**1. List<E> (Interface)**

* **Definition:** Root interface of ordered collections that allows **duplicate elements**.
* **Characteristics:**
  + Maintains **insertion order**.
  + Allows **random access** using index-based operations.
  + Implemented by **ArrayList**, **LinkedList**, **Vector**, and **Stack**

**2. ArrayList<E> (Class)**

* **Definition:** Implements List using a **dynamic array**.
* **Characteristics:**
  + **Faster random access (O(1))** due to indexed structure.
  + **Slower insertion/removal (O(n))** because shifting elements is needed.
  + **Allows null values**.
  + **Not thread-safe**, but can be synchronized manually (Collections.synchronizedList()).
* **Use Case:** Best for **fast retrieval operations**.

**3. LinkedList<E> (Class)**

* **Definition:** Implements List using a **doubly linked list**.
* **Characteristics:**
  + **Faster insertion/removal (O(1))** at the beginning or middle.
  + **Slower retrieval (O(n))** due to traversal needed.
  + **Allows null values**.
  + Implements **both List and Deque**, meaning it supports **queue operations** (offer(), poll()).
* **Use Case:** Best for **frequent insertions/deletions**.

#### ArrayList vs LinkedList:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **ArrayList** | **LinkedList** |
| Internal data structure | ArrayList uses dynamic array to store elements internally | LinkedList uses doubly Linked List to store elements internally |
| Manipulation | If  We need to insert or delete element in ArrayList, it may take O(n) complexity, as it internally uses array and we may have to shift elements to down in case of insertion or deletion operations | If  We need to insert or delete element in LinkedList, it will take O(1) complexity, as it internally uses doubly LinkedList |
| Search | Search is faster in ArrayList as uses array internally which is index based. So here time complexity is O(1) | Search is slower in LinkedList as uses doubly Linked List internally So here time complexity is O(n) |
| Interfaces | ArrayList implements List interface only, So it can be used as List only. | LinkedList implements List, Deque interfaces, so it can be used as List, Stack or Queue |
| Implements | ArrayList implements RandomAccess Interface | LinkedList does not implements RandomAccess interface |

**4. Vector<E> (Class)**

* **Definition:** Implements List using a **resizable array (like ArrayList) but thread-safe**.
* **Characteristics:**
  + **Thread-safe** (all methods synchronized).
  + **Slower than ArrayList** due to synchronization overhead.
  + **Allows null values**.
* **Use Case:** Best when **synchronization is required**, but **CopyOnWriteArrayList** is recommended instead.

**5. Stack<E> (Class)**

* **Definition:** A subclass of Vector that follows **LIFO (Last In, First Out)**.
* **Characteristics:**
  + Provides methods like **push()**, **pop()**, **peek()**.
  + **Thread-safe** (inherited from Vector).
  + **Slower than Deque** for stack operations.
* **Use Case:** When stack behavior is needed, but **Deque is preferred over Stack**

**6. CopyOnWriteArrayList<E> (Class)**

* **Definition:** A thread-safe version of ArrayList.
* **Characteristics:**
  + **Read operations are fast** as it maintains a separate copy for modifications.
  + **Write operations (add/remove) are slower** because a new copy is created for each update.
  + **Used in concurrent environments**.
* **Use Case:** Best for **multi-threaded applications where reads are more frequent than writes**

**Comparison Table of List Implementations**

| **Implementation** | **Ordering** | **Allows Null** | **Thread-Safe** | **Time Complexity (Avg)** |
| --- | --- | --- | --- | --- |
| **ArrayList** | Yes | ✅ Yes | ❌ No | O(1) (get), O(n) (insert/remove) |
| **LinkedList** | Yes | ✅ Yes | ❌ No | O(n) (get), O(1) (insert/remove) |
| **Vector** | Yes | ✅ Yes | ✅ Yes | O(1) (get), O(n) (insert/remove) |
| **Stack** | Yes | ✅ Yes | ✅ Yes | O(1) (push/pop) |
| **CopyOnWriteArrayList** | Yes | ✅ Yes | ✅ Yes | O(1) (get), O(n) (write) |

**1. Set<E> (Interface)**

* **Definition:** Root interface in Java Collections for a collection of **unique elements**.
* **Characteristics:**
  + **No duplicate elements allowed**.
  + **Does not guarantee any order** (depends on implementation).
  + Implemented by **HashSet**, **LinkedHashSet**, and **TreeSet**.

**2. HashSet<E> (Class)**

* **Definition:** Implements Set using a **hash table**.
* **Characteristics:**
  + **Unordered**, does not maintain insertion order.
  + **Allows null elements** (only one null value).
  + Uses **HashMap internally** (HashSet<E> is backed by HashMap<E, Object>).
  + **Best case: O(1) time complexity**, **Worst case: O(n) for high hash collisions**.
* **Use Case:** Best for **fast lookups and uniqueness constraints**.

Set<Integer> hashSet = new HashSet<>();

hashSet.add(10);

hashSet.add(20);

hashSet.add(10); // Duplicate ignored

System.out.println(hashSet); // Output: [10, 20] (unordered)

**3. LinkedHashSet<E> (Class)**

* **Definition:** Extends HashSet, maintains **insertion order**.
* **Characteristics:**
  + Uses **Linked List** + **HashMap** for order tracking.
  + **Faster than TreeSet**, but slower than HashSet due to ordering overhead.
* **Use Case:** When uniqueness is required **with preserved insertion order**.

Set<Integer> linkedHashSet = new LinkedHashSet<>();

linkedHashSet.add(30);

linkedHashSet.add(10);

linkedHashSet.add(20);

System.out.println(linkedHashSet); // Output: [30, 10, 20] (insertion order)

**4. TreeSet<E> (Class)**

* **Definition:** Implements Set using a **Red-Black Tree (Self-balancing BST)**.
* **Characteristics:**
  + **Sorted in natural order** (or custom comparator).
  + **Does not allow null elements**.
  + **O(log n) time complexity** for add, remove, and search operations.
* **Use Case:** Best when **sorted unique elements** are required.

Set<Integer> treeSet = new TreeSet<>();

treeSet.add(20);

treeSet.add(10);

treeSet.add(30);

System.out.println(treeSet); // Output: [10, 20, 30] (sorted order)

**5. EnumSet<E> (Class)**

* **Definition:** Optimized Set for **enum values**.
* **Characteristics:**
  + **Much faster than HashSet or TreeSet for enums**.
  + Uses **bitwise operations** internally.
  + Does **not allow null values**.
* **Use Case:** When a Set is needed for **enum types**.

enum Colors { RED, GREEN, BLUE }

Set<Colors> enumSet = EnumSet.of(Colors.RED, Colors.BLUE);

System.out.println(enumSet); // Output: [RED, BLUE]

**6. ConcurrentSkipListSet<E> (Class)**

* **Definition:** Thread-safe version of TreeSet, backed by **Skip List**.
* **Characteristics:**
  + **Sorted order maintained**, just like TreeSet.
  + **Thread-safe with better concurrency** than TreeSet.
  + **O(log n) time complexity**.
* **Use Case:** Used in **highly concurrent applications** where sorted order is required.

Set<Integer> concurrentSet = new ConcurrentSkipListSet<>();

concurrentSet.add(50);

concurrentSet.add(40);

concurrentSet.add(60);

System.out.println(concurrentSet); // Output: [40, 50, 60] (sorted)

**Comparison Table of Set Implementations**

| **Implementation** | **Ordering** | **Null Elements** | **Thread-Safe** | **Time Complexity (Avg)** |
| --- | --- | --- | --- | --- |
| **HashSet** | No order | ✅ Yes | ❌ No | O(1) |
| **LinkedHashSet** | Insertion Order | ✅ Yes | ❌ No | O(1) |
| **TreeSet** | Sorted Order | ❌ No | ❌ No | O(log n) |
| **EnumSet** | Enum Order | ❌ No | ❌ No | O(1) |
| **ConcurrentSkipListSet** | Sorted Order | ❌ No | ✅ Yes | O(log n) |

A screenshot of a hashtag

AI-generated content may be incorrect.

-------------------------------------------------------------------------------------------------------------------------------------------------

**1. Map<K, V> (Interface)**

* **Definition:** Root interface in Java Collections for key-value pairs.
* **Characteristics:**
  + **No duplicate keys allowed.**
  + Each key maps to **at most one value** (1:1 mapping).
  + Allows **null keys (only one)** and **multiple null values**.
  + Implemented by **HashMap**, **TreeMap**, **LinkedHashMap**, etc.

**2. HashMap<K, V> (Class)**

* **Definition:** Implements Map using a **hash table**.
* **Characteristics:**
  + **Unordered**, does not maintain insertion order.
  + Allows **one null key** and **multiple null values**.
  + Uses **Linked List (Java 7)** or **Tree (Java 8+)** for collision handling.
  + **Best case: O(1) time complexity**, **Worst case: O(n) for collision-heavy cases**.
* **Use Case:** General-purpose map where order **doesn’t matter**, and fast retrieval is needed.

**3. LinkedHashMap<K, V> (Class)**

* **Definition:** Extends HashMap but maintains **insertion order**.
* **Characteristics:**
  + Uses **Doubly Linked List** to maintain **insertion order**.
  + Slightly **slower** than HashMap due to order tracking.
* **Use Case:** When maintaining **insertion order** is important.

Map<String, Integer> linkedMap = new LinkedHashMap<>();

linkedMap.put("X", 10);

linkedMap.put("Y", 20);

System.out.println(linkedMap); // Output: {X=10, Y=20} (insertion order maintained)

**4. TreeMap<K, V> (Class)**

* **Definition:** Implements Map using a **Red-Black Tree** (Self-balancing BST).
* **Characteristics:**
  + **Sorted by keys (natural order or custom comparator).**
  + **Does not allow null keys** (throws NullPointerException for null keys).
  + **O(log n) time complexity** for operations due to tree structure.
* **Use Case:** When data needs to be **sorted by keys**.

Map<Integer, String> treeMap = new TreeMap<>();

treeMap.put(3, "Three");

treeMap.put(1, "One");

treeMap.put(2, "Two");

System.out.println(treeMap); // Output: {1=One, 2=Two, 3=Three} (sorted)

**5. Hashtable<K, V> (Class)**

* **Definition:** Legacy thread-safe Map implementation.
* **Characteristics:**
  + **Thread-safe** but **slower** than HashMap (synchronized methods).
  + **Does not allow null keys or null values**.
  + **Best case O(1), worst case O(n)** due to chaining.
* **Use Case:** Rarely used; prefer ConcurrentHashMap instead.

Map<String, Integer> table = new Hashtable<>();

table.put("A", 1);

table.put("B", 2);

System.out.println(table); // Output: {A=1, B=2} (unordered)

**6. ConcurrentHashMap<K, V> (Class)**

* **Definition:** Thread-safe version of HashMap with better concurrency.
* **Characteristics:**
  + **Uses segment-level locking (Java 7) or bucket-level CAS locks (Java 8)**.
  + **Much faster than Hashtable** due to reduced synchronization overhead.
  + **Allows null values but not null keys**.
* **Use Case:** When multiple threads access a Map concurrently.

Map<String, Integer> concurrentMap = new ConcurrentHashMap<>();

concurrentMap.put("P", 100);

concurrentMap.put("Q", 200);

System.out.println(concurrentMap); // Output: {P=100, Q=200}

**Comparison Table of Map Implementations**

| **Implementation** | **Ordering** | **Null Keys** | **Null Values** | **Thread-Safe** | **Time Complexity (Avg)** |
| --- | --- | --- | --- | --- | --- |
| **HashMap** | No order | ✅ Yes | ✅ Yes | ❌ No | O(1) |
| **LinkedHashMap** | Insertion Order | ✅ Yes | ✅ Yes | ❌ No | O(1) |
| **TreeMap** | Sorted Order | ❌ No | ✅ Yes | ❌ No | O(log n) |
| **Hashtable** | No order | ❌ No | ❌ No | ✅ Yes | O(1) |
| **ConcurrentHashMap** | No order | ❌ No | ✅ Yes | ✅ Yes | O(1) |

A screenshot of a computer

AI-generated content may be incorrect.

-------------------------------------------------------------------------------------------------------------------------------------------------**BlockingQueue**

* BlockingQueue makes it easy to implement producer-consumer design pattern by providing inbuilt blocking support for put() and take() method.
* Put() method will block if Queue is full while take() method will block if Queue is empty.
* Java 5 API provides two concrete implementation of BlockingQueue in form of ArrayBlockingQueue and LinkedBlockingQueue, both of them implement FIFO ordering of element.

**ArrayBlockingQueue**:

* **Underlying Structure:** Uses a fixed-size array.
* **Capacity:** Requires a capacity to be specified at the time of creation, which cannot be changed later.
* **Order:** Maintains elements in the order they were added (FIFO - First-In-First-Out).
* **Blocking Operations:** Supports blocking operations for both inserting and removing elements when the queue is full or empty, respectively.

**Use Cases:**

* Suitable for scenarios where you know the maximum number of elements before and require high throughput with low latency.
* Example: A bounded buffer in a producer-consumer scenario.

**LinkedBlockingQueue**:

* **Underlying Structure:** Uses a linked list.
* **Capacity:** Can be optionally bounded. If no capacity is specified, it acts as an unbounded queue limited only by available memory.
* **Order:** Maintains elements in the order they were added (FIFO).
* **Blocking Operations:** Supports blocking operations for both inserting and removing elements when the queue is full or empty, respectively.

**Use Cases:**

* Suitable for scenarios where the number of elements may grow dynamically and you don't need a fixed capacity.
* Example: Task queues in multi-threaded applications where the number of tasks can vary.

**PriorityBlockingQueue:**

* **Underlying Structure:** Uses a priority heap.
* **Capacity:** Unbounded queue (no fixed size).
* **Order:** Maintains elements according to their natural ordering or a specified comparator (priority order).
* **Blocking Operations:** Supports blocking operations for removing elements, but insertion does not block (since it is unbounded).

**Use Cases:**

* Suitable for scenarios where elements need to be processed based on their priority rather than their insertion order.
* Example: Task scheduling where tasks have different priorities.

**Concurrent HashMap**:

* **ConcurrentHashMap** is a data structure is designed to be used in multi-threaded applications where multiple threads are reading and writing to the hash map in concurrently.
* **Lock-free:** Concurrent HashMap is designed to be lock-free, which means that there is no need to acquire a lock in order to read or write the data.
* **Scalable**: ConcurrentHashMap is designed to be scalable, that means it can handle large amount of data also.

**Linked List**:

* **Single Linked List**: Navigation is forward only. It contains two parts

1. **Data**: Contains the actual data
2. **Link**: Contains the address of the next node of the list.

* **Double Linked List:** Forward and backward navigation is possible
* **Circular Linked List:** Last element is linked to first element

**Deque:**

* **Deque (Double-Ended Queue)** is a data structure that allows insertion and deletion from both ends (front and rear).
* Implemented by **ArrayDeque and LinkedList** in Java
* Supports **FIFO (Queue)** and **LIFO (Stack)** operations.
* Supports both **insertion and deletion from both ends**.
* Does not allow **null** elements.

**ArrayDeque:**

* **ArrayDeque** is a resizable array-based implementation of Deque.
* Faster than **LinkedList** for add/remove operations.
* Does **not** allow null elements.
* Provides better performance than **Stack** and **LinkedList**.

**PriorityQueue**

* A **PriorityQueue** in Java is a **heap-based queue** that orders elements based on **natural ordering** or a **custom comparator**.
* Implement **Queue** and uses a **binary heap**.
* Not thread-safe (use **PriorityBlockingQueue** for concurrency).
* Supports **ordering** using natural order (Comparable<T>) or **Comparator**.
* **Does not allow null values**.
* **Time complexity**: O(log n) for insert and remove operations.
* **Use Cases**: Task Scheduling, Dijkstra’s Algorithm (Shortest Path), Huffman Encoding

**LinkedBlockingQueue**

* A **LinkedBlockingQueue** is a **thread-safe, optionally bounded queue** based on a **linked list**.
* **Thread-safe** (uses **ReentrantLock** for concurrency control).
* Can be **bounded** (fixed capacity) or **unbounded**.
* **Uses linked nodes**, so there is **no capacity restriction** unless specified.
* Allows **FIFO (First-In-First-Out)** ordering.
* Supports **blocking operations** (e.g., take(), put()).
* **Time complexity**: O(1) for insert/remove operations.
* **Use Cases**: Multi-threaded Producer-Consumer Models, Task Queues in Executors (ThreadPoolExecutor)

Log Processing Systems

**ArrayBlockingQueue**

* An **ArrayBlockingQueue** is a **bounded blocking queue** backed by an **array**.
* Use Cases: Thread Pools (Executors.newFixedThreadPool()), Rate Limiting

**Key Features:**

* **Thread-safe** (internally uses a **ReentrantLock**).
* **Fixed-size queue** (must specify capacity at initialization).
* Maintains **FIFO (First-In-First-Out) ordering**.
* Uses **array-based storage**, leading to **better cache locality**.
* Supports **blocking operations** (put(), take()).
* **Time complexity**: O(1) for insert/remove operations.

| **Feature** | **PriorityQueue** | **LinkedBlockingQueue** | **ArrayBlockingQueue** |
| --- | --- | --- | --- |

|  |  |  |  |
| --- | --- | --- | --- |
| **Thread-Safe?** | ❌ No | ✅ Yes | ✅ Yes |

|  |  |  |  |
| --- | --- | --- | --- |
| **Ordering** | ✅ Sorted (Natural/Comparator) | ✅ FIFO | ✅ FIFO |

|  |  |  |  |
| --- | --- | --- | --- |
| **Bounded?** | ❌ Unbounded | ✅ Optionally Bounded | ✅ Bounded |

|  |  |  |  |
| --- | --- | --- | --- |
| **Blocking Operations?** | ❌ No | ✅ Yes | ✅ Yes |

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **Binary Heap** | **Linked List** | **Array** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Null Values?** | ❌ No | ❌ No | ❌ No |

|  |  |  |  |
| --- | --- | --- | --- |
| **Use Case** | Task Scheduling | Multi-threaded Processing | Fixed-Size Buffer |

| **Queue** | **Stack** |
| --- | --- |
| Queue works on the principle of the First-In-First-Out (FIFO) approach. | Stack works on a Last-In-First-Out (LIFO) basis. |
| Insertion and deletion in the queue take place at different ends. | Insertion and deletion are performed from the same end called the top of the stack. |
| Enqueue is the name of Insertion and dequeue is the deletion of elements. | Push is insertion and Pop is the deletion of elements in Stack. |
| It has two pointers- one to the first element of the list (front) and one to the last (rear). | It only has one pointer pointing to the top element. |

| **FailFast** | **FailSafe** |
| --- | --- |
| While iterating, no modification of a collection is allowed. | Allows modification while iterating. |
| Uses original collection for traversing. | Uses a copy of the original collection. |
| No extra memory required. | Needs extra memory. |
| Throws ConcurrentModificationException. | No exception is thrown. |

| **Singly Linked List** | **Doubly Linked List** |
| --- | --- |
| Each node of the singly linked list consists of a data and a pointer to the next node. | A doubly linked list consists of data, a pointer to the next node, and a pointer to the previous node. |
| The singly-linked list can be traversed using the next pointer. | A doubly linked list can be traversed using both previous and next pointer. |
| The singly-linked list takes less space compared to a doubly-linked list. | The doubly linked list takes up a lot of memory space. |
| Element access is not very efficient. | Element access is efficient. |

| **ArrayList** | **Vector** |
| --- | --- |
| ArrayList is non-synchronized. | Vector is synchronized. |
| ArrayList is not a legacy class. | Vector is a legacy class. |
| ArrayList increases size by half of ArrayList when an element is inserted beyond its size. | Vector increases its size by double when an element is inserted beyond its size. |
| ArrayList is not thread-safe | Vector is a thread-safe. |

| **Iterable** | **Iterator** |
| --- | --- |
| It is Java.lang package interface. | It is Java.util package interface. |
| Yields only one abstract method known as the Iterator. | It comes with two abstract methods- hasNext and next. |
| Represents a series of elements that can be traversed. | Stands for objects with iteration state. |

| **Set** | **List** |
| --- | --- |
| Set implements Set interface. | The list implements the List interface. |
| Set is an unordered set of elements. | The list is an ordered set of elements. |
| The set doesn’t maintain the order of elements during insertion. | List retains the order of elements during insertion. |
| The set doesn’t allow duplicate values. | The list allows duplicate values. |
| The set does not contain any legacy class. | List contains Vector, a legacy class. |
| The set allows only one null value. | No restriction on the number of null values in List. |
| We cannot use ListIterator to traverse a set. | ListIterator can traverse List in any direction. |

| **Collection** | **Collections** |
| --- | --- |
| It is an interface. | It is class. |
| The collection represents a group of objects as a single entity. | Collections define different methods of utility for collection objects. |
| It is Collection Framework’s root interface. | Collections are a utility class. |
| It derives Collection Framework’s data structures. | Collections contain many different static methods for aiding in manipulating the data structure. |

| **Array** | **ArrayList** |
| --- | --- |
| The array is a strongly typed class. | ArrayList is a loosely typed class. |
| Array can’t be resized dynamically, its dimension is static. | ArrayList can be resized dynamically. |
| An array doesn’t need boxing and unboxing of elements. | ArrayList needs boxing and unboxing of elements. |

| **HashMap** | **HashTable** |
| --- | --- |
| HashMap inherits AbstractMap class | HashTable inherits Dictionary class. |
| HashMap is not synchronized. | HashTable is synchronized. |
| HashMap allows multiple null values but only one null key. | HashTable does not allow a null value or key. |
| HashMap is faster. | HashTable is slower than HashMap. |
| HashMap can be traversed by Iterator. | HashTable cannot be traversed using iterator or enumerator. |

**Comparable (I):**

* Java Comparable interface is used to order the objects of user-defined class.
* This interface is found in java.lang package and contains only one method named compareTo(Object).
* It provide single sorting sequence only i.e. you can sort the elements on based on single data member only. For example it may be rollno, name, age or anything else.
* **public int compareTo(Object obj):** is used to compare the current object with the specified object.
* **public void sort(List list):** is used to sort the elements of List. List elements must be of Comparable type.

**Comparator (I):**

* Java Comparator interface is used to order the objects of user-defined class.
* This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2)
* It provides multiple sorting sequence i.e. you can sort the elements on the basis of any data member, for example rollno, name, age or anything else.
* **public int compare(Object obj1,Object obj2):** compares the first object with second object.
* **public void sort(List list, Comparator c):** is used to sort the elements of List by the given Comparator.

**Collections (C)**

* Java collection class is used exclusively with static methods that operate on or return collections.
* It inherits Object class.
* Java Collection class supports the **polymorphic algorithms** that operate on collections.
* Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.