
  - At end: link new node after last
  - At beginning: link before first
- **Remove element**:
  - Adjust pointers of neighboring nodes
- **Access by index**:
  - Traverse from head (or tail if index > size/2)

📌 No Capacity in LinkedList

Unlike ArrayList, which uses a **backing array** and has a **capacity**, LinkedList has **no fixed capacity** — it grows and shrinks as elements are added or removed.

♦️ What is Vector in Java?

Vector is a **legacy class** in Java that implements the **List interface** and provides a **resizable array**, just like ArrayList, but with one major difference:

🔒 Vector is **synchronized**, meaning it is thread-safe by default.

✅ Key Features of Vector:

- Maintains **insertion order**
- Allows **null and duplicate** elements
- **Synchronised** (thread-safe)
- Slower than ArrayList in **single-threaded** environments due to synchronization overhead
- Grows dynamically like ArrayList, but with a **different growth strategy**

🔍 Internal Working of Vector:

- Internally uses an **array** (Object[] elementData) to store elements.
- The array is **dynamically resized** when it becomes full.

📏 Capacity and Resizing:

- **Initial capacity**: Default is **10**.
- **Growth strategy**: When full, capacity increases by **100%** (doubles the current size).
    - This differs from ArrayList, which increases capacity by **50%**.

♦️ What is Stack in Java?

Stack is a **legacy class** in Java that represents a **Last-In-First-Out (LIFO)** data structure — the last element added is the first to be removed.
It extends the Vector class and is part of java.util.

✅ Key Features of Stack:

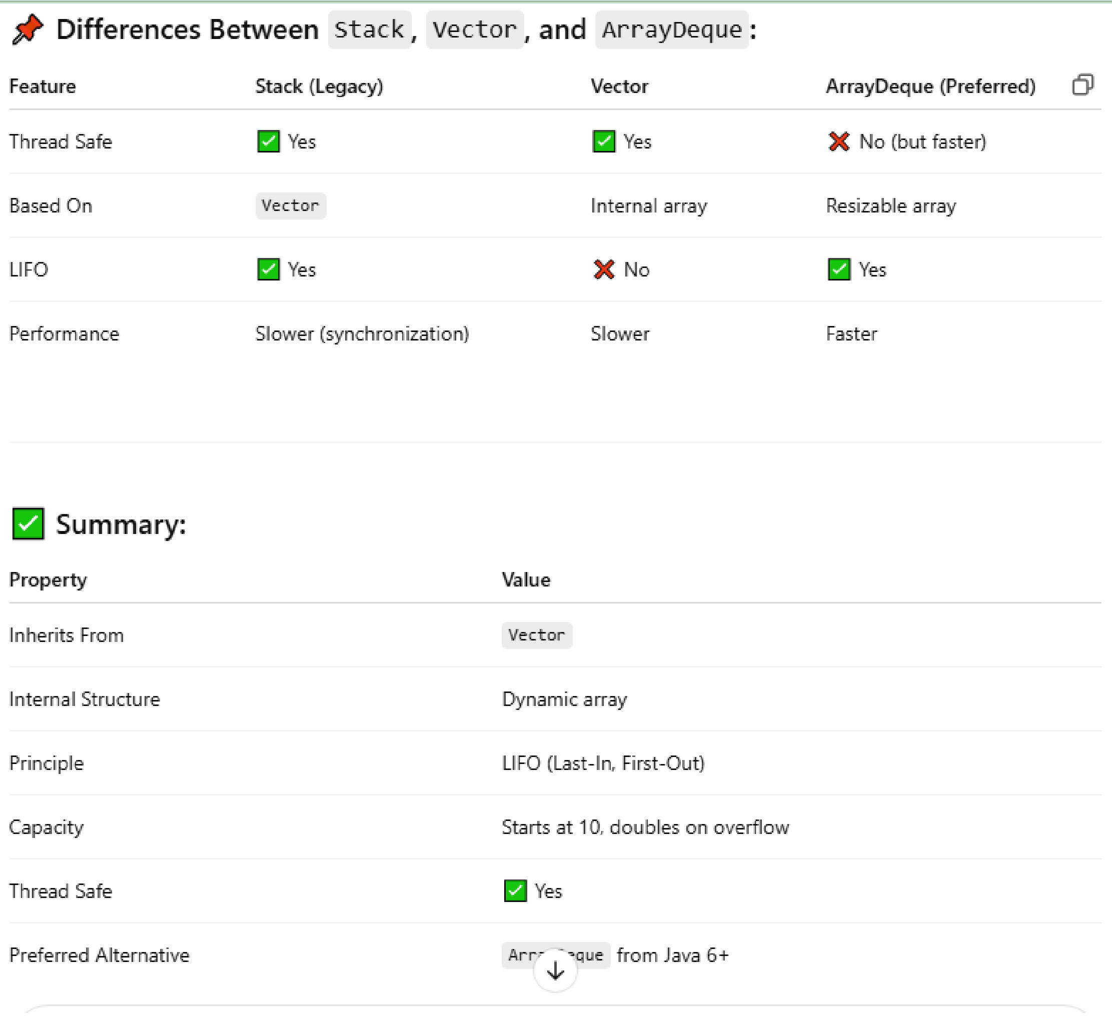
- Inherits all methods from Vector
- Operates on **LIFO principle**
- Thread-safe (because Vector is synchronized)
- Not recommended for new code (prefer Deque like ArrayDeque)

▯ Internal Working of Stack:

Since Stack extends Vector, it uses an **internal array** (Object[] elementData) to store elements and inherits all resizing behavior from Vector.

📏 Capacity and Resizing:

- Initial capacity: **10** (inherited from Vector)
- When full, capacity **doubles** (100% growth strategy)
- Capacity is the size of the underlying array — **not limited** (can grow as needed)

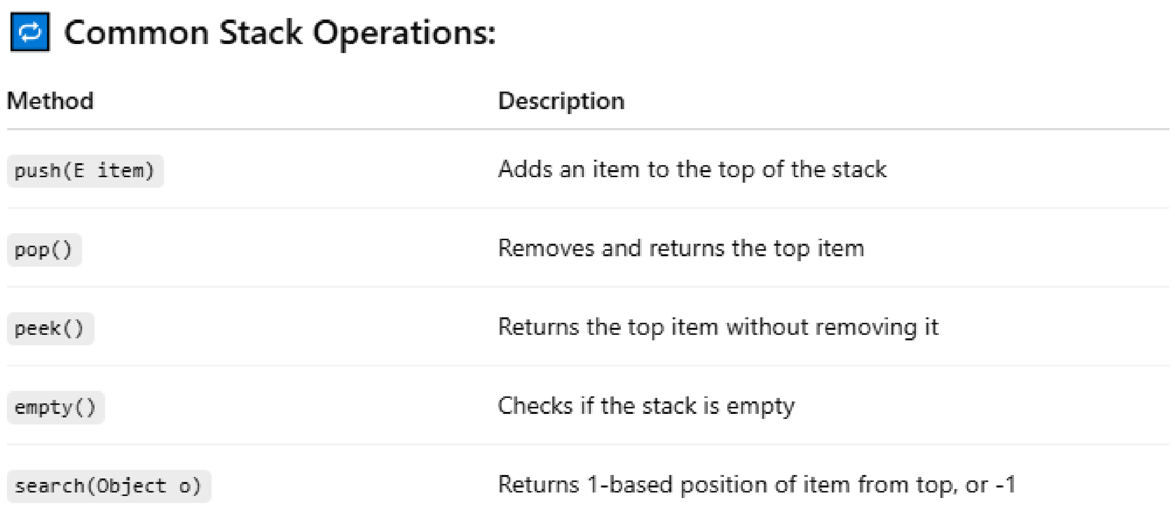


📌 Differences Between `Stack`, `Vector`, and `ArrayDeque`:

Feature	Stack (Legacy)	Vector	ArrayDeque (Preferred)
Thread Safe	✅ Yes	✅ Yes	❌ No (but faster)
Based On	`Vector`	Internal array	Resizable array
LIFO	✅ Yes	❌ No	✅ Yes
Performance	Slower (synchronization)	Slower	Faster

✅ Summary:

Property	Value
Inherits From	`Vector`
Internal Structure	Dynamic array
Principle	LIFO (Last-In, First-Out)
Capacity	Starts at 10, doubles on overflow
Thread Safe	✅ Yes
Preferred Alternative	`ArrayDeque` from Java 6+



🔁 Common Stack Operations:

Method	Description
`push(E item)`	Adds an item to the top of the stack
`pop()`	Removes and returns the top item
`peek()`	Returns the top item without removing it
`empty()`	Checks if the stack is empty
`search(Object o)`	Returns 1-based position of item from top, or -1

🔶 What is Set Interface in Java Collections?

The **Set interface** in Java is a part of the **Java Collections Framework** and represents a **collection of unique elements — duplicates are not allowed**.

🔍 **Key property**: A Set **does not allow duplicate elements**.

✅ **Key Features of Set Interface:**

- No duplicate elements
- No defined order (in basic Set)
- Can contain null (at most one null element, depending on implementation)
- Supports basic collection operations: add, remove, contains, size, iterator

🔄 Set Implementations:

Implementation	Ordering	Null Allowed	Sorted?	Thread-safe
HashSet	No order (unordered)	✅ Yes (one)	❌ No	❌ No
LinkedHashSet	Insertion order	✅ Yes (one)	❌ No	❌ No
TreeSet	Sorted (natural or custom)	❌ No (null not allowed in most cases)	✅ Yes	❌ No

❗ Set vs List:

Feature	Set	List
Allows Duplicates	❌ No	✅ Yes
Maintains Order	Depends on implementation	✅ Yes
Access by Index	❌ No	✅ Yes
Examples	`HashSet`, `TreeSet`	`ArrayList`, `LinkedList`

♦ What is HashSet in Java?

HashSet is a class in Java that implements the Set interface and **stores unique elements only**. It is backed by a **HashMap**, meaning it uses **hashing** to store elements.

✅ Key Features of `HashSet`:

Feature	Description
Duplicates	❌ Not allowed
Null	✅ Allows a single `null` element
Order	❌ Unordered (no insertion or sorted order)
Thread-safe	❌ Not synchronized
Performance	🔄 Constant-time (`O(1)`) for `add`, `remove`, `contains` in average case

▯ Internal Working of HashSet:

- Internally, HashSet uses a **HashMap**.
- When you add an element to a HashSet, it is added as a **key** to the HashMap with a constant dummy value (e.g., PRESENT = new Object()).

```
private transient HashMap<E,Object> map;
private static final Object PRESENT = new Object();
public boolean add(E e) {
   return map.put(e, PRESENT) == null;
}
```

So every element you add to HashSet is actually stored as a key in the underlying HashMap.

📏 Capacity and Load Factor:

- **Initial capacity**: **16** (default)
- **Load factor**: **0.75** (default)
- **Threshold** = capacity * loadFactor
  → When this threshold is exceeded, the internal **array is resized (doubled)**.

Example:

- Initial capacity = 16
- Load factor = 0.75
- Threshold = 16 × 0.75 = 12
  → When 13th element is added, it resizes the backing array.

🔍 Internal Steps When Adding an Element:

1. Compute **hash code** of the object.
2. Apply **hash function** to determine the bucket index.
3. Check if any existing object is in that bucket:
   - If yes: Use equals() to detect duplicates.
   - If no: Insert new entry.
4. If size exceeds threshold → **resize the internal array**.

⚖️ Time Complexity:

Operation	**Time Complexity**
add(E e)	O(1) avg, O(n) worst
remove(E e)	O(1) avg
contains(E e)	O(1) avg

⚠ Notes:

- **Hashing + equals()** determine uniqueness.
- To store custom objects in HashSet, override hashCode() and equals() methods.

What is LinkedHashSet in Java?

LinkedHashSet is a **child class of HashSet** that maintains **insertion order** of elements. It is a part of the Java Collections Framework and implements the Set interface.

✅ **Key Difference from HashSet**: LinkedHashSet preserves the **order** in which elements were inserted.

✅ **Key Features of LinkedHashSet:**

Feature	**Description**
Duplicate Elements	❌ Not allowed
Insertion Order	✅ Maintained
Null Element	✅ One null allowed
Backed By	**LinkedHashMap**
Thread-safe	❌ No
Performance	Similar to HashSet (slightly slower due to maintaining order)

▯ Internal Working of LinkedHashSet:

- Internally uses a **LinkedHashMap** to store elements.
- Elements are stored as **keys** in the map, and all values are a dummy constant (like PRESENT = new Object()).
- A **doubly linked list** is used to maintain the **insertion order** of elements.

Internal Representation:

```
private transient LinkedHashMap<E, Object> map;
private static final Object PRESENT = new Object();
public boolean add(E e) {
   return map.put(e, PRESENT) == null;
}
```

When an element is added, it is stored in LinkedHashMap as a **key**, and the value is always a constant dummy object.

📏 **Capacity and Load Factor:**

- **Initial capacity**: 16 (default, same as HashMap)
- **Load factor**: 0.75 (default)
- **Threshold** = capacity × load factor
  → When the threshold is crossed, the internal map resizes.

🔁 **How Insertion Order is Maintained:**

Each entry in LinkedHashMap (used internally) has:

- A key
- A pointer to the next and previous entry (forming a doubly-linked list)

This allows iteration over the elements in the **order they were inserted**.

⏱ **What is Time Complexity?**

Time complexity is a way to describe how the **runtime of an algorithm** increases as the size of the input (**n**) increases.
It helps us analyze the **efficiency** of an algorithm, especially for large inputs.

✅ **Why is it Important?**

- To **compare** different algorithms.
- To **predict** how long an algorithm will take.
- To **optimize** code performance.

:

🎄 **What is TreeSet in Java?**

TreeSet is a class in Java that implements the **NavigableSet** interface (which extends SortedSet) and stores **unique elements in **sorted (ascending)** order.

✅ **Key Features**:

- Stores **only unique elements**
- Maintains elements in **sorted (natural or custom) order**
- Internally uses a **Red-Black Tree (Self-Balancing Binary Search Tree)**

▯ **Internal Working of TreeSet:**

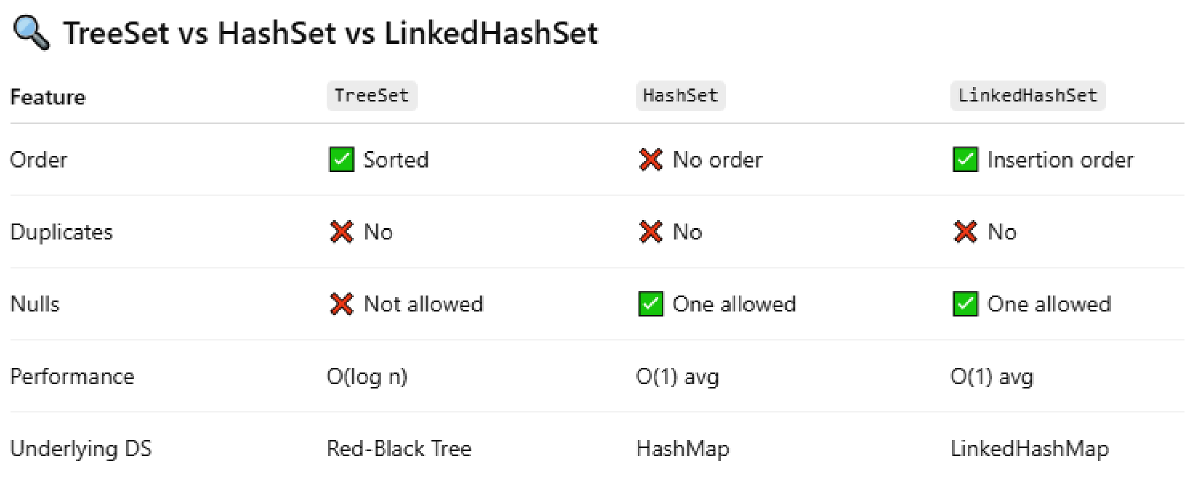
- Internally uses a **TreeMap** to store elements.
- All elements are stored as **keys** in the TreeMap.
- The sorting is done using:
  - **Natural ordering** (if elements implement Comparable)
  - Or a **custom Comparator** passed to the constructor

📐 **Sorting Rules:**

- **Natural Order**: Ascending (e.g., numbers from low to high, strings in dictionary order)
- **Custom Order**: You can pass a Comparator to sort based on your logic

📏 Capacity:

No initial capacity like ArrayList or HashSet.
TreeSet is dynamically resized as elements are added.
Internally relies on the Red-Black Tree structure which automatically balances itself.



🔍 TreeSet vs HashSet vs LinkedHashSet

Feature	`TreeSet`	`HashSet`	`LinkedHashSet`
Order	✅ Sorted	❌ No order	✅ Insertion order
Duplicates	❌ No	❌ No	❌ No
Nulls	❌ Not allowed	✅ One allowed	✅ One allowed
Performance	O(log n)	O(1) avg	O(1) avg
Underlying DS	Red-Black Tree	HashMap	LinkedHashMap

🎯 What is a Queue in Java?

A **Queue** is a **linear data structure** that follows the **FIFO** (First-In-First-Out) principle — the element added **first** is removed **first**.
It's like a **real-life queue**: the person who comes first, gets served first.

☑ Key Features of Queue Interface:

Feature	**Description**
Order	Maintains insertion order
Duplicates	☑ Allowed
Null Elements	✗ Not allowed (in some implementations)
Thread-safe	✗ Usually not (unless using ConcurrentLinkedQueue)

▯ Common Methods:

Method	**Description**
add(e)	Inserts element. Throws exception if full.
offer(e)	Inserts element. Returns false if full.
remove()	Removes and returns head. Throws exception if empty.
poll()	Removes and returns head. Returns null if empty.
element()	Returns head. Throws exception if empty.
peek()	Returns head. Returns null if empty.

🛠 Implementations of Queue in Java:

Class	**Description**
LinkedList	Implements both List and Queue
PriorityQueue	Elements are ordered based on priority (not FIFO)
ArrayDeque	Resizable array, efficient for queues

Class	Description
ConcurrentLinkedQueue	Thread-safe non-blocking queue
BlockingQueue (interface)	Used in concurrent programming

📌 Queue vs Stack

Feature	Queue	Stack
Order	FIFO	LIFO
Add	offer()	push()
Remove	poll()	pop()

✅ Summary

- A **Queue** stores elements in **FIFO order**
- Provides methods to **add**, **peek**, and **remove**
- Java provides multiple **implementations** based on different use cases (LinkedList, PriorityQueue, etc.)

‼ What is PriorityQueue in Java?

PriorityQueue is a class in Java that implements the **Queue interface**, but unlike regular queues (FIFO), it **orders elements based on priority** — not insertion order.

✅ The **head of the queue** is always the **least** (or highest priority) element based on **natural ordering** or a **custom comparator**.

✅ Key Features of PriorityQueue:

Feature	Description
Ordering	Elements are ordered by **priority**, not insertion
Nulls	❌ null elements **not allowed**
Duplicates	✅ Allowed
Thread-safe	❌ Not synchronized
Data Structure	Internally uses a **Min-Heap (binary heap)**
Time Complexity	O(log n) for add/remove

▯ Internal Working:

- PriorityQueue uses a **binary heap** (a complete binary tree stored in an array).
- The **root** (index 0) always contains the **smallest** element (by default).
- When you add or remove elements:
  - The heap **rebalances itself** using **heapify** operations (log n time).

📋 Common Methods:

Method	Description
add(e) / offer(e)	Adds element and reorders
peek()	Returns the head (smallest / highest priority)
poll()	Removes and returns the head
remove(e)	Removes a specific element
contains(e)	Checks if element exists

⏱ **Time Complexity:**

Operation	Time
add() / offer()	O(log n)
poll() / remove()	O(log n)
peek()	O(1)

🔁 **Comparison with Other Queues:**

Feature	Queue	PriorityQueue
Order	FIFO	Based on priority
Duplicates allowed	✅ Yes	✅ Yes
Null allowed	⚠ Sometimes	❌ No

- PriorityQueue is a **heap-based** queue where the **highest priority element is always at the front**.
- It is **not FIFO**.
- Used when you want to **process elements by priority**, not by insertion order.

🔁 What is Deque in Java?

A **Deque** (pronounced *"deck"*) stands for **Double-Ended Queue** — a **linear collection** that allows insertion and removal of elements **at both ends** (front and rear).

✅ Unlike a normal Queue (FIFO), a Deque can function as both **Queue** (FIFO) and **Stack** (LIFO).

Deque is a subinterface of Queue and is part of the java.util package:

public interface Deque<E> extends Queue<E>

📌 Key Methods in `Deque`:

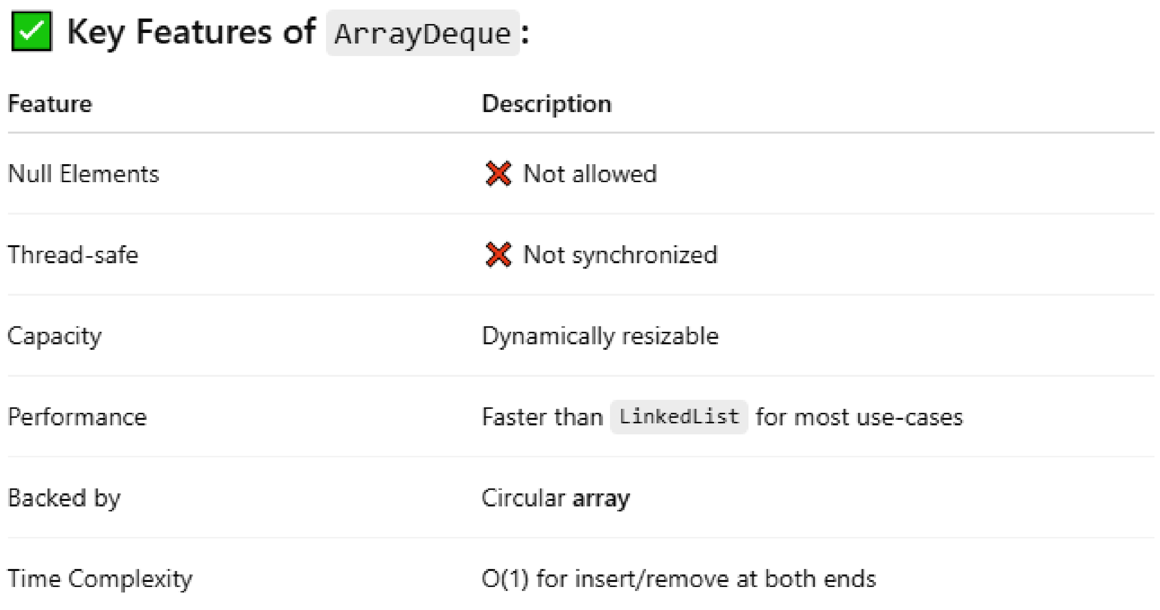
Operation	Front (Head)	Rear (Tail)
Add	`addFirst(e)`, `offerFirst(e)`	`addLast(e)`, `offerLast(e)`
Remove	`removeFirst()`, `pollFirst()`	`removeLast()`, `pollLast()`
Peek	`getFirst()`, `peekFirst()`	`getLast()`, `peekLast()`

☑ **Why Use Deque?**

- Acts as both **Queue** and **Stack**
- Efficient O(1) insert/delete from both ends
- Flexible for many algorithms (e.g., sliding window, undo/redo operations)

📦 What is ArrayDeque?

ArrayDeque is a **resizable array-based** implementation of the Deque interface.
It is faster than Stack and LinkedList for stack and queue operations.

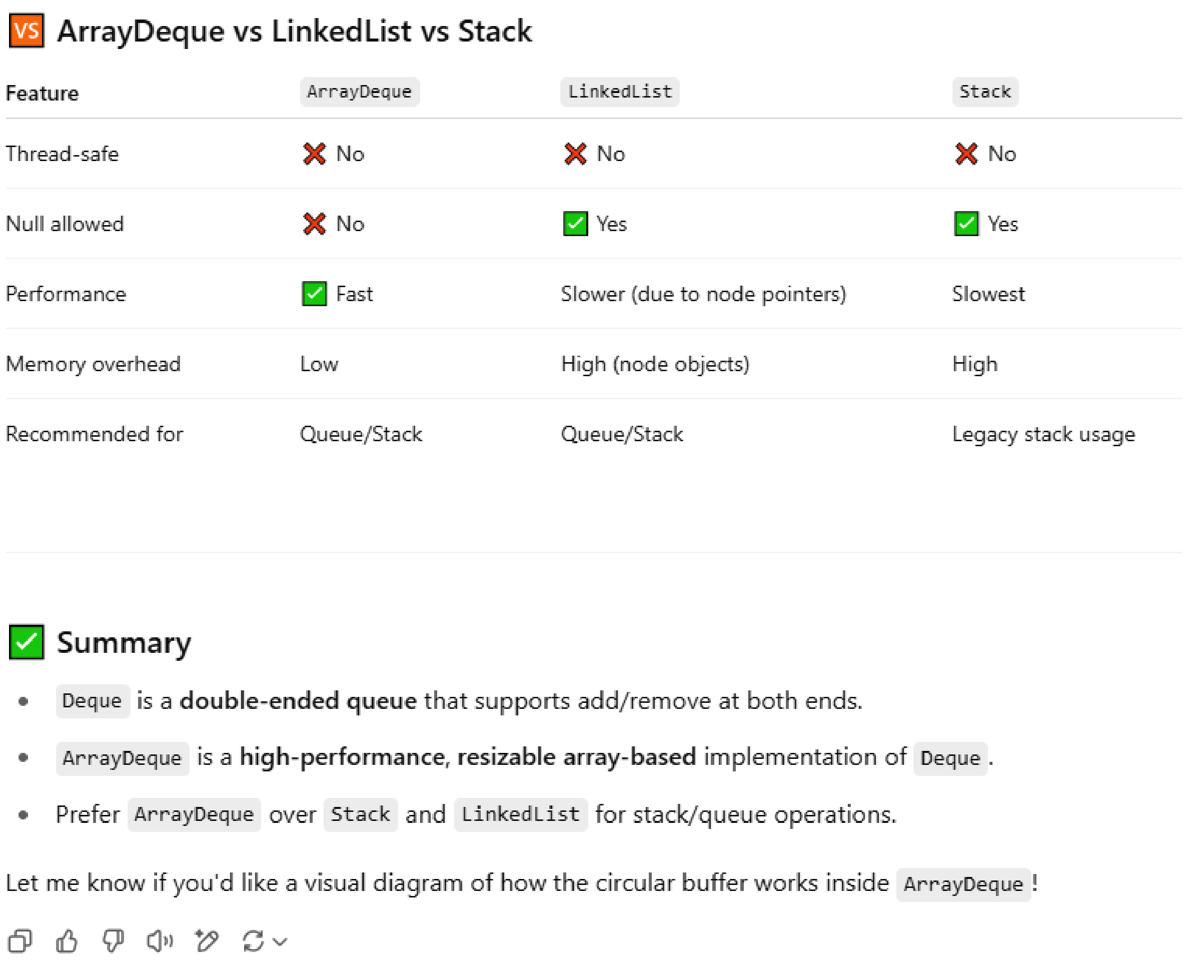


✅ Key Features of `ArrayDeque`:

Feature	Description
Null Elements	❌ Not allowed
Thread-safe	❌ Not synchronized
Capacity	Dynamically resizable
Performance	Faster than `LinkedList` for most use-cases
Backed by	Circular **array**
Time Complexity	O(1) for insert/remove at both ends

▯ **Internal Working:**

- Uses a **circular array** to store elements.
- Automatically **resizes** when full.
- Indexes wrap around using **modulo arithmetic** to efficiently utilize array space.



🆚 ArrayDeque vs LinkedList vs Stack

Feature	`ArrayDeque`	`LinkedList`	`Stack`
Thread-safe	❌ No	❌ No	❌ No
Null allowed	❌ No	✅ Yes	✅ Yes
Performance	✅ Fast	Slower (due to node pointers)	Slowest
Memory overhead	Low	High (node objects)	High
Recommended for	Queue/Stack	Queue/Stack	Legacy stack usage

✅ Summary

- `Deque` is a **double-ended queue** that supports add/remove at both ends.
- `ArrayDeque` is a **high-performance, resizable array-based** implementation of `Deque`.
- Prefer `ArrayDeque` over `Stack` and `LinkedList` for stack/queue operations.

Let me know if you'd like a visual diagram of how the circular buffer works inside `ArrayDeque`!

🗺️ What is the Map Interface in Java?

The **Map interface** in Java is part of the **Java Collections Framework** and represents a **collection of key-value pairs**, where **each key is unique** and **maps to exactly one value**.
Think of a Map like a **dictionary**: each word (key) has a meaning (value).

✅ Key Points about Map:

Feature	Description
Key uniqueness	✅ Each key must be **unique**
Value uniqueness	❌ Values **can be duplicated**
Allows null keys/values	Varies by implementation (e.g., HashMap allows 1 null key)
Not part of Collection	🚫 Map is **not a subtype** of Collection

📦 Common Implementations of `Map`:

Class	Ordering	Thread-Safe	Nulls Allowed
`HashMap`	❌ No order	❌ No	✅ 1 null key, many null values
`LinkedHashMap`	✅ Insertion order	❌ No	✅ Yes
`TreeMap`	✅ Sorted by keys	❌ No	❌ No null keys
`Hashtable`	❌ No order	✅ Yes	❌ No null key/value
`ConcurrentHashMap`	❌ No order	✅ Yes (concurrent)	❌ No nulls

🔑 Important Methods in Map Interface

Method	Description
`put(K key, V value)`	Adds a key-value pair
`get(Object key)`	Retrieves value for a key
`remove(Object key)`	Removes the mapping for the key
`containsKey(Object key)`	Checks if key exists
`containsValue(Object val)`	Checks if value exists
`keySet()`	Returns a Set of keys
`values()`	Returns a Collection of values
`entrySet()`	Returns a Set of key-value pairs (Map.Entry)

⚖️ Map vs Collection

Feature	`Map`	`Collection`
Stores	Key-Value pairs	Single values (elements)
Key uniqueness	✅ Required	❌ Not applicable
Value uniqueness	❌ Not required	Depends on collection
Allows nulls	Depends on impl.	Depends on impl.

✅ Summary

- `Map` stores **key-value** pairs where **keys are unique.**
- It is **not** a subtype of `Collection`.
- Popular implementations: `HashMap`, `LinkedHashMap`, `TreeMap`, and `Hashtable`.

What is HashMap in Java?

HashMap is a part of Java's java.util package and is the most commonly used implementation of the Map interface. It **stores data as key-value pairs**, and allows **one null key** and **multiple null values**.

✅ Key Features of `HashMap`:

Feature	Description
Key Uniqueness	Keys must be unique
Null Allowed	✅ One `null` key, many `null` values
Ordering	❌ No guaranteed order
Thread-safe	❌ Not synchronized (use `ConcurrentHashMap` for thread-safety)
Load Factor	Default: `0.75` (resize threshold)
Initial Capacity	Default: `16`
Time Complexity	O(1) on average for `put()` and `get()`

🔧 Internal Working of HashMap:

A HashMap works internally using:

1. **Hashing**: Each key is hashed using hashCode().

2. **Bucket Index**: The hash is transformed to a bucket index.
3. **Array + Linked List / Tree**:
    - It stores key-value pairs in **buckets**.
    - Each bucket can hold:
        - A **linked list** (for hash collisions)
        - A **balanced tree** (if too many collisions – Java 8+)

🧱 Structure Internally:

- HashMap uses an array of **Node<K, V>**.
- Each element in the array is called a **bucket**.
- Each bucket is a **linked list or tree** of entries with same hash index.

Java 8+: Tree-Based Buckets

- If a bucket has **more than 8 entries**, it becomes a **balanced tree** to improve performance (from O(n) to O(log n)).
- Converts back to a list if entries drop below 6.

🚫 Collisions in HashMap

When two keys have the same hash index, they go into the **same bucket** — this is called a **collision**.

- Earlier: used Linked List to store collided entries → O(n)
- Java 8+: uses Tree (Red-Black Tree) when entries > 8 → O(log n)

⚠ Note on Thread Safety

- HashMap is **not thread-safe**.
- Use Collections.synchronizedMap() or ConcurrentHashMap for thread safety.

✅ Summary

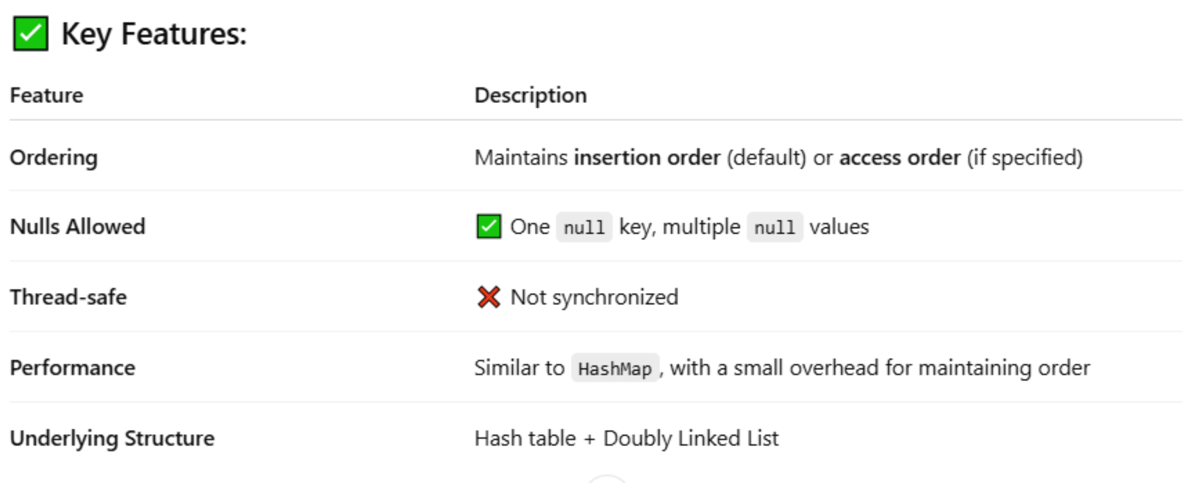
- HashMap stores key-value pairs using **hashing**.
- Average time complexity for put(), get(), remove() is **O(1)**.
- Internally uses **array + linked list/tree**.
- Automatically **resizes** when load factor threshold is crossed.

🔁 Resizing Behavior:

- When the number of entries exceeds **12**, the HashMap **doubles** its capacity.
- **New capacity = 32**, and the entries are **rehashed** and **redistributed** across the new bucket array.

▯ What is LinkedHashMap in Java?

LinkedHashMap is a subclass of HashMap that **maintains insertion order** or **access order** of entries. It is part of the java.util package and implements the Map interface.



✅ Key Features:

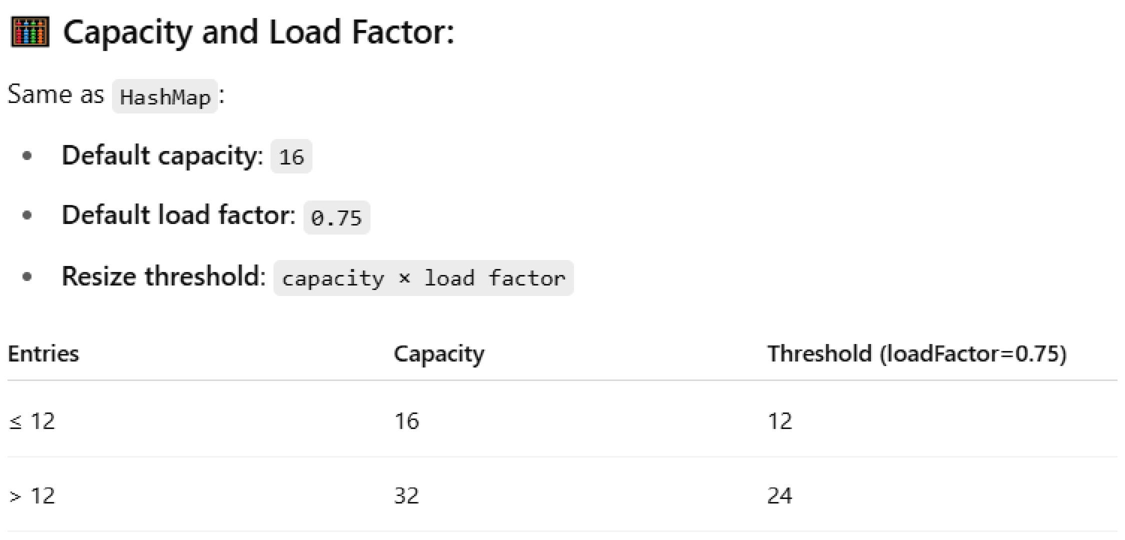
Feature	Description
Ordering	Maintains **insertion order** (default) or **access order** (if specified)
Nulls Allowed	✅ One `null` key, multiple `null` values
Thread-safe	❌ Not synchronized
Performance	Similar to `HashMap`, with a small overhead for maintaining order
Underlying Structure	Hash table + Doubly Linked List

🔧 Internal Working:

LinkedHashMap extends HashMap and adds a **doubly-linked list** to track the **order** of entries.
Each entry in LinkedHashMap is a special node:

```
static class Entry<K,V> extends HashMap.Node<K,V> {
   Entry<K,V> before, after; // for insertion/access order
}
```

□ before and after pointers form a **doubly-linked list**.
□ The linked list tracks **insertion order** or **access order**, depending on the constructor.



🧮 Capacity and Load Factor:

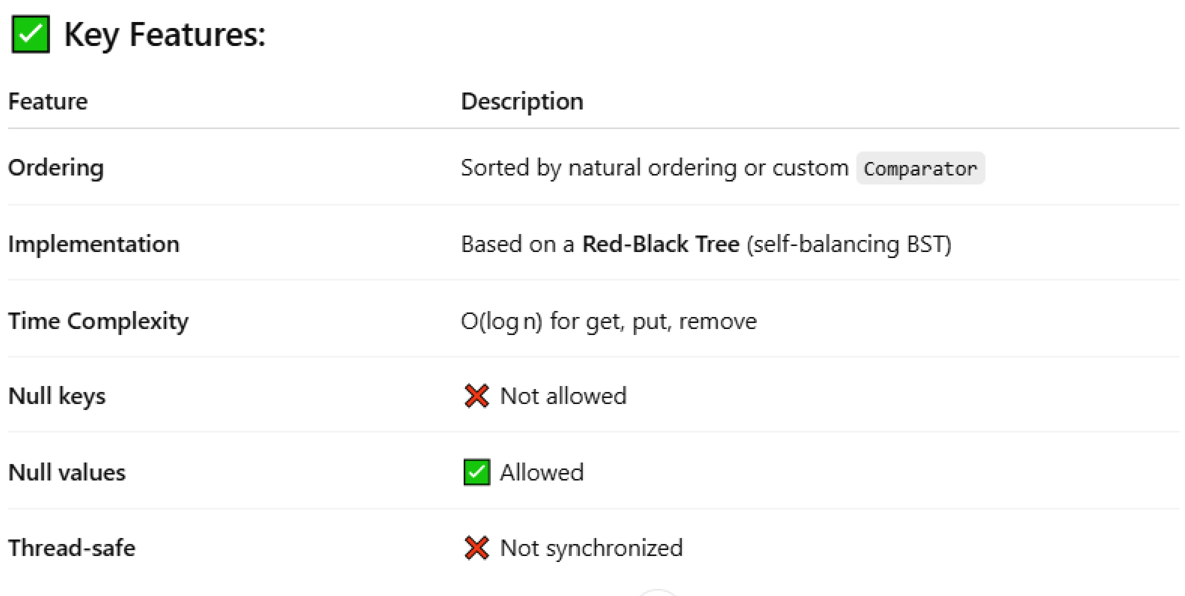
Same as `HashMap`:

- **Default capacity:** `16`
- **Default load factor:** `0.75`
- **Resize threshold:** `capacity × load factor`

Entries	Capacity	Threshold (loadFactor=0.75)
≤ 12	16	12
> 12	32	24

🌳 What is TreeMap in Java?

TreeMap is a class in Java's java.util package that implements the NavigableMap interface and stores key-value pairs **in sorted (ascending) order of keys**. It is **not synchronized** and **does not allow null keys** (but allows multiple null values).



✅ Key Features:

Feature	Description
Ordering	Sorted by natural ordering or custom `Comparator`
Implementation	Based on a **Red-Black Tree** (self-balancing BST)
Time Complexity	O(log n) for get, put, remove
Null keys	❌ Not allowed
Null values	✅ Allowed
Thread-safe	❌ Not synchronized

🔧 **Internal Working:**

TreeMap uses a **Red-Black Tree**, which is a type of self-balancing binary search tree. Keys are stored in sorted order.

Each node maintains:

- Key
- Value
- Left child
- Right child
- Parent
- Color (RED or BLACK)

✔ **Red-Black Tree Rules:**

- Every node is either red or black.
- The root is always black.
- No two red nodes appear consecutively.
- Every path from root to leaf has the same number of black nodes.

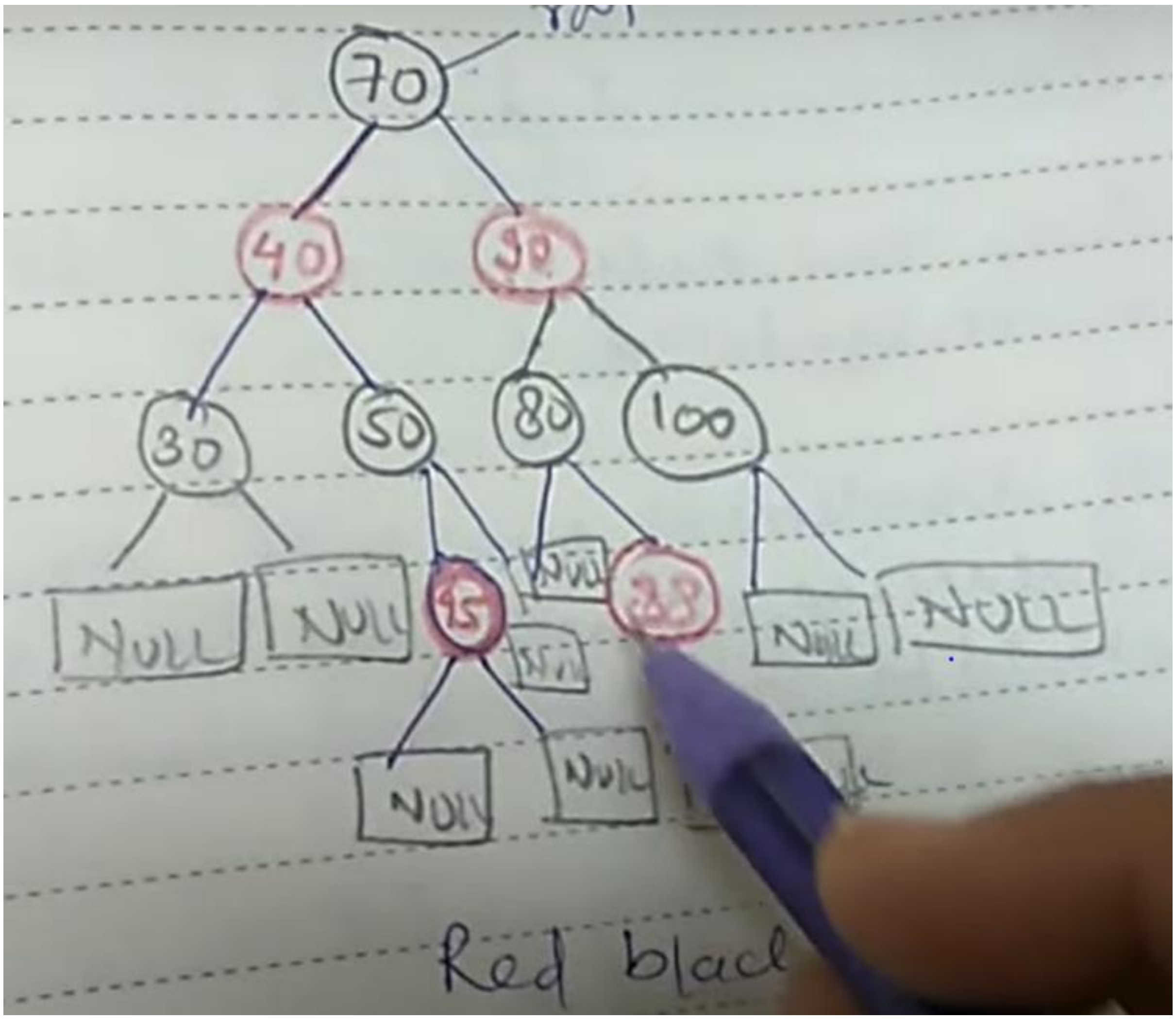
This balancing ensures O(log n) time for all basic operations.

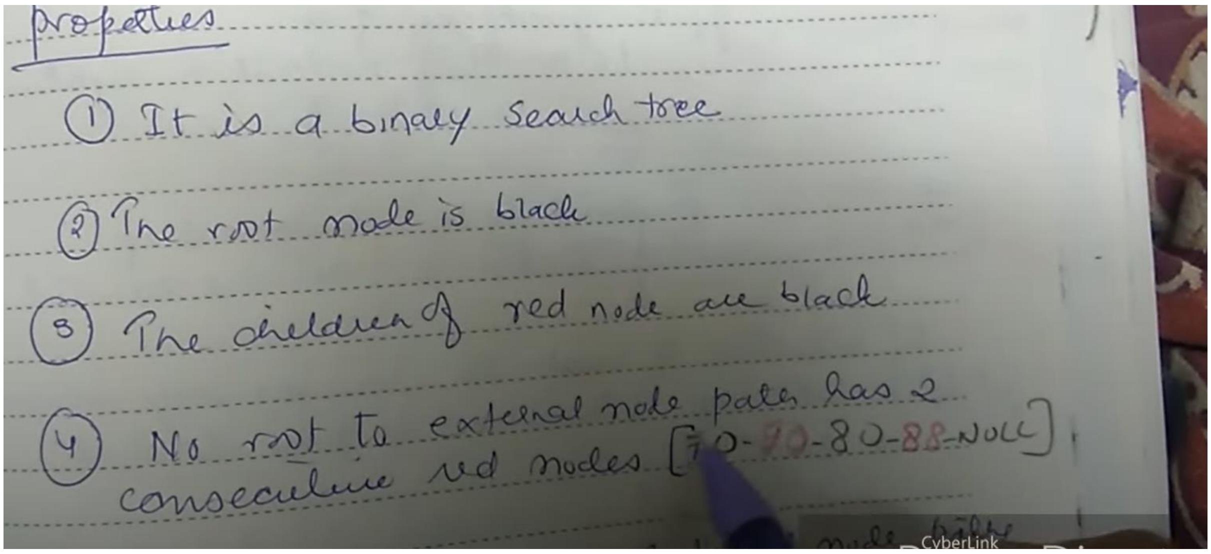
▯ **Capacity:**

Unlike HashMap or LinkedHashMap, **TreeMap does not have a fixed capacity or load factor** because:

- It is not backed by an array.
- It grows dynamically as nodes are added.
- The structure of the tree (balanced height) ensures efficiency.

Red-Balck-Tree




properties

① It is a binary search tree

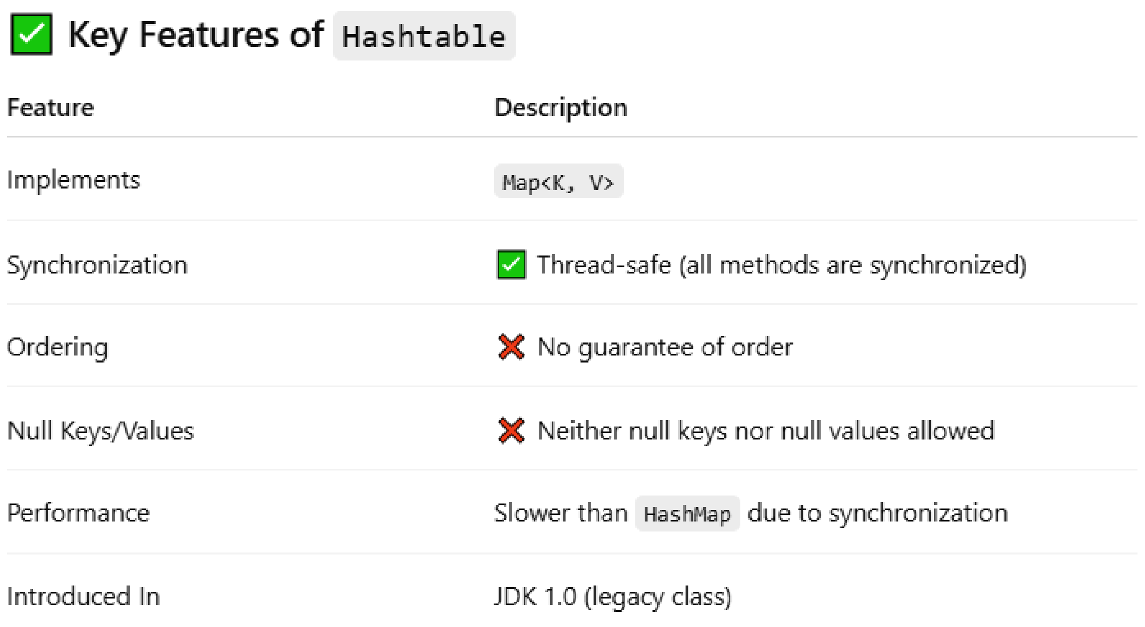
② The root node is black

③ The children of red node are black

④ No root to external node path has 2 consecutive red nodes [70-90-80-88-NULL]

□ What is Hashtable in Java?

Hashtable is a **legacy class** in Java that implements the Map interface and stores key-value pairs. It was part of the original JDK (prior to Java 2), and it is **synchronized**, meaning it is **thread-safe** for concurrent access by multiple threads.



✅ Key Features of `Hashtable`

Feature	Description
Implements	`Map<K, V>`
Synchronization	✅ Thread-safe (all methods are synchronized)
Ordering	❌ No guarantee of order
Null Keys/Values	❌ Neither null keys nor null values allowed
Performance	Slower than `HashMap` due to synchronization
Introduced In	JDK 1.0 (legacy class)

⚙️ Internal Working

Internally, Hashtable uses an **array of buckets** (i.e., an array of Entry objects or nodes), and it uses the **hashCode()** of the key to determine the index of the bucket.

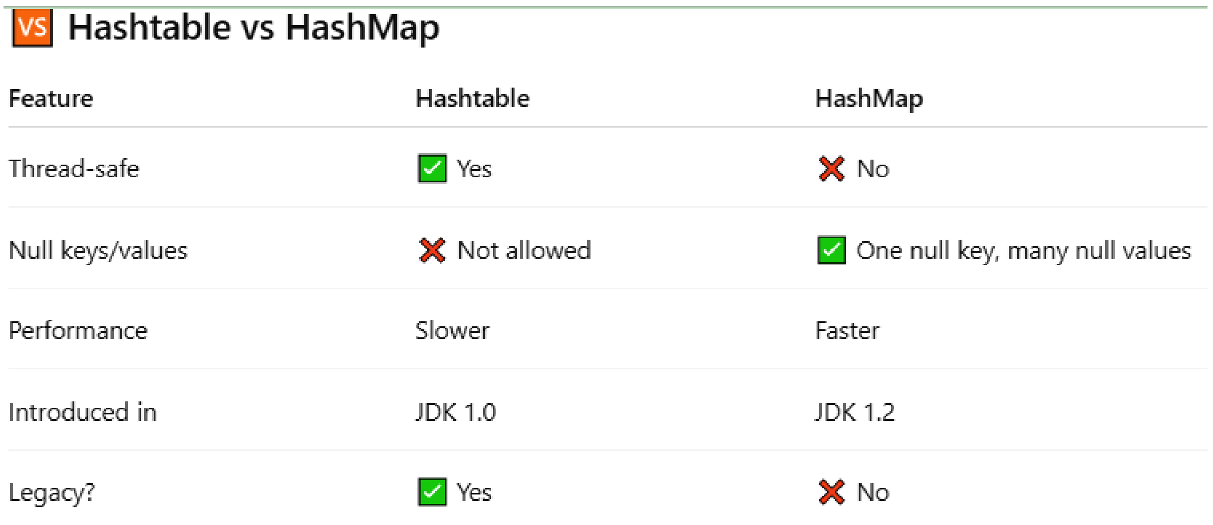
Steps:

1. Compute hash from the key using key.hashCode().
2. Apply a modulo operation to map it to the index: index = hash % table.length.
3. If multiple keys map to the same index (collision), it stores entries in a **linked list** at that index.
4. On retrieval, it compares keys using .equals() to find the right value.

💡 Capacity & Load Factor

Term	**Description**
Initial Capacity	Default = 11
Load Factor	Default = 0.75
Threshold	Capacity × Load Factor (e.g., 11 × 0.75 = 8.25) → resize at 8 entries

When the threshold is exceeded, the internal table size is doubled and incremented by 1 (2 * oldCapacity + 1), and all entries are **rehashed** into the new array.



🆚 Hashtable vs HashMap

Feature	Hashtable	HashMap
Thread-safe	✅ Yes	❌ No
Null keys/values	❌ Not allowed	✅ One null key, many null values
Performance	Slower	Faster
Introduced in	JDK 1.0	JDK 1.2
Legacy?	✅ Yes	❌ No

⚙️ Internal Working

The internal working of ConcurrentHashMap differs based on the Java version:

📌 Before Java 8 (Segmented Locking)

- Internally divided the map into **segments** (like buckets).
- Each segment had its own lock.
- Allowed multiple threads to read/write in **different segments** concurrently.

📌 Java 8 and Later (Bucket-based with CAS and Locking)

- Uses **array of Node<K,V>[]** just like HashMap.
- Collisions are handled using:
  - Linked list (for few entries)
  - Red-Black Tree (for many entries)
- Uses **Compare-And-Swap (CAS)** and **synchronized blocks** for thread safety on individual buckets — no global lock.

🔒 Locking Mechanism

- **Reads** are lock-free.
- **Writes** (put, remove) use a **lock only on the bucket** being modified, not the whole map.
- High concurrency is achieved as multiple threads can modify different buckets simultaneously.

□ Capacity and Resizing

Property	**Default Value**
Initial Capacity	16
Load Factor	0.75
Concurrency Level (Java 7)	16

- Resizing occurs when size exceeds capacity × loadFactor.
- Resize is also **thread-safe and efficient** due to fine-grained locking.

JavaHungry Interview question and answer

🔍 What is Comparable and Comparator Interface in Java?

Both Comparable and Comparator are interfaces in Java used to **sort objects**, but they serve **different purposes**.

✅ 1. Comparable Interface (Natural Ordering)

💠 Definition:

- Used to define the **default (natural)** sorting of objects.
- Belongs to the package java.lang.

```
public interface Comparable<T> {
   public int compareTo(T o);
}
```

💠 Behavior:

- If a.compareTo(b):
  - returns **negative** → a < b
  - returns **zero** → a == b
  - returns **positive** → a > b

```
class Student implements Comparable<Student> {
   int id;
   String name;

   public Student(int id, String name) {
      this.id = id;
      this.name = name;
   }

   public int compareTo(Student s) {
      return this.id - s.id;  // sort by id ascending
   }
}
```

✅ 2. Comparator Interface (Custom Ordering)

🔷 Definition:

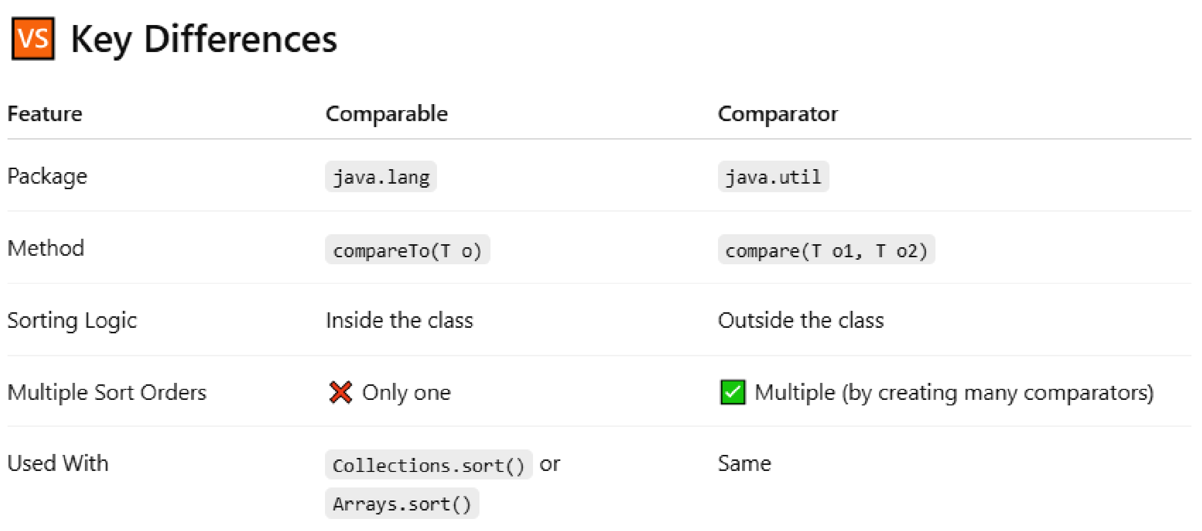
- Used to define **custom** sorting outside the class.
- Belongs to the package java.util.

```
public interface Comparator<T> {
  public int compare(T o1, T o2);
}
```

🔷 Behavior:

- If compare(a, b):
  - returns **negative** → a < b
  - returns **zero** → a == b
  - returns **positive** → a > b

```
class NameComparator implements Comparator<Student> {
  public int compare(Student s1, Student s2) {
    return s1.name.compareTo(s2.name); // sort by name
  }
}
```



🆚 Key Differences

Feature	Comparable	Comparator
Package	`java.lang`	`java.util`
Method	`compareTo(T o)`	`compare(T o1, T o2)`
Sorting Logic	Inside the class	Outside the class
Multiple Sort Orders	❌ Only one	✅ Multiple (by creating many comparators)
Used With	`Collections.sort()` or `Arrays.sort()`	Same

Q1 What is Collection? What is a Collections Framework? What are the benefits of the Java Collections Framework?

Collection : A collection (also called a container) is an object that groups multiple elements into a single unit.

Collections Framework : Collections framework provides a unified architecture for manipulating and representing collections.

Benefits of Collections Framework :

1. Improves program quality and speed
2. Increases the chances of reusability of software
3. Decreases programming effort.

Q2 What is the root interface in the collection hierarchy?

The root interface in the collection hierarchy is the **Collection interface.** Few interviewers may argue that

the Collection interface extends the **Iterable interface**. So iterable should be the root interface. But you should reply iterable interface present in java.lang package not in java.util package. It is clearly mentioned in Oracle Collection docs, that Collection interface is a member of the Java Collections framework. For the Iterable interface Oracle doc, the iterable interface is not mentioned as a part of the Java Collections framework. So if the question includes collection hierarchy, then you should answer the question as Collection interface (which is found in java.util package).

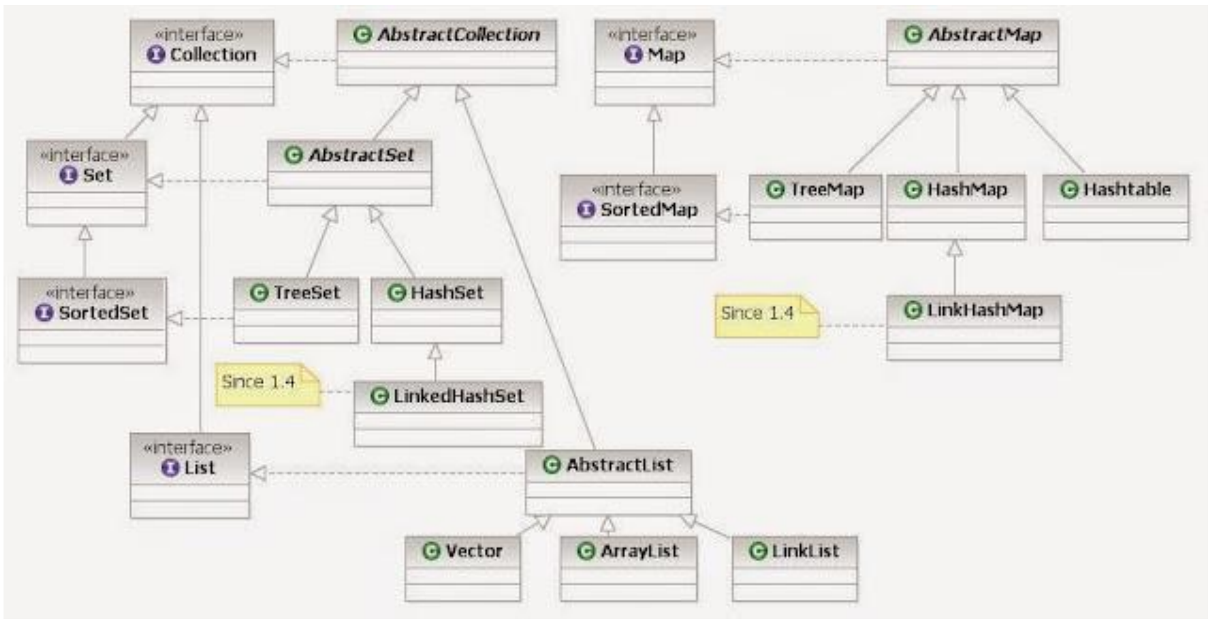
Q3 What is the difference between Collection and Collections?

The Collection is an interface while Collections is a java class, both are present in java.util package and part of the java collections framework. (answer)

Q4 Which collection classes are synchronized or thread-safe?

Stack, Properties, Vector, and Hashtable can be used in a multi-threaded environment because they are synchronized classes (or thread-safe).

Q5 Name the core Collection interfaces?



source of image : By Ervinn at en.wikibooks [CC BY-SA 3.0], from Wikimedia Commons

The list of core collection interfaces are : just mention the important ones

Important : Collection , Set , Queue , List , Map

Other interfaces also on the list : SortedSet, SortedMap, Deque, ListIterator, etc.

Q6 What is the difference between List and Set?

Set contains only unique elements while List can contain duplicate elements.
Set is unordered while the List is ordered. List maintains the order in which the objects are added.

Q7 What is the difference between Map and Set?

Map object has unique keys each containing some value, while Set contains only unique values.

Q8 What are the classes implementing List and Set interface?

Class implementing List interface : ArrayList, Vector, LinkedList

Class implementing Set interface : HashSet, TreeSet

Q9 What is an iterator?

The Iterator is an interface. It is found in java.util package. It provides methods to iterate over any Collection.

Q10 What is the difference between Iterator and Enumeration?

The main difference between Iterator and Enumeration is that Iterator has remove() method while Enumeration doesn't.
Hence, using Iterator we can manipulate objects by adding and removing the objects from the collections. Enumeration behaves like a read-only interface as it can only traverse the objects and fetch it.

Q11 Which design pattern followed by Iterator?

It follows the iterator design pattern. An iterator design pattern provides us to navigate through the collection of objects by using a common interface without letting us know about the underlying implementation.

Enumeration is an example of an Iterator design pattern.

Q12 Which methods you need to override to use any object as a key in HashMap?

To use any object as a key in HashMap, it needs to implement equals() and hashCode() method.

Q13 What is the difference between Queue and Stack?

The Queue is a data structure that is based on FIFO (first in first out) property. An example of a Queue in the real-world is buying movie tickets in the multiplex or cinema theaters.

The Stack is a data structure that is based on LIFO (last in first out) property. An example of Stack in the real-world is the insertion or removal of CD from the CD case.

Q14 How to reverse the List in Collections?

There is a built-in reverse method in the Collections class. reverse(List list) accepts the list as a parameter.

Collections.reverse(listobject);

Q15 How to convert the array of strings into the list?

Arrays class of java.util package contains the method asList() which accepts the array as a parameter.
So,

String[] wordArray = {"Love Yourself" , "Alive is Awesome" , "Be in present"};
List wordList = Arrays.asList(wordArray);

Intermediate Level (1-3 yrs): Java Collections Interview Questions and Answers

Q16 What is the difference between ArrayList and Vector?
It is one of the frequently asked collection interview questions, the main differences are Vector is synchronized while ArrayList is not. Vector is slow while ArrayList is fast. Every time when needed, Vector increases the capacity twice of its initial size while ArrayList increases its Array size by 50%. find detailed explanation ArrayList vs Vector.

Q17 What is the difference between HashMap and Hashtable?

It is one of the most popular collections interview questions for java developers. Make sure you go through this once before appearing for the interview.
Main differences between HashMap and Hashtable are :

a. HashMap allows one null key and any number of null values while Hashtable does not allow null keys and null values.
b. HashMap is not synchronized or thread-safe while Hashtable is synchronized or thread-safe.
find a detailed explanation here Hashtable vs HashMap in Java

Q18 What is the difference between peek(), poll() and remove() method of the Queue interface?

Both poll() and remove() method are used to remove the head object of the Queue. The main difference lies when the Queue is empty().
If the Queue is empty then the poll() method will return null. While in similar case , remove() method will throw NoSuchElementException .
peek() method retrieves but does not remove the head of the Queue. If the queue is empty then the peek() method also returns null.
Q19 What is the difference between Iterator and ListIterator.

Using Iterator we can traverse the list of objects in the forward direction. But ListIterator can traverse the collection in both directions that are forward as well as backward.

Q20 What is the difference between Array and ArrayList in Java?

This question checks whether the student understands the concept of the static and dynamic array. Some main differences between Array and ArrayList are :
a. Array is static in size while ArrayList is dynamic in size.
b. Array can contain primitive data types while ArrayList can not contain primitive data types.
find detailed explanation ArrayList vs Array in Java

Q21 What is the difference between HashSet and TreeSet?

Main differences between HashSet and TreeSet are :
a. HashSet maintains the inserted elements in random order while TreeSet maintains elements in the sorted order
b. HashSet can store the null object while TreeSet can not store the null object.
find a detailed explanation here TreeSet vs HashSet in Java

Q22 Write java code showing insertion, deletion, and retrieval of HashMap object?

Do it yourself (DIY), if found any difficulty or doubts then please mention in the comments.

Q23 What is the difference between HashMap and ConcurrentHashMap?

This is also one of the most popular java collections interview questions. Make sure this question is in your to-do list before appearing for the interview.
Main differences between HashMap and ConcurrentHashMap are :
a. HashMap is not synchronized while ConcurrentHashMap is synchronized.
b. HashMap can have one null key and any number of null values while ConcurrentHashMap does not allow null keys and null values.
find a detailed explanation here ConcurrentHashMap vs HashMap in Java

Q24 Arrange the following in the ascending order (performance):
HashMap, Hashtable, ConcurrentHashMap, and Collections.SynchronizedMap

Hashtable < Collections.SynchronizedMap < ConcurrentHashMap < HashMap

Q25 How HashMap works in Java?

This is one of the most important questions for java developers. HashMap works on the principle of Hashing. Find detailed information here to understand what is hashing and how hashmap works in java.

Q26 What is the difference between LinkedList and ArrayList in Java?

Main differences between LinkedList and ArrayList are :
a. LinkedList is the doubly linked list implementation of the List interface, while , ArrayList is the resizable array implementation of the List interface.
b. LinkedList can be traversed in the reverse direction using the descendingIterator() method provided by the Java API developers, while , we need to implement our own method to traverse ArrayList in the reverse direction. find the detailed explanation here ArrayList vs LinkedList in java.

Q27 What are Comparable and Comparator interfaces? List the difference between them?

We already explained what is comparable and comparator interface in detail along with examples here, Comparable vs Comparator in Java

Q28 Why Map interface does not extend the Collection interface in Java Collections

Framework?

One liner answer : **Map interface is not compatible with the Collection interface.**
Explanation : Since Map requires a key as well as value, for example, if we want to add key-value pair then we will use put(Object key, Object value). So there are two parameters required to add an element to the HashMap object. In Collection interface add(Object o) has only one parameter.
The other reasons are Map supports valueSet, keySet as well as other appropriate methods which have just different views from the Collection interface.

Q29 When to use ArrayList and LinkedList in the application?

ArrayList has a constant time search operation O(1). Hence, ArrayList is preferred when there are more get() or search operation.

Insertion, Deletion operations take constant time O(1) for LinkedList. Hence, LinkedList is preferred when there are more insertions or deletions involved in the application.

Q30 Write the code for iterating the list in different ways in java?

There are two ways to iterate over the list in java :
a. using Iterator
b. using for-each loop

Coding part : Do it yourself (DIY), in case of any doubts or difficulty please mention in the comments.

Q31 Give a practical example of BlockingQueue?

BlockingQueue can be used in the Producer-Consumer design pattern. You can find a detailed explanation with code here.

Q32 What is the default capacity of mostly used java collections (like ArrayList and HashMap)?

Make sure you understand the difference between the terms size and capacity.
Size represents the number of elements stored currently. Capacity indicates the maximum number of elements a collection can hold currently.

Default capacity of ArrayList : 10
Default capacity of HashMap : 16

Advance Level (3+ yrs): Java Collections Interview Questions and Answers

Q33 How HashSet works internally in java?

This is one of the popular interview questions. HashSet internally uses HashMap to maintain the uniqueness of elements. We have already discussed in detail hashset internal working in java.

Q34 What is CopyOnWriteArrayList? How it is different from ArrayList in Java?

CopyOnWriteArrayList is a thread-safe variant of ArrayList in which all mutative operations like add, set are implemented by creating a fresh copy of the underlying array.
It guaranteed not to throw ConcurrentModificationException.
It permits all elements including null. It is introduced in JDK 1.5.

Q35 How HashMap works in Java?

We are repeating this question, as it is one of the most important question for java developer.HashMap works on the principle of Hashing. Please find the detailed answer here hashmap internal working in java.

Q36 How remove(key) method works in HashMap?

This is the new question which is getting popular among java interviewers. We have shared a detailed explanation here about how remove method works internally in java.

Q37 What is BlockingQueue in Java Collections Framework?

BlockingQueue implements the java.util.Queue interface. BlockingQueue supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
It does not accept null elements.
Blocking queues are primarily designed for producer-consumer problems.
BlockingQueue implementations are thread-safe and can also be used in inter-thread communications.
This concurrent Collection class was added in JDK 1.5

Q38 How TreeMap works in Java?

TreeMap internally uses a Red-Black tree to sort the elements in a natural order. Please find the detailed answers here internal implementation of TreeMap in java.

Q39 All the questions related to HashSet class can be found here frequently asked HashSet interview questions

Q40 What is the difference between Fail- fast iterator and Fail-safe iterator?

This is one of the most popular interview questions for the higher experienced java developers.
Main differences between Fail-fast and Fail-safe iterators are :
a. Fail-fast throws ConcurrentModificationException while Fail-safe does not.
b. Fail-fast does not clone the original collection list of objects while Fail-safe creates a copy of the original collection list of objects.
The difference is explained in detail here fail-safe vs fail-fast iterator in java.

Q41 How ConcurrentHashMap works internally in Java?

The detailed answer is already explained here internal implementation of ConcurrentHashMap

Q42 How do you use a custom object as a key in Collection classes like HashMap?

If one is using the custom object as a key then one needs to override equals() and hashCode() method
and one also needs to fulfill the contract.
If you want to store the custom object in the SortedCollections like SortedMap then one needs to make sure that equals() method is consistent with the compareTo() method. If inconsistent, then the collection will not follow their contracts, that is, Sets may allow duplicate elements.
Q43 What is hash-collision in Hashtable? How it was handled in Java?

In Hashtable, if two different keys have the same hash value then it leads to hash -collision. A bucket of type linkedlist used to hold the different keys of same hash value.

Q44 Explain the importance of hashCode() and equals() method ? Explain the contract also?

HashMap object uses a Key object hashCode() method and equals() method to find out the index to put the key-value pair. If we want to get value from the HashMap same both methods are used. Somehow, if both methods are not implemented correctly, it will result in

two keys producing the same hashCode() and equals() output. The problem will arise that HashMap will treat both outputs the same instead of different and overwrite the most recent key-value pair with the previous key-value pair.
Similarly, all the collection classes that do not allow the duplicate values to use hashCode() and equals() method to find the duplicate elements. So it is very important to implement them correctly.

Contract of hashCode() and equals() method

a. If object1.equals(object2) , then object1.hashCode() == object2.hashCode() should always be true.

b. If object1.hashCode() == object2.hashCode() is true does not guarantee object1.equals(object2)

Q45 What is EnumSet in Java?

EnumSet is a specialized Set implementation for use with enum types. All of the elements in an enum set must come from a single enum type that is specified explicitly or implicitly when the set is created.
The iterator never throws ConcurrentModificationException and is weakly consistent.
Advantage over HashSet:
All basic operations of EnumSet execute in constant time. It is most likely to be much faster than HashSet counterparts.
It is a part of the Java Collections Framework since JDK 1.5.

Q46 What are concurrentCollectionClasses?

In jdk1.5, Java Api developers had introduced a new package called java.util.concurrent that has thread-safe collection classes as they allow collections to be modified while iterating. The iterator is fail-safe that is it will not throw ConcurrentModificationException.
Some examples of concurrentCollectionClasses are :
a. CopyOnWriteArrayList
b. ConcurrentHashMap

Q47 How do you convert a given Collection to SynchronizedCollection?

One line code : Collections.synchronizedCollection(Collection collectionObj) will convert a given collection to synchronized collection.

Q48 What is IdentityHashMap?

IdentityHashMap

IdentityHashMap is a class present in java.util package. It implements the Map interface with a hash table, using reference equality instead of object equality when comparing keys and values. In other words, in IdentityHashMap two keys, k1 and k2 are considered equal if only if (k1==k2).
IdentityHashMap is not synchronized.
Iterators returned by the iterator() method are fail-fast, hence, they will throw ConcurrentModificationException.

Q49 What is WeakHashMap?

WeakHashMap :

WeakHashMap is a class present in java.util package similar to IdentityHashMap. It is a Hashtable based implementation of Map interface with weak keys. An entry in WeakHashMap will automatically be removed when its key is no longer in ordinary use. More precisely the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector.
It permits null keys and null values.
Like most collection classes this class is not synchronized. A synchronized WeakHashMap may be constructed using the Collections.synchronizedMap() method.
Iterators returned by the iterator() method are fail-fast, hence, they will throw ConcurrentModificationException.

Q50 How will you make Collections readOnly?

We can make the Collection readOnly by using the following lines code:
General : Collections.unmodifiableCollection(Collection c)

Collections.unmodifiableMap(Map m)
Collections.unmodifiableList(List l)
Collections.unmodifiableSet(Set s)

Q51 What is UnsupportedOperationException?

This exception is thrown to indicate that the requested operation is not supported.
Example of UnsupportedOperationException:
In other words, if you call add() or remove() method on the readOnly collection. We know readOnly collection can not be modified. Hence, UnsupportedOperationException will be thrown.

Q52 Suppose there is an Employee class. We add Employee class objects to the ArrayList. Mention the steps that need to be taken if I want to sort the objects in ArrayList using the employeeId attribute present in Employee class.

a. Implement the Comparable interface for the Employee class and now to compare the objects by employeeId we will override the emp1.compareTo(emp2)
b. We will now call Collections class sort method and pass the list as an argument, that is,
   Collections.sort(empList)

If you want to add more java collections interview questions and answers or in case you have any doubts related to the Java Collections framework, then please mention in the comments.

Q53 What are common algorithms used in the Collections Framework?

Common algorithms used for searching and sorting. For example, a red-black algorithm is used in the sorting of elements in TreeMap. Most of the algorithms are used for List interface but few of them are applicable for all kinds of Collection.

Q54 How to sort ArrayList in descending order?

One liner will be
Collections.sort(arraylist, Collections.reverseOrder());