Collection Framework

What is a Framework?

A **framework** is a **set of pre-written code or tools** that provides a **structure** for developers to build programs more easily.

In Java, the **Collection Framework** is a **framework** that provides **ready-made classes and interfaces** to **store and manipulate groups of objects** like lists, sets, queues, etc.

Why use the Collection Framework?

Before the collection framework, Java had to rely on arrays or custom data structures. Arrays are fixed in size, and creating your own linked list, stack, or queue is time-consuming.

The **Collection Framework** solves this by providing:

- **Interfaces**: What functionalities should be there (like List, Set, Map)
- **Classes**: Actual implementations (like ArrayList, HashSet, HashMap)
- **Algorithms**: Like sorting, searching, etc.

Interface:

An interface in Java is like a set of rules. It only defines what methods should be present, but not how they work. Classes that use the interface must provide the actual code for those methods.

Class:

A class is like a blueprint. It defines the structure and behavior of objects — like what variables and methods they will have.

Object:

An object is the actual thing created from a class. It has real values and can perform actions using the methods defined in the class.

Quick one-line version (super simple):

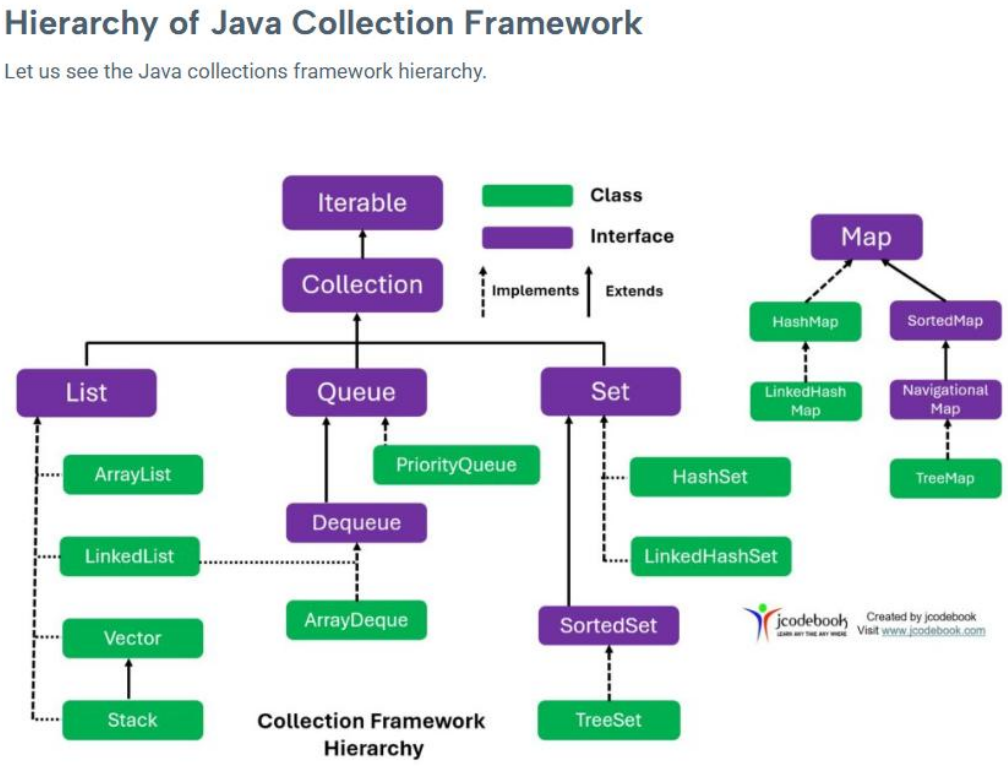
- **Interface** – A list of rules or functions without details.
- **Class** – A plan that defines variables and methods.
- **Object** – A real thing made from a class that we can use.

- **extends:**

This keyword establishes an inheritance relationship between two classes. A class using extends (subclass) inherits the properties and methods of another class (superclass). It promotes code reuse and establishes an "is-a" relationship. Only one class can be extended at a time.

- **implements:**

This keyword enables a class to adhere to a contract defined by an interface. An interface outlines methods that a class must implement. A class can implement multiple interfaces, ensuring it provides specific functionalities. It establishes a "can-do" relationship.



What is the List Interface?

The List interface in Java is a **part of the Collection Framework**.
It is used to **store a group of elements in an ordered way**, and it **allows duplicates**.

List is like an **ordered collection** where we can **store, access, and modify** elements using index numbers, and it **can have repeated values**.

Key Features of List:

- Ordered – elements are stored in the order they are added.
- Duplicates allowed – same value can appear more than once.
- Index-based – you can access elements using get(index).
- You can insert, update, or remove elements.

Common Classes that implement List:

Class	Description
ArrayList	Fast access, slow insert/delete (uses array internally)
LinkedList	Fast insert/delete, slow access (uses nodes/links)
Vector	Like ArrayList but thread-safe (rarely used now)
Stack	LIFO (Last-In-First-Out) structure built on Vector

Important Methods in List:

Method	Description
add()	Add element
get(index)	Get element at index
set(index, value)	Replace element
remove(index)	Remove element
size()	Get number of elements
contains(value)	Check if value is present

The List interface allows us to store elements in order, supports duplicates, and lets us access elements using index numbers. Common classes include ArrayList and LinkedList."

Differences Between ArrayList and LinkedList

Feature	**ArrayList**	**LinkedList**
Internal Structure	Dynamic array	Doubly linked list
Access (get/set)	Fast (O(1))	Slow (O(n))
Insertion/Deletion	Slow (shifting needed)	Fast (just change links)
Memory	Uses less memory	Uses more memory (extra pointers)
Best For	Reading/searching frequently	Adding/removing data frequently

Use ArrayList when:

- You need **fast access** to elements (search, get).
- Inserting/removing is rare or always at the end.
- You want to maintain insertion order.

Use LinkedList when:

- You need to **insert/delete elements frequently** (especially in the middle).
- Don't need random access much.
- Memory isn't a major concern.

Use Vector when:

- You need **thread-safe** (synchronized) list (used rarely now).
- Legacy code uses Vector (not recommended in new projects).

Use Stack when:

- You need **LIFO** (Last-In-First-Out) behavior.
- Use case: undo/redo, history tracking, backtracking.

Use ArrayList for fast access and when frequent read is required. Use LinkedList when you need to frequently add or remove items. Vector is like ArrayList but thread-safe. Stack is used when LIFO behavior is needed.

ArrayList

Implements: List interface

Commonly Used Methods:

Method	From	Purpose
add(E e)	List	Adds element at the end
add(int index, E e)	List	Adds element at a specific position
get(int index)	List	Returns element at given index
set(int index, E e)	List	Replaces element at index
remove(int index)	List	Removes element from index
size()	Collection	Returns number of elements
contains(Object o)	Collection	Checks if element exists
clear()	Collection	Removes all elements

LinkedList

Implements: List, Deque, Queue interfaces

Commonly Used Methods:

Method	From	Purpose
addFirst(E e)	Deque	Add at beginning
addLast(E e)	Deque	Add at end
removeFirst()	Deque	Remove first element
removeLast()	Deque	Remove last element
get(int index)	List	Get element at index
add(E e)	List	Add at end
size()	Collection	Total elements

Vector

Implements: List interface (thread-safe)

Commonly Used Methods:

Method	From	Purpose
add(E e)	List	Add element
get(int index)	List	Get element
size()	Collection	Number of elements
remove(int index)	List	Remove by index
capacity()	Vector class	Current capacity of Vector
trimToSize()	Vector class	Reduce memory to actual size

Stack

Extends: Vector
Implements: List

Commonly Used Methods:

Method	From	Purpose
push(E e)	Stack	Add element on top (LIFO)
pop()	Stack	Remove top element
peek()	Stack	View top element
empty()	Stack	Check if stack is empty
search(E e)	Stack	Find position from top (1-based)

What is the Set Interface?

- Set is a **Collection** that **does not allow duplicate elements**.
- It models the **mathematical set** — unordered and unique.
- It is an **interface**, so you cannot create objects of Set directly. You use **classes that implement** it.
-

Common Classes that implement `Set`:

Class Name	Maintains Order?	Allows Duplicates?	Sorted?	Thread-Safe?
`HashSet`	❌ (no order)	❌	❌	❌
`LinkedHashSet`	✅ (insertion)	❌	❌	❌
`TreeSet`	✅ (sorted)	❌	✅ (natural or custom)	❌

1. HashSet

When to Use:

- When you want a **unique collection** with **no duplicate elements**.
- You **don't care about order**.
- You want **fast performance** for insert/search/delete.

Internal Working:

- Uses a **HashMap** internally.
- Uses **hashing** for constant time operations.

Common Functions + Time Complexity:

Function	Defined In	Time Complexity	Use Case
add(E e)	Collection → Set	O(1) avg	Add a new element
remove(Object o)	Collection	O(1) avg	Remove element
contains(Object o)	Collection	O(1) avg	Check if element exists
size()	Collection	O(1)	Get number of elements
clear()	Collection	O(n)	Remove all elements

---

2. LinkedHashSet

When to Use:

- You want **unique elements** (like HashSet).
- But also want to **preserve insertion order**.

Internal Working:

- Uses **LinkedHashMap** internally to maintain order + hashing.

Common Functions + Time Complexity:

Function	**Defined In**	**Time Complexity**	**Use Case**
add(E e)	Collection → Set	O(1) avg	Add element + keep insertion order
remove(Object o)	Collection	O(1) avg	Remove while keeping order intact
contains(Object o)	Collection	O(1) avg	Lookup with order
iterator()	Iterable	O(1)	Traverse in insertion order

Slower than HashSet if insertion order isn't needed.

---

3. TreeSet

When to Use:

- You want **unique elements and** want them to be **sorted automatically**.
- When you need **range queries** (like headSet, tailSet, etc.)

Internal Working:

- Uses a **Red-Black Tree** (Self-balancing BST).

Common Functions + Time Complexity:

Function	**Defined In**	**Time Complexity**	**Use Case**
add(E e)	NavigableSet → SortedSet → Set	O(log n)	Insert sorted
remove(Object o)	Collection	O(log n)	Remove sorted element
contains(Object o)	Collection	O(log n)	Check if exists in sorted order
first() / last()	SortedSet	O(log n)	Get smallest/largest
ceiling(E e) / floor(E e)	NavigableSet	O(log n)	Nearest greater or smaller value

Slower than HashSet/LinkedHashSet, but useful for **sorted data or range operations**.

Summary Table

Feature	HashSet	LinkedHashSet	TreeSet
Order Maintained?	❌ No	✅ Insertion	✅ Sorted
Allows Duplicates?	❌ No	❌ No	❌ No
Underlying DS	HashMap	LinkedHashMap	Red-Black Tree
Insertion Time	O(1)	O(1)	O(log n)
Search Time	O(1)	O(1)	O(log n)
Remove Time	O(1)	O(1)	O(log n)
Best Use Case	Fast lookup	Ordered unique list	Sorted set with range ops

---

Interview Tip:

"I prefer HashSet for fast operations, LinkedHashSet when I want to keep the order of input, and TreeSet when I want my set to be sorted and perform range queries."

What is the Queue Interface?

- A **Queue** is a collection used to **hold elements before processing**.
- Follows the **FIFO** (First-In-First-Out) principle — just like a real-world queue.
- It is part of **java.util** package.
- Queue is an **interface**, so you need to use a class that implements it (like LinkedList, PriorityQueue, etc.)

Queue Interface Hierarchy

Collection (Interface)

↳ Queue (Interface)

↳ Deque (Interface for double-ended queue)

↳ ArrayDeque (Class)

↳ LinkedList (Class)

↳ PriorityQueue (Class)

Common Implementing Classes of Queue:

Class Name	Ordered?	Priority-Based?	Allows Nulls?	Thread-Safe?	Use Case
LinkedList	✅ FIFO	❌	✅ (1 null)	❌	Basic queue (FIFO) operations
PriorityQueue	✅ (sorted)	✅	❌	❌	Tasks with priorities
ArrayDeque	✅ (double-ended)	❌	❌	❌	Fast stack or queue (no nulls)

Common Methods in Queue Interface

Method	Defined In	Purpose
add(E e)	Queue	Inserts element, throws exception if fails
offer(E e)	Queue	Inserts element, returns false if fails (safe)
poll()	Queue	Removes and returns the head (null if empty)
remove()	Queue	Removes and returns head (throws exception if empty)
peek()	Queue	Returns head without removing (null if empty)
element()	Queue	Returns head without removing (throws if empty)

Time Complexities for Common Queue Classes

Class	add() / offer()	poll() / remove()	peek() / element()
LinkedList	O(1)	O(1)	O(1)
PriorityQueue	O(log n)	O(log n)	O(1)
ArrayDeque	O(1) (amortized)	O(1)	O(1)

When to Use Which Queue Class

Use Case	Recommended Class
Normal FIFO queue	LinkedList
Double-ended queue (insert/remove from both ends)	ArrayDeque
Priority-based task processing	PriorityQueue

🚀 **Interview-Ready Summary**

"The Queue interface is used when I want to process elements in order. I use LinkedList for simple queues, ArrayDeque for fast stack/queue, and PriorityQueue when elements must come out by priority, not insertion order."

What is the Map Interface?

- A **Map** stores **key-value pairs** (like a dictionary).
- Keys are **unique** (cannot be duplicated).
- Each key is mapped to **one value**.
- It's **not part of the Collection hierarchy**, but still belongs to the **Java Collections Framework**.

Defined in java.util package.

Map Interface Hierarchy

Map (Interface)

├── HashMap (Class)

├── LinkedHashMap (Class)

├── TreeMap (Class)

├── Hashtable (Legacy Class)

└── ConcurrentHashMap (Thread-safe)

Common Methods in Map Interface

Method	Description
put(K key, V value)	Adds or replaces a key-value pair
get(Object key)	Returns the value for the given key
containsKey(Object key)	Checks if the key exists
containsValue(Object val)	Checks if any key maps to this value
remove(Object key)	Removes key-value pair for given key
keySet()	Returns a Set of all keys
values()	Returns a Collection of all values
entrySet()	Returns a Set of Map.Entry (key-value pairs)
size()	Returns number of key-value pairs
clear()	Removes all entries

.

Map Implementations: Differences & When to Use

Map Type	Preserves Order?	Sorted?	Allows Null Key?	Thread-Safe?	Internal Structure	Time Complexity (get/put)
HashMap	❌ No	❌ No	✅ 1 null key	❌	Hash table	O(1) average
LinkedHashMap	✅ Insertion Order	❌ No	✅ 1 null key	❌	Hash table + LinkedList	O(1) average
TreeMap	✅ Sorted by key	✅ Yes	❌ No	❌	Red-Black Tree	O(log n)
Hashtable	❌ No	❌ No	❌ No	✅ Yes	Hash table	O(1) average

ConcurrentHashMap	❌ No	❌ No	❌ No	✅ Yes	Segmented buckets	O(1) average

---

When to Use Which Map

Use Case	**Recommended Map**
Fast, general-purpose key-value mapping	HashMap
Maintain order of keys as inserted	LinkedHashMap
Keep keys sorted automatically	TreeMap
Need thread-safe legacy map (older code)	Hashtable
Need high-performance thread-safe map	ConcurrentHashMap

🧠 **Interview Tip**

"I use HashMap for fast access, LinkedHashMap when I care about insertion order, and TreeMap when I need my keys sorted. For multithreaded use, I prefer ConcurrentHashMap."