Core - Java

Module - Data Structures and Collections

Collections

*In Java, **Collections** refers to the **Java Collections Framework (JCF)**, which provides a set of interfaces and classes for storing and manipulating groups of objects efficiently. It includes **List, Set, Queue, Map**, and utility classes to handle common collection operations.*

Collection Interface (java.util.Collection)

- *The root interface of the Collection framework (except Map).*
- *It provides methods like add(), remove(), size(), clear(), etc.*

List Interface (java.util.List)

- *Ordered collection (sequence).*
- *Allows duplicate elements.*
- *Implementations:*
- *ArrayList (Dynamic array, fast read, slow insert/delete)*
- *LinkedList (Doubly linked list, fast insert/delete, slow access)*
- *Vector (Thread-safe alternative to ArrayList)*
- *Stack (LIFO stack implementation*

Collections

Set Interface (java.util.Set)

- *Unordered collection, does not allow duplicates.*
- *Implementations:*
  - *HashSet (Uses HashMap internally, unordered)*
  - *LinkedHashSet (Maintains insertion order)*
  - *TreeSet (Sorted, uses Red-Black Tree)*

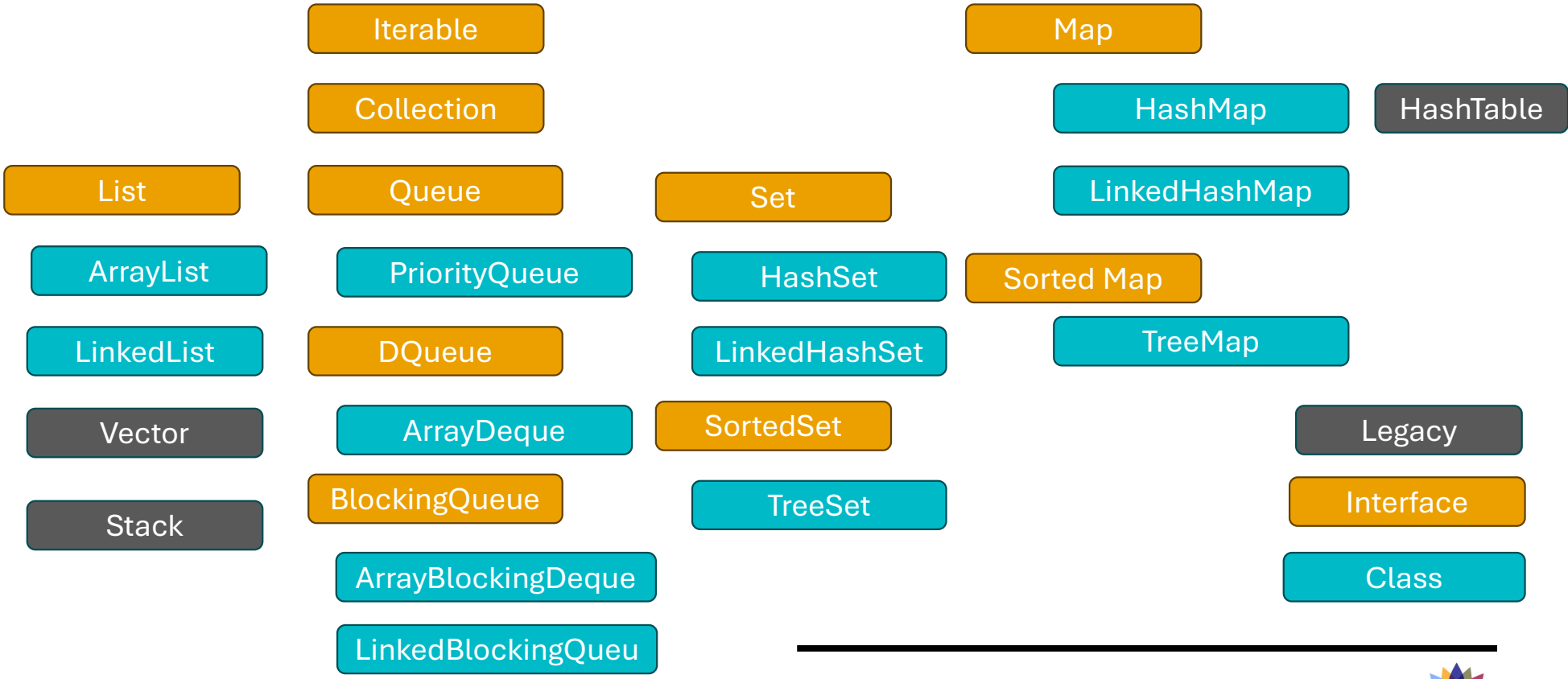
Queue Interface (java.util.Queue)

- *Follows FIFO (First-In-First-Out).*
- *Used for scheduling, buffering, etc.*
- *Implementations:*
  - *PriorityQueue (Elements sorted based on priority)*
  - *ArrayDeque (Efficient double-ended queue)*

Map Interface (java.util.Map)

- *Stores key-value pairs, keys must be unique.*
- *Implementations:*
  - *HashMap (Unordered, fast access)*
  - *LinkedHashMap (Maintains insertion order)*
  - *TreeMap (Sorted order using Red-Black Tree)*
  - *Hashtable (Thread-safe alternative to HashMap)*

Collections



Navarasam

Collections

Method	Description
iterator()	Returns an Iterator<T> to traverse elements.
forEach(Consumer<? super T> action)	Performs an action for each element in the collection (supports lambda expressions).
spliterator()	Returns a Spliterator<T> for parallel processing.

Navarasam

Iterable

Method	Description
iterator()	Returns an Iterator<T> to traverse elements.
forEach(Consumer<? super T> action)	Performs an action for each element in the collection (supports lambda expressions).
spliterator()	Returns a Spliterator<T> for parallel processing.

```
import java.util.*;
import java.util.function.Consumer;

public class IterableDemo {
  public static void main(String[] args) {
    // Creating a List (which implements Iterable)
    List<String> names = new ArrayList<>(Arrays.asList("Alice", "Bob", "Charlie", "David", "Eve"));

    // 1 Using iterator() to traverse manually
    System.out.println("Using iterator():");
    Iterator<String> iterator = names.iterator();
    while (iterator.hasNext()) {
      System.out.println(iterator.next());
    }

    // 2 Using forEach() with Lambda
    System.out.println("\nUsing forEach():");
    names.forEach(name -> System.out.println(name));

    // 3 Using Iterator to remove an element safely
    System.out.println("\nRemoving 'Charlie' using iterator.remove():");
    Iterator<String> removeIterator = names.iterator();
    while (removeIterator.hasNext()) {
      if (removeIterator.next().equals("Charlie")) {
        removeIterator.remove();
      }
    }
    System.out.println("List after removal: " + names);

    // 4 Using spliterator() for advanced iteration
    System.out.println("\nUsing spliterator():");
    Spliterator<String> spliterator = names.spliterator();
    spliterator.forEachRemaining(System.out::println);
  }
}
```

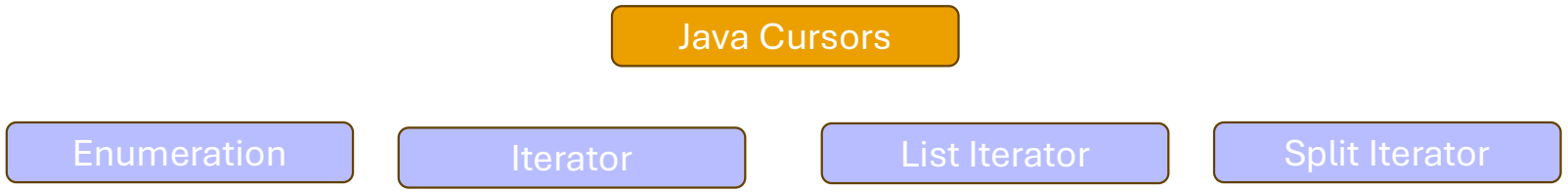
Navarasam

Iterator

An **Iterator in Java** is an interface used to traverse elements in a collection sequentially. It provides methods like **hasNext()**, **next()**, and **remove()** to loop through collections and perform manipulation. An Iterator is a part of the **Java Collection Framework**, and we can use it with collections like **ArrayList, LinkedList,** and other classes that implement the Collection interface

ListIterator

The **Iterable** interface was introduced in JDK 1.5. It belongs to **java.lang** package. In general, an object Implementing Iterable allows it to be iterated. An iterable interface allows an object to be the target of enhanced for loop(for-each loop).



Use Case	Best Choice
Need to traverse old Vector or Hashtable	**Enumeration**
Need to iterate through any collection (List, Set, Queue)	**Iterator**
Need both forward and backward iteration in a List	**ListIterator**
Need parallel iteration and functional programming	**Spliterator**

Comparison of Enumeration, Iterator, ListIterator, and Spliterator in Java

Feature	**Enumeration**	**Iterator**	**ListIterator**	**Spliterator**
Introduced In	Java 1.0	Java 1.2	Java 1.2	Java 8
Applicable To	Vector, Hashtable	List, Set, Queue	List (ArrayList, LinkedList)	List, Set, Queue, Map, Stream
Direction	Forward only	Forward only	Forward & Backward	Forward only
Can Modify Elements?	❌ No	✅ Yes (remove())	✅ Yes (remove(), set(), add())	❌ No
Fail-Safe or Fail-Fast?	✅ Fail-Safe	❌ Fail-Fast	❌ Fail-Fast	✅ Fail-Safe
Concurrent Modification Exception?	❌ No	✅ Yes	✅ Yes	❌ No
Allows Element Removal?	❌ No	✅ Yes (remove())	✅ Yes (remove())	❌ No
Allows Element Addition?	❌ No	❌ No	✅ Yes (add())	❌ No
Allows Element Replacement?	❌ No	❌ No	✅ Yes (set())	❌ No
Supports Parallel Processing?	❌ No	❌ No	❌ No	✅ Yes
Usage Scenario	Legacy collections (Vector, Hashtable)	General collections (List, Set, Queue)	Lists requiring bidirectional traversal and modification	Large datasets requiring parallel processing

Collection Interface Methods

Method	Description
add(E e)	Adds an element to the collection.
remove(Object o)	Removes an element.
contains(Object o)	Checks if an element exists.
addAll(Collection< ? Extends E> C)	Adds all elements from another collection
removeAll(Collection<?> c)	Removes all mathing elements from the collection
retainAll(Collection<?> c)	Retains only the elements that are also In the given collection
Clear()	Removes all elements in the collection
ContainsAll(Collection<?> c)	Checks if all elements in the given collection exist in this collection
toArray()	Converts the collections into an Array
toArray(T[] a)	Converts the collection into a typed array

List Interface – Quick Reference

- **Part of java.util package** and extends the Collection interface.
- **Maintains element order** as they are added.
- **Allows duplicates** (same values can be stored multiple times).
- **Common implementations**: ArrayList, LinkedList, Stack, Vector.
- **Supports null values** (depends on implementation).
- **Allows indexed access** to elements.
- **Supports ListIterator**, which enables **both forward & backward traversal**.

Navarasam

Important Methods in List Interface

Method	Description
add(E e)	Adds an element to the list.
add(int index, E e)	Inserts an element at the specified index.
get(int index)	Retrieves an element at the specified index.
set(int index, E e)	Replaces an element at the given index.
remove(int index)	Removes the element at the given index.
remove(Object o)	Removes the first occurrence of the specified element.
size()	Returns the number of elements in the list.
contains(Object o)	Checks if the list contains the specified element.
indexOf(Object o)	Returns the index of the first occurrence of an element.
lastIndexOf(Object o)	Returns the index of the last occurrence of an element.
subList(int fromIndex, int toIndex)	Returns a portion of the list.
listIterator()	Returns a ListIterator to traverse forward and backward.
sort(Comparator<? super E> c)	Sorts the list using a custom comparator.

Comparison of List Implementation

Feature	**ArrayList**	**LinkedList**	**Vector**	**Stack**
Implementation Type	Resizable array	Doubly linked list	Resizable array	Extends Vector
Access Speed (Indexing)	Fast	Slow	Fast	Fast
Insertion Speed	Slow at the beginning, fast at the end	Fast at both ends	Slow (due to synchronization)	Slow (due to synchronization)
Thread-Safety	Not thread-safe	Not thread-safe	Thread-safe	Thread-safe
Memory Overhead	Low	High	High	High
Null Values	Allows nulls	Allows nulls	Allows nulls	Allows nulls
Uses	Frequent random access, dynamic array-based lists	Ideal for frequent insertions and deletions	Similar to ArrayList but synchronized	Extends Vector with stack-specific methods (LIFO)
Synchronized	No	No	Yes	Yes

Memory Management In List

Feature	ArrayList	LinkedList	Vector	Stack
Internal Storage	**Resizable array**	**Doubly linked list**	**Resizable array (Synchronized)**	**Extends Vector (Synchronized)**
Memory Overhead	✅ Low	❌ High (extra pointers)	❌ High	❌ High
Growth Strategy	1.5x when full	Dynamic allocation (No resizing)	2x when full	2x when full
Random Access Speed	✅ Fast O(1)	❌ Slow O(n)	✅ Fast O(1)	✅ Fast O(1)
Insertion/Deletion Speed	❌ Slow O(n)	✅ Fast O(1)	❌ Slow O(n)	❌ Slow O(n)
Garbage Collection	✅ After resizing	✅ After removing nodes	✅ After resizing	✅ After resizing
Synchronization	❌ No	❌ No	✅ Yes	✅ Yes
Best For	Frequent access	Frequent insert/delete	Thread-safe collections	LIFO operations

Navarasam

Comparison Table: Data Structures Used

Feature	ArrayList	LinkedList	Vector	Stack	int[]
Underlying DS	**Dynamic Array**	**Doubly Linked List**	**Synchronized Array**	**Vector (Array-Based Stack)**	**Fixed Array**
Access Speed	✅ O(1)	❌ O(n)	✅ O(1)	✅ O(1)	✅ O(1)
Insertion Speed	❌ O(n) (shifts)	✅ O(1) (at start/end)	❌ O(n)	✅ O(1) (push())	❌ Fixed
Deletion Speed	❌ O(n)	✅ O(1)	❌ O(n)	✅ O(1) (pop())	❌ Fixed
Memory Usage	✅ Low	❌ High (extra pointers)	❌ High	❌ High	✅ Lowest
Resizable?	✅ Yes	✅ Yes	✅ Yes	✅ Yes	❌ No
Thread-Safe?	❌ No	❌ No	✅ Yes	✅ Yes	❌ No
Best Use Case	Fast access	Frequent insert/delete	Multi-threading	LIFO (Stack)	Fixed-size storage

Navarasam

To Store list of number

Scenario	Best Choice
Dynamic list of numbers	✅ ArrayList<Integer>
Fixed number of elements	✅ int[]
Frequent insertions/removals	✅ LinkedList<Integer>
Multi-threading environment	✅ Vector<Integer>

Navarasam

Set Interface – Quick Reference

•Package: java.util

•Extends: Collection<E>

•Definition: Unordered collection that does not allow duplicates

Key Features

• No specific order (except LinkedHashSet & TreeSet)

• Allows one null element (except TreeSet)

• Implements mathematical set behavior

• Efficient for unique element storage

Implementations

HashSet – Fast lookups, no order

•inkedHashSet – Maintains insertion order

•TreeSet – Sorted elements (natural order or custom comparator)

Thread-Safe Alternatives

•ConcurrentSkipListSet – Thread-safe & sorted

•Collections.synchronizedSet(new HashSet<>()) – Synchronized wrapper

Navarasam

Important Methods in Set Interface

Method	Description
boolean add(E e)	Adds an element to the set if it is not already present. Returns true if added, false otherwise.
boolean remove(Object o)	Removes the specified element from the set if present. Returns true if the element was removed.
boolean contains(Object o)	Checks if the set contains the specified element. Returns true if found.
int size()	Returns the number of elements in the set.
boolean isEmpty()	Checks if the set is empty (i.e., has no elements). Returns true if empty.
void clear()	Removes all elements from the set.
Iterator<E> iterator()	Returns an Iterator to traverse elements in the set.
Object[] toArray()	Converts the set into an array.
<T> T[] toArray(T[] a)	Returns an array containing all elements of the set in the specified type.
boolean addAll(Collection<? extends E> c)	Adds all elements from the specified collection to the set (ignoring duplicates).
boolean removeAll(Collection<?> c)	Removes all elements in the set that are present in the specified collection.
boolean retainAll(Collection<?> c)	Retains only the elements that exist in both the set and the specified collection.
boolean containsAll(Collection<?> c)	Checks if all elements of the specified collection exist in the set.

Comparison of Set Implementation

Feature	HashSet	LinkedHashSet	TreeSet	CopyOnWriteArraySet	ConcurrentSkipListSet
Ordering	❌ No Order	✅ Insertion Order	✅ Sorted Order (Natural/Comparator)	✅ Insertion Order	✅ Sorted Order (Natural/Comparator)
Duplicates	❌ Not Allowed	❌ Not Allowed	❌ Not Allowed	❌ Not Allowed	❌ Not Allowed
Null Elements	✅ Allowed (only one)	✅ Allowed (only one)	❌ Not Allowed	✅ Allowed (only one)	❌ Not Allowed
Thread-Safe	❌ No	❌ No	❌ No	✅ Yes (Copy-on-Write)	✅ Yes (Concurrent)
Performance	✅ Fast (O(1) for add, remove, contains)	🔹 Slightly slower than HashSet	⚠️ Slower (O(log n) due to tree operations)	⚠️ Slow (Copy-on-Write overhead)	⚠️ Slower (O(log n) due to concurrent tree)
Best Use Case	Fast unique storage without order	Maintaining insertion order	Sorted unique elements	Read-heavy, thread-safe	Thread-safe sorted set

Navarasam

Memory Management In Set

Set Implementation	Data Structure Used	Memory Efficiency	Key Factors Affecting Memory Usage
HashSet	Hash Table	🟢 High (due to hash table storage)	Load factor, capacity, hash collisions
LinkedHashSet	Hash Table + Linked List	🟡 Medium (extra memory for maintaining order)	Stores additional pointers for maintaining insertion order
TreeSet	Red-Black Tree (Self-Balancing BST)	🔴 Low (due to tree structure)	Each node stores left/right child pointers, causing higher memory usage
CopyOnWriteArraySet	Copy-on-Write Array	🔴 Low (memory overhead from array copies)	Creates a new copy on every modification, consuming more memory
ConcurrentSkipListSet	Concurrent Skip List (Multiple Linked Lists)	🔴 Low (linked list overhead)	Stores multiple levels of linked nodes, increasing memory usage

Navarasam

Queue Interface – Quick Reference

- **Package**: java.util
- **Extends**: Collection interface
- **Order**: Follows **FIFO (First In, First Out)**
- **Insertion & Removal**: Add at the end, remove from the front
- **No Nulls**: Most implementations (e.g., PriorityQueue) do not allow null elements
- **Common Implementations**: LinkedList, PriorityQueue, ArrayDeque, ConcurrentLinkedQueue (thread-safe)
- **Use Cases**: Task scheduling, message passing, buffer management
- **Iteration**: Supports element traversal; order depends on implementation

Important Methods in Queue Interface

Method	Description	Behavior when Queue is Empty
add(E e)	Inserts an element; throws an exception if it fails.	Throws an exception (IllegalStateException).
offer(E e)	Inserts an element; returns false if insertion fails.	Returns false.
remove()	Removes and returns the front element.	Throws an exception (NoSuchElementException).
poll()	Removes and returns the front element.	Returns null.
element()	Retrieves the front element without removing it.	Throws an exception (NoSuchElementException).
peek()	Retrieves the front element without removing it.	Returns null.

Navarasam

Comparison of Queue Implementation

Feature	LinkedList	PriorityQueue	ArrayDeque	ConcurrentLinkedQueue	BlockingQueue
Ordering	FIFO (First In, First Out)	Sorted (Natural/Custom Comparator)	FIFO (or LIFO if used as a stack)	FIFO	FIFO
Null Allowed?	✅ Yes	❌ No	✅ Yes	❌ No	❌ No
Thread-Safe?	❌ No	❌ No	❌ No	✅ Yes (Lock-free)	✅ Yes (Blocking operations)
Underlying Data Structure	**Doubly Linked List**	**Binary Heap (Array-based)**	**Resizable Array**	**Lock-Free Linked List**	**Array or Linked List (Depends on type)**
Performance	Moderate (O(1) add/remove, O(n) search)	Moderate (O(log n) add/remove, O(1) peek)	Fast (O(1) add/remove, O(n) search)	High (O(1) add/remove)	High (Blocking, depends on implementation)
Use Case	General-purpose queue	Task scheduling, priority-based processing	Stack & queue operations, buffer management	Multi-threaded, lock-free queue operations	Producer-consumer model, thread synchronization
Capacity Restrictions	No (Dynamic growth)	No (Dynamic growth)	No (Dynamic growth)	No (Dynamic growth)	✅ Yes (Can be bounded or unbounded)
When to Use?	Simple queue operations	Prioritized task execution	Faster than LinkedList for queue operations	Multi-threaded environments	Thread-safe queue with capacity control

Navarasam

Memory Management In Queue

Feature	LinkedList	PriorityQueue	ArrayDeque	ConcurrentLinkedQueue	BlockingQueue
Memory Storage	Node-based (Linked List)	Array-based (Resizable)	Array-based (Resizable)	Node-based (Lock-free)	Array or Node-based (Fixed Size)
Garbage Collection	Automatic when nodes are unlinked	Automatic when array resizes	Automatic when array resizes	Automatic when nodes are unlinked	Automatic when elements are dequeued
Capacity	Dynamic (grows as needed)	Dynamic (resizes array)	Dynamic (resizes array)	Dynamic	Fixed (predefined limit)
Risk of Memory Leak	High (if elements aren't removed)	Moderate (due to array resizing)	Low	Low	Low (bounded capacity prevents excessive growth)
Best Practices	Manually clear references	Avoid unnecessary resizing	Predefine an appropriate initial size	Use for multi-threading to avoid locks	Use for producer-consumer scenarios

Navarasam

Memory Management In Queue

Memory Management in Queue (Java)

Memory management in a **Queue** is crucial as elements are continuously added and removed. Here's how Java handles it:

1. Memory Allocation

•Java allocates memory dynamically for queue elements.

•Implementations like **LinkedList** use **node-based** storage, while **ArrayDeque** and **PriorityQueue** use **array-based** storage.

2. Automatic Garbage Collection

•When an element is dequeued (removed), it becomes eligible for **garbage collection (GC)** if no other references exist.

•**Example:** In LinkedList, removing a node releases its reference, allowing GC to reclaim memory.

3. Capacity Management

•**LinkedList**: Grows dynamically, no predefined capacity.

•**ArrayDeque & PriorityQueue**: Uses **resizable arrays** (expands when full, but resizing may cause performance overhead).

•**BlockingQueue** (e.g., ArrayBlockingQueue): Uses a **fixed capacity** to prevent excessive memory usage.

4. Memory Leaks in Queues

•**Holding References:** If elements are not explicitly removed, they stay in memory, leading to potential **memory leaks**.

•**Weak References:** Using WeakReference or SoftReference in queues (e.g., WeakLinkedQueue) helps avoid memory retention issues.

5. Best Practices for Efficient Memory Usage

•**Use a bounded queue** (ArrayBlockingQueue) when applicable to limit memory growth.

•**Manually remove references** in custom implementations to help GC.

•**Prefer poll() over remove()** to avoid exceptions when handling empty queues.

Map Interface – Quick Reference

Overview

•**Map** is a part of the Java Collections Framework (java.util package).

•It represents a collection of key-value pairs.

•Unlike a **List** or **Set**, a **Map** does **not** allow duplicate keys.

Key Features

1.Stores key-value pairs – Each key is unique, and each key maps to one value.

2.Allows null keys and values – Only **one null key** is allowed, but multiple null values are permitted.

3.Not a subtype of Collection – It is a separate interface in the Java Collections Framework.

4.Implements hashing and sorting – Depending on the implementation, a Map can be **unordered**, **sorted**, or **linked**.

Common Implementations –*HashMap, LinkedHashMap, TreeMap and HashTable*.

Navarasam

Important Methods in Map Interface

Category	Method	Description
Basic Operations	put(K key, V value)	Inserts or updates a key-value pair.
	get(Object key)	Retrieves the value for the given key. Returns null if key is absent.
	remove(Object key)	Removes a key-value pair and returns the value.
	containsKey(Object key)	Checks if the map contains the specified key.
	containsValue(Object value)	Checks if the map contains the specified value.
Bulk Operations	putAll(Map<? extends K, ? extends V> m)	Copies all key-value pairs from another map.
	clear()	Removes all key-value pairs from the map.
Size & Empty Check	size()	Returns the number of key-value pairs.
	isEmpty()	Checks if the map is empty.
Collection Views	keySet()	Returns a Set of all keys.
	values()	Returns a Collection of all values.
	entrySet()	Returns a Set of key-value pairs (Map.Entry<K, V>).
Java 8+ Methods	getOrDefault(Object key, V defaultValue)	Returns the value for a key or a default value if key is absent.
	putIfAbsent(K key, V value)	Inserts a key-value pair only if the key is not already present.
	remove(Object key, Object value)	Removes the key-value pair only if the key is mapped to the specified value.
	replace(K key, V value)	Replaces the value for a key if it is present.
	compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)	Updates the value based on a function.
	computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)	Computes and inserts a value if the key is absent.
	computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)	Updates an existing value using a function.

Navarasam

Comparison of Map Implementation

Feature	HashMap	LinkedHashMap	TreeMap	Hashtable
Ordering	No ordering (unordered)	Maintains **insertion order**	Sorted in **natural order** (or custom Comparator)	No ordering (unordered)
Null Keys	Allows **one** null key	Allows **one** null key	**Does not allow** null keys	**Does not allow** null keys
Null Values	Allows multiple null values	Allows multiple null values	Allows multiple null values	**Does not allow** null values
Performance	**O(1)** for put(), get() (fastest)	**O(1)** for put(), get() (slightly slower than HashMap)	**O(log n)** for put(), get()	**O(1)** for put(), get()
Thread Safety	**Not thread-safe**	**Not thread-safe**	**Not thread-safe**	**Thread-safe** (synchronized)
Use Case	Best for general-purpose usage	Best when maintaining **insertion order** is important	Best for **sorted data** or **range queries**	Best for **legacy applications** requiring thread safety
Internal Implementation	Uses **hash table** (based on hashCode())	Uses **doubly linked list** along with a hash table	Uses **Red-Black Tree**	Uses **hash table** with synchronization
Concurrency Support	Can be made thread-safe using Collections.synchronizedMap() or ConcurrentHashMap	Can be made thread-safe using Collections.synchronizedMap()	Can be made thread-safe using Collections.synchronizedMap()	Fully synchronized but slower

Here's a **comparison table** of different **Map implementations** in Java:

Which One to Use?

- Use **HashMap** if you need a **fast, unordered** key-value store.
- Use **LinkedHashMap** if you need to **preserve insertion order**.
- Use **TreeMap** if you need **sorted keys** or **range-based queries**.
- Use **Hashtable** only for **legacy code**; prefer ConcurrentHashMap instead.

Navarasam

Memory Management In Queue

Feature	HashMap	LinkedHashMap	TreeMap	Hashtable
Data Structure	**Array + Linked List** (or Tree in case of collisions - Java 8+)	**Array + Doubly Linked List**	**Red-Black Tree**	**Array + Synchronized Hash Table**
Memory Overhead	**Moderate** (Uses buckets, each holding an Entry object)	**High** (Maintains an extra doubly linked list for insertion order)	**High** (Uses self-balancing Red-Black tree with additional pointers)	**High** (Synchronized operations require more memory)
Handling of Collisions	Uses **chaining** (Linked List → Tree after threshold)	Same as HashMap but with additional links	N/A (No collisions, as it is a tree)	Uses **chaining** (Similar to HashMap)
Load Factor & Resizing	Default **0.75** (resizes when 75% full)	Same as HashMap	No resizing needed (Tree dynamically balances)	Default **0.75** (resizes when 75% full)
Garbage Collection Impact	Moderate (Entries are linked but not deeply nested)	Higher GC impact (due to extra linked list references)	Higher GC impact (More objects with parent-child relationships)	Higher GC impact (Synchronized methods create temporary objects)
Performance vs Memory Trade-off	Efficient in memory and performance	Consumes more memory for maintaining order	More memory due to tree structure	More memory due to synchronization

Data Structures

1. List Implementations and Their Data Structures

Implementation	Data Structure Used	Details
ArrayList	**Dynamic Array**	- Uses a resizable array (Object[]). - Supports **random access** (O(1) for get()). - Expands capacity when full (1.5x growth factor).
LinkedList	**Doubly Linked List**	- Each node contains **data + 2 pointers** (prev & next). - Efficient for **insertions/deletions** (O(1) at ends). - Slower **random access** (O(n) for get()).
Vector	**Dynamic Synchronized Array**	- Like ArrayList but **thread-safe**. - Uses **synchronized methods**, making it slower. - Grows **2x** its size when full.
Stack	**Dynamic Array (extends Vector)**	- LIFO (Last-In, First-Out) behavior. - Uses push(), pop(), and peek().

Navarasam

2. Set Implementations and Their Data Structures

Implementation	Data Structure Used	Details
HashSet	**Hash Table (Backed by HashMap)**	- Uses **buckets and linked list/tree** for collisions. - Stores elements in **unordered** manner. - O(1) for add(), remove(), and contains() (if no collisions).
LinkedHashSet	**Hash Table + Doubly Linked List**	- Maintains **insertion order**. - Uses a **linked list** to keep track of order. - Slightly more memory overhead than HashSet.
TreeSet	**Red-Black Tree (Self-balancing BST)**	- Stores elements in **sorted order**. - Operations (add(), remove(), contains()) take **O(log n)**. - Maintains **balanced height** to ensure efficiency.

Queue Implementations and Their Data Structures

Implementation	Data Structure Used	Details
PriorityQueue	**Binary Heap (Min-Heap by default)**	- Elements are **sorted based on priority**. - poll() removes the **smallest** element (O(log n)). - Implements **heap-based priority queue**.
ArrayDeque	**Resizable Circular Array**	- More efficient than Stack and LinkedList. - Supports **double-ended queue (Deque)**. - Faster than LinkedList for deque operations.
LinkedList (Queue mode)	**Doubly Linked List**	- FIFO (First-In, First-Out) behavior. - poll(), offer(), peek() operations.

Navarasam

Map Implementations and Their Data Structures

Implementation	Data Structure Used	Details
HashMap	**Array + Linked List (Java 7) → Array + Tree (Java 8+)**	- Uses **hash table** with buckets. - Uses **linked list** for collisions (Java 7). - Uses **balanced tree** for collisions (Java 8+ if bucket size > 8).
LinkedHashMap	**Hash Table + Doubly Linked List**	- Maintains **insertion order**. - Uses a **linked list** to track order. - Higher memory usage than HashMap.
TreeMap	**Red-Black Tree (Self-balancing BST)**	- Stores keys in **sorted order**. - O(log n) for put(), get(), remove().
Hashtable	**Synchronized Hash Table**	- Thread-safe but **synchronized**, so it's slower. - Does **not allow null keys or values**.

Navarasam