📖 Java Collection Framework - Complete Guide (Basic to Advanced)

📌 Chapter 1: Introduction to Java Collection Framework

- What is a Collection?
- Need for Collections over Arrays
- Java Collection Framework Overview
- Benefits of Using Collections
- Collection Framework Hierarchy

📌 Chapter 2: Iterable and Collection Interface

- Understanding `Iterable<T>`
- Methods of the `Iterable` Interface
- Understanding `Collection<T>` Interface
- Important Methods of `Collection` Interface

📌 Chapter 3: List Interface (Ordered Collection)

- Introduction to `List<T>` Interface
- Implementations:
  - `ArrayList` (Dynamic Array, Fast Read)
  - `LinkedList` (Doubly Linked List, Fast Insert/Delete)
  - `Vector` (Thread-Safe, Legacy)
  - `Stack` (LIFO, Legacy)
  - `CopyOnWriteArrayList` (Thread-Safe Variant of ArrayList)
- **Operations on List** (Add, Remove, Search, Sort)
- **Performance Comparison of List Implementations**

📌 Chapter 4: Set Interface (Unique Elements Collection)

- Introduction to `Set<T>` Interface
- Implementations:
  - `HashSet` (Unordered, Unique, Uses Hashing)

- `LinkedHashSet` (Maintains Insertion Order)
- `TreeSet` (Sorted, Uses Red-Black Tree)
- `EnumSet` (Efficient Enum Collection)
- `ConcurrentSkipListSet` (Thread-Safe Sorted Set)
- `CopyOnWriteArraySet` (Thread-Safe Set)

- **Set Operations (Union, Intersection, Difference, Subset)**

📌 Chapter 5: Queue Interface (FIFO Data Structure)

- Introduction to `Queue<T>` Interface
- Implementations:
  - `LinkedList` (Can Be Used as Queue)
  - `PriorityQueue` (Min-Heap Implementation)
  - `Deque` (Double-Ended Queue)
  - `ArrayDeque` (Resizable Array-Based Deque)
  - `ConcurrentLinkedQueue` (Thread-Safe Queue)
  - **BlockingQueues (Used in Multi-threading)**:
    - `BlockingQueue` (Interface Overview)
    - `ArrayBlockingQueue`
    - `LinkedBlockingQueue`
    - `PriorityBlockingQueue`
    - `SynchronousQueue`
    - `DelayQueue`

📌 Chapter 6: Map Interface (Key-Value Pair Collection)

- Introduction to `Map<K, V>` Interface
- Implementations:
  - `HashMap` (Unordered Key-Value Mapping)
  - `LinkedHashMap` (Maintains Insertion Order)
  - `TreeMap` (Sorted Key-Value Mapping)
  - `Hashtable` (Thread-Safe, Legacy)

- `ConcurrentHashMap` (Thread-Safe HashMap)

- **Map Operations (Put, Get, Remove, Search, Iterate)**

📌 Chapter 7: Comparators and Sorting in Collections

- `Comparable<T>` Interface (Natural Sorting)
- `Comparator<T>` Interface (Custom Sorting)
- Sorting Lists, Sets, and Maps with Comparators

📌 Chapter 8: Collections Utility Class (Helper Methods)

- Sorting Collections (`Collections.sort()`)
- Searching in Collections (`Collections.binarySearch()`)
- Immutable Collections (`Collections.unmodifiableList()`)
- Thread-Safe Collections (`Collections.synchronizedList()`)

📌 Chapter 9: Thread-Safety in Java Collections

- Synchronization and Concurrent Collections
- `CopyOnWriteArrayList`, `CopyOnWriteArraySet`
- `ConcurrentHashMap`, `ConcurrentSkipListSet`
- Performance Comparison of Synchronized and Concurrent Collections

---

📌 Chapter 10: Best Practices and Performance Optimization

- When to Use Which Collection?
- Performance Considerations for Different Data Structures
- Avoiding `NullPointerException` in Collections
- Optimizing Memory and CPU Usage in Collections

📌 Chapter 1: Introduction to Java Collection Framework

1️⃣ What is a Collection?

A **collection** in Java is a **group of objects** stored together. It helps in **storing, retrieving, manipulating, and processing** data efficiently.

Think of a **collection** as a **container** (like a box) where you can store multiple objects.

📍 Example:

Imagine you have a list of student names. You can store them using **collections** instead of creating multiple variables.

```
List<String> students = new ArrayList<>();
students.add("John");
students.add("Emma");
students.add("David");
```

Here, `students` is a **collection** that stores multiple student names **together**.

💡 Key Features of Collections:

✔ **Dynamic Size** - Unlike arrays, collections can grow and shrink in size dynamically.
✔ **Efficient Operations** - Collections provide powerful methods for searching, sorting, and filtering data.
✔ **Flexible Data Structures** - Supports different structures like **lists, sets, and queues**.

2️⃣ Need for Collections over Arrays

Before collections, **arrays** were the only way to store multiple elements in Java. But arrays have some **limitations**.

❌ Limitations of Arrays:

1️⃣ **Fixed Size** - Once an array is created, its size **cannot** be changed.
2️⃣ **No Built-in Methods** - Arrays do not provide methods for common tasks like searching or sorting.
3️⃣ **Only Works with Indexes** - Arrays can only be accessed using **index numbers**, which is not always convenient.
4️⃣ **Inefficient Insertion/Deletion** - Adding or removing elements in the middle of an array is difficult.

☑ Why Collections are Better?

✔ **Dynamic Size** - Collections **automatically resize** when adding/removing elements.
✔ **Rich APIs** - Collections have built-in methods for sorting, searching, and filtering.
✔ **More Flexibility** - Collections support different data structures like **lists, sets, and queues**.
✔ **Easy to Use** - No need to manually manage indexes; you can directly use powerful methods.

3️⃣ Java Collection Framework Overview

The **Java Collection Framework (JCF)** is a set of **predefined classes and interfaces** that help store and process data efficiently.

It provides **ready-made implementations** for **Lists, Sets, Queues, and Maps**, so we don't have to create them from scratch.

🛠️ Components of Java Collection Framework:

1. **Interfaces** - Define the structure (e.g., `List`, `Set`, `Queue`, `Map`).
2. **Classes** - Implement the interfaces (e.g., `ArrayList`, `HashSet`, `LinkedList`).
3. **Methods** - Predefined operations (e.g., `add()`, `remove()`, `contains()`, `sort()`).

4️⃣ Benefits of Using Collections

1️⃣ Dynamic Memory Allocation

Unlike arrays, collections do not require a **fixed size** at the beginning. They **grow and shrink** dynamically as needed.

2️⃣ Predefined Methods

Collections provide **built-in methods** like `add()`, `remove()`, `contains()`, `size()`, making operations **easier**.

3️⃣ Better Performance

Collections are optimized for **fast searching, insertion, and deletion** operations compared to arrays.

4️⃣ Easy Iteration

Collections support **iterators** and **enhanced for-loops**, making traversal **simpler**.

```
for (String name : students) {
    System.out.println(name);
}
```

This is much easier compared to using **indexes in arrays**.

5️⃣ Supports Thread Safety

Java provides **thread-safe** collections like `Vector` and `ConcurrentHashMap`, making them **safe for multi-threading**.

5️⃣ Collection Framework Hierarchy (Complete Structure)

The **Java Collection Framework** is structured as follows:

📌 Main Interfaces:

1. **Iterable** - The root interface for all collections.
2. **Collection** - Extends `Iterable` and is the base for `List`, `Set`, and `Queue`.
3. **Map** - Stores data in **key-value pairs** (not part of `Collection`).

📌 Collection Types:

🔹 List (Ordered, Allows Duplicates)

- `ArrayList`
- `LinkedList`
- `Vector`
- `Stack`
- `CopyOnWriteArrayList`

🔹 Set (Unique Elements, No Duplicates)

- `HashSet`
- `LinkedHashSet`
- `TreeSet`
- `EnumSet`
- `CopyOnWriteArraySet`

🔹 Queue (FIFO Data Structure)

- `PriorityQueue`
- `ArrayDeque`
- `BlockingQueue` (for multi-threading)

🔹 Map (Key-Value Pair Collection)

- `HashMap`
- `LinkedHashMap`
- `TreeMap`
- `Hashtable`
- `ConcurrentHashMap`

📌 Summary of Chapter 1

Feature	Arrays	Collections

Size	Fixed	Dynamic
Built-in Methods	No	Yes
Efficiency	Low (Slow Insert/Delete)	High (Optimized)
Thread Safety	No	Yes (Some classes)
Data Structure Options	Only One (Array)	List, Set, Queue, Map

🌟 Key Takeaways:

✔ **Collections are more powerful than arrays** because they provide **flexibility and efficiency**.
✔ The **Java Collection Framework (JCF)** provides **ready-made classes and methods** for handling data efficiently.
✔ Different **types of collections** (`List`, `Set`, `Queue`, `Map`) are available for different use cases.

📌 Chapter 2: Iterable and Collection Interface

1️⃣ Understanding `Iterable<T>` Interface

📍 What is `Iterable<T>`?

- `Iterable<T>` is the **root interface** of the Java Collection Framework.
- It **allows collections to be iterated (looped) using a `for-each` loop**.
- All major collection classes like `ArrayList`, `LinkedList`, `HashSet`, etc., implement `Iterable<T>`.

📍 Why is `Iterable<T>` Important?

1️⃣ It allows **for-each loop** to work on collections.
2️⃣ It provides an `Iterator` to iterate through elements **one by one**.

📍 Simple Example of `Iterable<T>`

```
import java.util.*;

public class IterableExample {
    public static void main(String[] args) {
        List<String> students = new ArrayList<>();
        students.add("Alice");
        students.add("Bob");
        students.add("Charlie");

        // Using for-each loop (Internally uses Iterable)
        for (String name : students) {
            System.out.println(name);
        }
```

```
    }
}
```

☑ **Here,** `ArrayList` **implements** `Iterable<T>`, so we can use a **for-each loop** to iterate through elements.

---

2 Methods of the `Iterable<T>` Interface

The `Iterable<T>` interface provides **only one method** that must be implemented:

- `Iterator<T> iterator()`
  - Returns an **Iterator** to go through elements one by one.
  - The `Iterator` provides **three important methods**:

Method	Description
`hasNext()`	Returns `true` if more elements are present.
`next()`	Returns the next element.
`remove()`	Removes the current element.

📌 Example: Using `Iterator`

```
import java.util.*;

public class IteratorExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(10, 20, 30, 40));

        Iterator<Integer> it = numbers.iterator();  // Getting the iterator

        while (it.hasNext()) {  // Checking if more elements exist
            System.out.println(it.next());  // Printing the next element
        }
    }
}
```

☑ **Here, we manually iterate over** `ArrayList` **using an** `Iterator`**.**

---

3 Understanding `Collection<T>` Interface

📌 What is `Collection<T>`?

- The `Collection<T>` interface **extends** `Iterable<T>`.
- It provides **basic functionalities** for handling collections of objects.
- **All major collection types (**`List`**,** `Set`**,** `Queue`**) implement** `Collection<T>`**.**

📌 Key Features of `Collection<T>`

✔ Allows adding and removing elements.
✔ Supports operations like checking size, clearing the collection, and checking if it's empty.
✔ Implements `Iterable<T>`, so it can be used in a **for-each loop**.

📌 Collection Interface Hierarchy

```
Iterable<T>
   │
   ├── Collection<T>
         ├── List<T>  (Ordered, Duplicates Allowed)
         ├── Set<T>  (Unordered, Unique Elements)
         ├── Queue<T>  (FIFO)
```

☑ **So, every `List`, `Set`, and `Queue` class is part of `Collection<T>`.**

📌 Example: Using `Collection<T>` Methods

```
import java.util.*;

public class CollectionExample {
    public static void main(String[] args) {
        Collection<String> names = new ArrayList<>();  // Collection interface
reference
        names.add("John");
        names.add("Emma");
        names.add("David");

        System.out.println("Collection: " + names);
    }
}
```

☑ **Even though `Collection` is an interface, we can use `ArrayList` as an implementation.**

4️⃣ Important Methods of `Collection<T>` Interface

The `Collection<T>` interface provides **various useful methods**. Let's discuss the most important ones:

Method	Description
`add(T element)`	Adds an element to the collection.
`remove(Object obj)`	Removes an element from the collection.
`size()`	Returns the number of elements in the collection.
`clear()`	Removes all elements from the collection.
`contains(Object obj)`	Checks if a specific element exists.
`isEmpty()`	Returns `true` if the collection is empty.

📌 Example: Using Collection Methods

```
import java.util.*;

public class CollectionMethodsExample {
    public static void main(String[] args) {
        Collection<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        System.out.println("Size: " + names.size());  // 3
        System.out.println("Contains Bob? " + names.contains("Bob"));  // true
        names.remove("Bob");
        System.out.println("After removal: " + names);
        names.clear();
        System.out.println("Is collection empty? " + names.isEmpty());  // true
    }
}
```

☑ **Here, we added elements, checked their existence, removed an element, and cleared the collection.**

📌 Summary of Chapter 2

Feature	`Iterable<T>`	`Collection<T>`
What is it?	Root interface for iteration	Extends `Iterable`, supports basic collection operations
Key Method(s)	`iterator()`	`add()`, `remove()`, `size()`, `clear()`, `contains()`
Usage	Enables `for-each` loops	Used for storing and managing collections
Implemented By	`Collection`, `List`, `Set`, `Queue`	`ArrayList`, `HashSet`, `LinkedList`, etc.

🌟 Key Takeaways:

✔ **`Iterable<T>` is the root interface** that allows iteration through collections.
✔ **`Collection<T>` extends `Iterable<T>` and adds basic collection functionalities.**
✔ `Collection` provides important methods like `add()`, `remove()`, `size()`, `clear()`, and `contains()`.
✔ **All major collection types (`List`, `Set`, `Queue`) implement `Collection<T>`.**

📌 Chapter 3: List Interface (Ordered Collection)

1️⃣ Understanding `List<T>` Interface

📍 What is `List<T>`?

- `List<T>` is an **ordered collection** in Java that allows **duplicate elements**.
- It extends the `Collection<T>` interface.
- **Order matters** in `List`, meaning elements are stored in the same sequence in which they are inserted.
- Unlike `Set<T>`, it **allows duplicate values**.

📍 Characteristics of `List<T>`

☑ **Maintains Insertion Order** – Elements are stored in the order they were added.
☑ **Allows Duplicates** – You can have multiple occurrences of the same element.
☑ **Indexed Access** – Elements can be accessed using **index positions (0, 1, 2, ...)**.
☑ **Can Contain `null` Values** – Unlike some `Set` implementations, `List` can store `null`.

📍 List Interface Hierarchy

```
Iterable<T>
   │
   ├── Collection<T>
         │
         ├── List<T>  (Ordered, Duplicates Allowed)
               ├── ArrayList<T>  (Fast Read, Dynamic Array)
               ├── LinkedList<T>  (Fast Insert/Delete, Doubly Linked List)
               ├── Vector<T>  (Thread-Safe, Legacy)
               ├── Stack<T>  (LIFO, Legacy)
               ├── CopyOnWriteArrayList<T>  (Thread-Safe Variant of ArrayList)
```

☑ **So, every `ArrayList`, `LinkedList`, `Vector`, and `Stack` is a part of `List<T>`.**

2️⃣ Methods of `List<T>` Interface

The `List<T>` interface provides various useful methods:

Method	Description
`add(T element)`	Adds an element to the list.
`add(int index, T element)`	Inserts an element at a specific index.
`remove(int index)`	Removes the element at a given index.
`remove(Object obj)`	Removes the first occurrence of a specified object.
`get(int index)`	Retrieves the element at a specific index.

`set(int index, T element)`	Replaces the element at the given index.
`indexOf(T element)`	Returns the first index of an element (or `-1` if not found).
`lastIndexOf(T element)`	Returns the last index of an element.
`subList(int fromIndex, int toIndex)`	Extracts a portion of the list.
`sort(Comparator<T> c)`	Sorts the list using a comparator.

3️⃣ Example: Basic Operations with `List<T>`

📍 Example: Using `ArrayList` as `List`

```
import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();  // Using List<T> reference
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        names.add("Alice");  // Duplicates allowed

        System.out.println("List: " + names);  // [Alice, Bob, Charlie, Alice]

        System.out.println("Element at index 1: " + names.get(1));  // Bob

        names.remove(2);  // Removing "Charlie"
        System.out.println("After removal: " + names);  // [Alice, Bob, Alice]

        names.set(1, "David");  // Replacing "Bob" with "David"
        System.out.println("After set: " + names);  // [Alice, David, Alice]
    }
}
```

4️⃣ Implementations of `List<T>` Interface

The `List<T>` interface has multiple implementations. Let's discuss each one in detail.

📌 1. `ArrayList<T>` (Dynamic Array, Fast Read)

- Uses **dynamic array** to store elements.
- **Fast retrieval (O(1))**, but **slower insertion & deletion (O(n))**.
- **Best when searching elements frequently**.
- **Not thread-safe** (use `CopyOnWriteArrayList` for thread safety).

📌 2. `LinkedList<T>` (Doubly Linked List, Fast Insert/Delete)

- Uses **doubly linked list** to store elements.
- **Fast insertion & deletion (O(1))**, but **slower retrieval (O(n))**.
- **Best when adding/removing elements frequently**.
- **Not thread-safe** (explicit synchronization needed).

📌 3. `Vector<T>` (Thread-Safe, Legacy)

- Similar to `ArrayList`, but **synchronized (thread-safe)**.
- **Slower than `ArrayList` due to synchronization overhead**.
- **Rarely used** today (use `CopyOnWriteArrayList` instead).

📌 4. `Stack<T>` (LIFO, Legacy)

- **Follows Last-In-First-Out (LIFO) order**.
- Used for **stack operations like undo, recursion, and function calls**.
- Internally extends `Vector<T>`, making it **thread-safe**.

📌 5. `CopyOnWriteArrayList<T>` (Thread-Safe Variant of `ArrayList`)

- **Best for concurrent applications** where **read operations are more frequent**.
- Each modification creates a **new copy of the list**, avoiding concurrent modification issues.
- **Higher memory consumption** due to copying.

📌 Summary of `List<T>` Implementations

Implementation	Internal Structure	Performance	Thread Safety	Best Use Case
ArrayList	Dynamic Array	Fast Read (O(1)), Slow Insert/Delete (O(n))	❌ No	Frequent Read Operations
LinkedList	Doubly Linked List	Slow Read (O(n)), Fast Insert/Delete (O(1))	❌ No	Frequent Insert/Delete
Vector	Dynamic Array	Similar to `ArrayList`, but slower due to synchronization	☑ Yes	Legacy Code, Multi-threading
Stack	Dynamic Array (LIFO)	LIFO operations, similar to `Vector`	☑ Yes	Stack Operations (Undo, Function Calls)
CopyOnWriteArrayList	Dynamic Array	Fast Read, Slow Write	☑ Yes	Concurrent Read

	(Copy on Write)			Operations

📌 Key Takeaways

✔ **`List<T>` is an ordered collection that allows duplicates and indexed access.**
✔ It has multiple implementations:

- `ArrayList` (Fast read, slow insert/delete)
- `LinkedList` (Slow read, fast insert/delete)
- `Vector` (Thread-safe, legacy)
- `Stack` (LIFO structure)
- `CopyOnWriteArrayList` (Thread-safe variant of `ArrayList`)
  ✔ Choose the right `List<T>` implementation based on **performance needs**.

📌 ArrayList<T> (Dynamic Array, Fast Read)

1️⃣ What is `ArrayList<T>`?

- `ArrayList<T>` is a **dynamic array implementation** of the `List<T>` interface.
- It can **grow and shrink dynamically** based on the number of elements.
- **Fast read operations (O(1))**, but **slower insert/delete (O(n))** compared to `LinkedList`.
- It **allows duplicate elements** and **maintains insertion order**.

2️⃣ How `ArrayList<T>` Works Internally?

📌 **Internal Structure**

- `ArrayList` internally uses an **array** to store elements.
- When the **capacity is full**, it **creates a new array** with **1.5x larger size** and **copies old elements into it**.
- This is why `ArrayList` is **fast for reading**, but **slow for insertion/deletion at the beginning or middle**.

🛠 **Example:**
If an `ArrayList` has **capacity 10**, and we try to add the 11th element:

- Java **creates a new array of size 15 (1.5x of 10)**.
- It **copies old 10 elements** into the new array.

- It **adds the 11th element** in the newly allocated space.

☑ **This process is called "dynamic resizing".**

3️⃣ How to Create an `ArrayList<T>`?

📍 Creating an `ArrayList` in Java

```
import java.util.*;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>(); // Creating an empty ArrayList
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        System.out.println("ArrayList: " + names);  // [Alice, Bob, Charlie]
    }
}
```

☑ **Here, we created an `ArrayList<String>` and added elements.**

4️⃣ ArrayList Methods (With Examples)

📌 1. `add(E element)` → Add element to the list

Adds an element at the **end of the list**.

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
System.out.println(list);  // Output: [Java, Python]
```

📌 2. `add(int index, E element)` → Insert at a specific index

Inserts an element at the given **index** (shifts existing elements).

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.add(1, "C++");  // Insert "C++" at index 1
System.out.println(list);  // Output: [Java, C++, Python]
```

⏳ **Time Complexity:** `O(n)`, because elements need to shift.

📌 3. `get(int index)` → Retrieve element at index

Gets the **element present at the given index**.

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
System.out.println(list.get(1));  // Output: Python
```

☑ **Fast (O(1)) since `ArrayList` provides direct access using an index.**

📌 4. `set(int index, E element)` → Update element at index

Replaces the element at the given **index** with a new value.

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.set(1, "C++");  // Replace Python with C++
System.out.println(list);  // Output: [Java, C++]
```

☑ **Efficient operation (O(1)).**

📌 5. `remove(int index)` → Remove element by index

Removes the **element at the specified index**, shifting elements left.

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.add("C++");
list.remove(1);  // Remove "Python" (index 1)
System.out.println(list);  // Output: [Java, C++]
```

⏳ **Time Complexity:** `O(n)`, because elements shift left.

📌 6. `remove(Object obj)` → Remove element by value

Removes the **first occurrence** of the given value.

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.add("C++");
list.remove("Python");  // Remove "Python"
System.out.println(list);  // Output: [Java, C++]
```

☑ **Returns `true` if element was found and removed.**

📌 7. `size()` → Get the number of elements

Returns the **total number of elements** in the list.

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
System.out.println(list.size());  // Output: 2
```

📌 8. `contains(E element)` → Check if element exists

Checks if the **list contains a specific element**.

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
System.out.println(list.contains("Python"));  // Output: true
System.out.println(list.contains("C++"));  // Output: false
```

📌 9. `indexOf(E element)` → Get index of first occurrence

Returns the **index of the first occurrence** of an element (`-1` if not found).

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.add("Java");
System.out.println(list.indexOf("Java"));  // Output: 0
System.out.println(list.indexOf("C++"));  // Output: -1 (not found)
```

📌 10. `lastIndexOf(E element)` → Get index of last occurrence

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.add("Java");
System.out.println(list.lastIndexOf("Java"));  // Output: 2
```

📌 11. `subList(int fromIndex, int toIndex)` → Get portion of list

Extracts a **portion of the list** (from `fromIndex` to `toIndex-1`).

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.add("C++");
list.add("JavaScript");
System.out.println(list.subList(1, 3));  // Output: [Python, C++]
```

📌 12. `clear()` → Remove all elements

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.clear();
System.out.println(list);  // Output: []
```

📌 When to Use `ArrayList<T>`?

✔ **Best for fast random access (O(1)).**
✔ **Use when searching elements frequently.**
✔ **Avoid if you need frequent insertions/deletions in the middle.**

📌 Summary

Method	Description
`add(E e)`	Adds an element to the end
`add(int index, E e)`	Inserts element at a specific index
`get(int index)`	Retrieves element at an index
`set(int index, E e)`	Replaces element at an index
`remove(int index)`	Removes element at index
`remove(Object obj)`	Removes first occurrence of element
`contains(E e)`	Checks if element exists
`size()`	Returns the number of elements
`clear()`	Removes all elements
`subList(int from, int to)`	Gets a portion of the list

📌 Deep Dive into `LinkedList<T>` (Doubly Linked List, Fast Insert/Delete)

🚀 **In this chapter, we will explore `LinkedList<T>` in depth**. We will cover **how it works internally, when to use it, and all its methods with easy explanations and examples.**

1️⃣ What is `LinkedList<T>`?

- `LinkedList<T>` is a **doubly linked list** implementation of the `List<T>` interface.

- Unlike `ArrayList`, it does **not use an array internally**. Instead, it uses **nodes (objects) that are linked together**.
- Each node contains **3 parts**:
    1. **Data (Element)**
    2. **Reference to the next node**
    3. **Reference to the previous node**

🔗 **Structure of a LinkedList node:**

```
[Prev | Data | Next]  <-->  [Prev | Data | Next]  <-->  [Prev | Data | Next]
```

☑ **Key Features of `LinkedList<T>`:**
✔ **Fast insertions and deletions (`O(1)`)** at the beginning and middle.
✔ **Slower searching (`O(n)`)** because elements are not indexed.
✔ **Can be used as a `Queue` or `Stack` (since it has `addFirst()` and `removeFirst()`).**

2️⃣ How `LinkedList<T>` Works Internally?

📌 `LinkedList<T>` maintains a reference to:

- **First Node (`head`)** → Points to the first element.
- **Last Node (`tail`)** → Points to the last element.

Insertion Process:

- If inserting at the **beginning** (`addFirst()`), it updates the `head` to the new node.
- If inserting at the **end** (`addLast()`), it updates the `tail` to the new node.

Deletion Process:

- If deleting at the **beginning** (`removeFirst()`), the `head` moves to the next node.
- If deleting at the **end** (`removeLast()`), the `tail` moves to the previous node.

☑ **This makes insertion and deletion fast (`O(1)`).**

3️⃣ How to Create a `LinkedList<T>`?

📍 Creating a `LinkedList` in Java

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> names = new LinkedList<>(); // Creating a LinkedList
        names.add("Alice");
```

```
        names.add("Bob");
        names.add("Charlie");

        System.out.println("LinkedList: " + names);  // Output: [Alice, Bob, Charlie]
    }
}
```

☑ **Here, we created a `LinkedList<String>` and added elements.**

🔷 `LinkedList<T>` Methods (With Examples)

📌 1. `add(E element)` → Add element at the end

Adds an element to the **end of the list**.

```
LinkedList<String> list = new LinkedList<>();
list.add("Java");
list.add("Python");
System.out.println(list);  // Output: [Java, Python]
```

📌 2. `add(int index, E element)` → Insert at a specific index

Inserts an element at the given **index** (shifts existing elements).

```
LinkedList<String> list = new LinkedList<>();
list.add("Java");
list.add("Python");
list.add(1, "C++");  // Insert "C++" at index 1
System.out.println(list);  // Output: [Java, C++, Python]
```

☑ **Faster than `ArrayList` for insertions in the middle (`O(1)`).**

📌 3. `addFirst(E element)` → Insert at the beginning

Inserts an element **at the start of the list**.

```
LinkedList<String> list = new LinkedList<>();
list.add("Python");
list.addFirst("Java");  // Add "Java" at the beginning
System.out.println(list);  // Output: [Java, Python]
```

📌 4. `addLast(E element)` → Insert at the end

Same as `add()`, but explicitly adds at the end.

```
LinkedList<String> list = new LinkedList<>();
list.add("Python");
list.addLast("JavaScript");
System.out.println(list);  // Output: [Python, JavaScript]
```

📌 5. `get(int index)` → Retrieve element at index

Gets the **element present at the given index**.

```
LinkedList<String> list = new LinkedList<>();
list.add("Java");
list.add("Python");
System.out.println(list.get(1));  // Output: Python
```

❌ **Slower (`O(n)`) than `ArrayList` because it has to traverse the list.**

📌 6. `getFirst()` → Get first element

```
LinkedList<String> list = new LinkedList<>();
list.add("Java");
list.add("Python");
System.out.println(list.getFirst());  // Output: Java
```

📌 7. `getLast()` → Get last element

```
LinkedList<String> list = new LinkedList<>();
list.add("Java");
list.add("Python");
System.out.println(list.getLast());  // Output: Python
```

📌 8. `remove(int index)` → Remove element by index

Removes the **element at the specified index**.

```
LinkedList<String> list = new LinkedList<>();
list.add("Java");
list.add("Python");
list.add("C++");
list.remove(1);  // Remove "Python" (index 1)
System.out.println(list);  // Output: [Java, C++]
```

📌 9. `remove(Object obj)` → Remove element by value

```
LinkedList<String> list = new LinkedList<>();
list.add("Java");
list.add("Python");
list.remove("Python");  // Remove "Python"
System.out.println(list);  // Output: [Java]
```

📌 10. `removeFirst()` → Remove first element

```
LinkedList<String> list = new LinkedList<>();
list.add("Java");
list.add("Python");
list.removeFirst();
System.out.println(list);  // Output: [Python]
```

📌 11. `removeLast()` → Remove last element

```
LinkedList<String> list = new LinkedList<>();
list.add("Java");
list.add("Python");
list.removeLast();
System.out.println(list);  // Output: [Java]
```

📌 12. `size()` → Get the number of elements

```
LinkedList<String> list = new LinkedList<>();
list.add("Java");
list.add("Python");
System.out.println(list.size());  // Output: 2
```

📌 13. `clear()` → Remove all elements

```
LinkedList<String> list = new LinkedList<>();
list.add("Java");
list.add("Python");
list.clear();
System.out.println(list);  // Output: []
```

📌 When to Use `LinkedList<T>`?

✔ **Best for fast insertions and deletions (`O(1)`).**
✔ **Use when frequently adding/removing elements from the start or middle.**
✘ **Avoid if you need fast random access (`O(n)`).**

📌 Summary

Method	Description
`add(E e)`	Adds an element at the end
`addFirst(E e)`	Adds an element at the beginning
`get(int index)`	Retrieves element at an index

`remove(int index)`	Removes element at index
`removeFirst()`	Removes the first element
`removeLast()`	Removes the last element
`clear()`	Removes all elements

📌 Deep Dive: How `LinkedList<T>` Works Internally in Java

🚀 In this section, we will **break down the internal working of `LinkedList<T>`** step by step.
I will explain **how elements are stored, inserted, deleted, and accessed internally** using a **doubly linked list structure**.

1️⃣ What is a Linked List?

A **Linked List** is a **linear data structure** that consists of a **sequence of nodes** where:

1. **Each node stores two things:**
   - **Data** (the actual element)
   - **References (pointers) to the next and previous nodes**
2. Unlike an **array**, which stores elements **contiguously in memory**, a **linked list stores elements in separate memory locations**, connected using pointers.

📍 Structure of a `LinkedList<T>` Node:

Each node contains **three parts**:

```
[Prev | Data | Next]
```

- **Prev** → Points to the **previous** node
- **Data** → Stores the **actual value**
- **Next** → Points to the **next** node

📌 Example of a `LinkedList` with three elements:

```
Head -> [null | A | ⬇] <--> [⬆ | B | ⬇] <--> [⬆ | C | null] <- Tail
```

- **Head** points to the **first node (A)**
- **Tail** points to the **last node (C)**
- Each node is **connected both ways** (Doubly Linked List)

2️⃣ How Java's `LinkedList<T>` Works Internally?

📌 **Java's `LinkedList<T>` is implemented as a Doubly Linked List (`DLL`).**

- The `LinkedList` class has **two important instance variables**:
  - `Node first` → **Points to the first node** (head)
  - `Node last` → **Points to the last node** (tail)

📌 **Internal Node Class (`LinkedList.Node<T>`)**

```
private static class Node<T> {
    T item;       // The actual data stored
    Node<T> next; // Pointer to the next node
    Node<T> prev; // Pointer to the previous node

    Node(Node<T> prev, T item, Node<T> next) {
        this.item = item;
        this.next = next;
        this.prev = prev;
    }
}
```

☑ **Each node stores:**

- **Data (`item`)**
- **Next node reference (`next`)**
- **Previous node reference (`prev`)**

3️⃣ How `add(E element)` Works Internally?

📌 **Adding elements to a `LinkedList` (appending to the end)**

```
LinkedList<String> list = new LinkedList<>();
list.add("A");
list.add("B");
list.add("C");
```

Step-by-Step Execution:

1. **First Element "A" is added** → A new node is created

   ```
   [null | A | null]
   ```

   - **Head and Tail both point to A**

2. **Second Element "B" is added**

   ```
   [null | A | ⬇] <--> [⬆ | B | null]
   ```

   - **A's `next` pointer points to B**
   - **B's `prev` pointer points to A**

- **Tail now points to B**

3. **Third Element "C" is added**

```
[null | A | ⬇] <--> [⬆ | B | ⬇] <--> [⬆ | C | null]
```

- **B's `next` pointer points to C**
- **C's `prev` pointer points to B**
- **Tail now points to C**

☑ **Insertion is O(1) because we only update pointers.**

4️ How `remove(E element)` Works Internally?

📌 **Removing an element from `LinkedList`**

```
list.remove("B");  // Remove "B"
```

Step-by-Step Execution:

Before removing:

```
[null | A | ⬇] <--> [⬆ | B | ⬇] <--> [⬆ | C | null]
```

4. **Find "B"** → Traverse nodes until we reach "B"
5. **Update Pointers**
   - **A's `next` now points to C**
   - **C's `prev` now points to A**
6. **Remove "B"**

After removing "B":

```
[null | A | ⬇] <--> [⬆ | C | null]
```

☑ **Removal is O(1) if we already have a reference, otherwise O(n) if we search first.**

5️ How `get(int index)` Works Internally?

📌 **Retrieving an element by index (`O(n)`)**

```
String value = list.get(2);  // Fetch element at index 2
```

1. **If `index < size/2`**, start from **head** and move forward
2. **If `index > size/2`**, start from **tail** and move backward

Example for `list.get(2)`:

```
[null | A | ⬇] <--> [⬆ | B | ⬇] <--> [⬆ | C | null]
```

1. **Start from `head` and move `next` twice** → Reached C
2. **Return C**

☑ **Slower than `ArrayList` because there is no direct index access (O(n) complexity).**

---

6 How `addFirst()` and `addLast()` Work?

📌 **`addFirst(E e)` → Adds element at the start**

```
list.addFirst("X");
```

Before:

```
[null | A | ⬇] <--> [⬆ | B | ⬇] <--> [⬆ | C | null]
```

After adding `"X"` at the start:

```
[null | X | ⬇] <--> [⬆ | A | ⬇] <--> [⬆ | B | ⬇] <--> [⬆ | C | null]
```

☑ **O(1) complexity since only head pointer changes.**

📌 **`addLast(E e)` → Adds element at the end**

```
list.addLast("Y");

[null | X | ⬇] <--> [⬆ | A | ⬇] <--> [⬆ | B | ⬇] <--> [⬆ | C | ⬇]
<--> [⬆ | Y | null]
```

☑ **O(1) complexity since only tail pointer changes.**

---

7 Time Complexity Comparison

Operation	LinkedList	ArrayList
Insert at End	O(1)	O(1)
Insert at Beginning	O(1)	O(n)
Insert in Middle	O(1) (if reference is known)	O(n)
Delete at Beginning	O(1)	O(n)
Delete at End	O(1)	O(1)
Delete in Middle	O(1) (if reference is known)	O(n)
Access (get)	O(n)	O(1)

- **LinkedList is better for frequent insertions and deletions.**
- **ArrayList is better for fast access (`get(index)`).**

📌 Conclusion

☑ **LinkedList<T> works internally as a doubly linked list.**
☑ **Each node stores data, a pointer to the next node, and a pointer to the previous node.**
☑ **Insertions and deletions are O(1) if the reference is known.**
☑ **Retrieving elements by index is O(n) (slower than `ArrayList`).**

📌 Deep Dive into `Vector<T>` in Java

1️⃣ What is a `Vector<T>`?

🔹 `Vector<T>` is a **dynamic array** in Java, just like `ArrayList<T>`, but with **one key difference**: **Vector is thread-safe** (i.e., multiple threads can access it safely).

🔹 `Vector<T>` is a part of **Java's legacy collection framework**, but it is still used when we need **a synchronized (thread-safe) list**.

🔹 **Package:**

```
import java.util.Vector;
```

2️⃣ Why Use `Vector<T>`?

💡 **Why do we need Vector when we have ArrayList?**

☑ **Thread Safety**: `Vector<T>` is synchronized, so it can be used safely in multi-threaded environments.

☑ **Dynamic Resizing**: Unlike an array, `Vector` grows dynamically when more elements are added.

☑ **Fast Random Access**: Since it is an **array-based** structure, `Vector` allows fast access via an index.

🚫 **But...**

- **`Vector<T>` is slower than `ArrayList<T>`** because **each method in Vector is synchronized**, making it thread-safe but slower.
- **If you don't need thread safety, use `ArrayList<T>` instead.**

3️⃣ Internal Working of `Vector<T>`

📌 **How `Vector<T>` stores elements internally?**

- `Vector<T>` is implemented **internally as a resizable array**.
- It has an **initial capacity** (default = 10), and when it is full, **it grows by doubling its size**.

Internal Structure of `Vector<T>` (Before Resizing)

```
[ A ] [ B ] [ C ] [ D ] [ E ] [ - ] [ - ] [ - ] [ - ] [ - ]
(size = 5, capacity = 10)
```

After Adding More Elements (Resizing Happens)

```
[ A ] [ B ] [ C ] [ D ] [ E ] [ F ] [ G ] [ H ] [ I ] [ J ]
(size = 10, capacity = 10)
```

☑ **If one more element is added, Vector resizes (doubles capacity from 10 → 20):**

```
[ A ] [ B ] [ C ] [ D ] [ E ] [ F ] [ G ] [ H ] [ I ] [ J ] [ K ] [ - ] [ - ] [ - ] [
- ] [ - ] ...
(size = 11, capacity = 20)
```

📌 **Key Point:** Vector **doubles its capacity when it exceeds the limit**, while `ArrayList` grows by **50% of its size**.

4️⃣ How to Create a `Vector<T>`?

📍 Default Constructor (Capacity = 10)

```
Vector<Integer> vector = new Vector<>();
```

📌 This creates a `Vector` with **default capacity = 10**.

📍 Specifying Initial Capacity

```
Vector<Integer> vector = new Vector<>(20);
```

📌 This creates a `Vector` with **initial capacity = 20**.

📍 Specifying Capacity Increment

```
Vector<Integer> vector = new Vector<>(10, 5);
```

📌 **Initial capacity = 10**
📌 **Increases by 5 when full** (instead of doubling).

5️⃣ Important Methods in `Vector<T>` (with Examples)

📍 1. `add(E e)` - Add an element at the end

```
Vector<String> vector = new Vector<>();
vector.add("A");
vector.add("B");
```

```
vector.add("C");
System.out.println(vector);  // Output: [A, B, C]
```

---

2. `add(int index, E element)` - Insert at a specific position

```
vector.add(1, "X");
System.out.println(vector);  // Output: [A, X, B, C]
```

---

3. `get(int index)` - Retrieve element at index

```
String element = vector.get(2);
System.out.println(element);  // Output: B
```

---

4. `remove(int index)` - Remove element at index

```
vector.remove(1);
System.out.println(vector);  // Output: [A, B, C]
```

---

5. `remove(Object obj)` - Remove specific object

```
vector.remove("B");
System.out.println(vector);  // Output: [A, C]
```

---

6. `set(int index, E element)` - Replace an element

```
vector.set(1, "Z");
System.out.println(vector);  // Output: [A, Z, C]
```

---

7. `size()` - Get the number of elements

```
System.out.println(vector.size());  // Output: 3
```

---

8. `capacity()` - Get current capacity

```
System.out.println(vector.capacity());  // Output: 10 (default)
```

---

9. `isEmpty()` - Check if Vector is empty

```
System.out.println(vector.isEmpty());  // Output: false
```

📍 10. `contains(E e)` - Check if element exists

```
System.out.println(vector.contains("Z"));  // Output: true
```

📍 11. `clear()` - Remove all elements

```
vector.clear();
System.out.println(vector);  // Output: []
```

📍 12. `forEach()` - Iterate using forEach loop

```
vector.add("A");
vector.add("B");
vector.add("C");

vector.forEach(e -> System.out.println(e));
// Output:
// A
// B
// C
```

6️⃣ Performance Comparison (`Vector<T>` vs `ArrayList<T>`)

Operation	`Vector<T>`	`ArrayList<T>`
Thread Safety	☑ Yes (Synchronized)	❌ No
Performance	🚫 Slower	☑ Faster
Resizing Policy	Doubles capacity	Increases by 50%
Random Access (`get(int)`)	☑ O(1)	☑ O(1)
Insert/Delete in Middle	🚫 O(n)	🚫 O(n)
Insert/Delete at End	☑ O(1)	☑ O(1)

🚀 **Use `Vector<T>` only when thread safety is required.**
🚀 **For better performance in single-threaded applications, use `ArrayList<T>`.**

📌 Conclusion

☑ `Vector<T>` is a **resizable array** that is **synchronized (thread-safe)**.
☑ It has **slower performance** than `ArrayList<T>` due to **synchronization overhead**.
☑ **Use `Vector<T>` when multiple threads modify the list simultaneously**.
☑ **Prefer `ArrayList<T>` for better performance in a single-threaded environment**.

📌 Deep Dive into `Stack<T>` in Java

1️⃣ What is a `Stack<T>`?

- `Stack<T>` is a **Last In, First Out (LIFO)** data structure in Java.
- It is a **special type of** `Vector<T>` that **allows only specific operations**.
- The **last element added is the first to be removed** (just like a stack of plates 🍽️).
- **Stack is synchronized**, meaning it is **thread-safe**, but **slower than non-synchronized alternatives**.
- **Package:**

```
import java.util.Stack;
```

📌 **Key Concept: LIFO (Last In, First Out)**

```
Push -> [ A ] [ B ] [ C ] (C is the last added)
Pop -> [ A ] [ B ] (C is removed first)
```

2️⃣ Why Use `Stack<T>`?

💡 **When should you use a Stack?**

☑ **When you need LIFO order** (Last In, First Out).
☑ **Undo/Redo operations** (e.g., in text editors).
☑ **Browser back/forward history**.
☑ **Expression evaluation** (e.g., parsing arithmetic expressions).
☑ **Recursion tracking** (call stack in programming).

🚫 **But...**

- **Stack<T> is slower than alternatives like** `Deque<T>` because it is synchronized.
- **For better performance, use** `ArrayDeque<T>` **instead of** `Stack<T>`.

3️⃣ Internal Working of `Stack<T>`

📌 **How** `Stack<T>` **stores elements internally?**

- `Stack<T>` extends `Vector<T>` → **It is a dynamic array** that resizes itself when full.
- It provides **extra methods** like `push()`, `pop()`, `peek()`, etc., for stack operations.
- Stack **inherits all properties of** `Vector<T>`, including thread safety.

Internal Structure of `Stack<T>`

```
Bottom → [ A ] [ B ] [ C ] ← Top
```

- **Push(D) →** `[ A ] [ B ] [ C ] [ D ]`
- **Pop() →** `[ A ] [ B ] [ C ]` (removes `D`)

4️⃣ How to Create a `Stack<T>`?

📌 Creating a Stack

```
Stack<Integer> stack = new Stack<>();
```

📌 This creates an **empty Stack**.

5️⃣ Important Methods in `Stack<T>` (with Examples)

📌 1. `push(E e)` - Add an element to the top

```
stack.push(10);
stack.push(20);
stack.push(30);
System.out.println(stack);  // Output: [10, 20, 30]
```

📌 2. `pop()` - Remove and return the top element

```
int topElement = stack.pop();
System.out.println(topElement);  // Output: 30
System.out.println(stack);  // Output: [10, 20]
```

📌 3. `peek()` - Get the top element without removing it

```
int topElement = stack.peek();
System.out.println(topElement);  // Output: 20
System.out.println(stack);  // Output: [10, 20] (unchanged)
```

📌 4. `isEmpty()` - Check if stack is empty

```
System.out.println(stack.isEmpty());  // Output: false
```

📌 5. `search(E e)` - Find an element's position from the top

```
int position = stack.search(10);
System.out.println(position);  // Output: 2 (position from top, 1-based index)
```

📌 **Returns -1 if element is not found.**

📍 6. `size()` - Get number of elements in the stack

```
System.out.println(stack.size());  // Output: 2
```

📍 7. `contains(E e)` - Check if an element exists

```
System.out.println(stack.contains(20));  // Output: true
System.out.println(stack.contains(40));  // Output: false
```

📍 8. `clear()` - Remove all elements

```
stack.clear();
System.out.println(stack);  // Output: []
```

6️⃣ Performance of `Stack<T>`

Operation	Time Complexity
`push(E e)`	O(1)
`pop()`	O(1)
`peek()`	O(1)
`search(E e)`	O(n)
`isEmpty()`	O(1)

📌 **Stack operations are generally fast (O(1))**, but searching takes **O(n)** time.

7️⃣ `Stack<T>` vs `ArrayDeque<T>` (Which is better?)

Feature	`Stack<T>`	`ArrayDeque<T>`
Thread Safe?	☑ Yes	❌ No
Performance	🚫 Slower (synchronized)	☑ Faster (unsynchronized)
LIFO Support	☑ Yes	☑ Yes
Used for?	Legacy Code, Thread Safety	Better Performance

🚀 **Use `ArrayDeque<T>` instead of `Stack<T>` for better performance!**

📌 Conclusion

☑ `Stack<T>` is a **LIFO (Last In, First Out) data structure** in Java.
☑ It is **thread-safe**, but **slower than alternatives like** `ArrayDeque<T>`.
☑ It is used in **undo/redo, recursion, expression evaluation, etc.**
☑ **Use** `Stack<T>` **when you need thread safety, but prefer** `ArrayDeque<T>` **for better performance.**

📌 Deep Dive into `CopyOnWriteArrayList<T>` in Java

1️⃣ What is `CopyOnWriteArrayList<T>`?

🔹 `CopyOnWriteArrayList<T>` is a **thread-safe** version of `ArrayList<T>`.
🔹 It belongs to the `java.util.concurrent` package.
🔹 It **allows multiple threads to read the list concurrently without locking**.
🔹 But, **modifications (add, remove, set) create a new copy of the array**, making it different from `ArrayList<T>`.
🔹 Best suited for **scenarios where reads are more frequent than writes**.

2️⃣ Why Use `CopyOnWriteArrayList<T>`?

💡 **When should you use** `CopyOnWriteArrayList<T>`**?**

☑ **If multiple threads need to read the list simultaneously**.
☑ **If reads happen more often than writes** (because writes are costly).
☑ **If you want to avoid** `ConcurrentModificationException` during iteration.
☑ **Best for caching, notifications, and event handling systems**.

🚫 **But...**

- **Every modification (add, remove) creates a new copy of the list**, which makes it **memory-heavy**.
- **Slower for frequent modifications** compared to `ArrayList<T>` and `LinkedList<T>`.

3️⃣ Internal Working of `CopyOnWriteArrayList<T>`

📌 **How does it work?**

- `CopyOnWriteArrayList<T>` uses **an internal array (`Object[] array`) to store elements**.
- Every time a modification occurs (add, remove, set), **a new copy of the entire array is created**.
- This ensures that **read operations are never blocked**, but **modifications are expensive**.

Internal Structure

```
Original List: [ A ] [ B ] [ C ]
Modification (add D) → New List: [ A ] [ B ] [ C ] [ D ]
```

- **Reads use the old array** until the modification is complete.
- **After modification, the reference is updated to the new array.**

4️⃣ How to Create a `CopyOnWriteArrayList<T>`?

📌 **Import the package:**

```
import java.util.concurrent.CopyOnWriteArrayList;
```

📍 Creating a CopyOnWriteArrayList

```
CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>();
```

📌 This creates an **empty thread-safe list**.

5️⃣ Important Methods in `CopyOnWriteArrayList<T>` (with Examples)

📍 1. `add(E e)` - Add an element to the list

```
list.add(10);
list.add(20);
list.add(30);
System.out.println(list);  // Output: [10, 20, 30]
```

📍 2. `remove(int index)` - Remove element at a specific index

```
list.remove(1);
System.out.println(list);  // Output: [10, 30]
```

📍 3. `get(int index)` - Get element at a specific index

```
int element = list.get(0);
System.out.println(element);  // Output: 10
```

📍 4. `size()` - Get number of elements in the list

```
System.out.println(list.size());  // Output: 2
```

📌 5. `contains(E e)` - Check if an element exists

```
System.out.println(list.contains(30));  // Output: true
System.out.println(list.contains(50));  // Output: false
```

📌 6. `set(int index, E e)` - Update an element at a specific index

```
list.set(1, 40);
System.out.println(list);  // Output: [10, 40]
```

📌 7. `iterator()` - Get an iterator (safe from `ConcurrentModificationException`)

```
for (Integer num : list) {
    System.out.println(num);
}
```

📌 Unlike `ArrayList<T>`, this **will NOT throw `ConcurrentModificationException`** even if another thread modifies the list while iterating.

📌 8. `clear()` - Remove all elements

```
list.clear();
System.out.println(list);  // Output: []
```

6️⃣ Performance of `CopyOnWriteArrayList<T>`

Operation	Time Complexity
`add(E e)`	O(n) (creates a new array)
`remove(int index)`	O(n) (creates a new array)
`get(int index)`	O(1)
`contains(E e)`	O(n)
`set(int index, E e)`	O(n) (creates a new array)
`iterator()`	O(n) (creates a snapshot)

📌 **Read operations (`get()`) are fast (O(1)), but modifications (`add()`, `set()`, `remove()`) are slow (O(n)).**

7️⃣ `CopyOnWriteArrayList<T>` VS `ArrayList<T>` VS `Vector<T>`

Feature	`CopyOnWriteArrayList<T>`	`ArrayList<T>`	`Vector<T>`

Thread Safe?	☑ Yes	❌ No	☑ Yes
Performance (Read)	☑ Fast	☑ Fast	❌ Slow
Performance (Write)	🚫 Slow	☑ Fast	🚫 Slow
Best Use Case	Many reads, few writes	General purpose	Legacy multi-threading

🚀 **Use `CopyOnWriteArrayList<T>` when multiple threads need fast reads, but few writes.**

📌 Conclusion

☑ `CopyOnWriteArrayList<T>` is a **thread-safe alternative to `ArrayList<T>`**.
☑ It **allows multiple threads to read safely without locking**.
☑ **Every modification creates a new copy of the list, making writes expensive**.
☑ **Best suited for scenarios where reads are more frequent than writes**.
☑ **Avoid using it when frequent modifications are needed (use `ArrayList<T>` or `ConcurrentLinkedQueue<T>` instead).**

📌 Chapter 4: Set Interface (Unique Elements Collection) in Java

1️⃣ What is a `Set<T>`?

📌 **Definition:**
A `Set<T>` is a collection that **stores unique elements** and does **not allow duplicates**.

📌 **Key Features of `Set<T>`:**
☑ **No duplicate elements allowed** (Each element is unique)
☑ **Can be unordered or ordered** (depends on the implementation)
☑ **Efficient for search and lookup operations**
☑ **Useful for mathematical set operations (union, intersection, etc.)**

2️⃣ Why Use `Set<T>` Instead of `List<T>`?

Feature	`List<T>`	`Set<T>`
Allows Duplicates?	☑ Yes	❌ No
Maintains Order?	☑ Yes	❌ (Depends on implementation)
Search Performance	❌ Slower (O(n) for `ArrayList`, O(log n) for `LinkedList`)	☑ Faster (O(1) in `HashSet`)

Best Use Case	**If duplicates are allowed & order matters**	**If you need only unique elements**

📌 **Use `Set<T>` when you need to store unique elements and don't care about order.**

3️⃣ Implementations of `Set<T>` in Java

📌 There are **six main implementations** of `Set<T>` in Java:

1️⃣ `HashSet<T>`

- ☑ **Unordered**
- ☑ Uses **hashing** for fast search operations
- ☑ Best for **fast access and uniqueness**

2️⃣ `LinkedHashSet<T>`

- ☑ **Maintains insertion order**
- ☑ Uses a **linked list + hash table**
- ☑ Best for **unique elements while maintaining order**

3️⃣ `TreeSet<T>`

- ☑ **Sorted set (Natural ordering)**
- ☑ Uses a **Red-Black Tree (Self-balancing BST)**
- ☑ Best for **keeping elements sorted**

4️⃣ `EnumSet<T>`

- ☑ **Specialized set for Enums**
- ☑ **Very fast and memory-efficient**

5️⃣ `ConcurrentSkipListSet<T>`

- ☑ **Thread-safe sorted set**
- ☑ Uses a **Skip List for ordering**
- ☑ Best for **multi-threaded applications**

6️⃣ `CopyOnWriteArraySet<T>`

- ☑ **Thread-safe Set**
- ☑ **Good for concurrent read-heavy operations**

4️⃣ Internal Working of `Set<T>` Implementations

📌 **How does `HashSet<T>` store elements?**

- Uses **a Hash Table (based on HashMap)**.

- Uses **hashing** to store elements efficiently.
- **Search, Insert, Delete → O(1) time complexity** (best case).

📌 **How does `LinkedHashSet<T>` work?**

- Same as `HashSet<T>`, but **maintains insertion order** using a **doubly linked list**.

📌 **How does `TreeSet<T>` work?**

- Uses a **self-balancing Red-Black Tree**.
- Always **keeps elements sorted**.
- **Insert, Delete, Search → O(log n) time complexity**.

📌 **How does `ConcurrentSkipListSet<T>` work?**

- Uses a **Skip List (multiple linked lists at different levels for fast search)**.
- **Thread-safe and sorted**.
- **Slower than `TreeSet` in single-threaded cases but better in multi-threading**.

5️⃣ Set Operations (Mathematical Operations on Sets)

📌 **Java provides useful methods to perform operations like:**

📍 1. Union (A ∪ B) → Combine elements from both sets

```
Set<Integer> set1 = new HashSet<>(Arrays.asList(1, 2, 3));
Set<Integer> set2 = new HashSet<>(Arrays.asList(3, 4, 5));

set1.addAll(set2);
System.out.println(set1);  // Output: [1, 2, 3, 4, 5]
```

📍 2. Intersection (A ∩ B) → Common elements

```
Set<Integer> set1 = new HashSet<>(Arrays.asList(1, 2, 3));
Set<Integer> set2 = new HashSet<>(Arrays.asList(3, 4, 5));

set1.retainAll(set2);
System.out.println(set1);  // Output: [3]
```

📍 3. Difference (A - B) → Elements in A but not in B

```
Set<Integer> set1 = new HashSet<>(Arrays.asList(1, 2, 3));
Set<Integer> set2 = new HashSet<>(Arrays.asList(3, 4, 5));

set1.removeAll(set2);
System.out.println(set1);  // Output: [1, 2]
```

📍 4. Subset Check (A ⊆ B) → Check if A is a subset of B

```
Set<Integer> set1 = new HashSet<>(Arrays.asList(1, 2));
Set<Integer> set2 = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5));

System.out.println(set2.containsAll(set1));  // Output: true
```

6️⃣ Choosing the Right `Set<T>` Implementation

Feature	`HashSet<T>`	`LinkedHashSet<T>`	`TreeSet<T>`	`EnumSet<T>`	`ConcurrentSkipListSet<T>`	`CopyOnWriteArraySet<T>`
Duplicate Elements	❌ No	❌ No	❌ No	❌ No	❌ No	❌ No
Maintains Order?	❌ No	☑ Yes (Insertion Order)	☑ Yes (Sorted)	❌ No	☑ Yes (Sorted)	❌ No
Thread-Safe?	❌ No	❌ No	❌ No	❌ No	☑ Yes	☑ Yes
Performance	☑ Fast	☑ Fast (Slightly Slower than `HashSet`)	🚫 Slower	☑ Fastest for Enums	🚫 Slower	🚫 Slowest (Creates Copy on Modification)
Best Use Case	**General Purpose (Fast)**	**Fast with Order**	**Sorted Data**	**Efficient Enum Handling**	**Thread-Safe Sorted Set**	**Thread-Safe Reads**

📌 Conclusion

☑ **Use `HashSet<T>` if order doesn't matter & you want the fastest performance.**
☑ **Use `LinkedHashSet<T>` if you need insertion order.**
☑ **Use `TreeSet<T>` if you need sorting.**
☑ **Use `EnumSet<T>` if you work with Enums.**
☑ **Use `ConcurrentSkipListSet<T>` if you need thread-safe sorting.**
☑ **Use `CopyOnWriteArraySet<T>` for thread-safe read-heavy scenarios.**

📌 All Methods of Set Interface in Java (Deep Dive)

1️⃣ Overview of Set<T> Methods

The `Set<T>` interface in Java is part of the Java Collection Framework and extends the `Collection<T>` interface. Since it represents a unique collection of elements, it provides various methods to **add, remove, check, and manipulate elements efficiently**.

These methods are **inherited from the Collection Interface** and implemented in classes like `HashSet<T>`, `LinkedHashSet<T>`, `TreeSet<T>`, etc.

2️⃣ Methods of the Set Interface

Here's a deep dive into **all the important methods** of `Set<T>`, with examples:

📍 1. `add(E e)` → Adds an element

☑ **Adds an element to the set if it is not already present.**
❌ **If the element already exists, it is ignored.**

```
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Apple"); // Duplicate, won't be added

System.out.println(set); // Output: [Apple, Banana]
```

📍 2. `addAll(Collection<? extends E> c)` → Adds all elements from another collection

☑ **Adds all elements from another collection to the current set.**
❌ **Duplicate elements are ignored.**

```
Set<Integer> set1 = new HashSet<>();
set1.add(1);
set1.add(2);

Set<Integer> set2 = new HashSet<>();
set2.add(2);
set2.add(3);

set1.addAll(set2);
System.out.println(set1); // Output: [1, 2, 3]
```

📍 3. `remove(Object o)` → Removes an element

☑ **Removes a specific element from the set.**
❌ **If the element is not found, nothing happens.**

```
Set<String> set = new HashSet<>();
set.add("Car");
set.add("Bike");

set.remove("Bike");
System.out.println(set); // Output: [Car]
```

📌 4. `removeAll(Collection<?> c)` → Removes all elements in a given collection

☑ **Removes all elements from the set that exist in another collection.**

```
Set<Integer> set = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5));
Set<Integer> removeSet = new HashSet<>(Arrays.asList(2, 4));

set.removeAll(removeSet);
System.out.println(set); // Output: [1, 3, 5]
```

📌 5. `clear()` → Removes all elements from the set

☑ **Empties the set completely.**

```
Set<String> set = new HashSet<>(Arrays.asList("A", "B", "C"));
set.clear();

System.out.println(set); // Output: []
```

📌 6. `contains(Object o)` → Checks if an element exists

☑ **Returns `true` if the element is in the set, otherwise `false`.**

```
Set<String> set = new HashSet<>(Arrays.asList("Apple", "Orange"));
System.out.println(set.contains("Apple"));  // Output: true
System.out.println(set.contains("Banana")); // Output: false
```

📌 7. `containsAll(Collection<?> c)` → Checks if all elements in a collection exist

☑ **Returns `true` if the set contains all elements in the given collection.**

```
Set<Integer> set = new HashSet<>(Arrays.asList(1, 2, 3, 4));
Set<Integer> subSet = new HashSet<>(Arrays.asList(2, 3));

System.out.println(set.containsAll(subSet));  // Output: true
```

📌 8. `size()` → Returns the number of elements in the set

☑ **Returns the total number of elements.**

```
Set<String> set = new HashSet<>(Arrays.asList("A", "B", "C"));
System.out.println(set.size());  // Output: 3
```

📌 9. `isEmpty()` → Checks if the set is empty

☑ **Returns `true` if the set contains no elements.**

```
Set<Integer> set = new HashSet<>();
System.out.println(set.isEmpty());  // Output: true
```

📌 10. `retainAll(Collection<?> c)` → Keeps only common elements (Intersection)

☑ **Removes elements that are NOT in the given collection (performs intersection).**

```
Set<Integer> set = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5));
Set<Integer> commonSet = new HashSet<>(Arrays.asList(2, 4));

set.retainAll(commonSet);
System.out.println(set); // Output: [2, 4]
```

📌 11. `iterator()` → Returns an iterator to traverse the set

☑ **Useful for looping through the set elements.**

```
Set<String> set = new HashSet<>(Arrays.asList("Red", "Blue", "Green"));
Iterator<String> itr = set.iterator();

while (itr.hasNext()) {
    System.out.println(itr.next());
}
```

📌 12. `toArray()` → Converts the set into an array

☑ **Returns an array containing all elements in the set.**

```
Set<String> set = new HashSet<>(Arrays.asList("X", "Y", "Z"));
Object[] arr = set.toArray();

System.out.println(Arrays.toString(arr)); // Output: [X, Y, Z]
```

📌 13. `toArray(T[] a)` → Converts the set into a typed array

☑ **Returns an array of type `T` containing all elements.**

```
Set<String> set = new HashSet<>(Arrays.asList("Java", "Python"));
String[] arr = set.toArray(new String[0]);
```

```
System.out.println(Arrays.toString(arr)); // Output: [Java, Python]
```

📌 14. `equals(Object o)` → Checks if two sets are equal

☑ **Returns `true` if the sets contain the same elements (order doesn't matter).**

```
Set<Integer> set1 = new HashSet<>(Arrays.asList(1, 2, 3));
Set<Integer> set2 = new HashSet<>(Arrays.asList(3, 2, 1));

System.out.println(set1.equals(set2)); // Output: true
```

📌 15. `hashCode()` → Returns the hash code of the set

☑ **Used for hashing-based storage.**

```
Set<Integer> set = new HashSet<>(Arrays.asList(10, 20, 30));
System.out.println(set.hashCode()); // Example Output: 32015
```

3️⃣ Summary of All Methods

Method	Description
`add(E e)`	Adds an element if it's not already present.
`addAll(Collection<?> c)`	Adds all elements from another collection.
`remove(Object o)`	Removes the specified element.
`removeAll(Collection<?> c)`	Removes all elements present in another collection.
`clear()`	Removes all elements from the set.
`contains(Object o)`	Checks if an element exists in the set.
`containsAll(Collection<?> c)`	Checks if all elements of a collection exist.
`size()`	Returns the number of elements.
`isEmpty()`	Checks if the set is empty.
`retainAll(Collection<?> c)`	Keeps only common elements (Intersection).
`iterator()`	Returns an iterator to traverse elements.
`toArray()`	Converts the set into an object array.
`toArray(T[] a)`	Converts the set into a typed array.
`equals(Object o)`	Checks if two sets are equal.

`hashCode()`	Returns the hash code of the set.

📌 Deep Dive into `HashSet<T>` in Java (Easy Explanation)

1️⃣ What is `HashSet<T>`?

A `HashSet<T>` in Java is a class that implements the `Set<T>` interface and **stores unique elements in an unordered way** using **hashing** for fast retrieval.

- ✔ **Stores only unique elements (No duplicates allowed).**
- ✔ **Uses `HashMap` internally for storage.**
- ✔ **Allows `null` values (only one).**
- ✔ **Unordered (No guarantee of insertion order).**
- ✔ **Fast operations (O(1) time complexity for add, remove, contains).**

2️⃣ How `HashSet` Works Internally?

🔹 Step 1: Uses `HashMap` for storage

Internally, `HashSet<T>` uses a `HashMap<T, Object>` where:

- Each element is stored as **a key** in the `HashMap`.
- A dummy value (like `PRESENT`) is used as a value.

```
private static final Object PRESENT = new Object();
private transient HashMap<E,Object> map;
```

🔹 Step 2: Hashing Process

1. When you **add** an element, it calculates the **hash code** of the element.
2. Finds a suitable **bucket (index)** in the Hash Table.
3. Stores the element **only if it does not already exist.**

🔹 Step 3: No Duplicate Elements

Since `HashMap` does not allow duplicate keys, `HashSet` ensures **no duplicate elements**.

3️⃣ Creating a `HashSet` (Basic Example)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();
```

```
        set.add("Apple");
        set.add("Banana");
        set.add("Mango");
        set.add("Apple"); // Duplicate, will not be added

        System.out.println(set); // Output: [Banana, Apple, Mango] (Unordered)
    }
}
```

- **Duplicates are ignored** (Only one "Apple" is stored).
- **Order is not maintained**.

---

4 Important Methods of `HashSet<T>`

Method	Description
`add(E e)`	Adds an element if it is not already present.
`remove(Object o)`	Removes the specified element from the set.
`contains(Object o)`	Returns `true` if the element exists.
`size()`	Returns the number of elements in the set.
`isEmpty()`	Checks if the set is empty.
`clear()`	Removes all elements from the set.
`iterator()`	Returns an iterator to traverse the set.

---

5 Detailed Examples of `HashSet<T>` Methods

1. `add(E e)` → Adds an element

```
HashSet<Integer> numbers = new HashSet<>();
numbers.add(10);
numbers.add(20);
numbers.add(10); // Duplicate, ignored

System.out.println(numbers); // Output: [20, 10] (Unordered)
```

---

2. `remove(Object o)` → Removes an element

```
HashSet<String> set = new HashSet<>(Arrays.asList("Java", "Python", "C++"));
set.remove("Python");

System.out.println(set); // Output: [Java, C++]
```

📌 3. `contains(Object o)` → Checks if an element exists

```
HashSet<Integer> numbers = new HashSet<>(Arrays.asList(1, 2, 3));
System.out.println(numbers.contains(2)); // Output: true
System.out.println(numbers.contains(5)); // Output: false
```

📌 4. `size()` → Returns the total number of elements

```
HashSet<String> set = new HashSet<>(Arrays.asList("Apple", "Banana"));
System.out.println(set.size()); // Output: 2
```

📌 5. `isEmpty()` → Checks if the set is empty

```
HashSet<Integer> set = new HashSet<>();
System.out.println(set.isEmpty()); // Output: true
```

📌 6. `clear()` → Removes all elements

```
HashSet<String> set = new HashSet<>(Arrays.asList("A", "B", "C"));
set.clear();

System.out.println(set); // Output: []
```

📌 7. `iterator()` → Traversing the set

```
HashSet<String> set = new HashSet<>(Arrays.asList("Red", "Blue", "Green"));
Iterator<String> itr = set.iterator();

while (itr.hasNext()) {
    System.out.println(itr.next());
}
```

6️⃣ Performance Analysis

Operation	Time Complexity
`add(E e)`	O(1)
`remove(Object o)`	O(1)
`contains(Object o)`	O(1)
`size()`	O(1)
`iterator()`	O(n)

7️⃣ When to Use `HashSet<T>`?

Use `HashSet` When...	Avoid `HashSet` When...
You need **fast lookups**.	You need **ordered elements** (Use `LinkedHashSet`).
You don't care about insertion order.	You need **sorted elements** (Use `TreeSet`).
You want to store unique elements.	You need **index-based access** (Use `ArrayList`).

8️⃣ Summary

- ✔ **HashSet is a Set implementation that uses hashing to store unique elements.**
- ✔ **Elements are stored in an unordered manner.**
- ✔ **Uses `HashMap` internally for storage.**
- ✔ **Fast operations: O(1) for adding, removing, and searching elements.**
- ✔ **Best choice when you need unique elements and fast access.**

📌 Deep Dive into `LinkedHashSet<T>` in Java (Easy Explanation)

1️⃣ What is `LinkedHashSet<T>`?

A `LinkedHashSet<T>` is a class in Java that implements the `Set<T>` interface and **maintains the insertion order** while ensuring **unique elements**.

- ✔ **Stores only unique elements (No duplicates allowed).**
- ✔ **Maintains insertion order (Unlike `HashSet`).**
- ✔ **Uses a combination of `HashSet` and `LinkedList`.**
- ✔ **Allows `null` values (only one).**
- ✔ **Faster than `TreeSet`, but slightly slower than `HashSet`.**

2️⃣ How `LinkedHashSet<T>` Works Internally?

🔹 Step 1: Uses `LinkedHashMap` for storage

Internally, `LinkedHashSet<T>` uses a **`LinkedHashMap<T, Object>`** where:

- Each element is stored as **a key** in the `LinkedHashMap`.
- A dummy value (like `PRESENT`) is used as a value.

```
private static final Object PRESENT = new Object();
private transient LinkedHashMap<E,Object> map;
```

🔹 Step 2: Maintains Insertion Order

- Unlike `HashSet`, `LinkedHashSet` **preserves the order in which elements are added**.

- This happens because `LinkedHashMap` maintains a **doubly linked list** of its entries.

---

3 Creating a `LinkedHashSet` (Basic Example)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        LinkedHashSet<String> set = new LinkedHashSet<>();

        set.add("Apple");
        set.add("Banana");
        set.add("Mango");
        set.add("Apple"); // Duplicate, will not be added

        System.out.println(set); // Output: [Apple, Banana, Mango] (Maintains order)
    }
}
```

✔ **Order is maintained as elements were inserted (`Apple → Banana → Mango`).**
✔ **Duplicates are ignored.**

---

4 Important Methods of `LinkedHashSet<T>`

Method	Description
`add(E e)`	Adds an element if it is not already present.
`remove(Object o)`	Removes the specified element from the set.
`contains(Object o)`	Returns `true` if the element exists.
`size()`	Returns the number of elements in the set.
`isEmpty()`	Checks if the set is empty.
`clear()`	Removes all elements from the set.
`iterator()`	Returns an iterator to traverse the set.

---

5 Detailed Examples of `LinkedHashSet<T>` Methods

📍 1. `add(E e)` → Adds an element

```
LinkedHashSet<Integer> numbers = new LinkedHashSet<>();
numbers.add(10);
numbers.add(20);
numbers.add(10); // Duplicate, ignored

System.out.println(numbers); // Output: [10, 20] (Maintains order)
```

📌 2. `remove(Object o)` → Removes an element

```
LinkedHashSet<String> set = new LinkedHashSet<>(Arrays.asList("Java", "Python",
"C++"));
set.remove("Python");

System.out.println(set); // Output: [Java, C++]
```

📌 3. `contains(Object o)` → Checks if an element exists

```
LinkedHashSet<Integer> numbers = new LinkedHashSet<>(Arrays.asList(1, 2, 3));
System.out.println(numbers.contains(2)); // Output: true
System.out.println(numbers.contains(5)); // Output: false
```

📌 4. `size()` → Returns the total number of elements

```
LinkedHashSet<String> set = new LinkedHashSet<>(Arrays.asList("Apple", "Banana"));
System.out.println(set.size()); // Output: 2
```

📌 5. `isEmpty()` → Checks if the set is empty

```
LinkedHashSet<Integer> set = new LinkedHashSet<>();
System.out.println(set.isEmpty()); // Output: true
```

📌 6. `clear()` → Removes all elements

```
LinkedHashSet<String> set = new LinkedHashSet<>(Arrays.asList("A", "B", "C"));
set.clear();

System.out.println(set); // Output: []
```

📌 7. `iterator()` → Traversing the set

```
LinkedHashSet<String> set = new LinkedHashSet<>(Arrays.asList("Red", "Blue",
"Green"));
Iterator<String> itr = set.iterator();

while (itr.hasNext()) {
    System.out.println(itr.next());
}
```

---

6 Performance Analysis

Operation	Time Complexity
`add(E e)`	O(1)
`remove(Object o)`	O(1)
`contains(Object o)`	O(1)
`size()`	O(1)
`iterator()`	O(n)

---

7 When to Use `LinkedHashSet<T>`?

Use `LinkedHashSet` When...	Avoid `LinkedHashSet` When...
You need **fast lookups** with **insertion order maintained**.	You need **sorted elements** (Use `TreeSet`).
You want a **unique collection that preserves order**.	You need **index-based access** (Use `ArrayList`).
You need moderate performance with predictable order.	You need **higher performance** (Use `HashSet`).

---

8 Summary

- ✔ **LinkedHashSet is a Set implementation that maintains insertion order.**
- ✔ **Uses `LinkedHashMap` internally to store unique elements.**
- ✔ **Allows fast lookups, insertions, and deletions (O(1) time complexity).**
- ✔ **Best choice when you need unique elements with predictable order.**

📌 Deep Dive into `TreeSet<T>` in Java (Easy Explanation)

1 What is `TreeSet<T>`?

A `TreeSet<T>` in Java is a class that implements the `NavigableSet<T>` interface and maintains **sorted unique elements**.

- ✔ **Stores only unique elements (No duplicates allowed).**
- ✔ **Maintains elements in sorted (ascending) order.**
- ✔ **Implements `NavigableSet<T>`, which extends `SortedSet<T>`.**
- ✔ **Uses a self-balancing Red-Black Tree for storage.**
- ✔ **Faster than `LinkedList`, but slower than `HashSet`.**

---

2 How `TreeSet<T>` Works Internally?

🔹 Step 1: Uses a Red-Black Tree

A **Red-Black Tree** is a type of **self-balancing binary search tree (BST)**.
Whenever a new element is added:

- It is first inserted in BST order.
- If the tree becomes unbalanced, rotations and color changes occur to maintain balance.
- The height of the tree is maintained as **O(log n)**, ensuring efficient operations.

🔹 Step 2: Maintains Sorted Order

`TreeSet<T>` sorts elements **automatically in natural order** (`Comparable`) or based on a custom comparator (`Comparator`).

🔹 Step 3: No Duplicates Allowed

Duplicate elements are ignored while maintaining order.

---

3 Creating a `TreeSet` (Basic Example)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();

        numbers.add(50);
        numbers.add(20);
        numbers.add(10);
        numbers.add(40);
        numbers.add(30);
        numbers.add(10); // Duplicate, ignored

        System.out.println(numbers); // Output: [10, 20, 30, 40, 50] (Sorted)
    }
}
```

✔ **Sorted Order (`10 → 20 → 30 → 40 → 50`)**
✔ **Duplicates are ignored.**

---

4 Important Methods of `TreeSet<T>`

Method	Description

`add(E e)`	Adds an element if it is not already present (sorted).
`remove(Object o)`	Removes the specified element from the set.
`contains(Object o)`	Returns `true` if the element exists.
`size()`	Returns the number of elements in the set.
`isEmpty()`	Checks if the set is empty.
`clear()`	Removes all elements from the set.
`iterator()`	Returns an iterator to traverse the set.
`first()`	Returns the smallest (first) element.
`last()`	Returns the largest (last) element.
`higher(E e)`	Returns the smallest element greater than `e`.
`lower(E e)`	Returns the largest element smaller than `e`.
`ceiling(E e)`	Returns the smallest element greater than or equal to `e`.
`floor(E e)`	Returns the largest element smaller than or equal to `e`.
`pollFirst()`	Removes and returns the first element.
`pollLast()`	Removes and returns the last element.

---

5 Detailed Examples of `TreeSet<T>` Methods

1. `add(E e)` → Adds an element

```
TreeSet<String> set = new TreeSet<>();
set.add("Banana");
set.add("Apple");
set.add("Mango");

System.out.println(set); // Output: [Apple, Banana, Mango] (Sorted order)
```

---

2. `remove(Object o)` → Removes an element

```
TreeSet<Integer> numbers = new TreeSet<>(Arrays.asList(10, 20, 30, 40));
numbers.remove(20);

System.out.println(numbers); // Output: [10, 30, 40]
```

📌 3. `contains(Object o)` → Checks if an element exists

```
TreeSet<Integer> numbers = new TreeSet<>(Arrays.asList(5, 10, 15));
System.out.println(numbers.contains(10)); // Output: true
System.out.println(numbers.contains(25)); // Output: false
```

📌 4. `first()` and `last()` → Get first and last elements

```
TreeSet<Integer> numbers = new TreeSet<>(Arrays.asList(100, 50, 75, 25));
System.out.println(numbers.first()); // Output: 25 (Smallest)
System.out.println(numbers.last());  // Output: 100 (Largest)
```

📌 5. `higher(E e)` and `lower(E e)` → Get next and previous elements

```
TreeSet<Integer> numbers = new TreeSet<>(Arrays.asList(10, 20, 30, 40));
System.out.println(numbers.higher(20)); // Output: 30 (Next higher)
System.out.println(numbers.lower(20));  // Output: 10 (Previous lower)
```

📌 6. `ceiling(E e)` and `floor(E e)` → Get equal or closest values

```
TreeSet<Integer> numbers = new TreeSet<>(Arrays.asList(10, 20, 30, 40));
System.out.println(numbers.ceiling(25)); // Output: 30 (Next greater or equal)
System.out.println(numbers.floor(25));   // Output: 20 (Next smaller or equal)
```

📌 7. `pollFirst()` and `pollLast()` → Remove first and last elements

```
TreeSet<Integer> numbers = new TreeSet<>(Arrays.asList(5, 10, 15, 20));
System.out.println(numbers.pollFirst()); // Output: 5 (Removes first)
System.out.println(numbers.pollLast());  // Output: 20 (Removes last)
```

📌 8. Custom Sorting with `Comparator`

```
TreeSet<String> set = new TreeSet<>(Comparator.reverseOrder());
set.add("Banana");
set.add("Apple");
set.add("Mango");

System.out.println(set); // Output: [Mango, Banana, Apple] (Reverse Order)
```

6️⃣ Performance Analysis

Operation	Time Complexity
`add(E e)`	O(log n)

`remove(Object o)`	O(log n)
`contains(Object o)`	O(log n)
`size()`	O(1)
`iterator()`	O(n)

7️⃣ When to Use `TreeSet<T>`?

Use `TreeSet` When...	Avoid `TreeSet` When...
You need elements to be **sorted automatically**.	You need **unordered but fast access** (Use `HashSet`).
You want **logarithmic time complexity** (`O(log n)`).	You need **constant time lookups** (`O(1)`, Use `HashSet`).
You need efficient **range queries** (`higher()`, `lower()`).	You need **insertion order to be maintained** (Use `LinkedHashSet`).

8️⃣ Summary

✔ **TreeSet is a Set implementation that maintains sorted order.**
✔ **Uses a self-balancing Red-Black Tree for internal storage.**
✔ **Offers `O(log n)` time complexity for insert, delete, and search.**
✔ **Best for scenarios where sorted order is required.**

📌 Deep Dive into `EnumSet<T>` in Java (Easy Explanation)

1️⃣ What is `EnumSet<T>`?

`EnumSet<T>` is a specialized **Set implementation for Enums** in Java. It is designed to work **only with Enums** and is much **faster and memory-efficient** than other Set implementations like `HashSet` or `TreeSet`.

✔ **Stores only `enum` values.**
✔ **Extremely fast (Better than `HashSet` and `TreeSet`).**
✔ **Compact memory usage (Uses bitwise operations).**
✔ **Maintains natural order of Enums.**

2️⃣ How `EnumSet<T>` Works Internally?

- Unlike `HashSet`, which uses a **HashMap**, `EnumSet` uses a **bitwise representation** to store elements.

- Each `enum` constant is assigned a **bit position**, making operations **very fast (`O(1)`)**.
- Since `EnumSet` is backed by a **bit vector**, it **does not allow null values**.
- It maintains **insertion order** based on how `enum` constants are declared.

3️⃣ Creating an `EnumSet` (Basic Example)

Let's define an `enum` first:

```
enum Days {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Now, let's create an `EnumSet` and add elements:

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        EnumSet<Days> weekend = EnumSet.of(Days.SATURDAY, Days.SUNDAY);
        System.out.println(weekend); // Output: [SATURDAY, SUNDAY]
    }
}
```

✔ **Stores only `enum` values**
✔ **Maintains insertion order**

4️⃣ Ways to Create an `EnumSet<T>`

📌 1. `EnumSet.of(E... elements)` → Create from specific values

```
EnumSet<Days> set = EnumSet.of(Days.MONDAY, Days.WEDNESDAY);
System.out.println(set); // Output: [MONDAY, WEDNESDAY]
```

📌 2. `EnumSet.allOf(EnumType.class)` → Create a set of all Enum values

```
EnumSet<Days> allDays = EnumSet.allOf(Days.class);
System.out.println(allDays); // Output: [MONDAY, TUESDAY, WEDNESDAY, ...]
```

📌 3. `EnumSet.noneOf(EnumType.class)` → Create an empty set

```
EnumSet<Days> emptySet = EnumSet.noneOf(Days.class);
System.out.println(emptySet); // Output: []
```

📌 4. `EnumSet.range(E from, E to)` → Create a range of Enum values

```
EnumSet<Days> midWeek = EnumSet.range(Days.TUESDAY, Days.THURSDAY);
System.out.println(midWeek); // Output: [TUESDAY, WEDNESDAY, THURSDAY]
```

📌 5. `EnumSet.copyOf(Collection<E> c)` → Create from another collection

```
List<Days> list = Arrays.asList(Days.MONDAY, Days.FRIDAY);
EnumSet<Days> copiedSet = EnumSet.copyOf(list);
System.out.println(copiedSet); // Output: [MONDAY, FRIDAY]
```

5️⃣ Important Methods of `EnumSet<T>`

Method	Description
`add(E e)`	Adds an element to the set.
`remove(E e)`	Removes an element from the set.
`contains(E e)`	Checks if the set contains an element.
`size()`	Returns the number of elements in the set.
`isEmpty()`	Checks if the set is empty.
`clear()`	Removes all elements from the set.
`iterator()`	Returns an iterator to traverse the set.
`complementOf(EnumSet<E> s)`	Returns a set containing all elements **except** those in `s`.

6️⃣ Examples of `EnumSet<T>` Methods

📌 1. `add(E e)` and `remove(E e)` → Add & Remove Elements

```
EnumSet<Days> set = EnumSet.noneOf(Days.class);
set.add(Days.MONDAY);
set.add(Days.FRIDAY);
set.remove(Days.MONDAY);

System.out.println(set); // Output: [FRIDAY]
```

📌 2. `contains(E e)` → Check if an element exists

```
EnumSet<Days> set = EnumSet.of(Days.WEDNESDAY, Days.FRIDAY);
System.out.println(set.contains(Days.FRIDAY)); // Output: true
System.out.println(set.contains(Days.SUNDAY)); // Output: false
```

📌 3. `complementOf(EnumSet<E> s)` → Get the complement set

```
EnumSet<Days> workingDays = EnumSet.range(Days.MONDAY, Days.FRIDAY);
EnumSet<Days> nonWorkingDays = EnumSet.complementOf(workingDays);

System.out.println(workingDays);  // Output: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
```

```
FRIDAY]
System.out.println(nonWorkingDays); // Output: [SATURDAY, SUNDAY]
```

---

7 Performance Analysis

Operation	Time Complexity
`add(E e)`	O(1)
`remove(E e)`	O(1)
`contains(E e)`	O(1)
`size()`	O(1)
`iterator()`	O(n)

✔ **Extremely fast because it uses bitwise operations.**

---

8 When to Use `EnumSet<T>`?

Use `EnumSet` When...	Avoid `EnumSet` When...
You have **enum values** to store.	You need to store **non-enum values**.
You need a **faster and memory-efficient** Set.	You need to store **null values** (`EnumSet` does not allow `null`).
You want **ordered enum storage**.	You need a **hashed or sorted collection** (Use `HashSet` or `TreeSet`).

---

9 Summary

✔ **EnumSet is the best choice for storing Enums in a Set.**
✔ **Much faster and memory-efficient than `HashSet` and `TreeSet`.**
✔ **Uses bitwise operations for fast access.**
✔ **Maintains natural order of Enum constants.**
✔ **Does not allow `null` values.**

📌 Deep Dive into `ConcurrentSkipListSet<T>` in Java (Easy Explanation)

1️⃣ What is `ConcurrentSkipListSet<T>`?

`ConcurrentSkipListSet<T>` is a **thread-safe, sorted Set implementation** in Java.
It is part of the **java.util.concurrent** package and is designed for **concurrent (multi-threaded) environments**.

- ✔ **Thread-Safe** (Multiple threads can modify it safely).
- ✔ **Sorted Set** (Maintains natural order of elements).
- ✔ **Non-Synchronized Alternative to `TreeSet`**.
- ✔ **Uses a Skip List (Efficient for concurrent reads/writes).**
- ✔ **Does not allow `null` elements.**

2️⃣ How `ConcurrentSkipListSet<T>` Works Internally?

- It uses a **Skip List** instead of a Tree or Hash structure.
- A Skip List is like a **linked list with multiple levels** to speed up searches.
- It provides **logarithmic time complexity (`O(log n)`) for add, remove, and search** operations.
- Unlike `TreeSet`, which uses **synchronized locks**, `ConcurrentSkipListSet` allows **lock-free concurrent access**, making it much faster in multi-threaded scenarios.

🔹 **Comparison with Other Sets**

Feature	`ConcurrentSkipListSet`	`TreeSet`	`HashSet`
Thread-Safe?	☑ Yes	❌ No	❌ No
Sorted?	☑ Yes (Natural Order)	☑ Yes (Natural Order)	❌ No
Performance (Insert/Search)	⚡ `O(log n)`	⚡ `O(log n)`	🔥 `O(1)`
Allows `null`?	❌ No	❌ No	☑ Yes

3️⃣ Creating a `ConcurrentSkipListSet<T>` (Basic Example)

Let's create a `ConcurrentSkipListSet` and add elements to it:

```
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) {
        ConcurrentSkipListSet<Integer> set = new ConcurrentSkipListSet<>();

        set.add(10);
```

```
        set.add(5);
        set.add(20);
        set.add(15);

        System.out.println(set); // Output: [5, 10, 15, 20] (Sorted Order)
    }
}
```

✔ **Elements are always sorted in natural order**.
✔ **Thread-Safe operations without explicit locking**.

---

4 Important Methods of `ConcurrentSkipListSet<T>`

Method	Description
`add(E e)`	Adds an element to the set.
`remove(E e)`	Removes an element from the set.
`contains(E e)`	Checks if the set contains an element.
`size()`	Returns the number of elements in the set.
`isEmpty()`	Checks if the set is empty.
`pollFirst()`	Retrieves and removes the **smallest** element.
`pollLast()`	Retrieves and removes the **largest** element.
`headSet(E toElement)`	Returns elements **less than** `toElement`.
`tailSet(E fromElement)`	Returns elements **greater than or equal to** `fromElement`.
`subSet(E fromElement, E toElement)`	Returns a range of elements.

---

5 Examples of `ConcurrentSkipListSet<T>` Methods

📍 1. `add(E e)`, `remove(E e)`, and `contains(E e)`

```
ConcurrentSkipListSet<Integer> set = new ConcurrentSkipListSet<>();
set.add(10);
set.add(5);
set.add(20);
set.remove(10);

System.out.println(set.contains(10)); // Output: false
System.out.println(set); // Output: [5, 20]
```

✔ `add()` **inserts elements in sorted order.**
✔ `remove()` **deletes elements safely in multi-threaded environments.**
✔ `contains()` **checks if an element exists.**

📌 2. `pollFirst()` and `pollLast()` → Retrieve & Remove First/Last Element

```
ConcurrentSkipListSet<Integer> set = new ConcurrentSkipListSet<>();
set.add(10);
set.add(5);
set.add(20);

System.out.println(set.pollFirst()); // Output: 5 (Removes Smallest Element)
System.out.println(set.pollLast());  // Output: 20 (Removes Largest Element)
System.out.println(set); // Output: [10]
```

📌 3. `headSet(E toElement)` → Get elements less than a value

```
ConcurrentSkipListSet<Integer> set = new ConcurrentSkipListSet<>();
set.add(10);
set.add(5);
set.add(20);
set.add(15);

System.out.println(set.headSet(15)); // Output: [5, 10]
```

✔ **Returns all elements smaller than `15`**

📌 4. `tailSet(E fromElement)` → Get elements greater than or equal to a value

```
ConcurrentSkipListSet<Integer> set = new ConcurrentSkipListSet<>();
set.add(10);
set.add(5);
set.add(20);
set.add(15);

System.out.println(set.tailSet(15)); // Output: [15, 20]
```

✔ **Returns all elements `>= 15`**

📌 5. `subSet(E fromElement, E toElement)` → Get a range of elements

```
ConcurrentSkipListSet<Integer> set = new ConcurrentSkipListSet<>();
set.add(10);
set.add(5);
set.add(20);
set.add(15);

System.out.println(set.subSet(10, 20)); // Output: [10, 15]
```

✔ **Returns elements in the range `[10, 20)` (exclusive of 20)**

---

6 Performance Analysis

Operation	Time Complexity
`add(E e)`	`O(log n)`
`remove(E e)`	`O(log n)`
`contains(E e)`	`O(log n)`
`pollFirst() / pollLast()`	`O(log n)`
`headSet(E e) / tailSet(E e) / subSet(E e, E e)`	`O(log n)`

✔ **Faster than `TreeSet` in concurrent scenarios.**
✔ **Performs better in multi-threaded applications.**

---

7 When to Use `ConcurrentSkipListSet<T>`?

Use `ConcurrentSkipListSet` When...	Avoid `ConcurrentSkipListSet` When...
You need a **thread-safe sorted Set**.	You don't need sorting (Use `ConcurrentHashMap`).
You need **fast concurrent reads & writes**.	You need **faster writes** (`HashSet` is faster for single-threaded use).
You need **logarithmic time complexity (`O(log n)`)**.	You need constant-time lookups (`HashSet` provides `O(1)`).

---

8 Summary

✔ **Thread-Safe alternative to `TreeSet`.**
✔ **Faster concurrent operations than `TreeSet`.**
✔ **Uses Skip List for `O(log n)` operations.**
✔ **Maintains elements in sorted order.**
✔ **Does not allow `null` values.**

Yes! Before moving to `CopyOnWriteArraySet<T>`, let's first understand **SortedSet<T>** in deep detail.

📌 Deep Dive into `SortedSet<T>` in Java (Easy Explanation)

1 What is `SortedSet<T>`?

A **`SortedSet<T>`** is a specialized version of the `Set<T>` interface that **maintains elements in a sorted order**.
It is part of the **`java.util` package** and is implemented by `TreeSet<T>`.

✔ **No Duplicate Elements** (Like `Set<T>`)
✔ **Maintains Sorted Order** (Ascending order by default)
✔ **Supports Range Queries** (`headSet()`, `tailSet()`, `subSet()`)
✔ **Implements `NavigableSet<T>`** (which extends `SortedSet<T>` for more flexibility)

---

2 How `SortedSet<T>` Works Internally?

- It **extends `Set<T>`** and enforces a **sorting order**.
- It can use **natural ordering (Comparable)** or **custom ordering (Comparator)**.
- The most common implementation is **`TreeSet<T>`**, which is based on a **Red-Black Tree**.
- The **sorting mechanism is automatic**, meaning elements are always stored in sorted order.

---

3 Declaring a `SortedSet<T>` in Java

Since `SortedSet<T>` is an interface, we use `TreeSet<T>` as an implementation.

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        SortedSet<Integer> sortedSet = new TreeSet<>();

        sortedSet.add(30);
        sortedSet.add(10);
        sortedSet.add(20);
        sortedSet.add(50);
        sortedSet.add(40);

        System.out.println(sortedSet); // Output: [10, 20, 30, 40, 50] (Sorted Order)
    }
}
```

✔ **Automatically maintains sorted order**.
✔ **Duplicates are not allowed**.

---

4 Important Methods of `SortedSet<T>`

Method	Description
`first()`	Returns the first (smallest) element.
`last()`	Returns the last (largest) element.
`headSet(E toElement)`	Returns elements **less than** `toElement`.
`tailSet(E fromElement)`	Returns elements **greater than or equal to** `fromElement`.

`subSet(E fromElement, E toElement)`	Returns elements within the range `[fromElement, toElement)`.
`comparator()`	Returns the comparator used for ordering (or `null` for natural ordering).

---

5 Examples of `SortedSet<T>` Methods

1. `first()` and `last()` → Get First and Last Element

```
SortedSet<Integer> set = new TreeSet<>();
set.add(10);
set.add(30);
set.add(20);
set.add(50);
set.add(40);

System.out.println(set.first()); // Output: 10
System.out.println(set.last());  // Output: 50
```

✔ **Retrieves the smallest and largest elements.**

---

2. `headSet(E toElement)` → Get Elements Less Than a Value

```
SortedSet<Integer> set = new TreeSet<>();
set.add(10);
set.add(20);
set.add(30);
set.add(40);
set.add(50);

System.out.println(set.headSet(30)); // Output: [10, 20]
```

✔ **Returns elements smaller than `30`.**

---

3. `tailSet(E fromElement)` → Get Elements Greater Than or Equal to a Value

```
SortedSet<Integer> set = new TreeSet<>();
set.add(10);
set.add(20);
set.add(30);
set.add(40);
set.add(50);

System.out.println(set.tailSet(30)); // Output: [30, 40, 50]
```

✔ **Returns elements `>= 30`.**

📌 4. `subSet(E fromElement, E toElement)` → Get a Range of Elements

```
SortedSet<Integer> set = new TreeSet<>();
set.add(10);
set.add(20);
set.add(30);
set.add(40);
set.add(50);

System.out.println(set.subSet(20, 40)); // Output: [20, 30]
```

✔ **Returns elements in the range `[20, 40)` (exclusive of 40).**

📌 5. `comparator()` → Get the Sorting Comparator

```
SortedSet<Integer> set = new TreeSet<>();
System.out.println(set.comparator()); // Output: null (Natural Ordering)
```

✔ **Returns `null` if elements are sorted naturally.**

6️⃣ Performance Analysis

Operation	Time Complexity
`add(E e)`	`O(log n)`
`remove(E e)`	`O(log n)`
`contains(E e)`	`O(log n)`
`first() / last()`	`O(1)`
`headSet(E e) / tailSet(E e) / subSet(E e, E e)`	`O(log n)`

✔ **Uses a Red-Black Tree for efficient sorting.**
✔ **Faster than `LinkedList` but slower than `HashSet` for insertions/removals.**

7️⃣ When to Use `SortedSet<T>`?

Use `SortedSet` When...	Avoid `SortedSet` When...
You need **automatically sorted elements**.	You don't care about ordering (Use `HashSet`).
You need **range queries** (`headSet()`, `tailSet()`, etc.).	You need **fast lookups (`O(1)`)** (Use `HashSet`).
You need **logarithmic time complexity (`O(log n)`)**.	You need **constant-time inserts/deletes (`O(1)`)**.

8️⃣ Summary

✔ `SortedSet<T>` **maintains sorted order**.
✔ **Implemented using** `TreeSet<T>` **(Red-Black Tree)**.
✔ **Supports range queries like** `headSet()`, `tailSet()`, `subSet()`.
✔ **Does not allow duplicate elements**.
✔ **Faster than** `LinkedList`, **but slower than** `HashSet`.

📌 Deep Dive into `CopyOnWriteArraySet<T>` in Java (Easy Explanation)

1️⃣ What is `CopyOnWriteArraySet<T>`?

A `CopyOnWriteArraySet<T>` is a thread-safe implementation of the `Set<T>` interface. It is part of the `java.util.concurrent` package and is designed for **concurrent environments** where reads are frequent but writes (modifications) are rare.

✔ **Thread-Safe** (No need for manual synchronization)
✔ **No Duplicate Elements** (Like `Set<T>`)
✔ **Based on** `CopyOnWriteArrayList<T>` (Internally uses an array)
✔ **Best for Scenarios with Frequent Reads and Rare Writes**

2️⃣ How `CopyOnWriteArraySet<T>` Works Internally?

- It is backed by a `CopyOnWriteArrayList<T>`.
- Every time a **modification (add/remove) occurs**, it creates a **new copy of the underlying array**.
- **Iterators are fail-safe**, meaning they do not throw `ConcurrentModificationException`.
- **Best suited for scenarios where reading happens more frequently than writing**.

3️⃣ Declaring a `CopyOnWriteArraySet<T>` in Java

Since `CopyOnWriteArraySet<T>` is a concrete class, we can instantiate it directly.

```
import java.util.concurrent.CopyOnWriteArraySet;

public class Main {
    public static void main(String[] args) {
        CopyOnWriteArraySet<Integer> set = new CopyOnWriteArraySet<>();

        set.add(10);
        set.add(20);
        set.add(30);
        set.add(10); // Duplicate, will not be added
```

```
        System.out.println(set); // Output: [10, 20, 30]
    }
}
```

✔ **Automatically avoids duplicates**.
✔ **Thread-safe without explicit locks**.

4️⃣ Important Methods of `CopyOnWriteArraySet<T>`

Method	Description
`add(E e)`	Adds an element to the set (if not already present).
`remove(E e)`	Removes the element from the set.
`contains(E e)`	Checks if an element is present in the set.
`size()`	Returns the number of elements in the set.
`iterator()`	Returns a fail-safe iterator.
`toArray()`	Converts the set into an array.

5️⃣ Examples of `CopyOnWriteArraySet<T>` Methods

📌 1. `add()` and `remove()` → Add and Remove Elements

```
import java.util.concurrent.CopyOnWriteArraySet;

public class Main {
    public static void main(String[] args) {
        CopyOnWriteArraySet<String> set = new CopyOnWriteArraySet<>();

        set.add("Apple");
        set.add("Banana");
        set.add("Cherry");

        System.out.println(set); // Output: [Apple, Banana, Cherry]

        set.remove("Banana");
        System.out.println(set); // Output: [Apple, Cherry]
    }
}
```

✔ **Handles duplicates and thread safety automatically**.

📌 2. `contains()` → Check If an Element Exists

```
CopyOnWriteArraySet<Integer> set = new CopyOnWriteArraySet<>();
set.add(100);
set.add(200);
set.add(300);

System.out.println(set.contains(200)); // Output: true
System.out.println(set.contains(400)); // Output: false
```

✔ **Efficiently checks for element presence**.

📌 3. `iterator()` → Fail-Safe Iterator

```
import java.util.concurrent.CopyOnWriteArraySet;
import java.util.Iterator;

public class Main {
    public static void main(String[] args) {
        CopyOnWriteArraySet<String> set = new CopyOnWriteArraySet<>();
        set.add("A");
        set.add("B");
        set.add("C");

        Iterator<String> iterator = set.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
            set.add("D"); // No ConcurrentModificationException!
        }

        System.out.println(set); // Output: [A, B, C, D]
    }
}
```

✔ **Iterator does not throw `ConcurrentModificationException`.**
✔ **Changes made while iterating will not affect the current iterator**.

6️⃣ Performance Analysis

Operation	Time Complexity
`add(E e)`	`O(n)` (Creates a new copy of the array)
`remove(E e)`	`O(n)`
`contains(E e)`	`O(n)`
`iteration`	`O(n)`

✔ **Best for multi-threaded environments with frequent reads and rare writes**.
✔ **Not suitable for scenarios with frequent insertions/removals (`O(n)`)**.

7️⃣ When to Use `CopyOnWriteArraySet<T>`?

Use `CopyOnWriteArraySet<T>` When...	Avoid `CopyOnWriteArraySet<T>` When...
Frequent reads, rare writes.	**Frequent additions/removals** (`O(n)`).
Multiple threads accessing the set.	**Performance is critical** (Use `HashSet` for faster operations).
You need fail-safe iterators.	**You have large datasets** (Memory overhead is high).

8️⃣ Summary

✔ **Thread-safe `Set<T>` implementation** (No need for manual synchronization).
✔ **Uses `CopyOnWriteArrayList<T>` internally** (Every modification creates a new copy).
✔ **Best for read-heavy operations in multi-threaded environments**.
✔ **Fail-safe iterators (No `ConcurrentModificationException`)**.
✔ **Not suitable for frequent writes (`O(n)` complexity)**.

📌 Chapter 5: Queue Interface (FIFO Data Structure) in Java (Deep and Easy Explanation)

1️⃣ What is a Queue?

A **Queue** is a **FIFO (First-In-First-Out)** data structure, meaning the **first element added** will be the **first element removed**.

✔ **Imagine a queue at a movie ticket counter:**

- The first person who arrives will be the first one to get the ticket.
- The next person waits in line until it's their turn.

✔ **Real-Life Examples of Queues:**

- **Print Queue:** The first document sent to the printer gets printed first.
- **Call Center Support:** The first customer in line gets connected to an agent first.

✔ **Java Provides `Queue<T>` Interface**

- It is part of **`java.util` package** and extends the `Collection<T>` interface.
- **Different Implementations** are available based on requirements.

2️⃣ Why Do We Need a Queue in Java?

- **Problem with Arrays & Lists:**
  - `ArrayList` and `LinkedList` allow insertion and removal, but they do **not follow FIFO automatically**.
  - Using **`remove(0)`** in an `ArrayList` is slow (`O(n)`) because all elements shift left.
- **Queue is the Solution:**
  - Efficiently **adds elements at the rear** and **removes from the front** (`O(1)`).
  - Provides **built-in methods** for managing elements.

3️⃣ Queue Interface and Its Methods

Method	Description
`add(E e)`	Adds an element at the end (throws exception if full).
`offer(E e)`	Adds an element at the end (returns false if full).
`remove()`	Removes and returns the front element (throws exception if empty).
`poll()`	Removes and returns the front element (returns null if empty).
`element()`	Retrieves the front element without removing (throws exception if empty).
`peek()`	Retrieves the front element without removing (returns null if empty).

✔ **Use `offer()` and `poll()` instead of `add()` and `remove()` to avoid exceptions.**

4️⃣ Queue Hierarchy in Java

```
Queue<T>  (Interface)
│
├── LinkedList<T>  (Doubly Linked List Implementation)
│
├── PriorityQueue<T>  (Min-Heap Implementation)
│
├── Deque<T>  (Double-Ended Queue Interface)
│    ├── ArrayDeque<T>  (Array-Based Deque)
│    ├── LinkedList<T>  (Also Implements Deque)
│
├── ConcurrentLinkedQueue<T>  (Thread-Safe, Non-Blocking Queue)
│
├── BlockingQueue<T>  (Used in Multi-Threading)
│    ├── ArrayBlockingQueue<T>
│    ├── LinkedBlockingQueue<T>
│    ├── PriorityBlockingQueue<T>
```

```
│   ├── SynchronousQueue<T>
│   ├── DelayQueue<T>
```

---

5 Types of Queue Implementations in Java

Let's go through the different types of `Queue<T>` implementations.

1 LinkedList<T> as a Queue

- Implements `Queue<T>`, `Deque<T>`, and `List<T>`.
- Can be used as **FIFO Queue** or **Deque**.
- Not thread-safe.

2 PriorityQueue<T>

- Uses **Min-Heap** internally.
- Orders elements based on **natural ordering or custom comparator**.
- Does **not** guarantee FIFO.

3 Deque<T> (Double-Ended Queue)

- Allows insertion and deletion from **both ends**.
- `ArrayDeque<T>` and `LinkedList<T>` implement `Deque<T>`.

4 ArrayDeque<T> (Resizable Array-Based Deque)

- Faster than `Stack<T>` for LIFO.
- Faster than `LinkedList<T>` for FIFO.

5 ConcurrentLinkedQueue<T>

- **Thread-safe implementation** of `Queue<T>`.
- **Non-blocking** (uses **CAS** instead of locks).

6 BlockingQueue<T> (Used in Multi-Threading)

- Designed for **multi-threading scenarios**.
- Blocks producer/consumer threads if the queue is full/empty.
- Common implementations:
    - `ArrayBlockingQueue<T>` → **Fixed-size array-based blocking queue**.
    - `LinkedBlockingQueue<T>` → **Linked list-based blocking queue**.
    - `PriorityBlockingQueue<T>` → **Priority-based blocking queue**.
    - `SynchronousQueue<T>` → **Transfers elements between threads directly**.

- `DelayQueue<T>` → **Stores elements with delayed processing**.

6️⃣ Performance Comparison of Queue Implementations

Queue Type	Insertion (O)	Deletion (O)	Thread-Safe?
`LinkedList<T>`	`O(1)`	`O(1)`	❌ No
`PriorityQueue<T>`	`O(log n)`	`O(log n)`	❌ No
`ArrayDeque<T>`	`O(1)`	`O(1)`	❌ No
`ConcurrentLinkedQueue<T>`	`O(1)`	`O(1)`	☑ Yes (Non-Blocking)
`BlockingQueue<T>`	`O(1)`	`O(1)`	☑ Yes (Blocking)

7️⃣ When to Use Which Queue?

Use Case	Best Queue Implementation
Simple FIFO operations	`LinkedList<T>`
Priority-based processing	`PriorityQueue<T>`
Double-ended queue operations	`ArrayDeque<T>`
Multi-threaded queue (non-blocking)	`ConcurrentLinkedQueue<T>`
Multi-threaded queue (blocking)	`BlockingQueue<T>`

📌 Summary

✔ **Queue<T> follows FIFO (First-In-First-Out).**
✔ **Different implementations available:** `LinkedList<T>`, `PriorityQueue<T>`, `ArrayDeque<T>`, `ConcurrentLinkedQueue<T>`, `BlockingQueue<T>`.
✔ **Use `offer()` and `poll()` instead of `add()` and `remove()` to avoid exceptions.**
✔ **Choose the right queue based on performance needs (thread-safety, ordering, blocking, etc.).**

🚀 LinkedList<T> as a Queue (Deep & Easy Explanation)

1️⃣ What is LinkedList<T> as a Queue?

`LinkedList<T>` is a **doubly linked list** that implements the `Queue<T>` interface.
It allows **FIFO (First-In-First-Out) operations**, making it a **good choice** for a queue.

✔ **Key Features of LinkedList as a Queue:**

- **Uses Nodes** (Each element points to the next and previous element).
- **Fast Insertions & Deletions (`O(1)`)** at both ends.
- **Maintains Order** (Insertion order is preserved).
- **Allows Null Values.**
- **Not Thread-Safe** (Needs external synchronization for multi-threading).

✔ **Real-Life Example:**

- **Train Coaches:** The first coach attached is the first to leave the station.

---

2 How LinkedList<T> Works as a Queue?

✔ **Queue Operations:**

1 **Enqueue (Add element at the rear)** → `offer(E e)` / `add(E e)`
2 **Dequeue (Remove element from the front)** → `poll()` / `remove()`
3 **Peek (Retrieve front element without removing)** → `peek()` / `element()`

📌 **Internal Working:**

- **Each element is stored in a Node (`Node<E>`)**
- **Two pointers (`head` and `tail`) keep track of the front & rear.**
- **Adding is done at `tail`**, removing is done from `head`.**

```
HEAD → [10] → [20] → [30] → TAIL
```

✔ **Adding 40 to Queue (`offer(40)`)**

```
HEAD → [10] → [20] → [30] → [40] → TAIL
```

✔ **Removing (`poll()`)**

```
HEAD → [20] → [30] → [40] → TAIL  (10 is removed)
```

---

3 LinkedList<T> Methods for Queue

Method	Description
`add(E e)`	Adds an element to the queue (throws exception if full).
`offer(E e)`	Adds an element to the queue (returns `false` if full).
`remove()`	Removes the front element (throws exception if empty).
`poll()`	Removes the front element (returns `null` if empty).

`element()`	Retrieves the front element without removing (throws exception if empty).
`peek()`	Retrieves the front element without removing (returns `null` if empty).

4️⃣ Implementation of LinkedList<T> as a Queue

```
import java.util.LinkedList;
import java.util.Queue;

public class LinkedListQueueExample {
    public static void main(String[] args) {
        // Create a Queue using LinkedList
        Queue<Integer> queue = new LinkedList<>();

        // Adding elements to the queue
        queue.offer(10);
        queue.offer(20);
        queue.offer(30);

        System.out.println("Queue: " + queue); // [10, 20, 30]

        // Peek (front element without removing)
        System.out.println("Front Element: " + queue.peek()); // 10

        // Removing elements
        System.out.println("Removed: " + queue.poll()); // 10
        System.out.println("Queue after removal: " + queue); // [20, 30]

        // Checking if queue is empty
        System.out.println("Is queue empty? " + queue.isEmpty()); // false
    }
}
```

✔ **Output:**

```
Queue: [10, 20, 30]
Front Element: 10
Removed: 10
Queue after removal: [20, 30]
Is queue empty? false
```

5️⃣ How LinkedList<T> Works Internally as a Queue

✔ **Structure:**

- **Each element is stored in a `Node<E>`.**
- **Each Node contains:**
  - `E data` (Element Value)
  - `Node<E> next` (Pointer to next node)

- `Node<E> prev` (Pointer to previous node)

✔ **Internal Representation:**

```
head → [10] ⇔ [20] ⇔ [30] → tail
```

✔ **Adding an Element (`offer(40)`)**

```
head → [10] ⇔ [20] ⇔ [30] ⇔ [40] → tail
```

✔ **Removing an Element (`poll()`)**

```
head → [20] ⇔ [30] ⇔ [40] → tail
```

---

6 Performance Analysis of LinkedList as a Queue

Operation	Complexity (O)
`offer(E e)` (Add to rear)	`O(1)`
`poll()` (Remove from front)	`O(1)`
`peek()` (Retrieve front)	`O(1)`
Search	`O(n)`

✔ **Why is LinkedList Fast for Queue?**

- `O(1)` **insertion and deletion at both ends** (No shifting needed).
- `O(n)` **search** (Not efficient for finding elements).

---

7 When to Use LinkedList as a Queue?

✔ **Use `LinkedList<T>` when:**
☑ **Fast Insertion & Deletion (`O(1)`) are needed.**
☑ **You don't need random access (`O(n)`).**
☑ **Maintaining insertion order is important.**
☑ **You need a flexible data structure (Can act as a Queue & Deque).**

❌ **Don't use LinkedList<T> when:**
🚫 **You need frequent searching (`O(n)`).**
🚫 **Memory consumption is a concern (Each node requires extra pointers).**

📌 Summary

✔ `LinkedList<T>` **implements** `Queue<T>`.
✔ **FIFO operations:** Insert at the tail, remove from the head.
✔ **Efficient** `O(1)` **insertion & deletion, but** `O(n)` **search.**

✔ **Uses `Node<E>` (doubly linked list structure).**
✔ **Best for scenarios needing fast insert/remove, but not for random access.**

🚀 PriorityQueue<T> (Deep & Easy Explanation)

1️⃣ What is a PriorityQueue<T>?

A **PriorityQueue<T>** is a special type of queue where **elements are ordered based on priority** rather than insertion order.
It is based on **Heap Data Structure** (Min-Heap or Max-Heap).

✔ **Key Features of PriorityQueue:**

- **Elements are sorted based on priority (Natural or Custom Comparator).**
- **By default, it is a Min-Heap (Smallest element at the top).**
- **Does NOT allow `null` values.**
- **Not thread-safe** (Use `PriorityBlockingQueue` for multi-threading).

✔ **Real-Life Example:**

- **Hospital Emergency Room:** Patients with serious conditions are treated first.
- **Dijkstra's Algorithm:** Used in shortest path finding.

2️⃣ How PriorityQueue<T> Works?

✔ **Queue Operations:**
1️⃣ **Enqueue (Add element in the correct position based on priority)** → `offer(E e)` / `add(E e)`
2️⃣ **Dequeue (Remove element with highest priority)** → `poll()` / `remove()`
3️⃣ **Peek (Retrieve highest-priority element without removing)** → `peek()` / `element()`

📌 **Default Behavior:**

- **Min-Heap (Smallest element first).**
- **Max-Heap (Largest element first) needs a custom comparator.**

```
Min-Heap:
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.offer(30);
pq.offer(10);
pq.offer(20);

Internally Stored:
[10, 30, 20]   → 10 is the highest priority (Min-Heap)
```

✔ **Adding 5 to Queue (`offer(5)`)**

```
[5, 10, 20, 30]   → 5 moves to the top
```

✔ **Removing (`poll()`)**

```
[10, 30, 20]   → 5 is removed
```

---

3 PriorityQueue<T> Methods

Method	Description
`add(E e)`	Adds an element to the queue (throws exception if full).
`offer(E e)`	Adds an element to the queue (returns `false` if full).
`remove()`	Removes the highest-priority element (throws exception if empty).
`poll()`	Removes the highest-priority element (returns `null` if empty).
`element()`	Retrieves the highest-priority element without removing (throws exception if empty).
`peek()`	Retrieves the highest-priority element without removing (returns `null` if empty).

---

4 Implementation of PriorityQueue<T>

```
import java.util.PriorityQueue;

public class PriorityQueueExample {
    public static void main(String[] args) {
        // Create a Min-Heap (default)
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        // Adding elements
        pq.offer(30);
        pq.offer(10);
        pq.offer(20);

        System.out.println("PriorityQueue: " + pq); // Output: [10, 30, 20]

        // Peek (Retrieve highest priority element)
        System.out.println("Top Element: " + pq.peek()); // Output: 10

        // Removing elements
        System.out.println("Removed: " + pq.poll()); // Output: 10
        System.out.println("PriorityQueue after removal: " + pq); // Output: [20, 30]
    }
}
```

✔ **Output:**

```
PriorityQueue: [10, 30, 20]
Top Element: 10
```

```
Removed: 10
PriorityQueue after removal: [20, 30]
```

---

5 How PriorityQueue Works Internally?

✔ **Structure:**

- **Uses a Min-Heap (Binary Heap) internally.**
- **Heap is stored in an array for efficient retrieval.**
- **Insertion follows heap properties (smallest at root).**
- **Removal maintains heap properties (restructure after deletion).**

✔ **Internal Representation (Heap Structure)**

```
      10
     /  \
   30    20
```

✔ **Adding 5 (`offer(5)`)**

```
      5
     /  \
   10    20
  /
 30
```

✔ **Removing (`poll()`)**

```
      10
     /  \
   30    20
```

---

6 Custom Comparator for Max-Heap (Highest First)

By default, `PriorityQueue` is a **Min-Heap**. To make it a **Max-Heap**, use a custom comparator.

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

public class MaxHeapExample {
    public static void main(String[] args) {
        // Max-Heap using Comparator
        PriorityQueue<Integer> maxHeap = new
PriorityQueue<>(Collections.reverseOrder());

        maxHeap.offer(30);
        maxHeap.offer(10);
        maxHeap.offer(20);

        System.out.println("Max-Heap PriorityQueue: " + maxHeap); // Output: [30, 10,
```

```
20]

        System.out.println("Top Element: " + maxHeap.peek()); // Output: 30
        System.out.println("Removed: " + maxHeap.poll()); // Output: 30
    }
}
```

✔ **Output:**

```
Max-Heap PriorityQueue: [30, 10, 20]
Top Element: 30
Removed: 30
```

---

7 Performance Analysis of PriorityQueue<T>

Operation	Complexity (O)
`offer(E e)` (Insertion)	`O(log n)`
`poll()` (Remove highest priority)	`O(log n)`
`peek()` (Retrieve highest priority)	`O(1)`

✔ **Why is PriorityQueue Fast?**

- **Uses Heap structure (Efficient insertion/removal).**
- **Heap properties ensure quick access to the highest priority.**

---

8 When to Use PriorityQueue?

✔ **Use `PriorityQueue<T>` when:**
☑ **You need efficient priority-based retrieval.**
☑ **You need a Min-Heap (`O(log n)` operations).**
☑ **You need a Max-Heap (With custom comparator).**

❌ **Don't use PriorityQueue when:**
🚫 **You need FIFO ordering (Use `LinkedList` for Queue).**
🚫 **You need thread-safety (Use `PriorityBlockingQueue`).**
🚫 **You need fast random access (`O(n)`).**

📌 Summary

✔ **`PriorityQueue<T>` orders elements based on priority.**
✔ **Min-Heap by default (Smallest element first).**
✔ **Supports custom comparator for Max-Heap.**
✔ **Operations are `O(log n)`, making it efficient.**
✔ **Best for priority-based tasks like scheduling, pathfinding, etc.**

🚀 Deque<T> (Double-Ended Queue) - Deep & Easy Explanation

1️⃣ What is a Deque<T>?

A **Deque (Double-Ended Queue)** is a special type of queue where **elements can be added or removed from both ends (front & rear).**

✔ **Key Features of Deque:**

- **Supports FIFO & LIFO operations.**
- **Efficient insertions/removals from both ends.**
- **Allows `null` values (except in concurrent implementations).**
- **Faster than `LinkedList` for queue operations.**
- **Thread-safe versions exist (`ConcurrentLinkedDeque`).**

✔ **Real-Life Example:**

- **Deque in Browsers:** Back & Forward navigation history.
- **Job Scheduling:** Tasks can be added at the beginning or end.

2️⃣ How Deque<T> Works?

✔ **Operations on Both Ends:**

1. **Add to Front** → `addFirst(E e)` / `offerFirst(E e)`
2. **Remove from Front** → `removeFirst()` / `pollFirst()`
3. **Add to Rear** → `addLast(E e)` / `offerLast(E e)`
4. **Remove from Rear** → `removeLast()` / `pollLast()`
5. **Peek (Retrieve without removing)** → `peekFirst()` / `peekLast()`

✔ **Deque as a Queue (FIFO)**

```
Front → [1, 2, 3, 4, 5] → Rear
```

✔ **Deque as a Stack (LIFO)**

```
Top → [1, 2, 3, 4, 5] (Last In First Out)
```

3️⃣ Deque<T> Methods

Method	Description

`addFirst(E e)`	Adds element at the front (throws exception if full).
`offerFirst(E e)`	Adds element at the front (returns `false` if full).
`addLast(E e)`	Adds element at the rear (throws exception if full).
`offerLast(E e)`	Adds element at the rear (returns `false` if full).
`removeFirst()`	Removes the first element (throws exception if empty).
`pollFirst()`	Removes the first element (returns `null` if empty).
`removeLast()`	Removes the last element (throws exception if empty).
`pollLast()`	Removes the last element (returns `null` if empty).
`peekFirst()`	Retrieves the first element without removing.
`peekLast()`	Retrieves the last element without removing.

Implementation of Deque<T>

```
import java.util.Deque;
import java.util.LinkedList;

public class DequeExample {
    public static void main(String[] args) {
        // Creating a Deque
        Deque<Integer> deque = new LinkedList<>();

        // Adding elements at both ends
        deque.addFirst(10);
        deque.addLast(20);
        deque.offerFirst(5);
        deque.offerLast(25);

        System.out.println("Deque: " + deque); // Output: [5, 10, 20, 25]

        // Retrieving elements
        System.out.println("First Element: " + deque.peekFirst()); // Output: 5
        System.out.println("Last Element: " + deque.peekLast()); // Output: 25

        // Removing elements from both ends
        System.out.println("Removed First: " + deque.pollFirst()); // Output: 5
        System.out.println("Removed Last: " + deque.pollLast()); // Output: 25

        System.out.println("Deque after removal: " + deque); // Output: [10, 20]
    }
}
```

Output:

```
Deque: [5, 10, 20, 25]
First Element: 5
```

```
Last Element: 25
Removed First: 5
Removed Last: 25
Deque after removal: [10, 20]
```

---

5 How Deque Works Internally?

✔ **Structure:**

- **Uses a Doubly Linked List or Resizable Array (ArrayDeque).**
- **Efficient insertions/removals at both ends (`O(1)`).**

✔ **Internal Representation (Doubly Linked List)**

```
NULL ← [5] ⇄ [10] ⇄ [20] ⇄ [25] → NULL
```

✔ **Adding 30 at front (`addFirst(30)`)**

```
NULL ← [30] ⇄ [5] ⇄ [10] ⇄ [20] ⇄ [25] → NULL
```

✔ **Removing last (`pollLast()`)**

```
NULL ← [30] ⇄ [5] ⇄ [10] ⇄ [20] → NULL
```

---

6 ArrayDeque<T> (Faster Alternative to LinkedList)

ArrayDeque is an array-based **double-ended queue**, faster than `LinkedList`.

✔ **Why Use `ArrayDeque` Instead of `LinkedList`?**

- **No overhead of node pointers (faster).**
- **Resizable array grows automatically.**
- **Faster insertion/removal (`O(1)`).**

```
import java.util.ArrayDeque;
import java.util.Deque;

public class ArrayDequeExample {
    public static void main(String[] args) {
        Deque<Integer> arrayDeque = new ArrayDeque<>();

        arrayDeque.addFirst(10);
        arrayDeque.addLast(20);
        arrayDeque.offerFirst(5);
        arrayDeque.offerLast(25);

        System.out.println("ArrayDeque: " + arrayDeque); // Output: [5, 10, 20, 25]
    }
}
```

7️⃣ Performance Analysis of Deque<T>

Operation	LinkedList `O(n)`	ArrayDeque `O(1)`
`addFirst(E e)`	`O(1)`	`O(1)`
`addLast(E e)`	`O(1)`	`O(1)`
`removeFirst()`	`O(1)`	`O(1)`
`removeLast()`	`O(1)`	`O(1)`
`getFirst()`	`O(1)`	`O(1)`
`getLast()`	`O(1)`	`O(1)`

✔ **`ArrayDeque` is the best choice for Deque operations.**

8️⃣ When to Use Deque?

✔ **Use `Deque<T>` when:**
☑ **You need insertion/removal from both ends.**
☑ **You need a fast, resizable queue.**
☑ **You need LIFO & FIFO behavior.**

❌ **Don't use Deque when:**
🚫 **You need indexed access (Use `ArrayList`).**
🚫 **You need thread-safety (Use `ConcurrentLinkedDeque`).**

📌 Summary

✔ **Deque supports adding/removing from both ends.**
✔ **Uses `LinkedList` (Doubly Linked List) or `ArrayDeque` (Resizable Array).**
✔ **Faster than `LinkedList` for queue operations.**
✔ **Best choice: `ArrayDeque` (Faster than `LinkedList`).**
✔ **Operations are `O(1)`, making it efficient.**

🚀 ArrayDeque<T> – Deep Dive & Easy Explanation

1️⃣ What is ArrayDeque<T>?

An **ArrayDeque (Array Double-Ended Queue)** is a **resizable array-based implementation of Deque**, which allows **efficient insertion and removal of elements from both ends**.

✔ **Key Features:**

- **Faster than `LinkedList<T>` for Deque operations.**
- **Dynamic resizing (no fixed capacity like an array).**
- **Does NOT allow `null` elements (unlike `LinkedList`).**
- **Not thread-safe (use `ConcurrentLinkedDeque` for multi-threading).**

📌 **Real-Life Example:**

- **Task Scheduling** – Jobs added at the front or end of the queue.
- **Undo-Redo Feature** – Last action undone (LIFO), or first action redone (FIFO).

---

2 How Does ArrayDeque<T> Work?

📌 **Operations on Both Ends**

1 **Add at Front** → `addFirst(E e)` / `offerFirst(E e)`
2 **Remove from Front** → `removeFirst()` / `pollFirst()`
3 **Add at Rear** → `addLast(E e)` / `offerLast(E e)`
4 **Remove from Rear** → `removeLast()` / `pollLast()`
5 **Peek (Retrieve without removing)** → `peekFirst()` / `peekLast()`

✔ **ArrayDeque as a Queue (FIFO)**

```
Front → [1, 2, 3, 4, 5] → Rear
```

✔ **ArrayDeque as a Stack (LIFO)**

```
Top → [1, 2, 3, 4, 5] (Last In First Out)
```

---

3 Methods of ArrayDeque<T>

Method	Description
`addFirst(E e)`	Adds an element at the front (throws exception if full).
`offerFirst(E e)`	Adds an element at the front (returns `false` if full).
`addLast(E e)`	Adds an element at the rear (throws exception if full).
`offerLast(E e)`	Adds an element at the rear (returns `false` if full).
`removeFirst()`	Removes the first element (throws exception if empty).
`pollFirst()`	Removes the first element (returns `null` if empty).
`removeLast()`	Removes the last element (throws exception if empty).

`pollLast()`	Removes the last element (returns `null` if empty).
`peekFirst()`	Retrieves the first element without removing.
`peekLast()`	Retrieves the last element without removing.

4️⃣ Implementation of ArrayDeque<T>

```
import java.util.ArrayDeque;
import java.util.Deque;

public class ArrayDequeExample {
    public static void main(String[] args) {
        // Creating an ArrayDeque
        Deque<Integer> deque = new ArrayDeque<>();

        // Adding elements at both ends
        deque.addFirst(10);
        deque.addLast(20);
        deque.offerFirst(5);
        deque.offerLast(25);

        System.out.println("ArrayDeque: " + deque); // Output: [5, 10, 20, 25]

        // Retrieving elements
        System.out.println("First Element: " + deque.peekFirst()); // Output: 5
        System.out.println("Last Element: " + deque.peekLast()); // Output: 25

        // Removing elements from both ends
        System.out.println("Removed First: " + deque.pollFirst()); // Output: 5
        System.out.println("Removed Last: " + deque.pollLast()); // Output: 25

        System.out.println("ArrayDeque after removal: " + deque); // Output: [10, 20]
    }
}
```

✔ **Output:**

```
ArrayDeque: [5, 10, 20, 25]
First Element: 5
Last Element: 25
Removed First: 5
Removed Last: 25
ArrayDeque after removal: [10, 20]
```

5️⃣ How ArrayDeque Works Internally?

📌 **Structure:**

- Uses **a dynamically resizable circular array**.
- **Elements wrap around when reaching array capacity.**

- **Insertion/removal from both ends is O(1)** because it doesn't require shifting like `ArrayList`.

✔ **Internal Representation (Circular Array)**

```
[ _, _, 10, 20, 30, _, _, _ ]
     ↑   ↑    ↑
   Front  Elements  Rear
```

✔ **Adding `40` at front (`addFirst(40)`)**

```
[ _, _, 40, 10, 20, 30, _, _ ]
     ↑   ↑    ↑
   Front  Elements  Rear
```

✔ **Removing last (`pollLast()`)**

```
[ _, _, 40, 10, 20, _, _, _ ]
     ↑   ↑    ↑
   Front  Elements  Rear
```

---

6 Performance Analysis of ArrayDeque<T>

Operation	ArrayDeque `O(1)`	LinkedList `O(n)`
`addFirst(E e)`	`O(1)`	`O(1)`
`addLast(E e)`	`O(1)`	`O(1)`
`removeFirst()`	`O(1)`	`O(1)`
`removeLast()`	`O(1)`	`O(1)`
`getFirst()`	`O(1)`	`O(1)`
`getLast()`	`O(1)`	`O(1)`

✔ **ArrayDeque is the best choice for Deque operations.**

---

7 When to Use ArrayDeque?

✔ **Use `ArrayDeque<T>` when:**
☑ **You need fast insertion/removal from both ends.**
☑ **You need a resizable array-backed deque.**
☑ **You don't need thread-safety.**

❌ **Don't use ArrayDeque when:**
🚫 **You need indexed access (Use `ArrayList`).**
🚫 **You need thread-safety (Use `ConcurrentLinkedDeque`).**

📌 Summary

✔ **ArrayDeque supports adding/removing from both ends.**
✔ **Uses a dynamic resizable array (circular buffer).**
✔ **Faster than `LinkedList` for queue operations.**
✔ **Best choice: `ArrayDeque` (Faster than `LinkedList`).**
✔ **Operations are `O(1)`, making it efficient.**

🚀 ConcurrentLinkedQueue<T> – Deep Dive & Easy Explanation

1️⃣ What is ConcurrentLinkedQueue<T>?

A **ConcurrentLinkedQueue** is a **thread-safe, non-blocking, FIFO (First-In-First-Out) queue** that allows multiple threads to access and modify it **without explicit locking**.

✔ **Key Features:**

- ☑ **Thread-safe** (Multiple threads can modify it safely).
- ☑ **Non-blocking** (Uses **CAS (Compare-And-Swap) operations** instead of locks).
- ☑ **FIFO Order** (Elements are processed in order of insertion).
- ☑ **Does NOT allow `null` elements**.
- ☑ **Uses a **linked-list** internally (Each element points to the next).

📌 **Real-Life Example:**

- **Producer-Consumer Pattern** – Multiple producer threads add tasks, while consumer threads process them.
- **Multi-threaded Job Queue** – A system where multiple users submit jobs for processing.

2️⃣ How Does ConcurrentLinkedQueue<T> Work Internally?

📌 **Non-blocking Mechanism**

- Instead of locks (`synchronized` keyword), it uses **atomic operations (CAS - Compare-And-Swap)**.
- This makes it **faster than blocking queues (`BlockingQueue`) in high-concurrency situations**.

✔ **Internal Structure (Linked List Implementation)**

```
Head → [1] → [2] → [3] → Tail
```

- **New elements are added at the tail.**
- **Elements are removed from the head.**
- **Each node contains a reference to the next node.**

3️⃣ Methods of ConcurrentLinkedQueue<T>

Method	Description
`add(E e)`	Adds an element at the tail (throws exception if `null`).
`offer(E e)`	Adds an element at the tail (returns `false` if `null`).
`poll()`	Retrieves and removes the head of the queue (returns `null` if empty).
`peek()`	Retrieves but does not remove the head (returns `null` if empty).
`size()`	Returns the number of elements (not always accurate in multi-threading).
`isEmpty()`	Checks if the queue is empty.
`iterator()`	Returns an iterator over the elements (weakly consistent).

✔ **Important Notes:**

- 🚀 `size()` may not be accurate in multi-threading because other threads might modify the queue simultaneously.
- 🚀 **`poll()` is better than `remove()`** since it doesn't throw an exception if the queue is empty.

4️⃣ Implementation of ConcurrentLinkedQueue<T>

```
import java.util.concurrent.ConcurrentLinkedQueue;

public class ConcurrentLinkedQueueExample {
    public static void main(String[] args) {
        // Creating a ConcurrentLinkedQueue
        ConcurrentLinkedQueue<Integer> queue = new ConcurrentLinkedQueue<>();

        // Adding elements
        queue.add(10);
        queue.offer(20);
        queue.offer(30);

        System.out.println("Queue: " + queue); // Output: [10, 20, 30]

        // Retrieving elements
        System.out.println("Head Element (peek): " + queue.peek()); // Output: 10

        // Removing elements
        System.out.println("Removed Element (poll): " + queue.poll()); // Output: 10
```

```
        System.out.println("Queue after removal: " + queue); // Output: [20, 30]
    }
}
```

✔ **Output:**

```
Queue: [10, 20, 30]
Head Element (peek): 10
Removed Element (poll): 10
Queue after removal: [20, 30]
```

---

5 How ConcurrentLinkedQueue Works Internally?

📌 **Uses Atomic References for Thread-Safety**

- **Each node contains:**
  - **Value**
  - **Reference to next node**
- **CAS (Compare-And-Swap) is used to modify nodes without locks.**

✔ **Example: Adding Elements**

```
Head → [10] → [20] → [30] → Tail
```

✔ **Example: Polling (Removing Head)**

```
Before poll(): Head → [10] → [20] → [30] → Tail
After poll():  Head → [20] → [30] → Tail
```

📌 **Why CAS (Compare-And-Swap)?**
Instead of **synchronized locks**, CAS ensures that:

- **If the reference is still the same (no change by another thread), it updates the value.**
- **If another thread modified it, retry until successful.**
- **This makes operations faster and scalable in multi-threading.**

---

6 Performance Analysis of ConcurrentLinkedQueue<T>

Operation	Complexity `O(n)`	Notes
`add(E e)`	`O(1)`	Fast insert at the tail
`offer(E e)`	`O(1)`	Fast insert at the tail
`poll()`	`O(1)`	Fast removal from head
`peek()`	`O(1)`	Constant time retrieval

`size()`	`O(n)`	Not always accurate

✔ **Why use `ConcurrentLinkedQueue`?**

- **No locking overhead (`synchronized`).**
- **Scales better in high-concurrency environments.**
- **Best for multi-threaded queue processing.**

7️⃣ When to Use ConcurrentLinkedQueue?

✔ **Use `ConcurrentLinkedQueue<T>` when:**
☑ **Multiple threads need to access a queue concurrently.**
☑ **You want a non-blocking, lock-free queue.**
☑ **Elements should be processed in FIFO order.**
☑ **Performance is critical in a multi-threaded environment.**

❌ **Don't use `ConcurrentLinkedQueue<T>` when:**
🚫 **You need blocking operations (use `BlockingQueue<T>` instead).**
🚫 **You require precise `size()` calculation.**

📌 Summary

✔ **ConcurrentLinkedQueue is a non-blocking, thread-safe queue.**
✔ **FIFO order is maintained.**
✔ **Uses CAS (Compare-And-Swap) for efficient updates.**
✔ **Faster than `BlockingQueue` in high-concurrency situations.**
✔ **Best for producer-consumer scenarios in multi-threading.**

📌 Chapter 6: Map Interface (Key-Value Pair Collection) – Deep Dive & Easy Explanation

---

1️⃣ What is `Map<K, V>` Interface?

A **Map** is a data structure that **stores elements in key-value pairs**. Unlike `List` or `Set`, a `Map` does not store individual elements but rather a **mapping of keys to values**.

✔ **Key Features of Map<K, V>:**

- ☑ **Stores data in the form of key-value pairs (`K → V`).**
- ☑ **Each key is unique (no duplicates).**

- ☑ **Values can be duplicated.**
- ☑ **Efficient retrieval based on keys (`O(1)` for HashMap, `O(log n)` for TreeMap).**
- ☑ **Provides various implementations with different characteristics.**

📌 **Real-Life Example of a Map:**
A **dictionary** is a great example of a Map.

- **Key** → Word
- **Value** → Meaning
  Example:

```
{"apple" → "A fruit", "car" → "A vehicle", "java" → "A programming language"}
```

2️⃣ Why Use Map Over List/Set?

Feature	List	Set	Map
Stores Elements	☑ Yes	☑ Yes	🚫 No (Stores Key-Value pairs)
Allows Duplicates	☑ Yes	🚫 No	🚫 No (Keys must be unique)
Ordered	☑ Yes (List is ordered)	❌ No (HashSet is unordered)	☑ Depends on implementation
Fast Lookup	❌ No (O(n) for search)	❌ No (O(n) for search)	☑ Yes (O(1) for HashMap)
Key-Value Mapping	❌ No	❌ No	☑ Yes

📌 **Use Map when:**
✔ You need **fast retrieval of values using keys**.
✔ You want **unique keys with associated values**.
✔ You require **efficient search and updates**.

3️⃣ Map Interface – Important Methods

Method	Description
`put(K key, V value)`	Adds a key-value pair to the map.
`get(K key)`	Retrieves the value associated with the key.
`remove(K key)`	Removes the key-value pair from the map.
`containsKey(K key)`	Checks if the key exists in the map.

`containsValue(V value)`	Checks if the value exists in the map.
`keySet()`	Returns a set of all keys.
`values()`	Returns a collection of all values.
`entrySet()`	Returns a set of all key-value pairs.
`size()`	Returns the number of key-value pairs in the map.
`isEmpty()`	Checks if the map is empty.

---

4 Implementations of Map<K, V>

Implementation	Order	Thread-Safe	Null Keys Allowed?	Performance
`HashMap`	❌ No Order	❌ No	☑ Yes (Only one null key)	🚀 Fast (O(1) for put/get)
`LinkedHashMap`	☑ Insertion Order	❌ No	☑ Yes	🚀 Fast (O(1) for put/get)
`TreeMap`	☑ Sorted Order	❌ No	❌ No	🐢 Slower (O(log n) for put/get)
`Hashtable`	❌ No Order	☑ Yes	❌ No	🐢 Slower (Thread-safe)
`ConcurrentHashMap`	❌ No Order	☑ Yes	❌ No	🚀 Fast (Thread-safe, better than Hashtable)

📌 **Choosing the Right Map:**

- **If you need fast access:** ☑ `HashMap`
- **If you need insertion order:** ☑ `LinkedHashMap`
- **If you need sorted keys:** ☑ `TreeMap`
- **If you need thread safety:** ☑ `ConcurrentHashMap`

---

5 Basic Implementation of Map

```
import java.util.*;

public class MapExample {
    public static void main(String[] args) {
        // Creating a Map
        Map<String, Integer> map = new HashMap<>();

        // Adding key-value pairs
```

```
        map.put("Apple", 10);
        map.put("Banana", 20);
        map.put("Mango", 30);

        // Retrieving a value
        System.out.println("Value for 'Apple': " + map.get("Apple")); // 10

        // Checking key existence
        System.out.println("Contains 'Banana'? " + map.containsKey("Banana")); //
true

        // Removing a key-value pair
        map.remove("Banana");

        // Iterating over the Map
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " → " + entry.getValue());
        }
    }
}
```

✔️ **Output:**

```
Value for 'Apple': 10
Contains 'Banana'? true
Apple → 10
Mango → 30
```

6️⃣ How Map Works Internally?

🔹 Internal Working of `HashMap`

- Uses a **hashing algorithm** to store key-value pairs.
- **Keys are converted into hash codes**, which determine their storage location in an array.
- **Collisions are handled using linked lists (before Java 8) or balanced trees (after Java 8 for large collisions).**

📌 **Example Storage Mechanism (Hash Buckets)**

```
HashMap<K, V> Internal Structure:
Index | Key | Value
------|-----|------
0     | null | null
1     | "Apple" | 10
2     | "Banana" | 20
3     | "Mango" | 30
```

- When `map.put("Apple", 10);` is called:
  - `"Apple"` is **hashed**.
  - It is placed in the corresponding **bucket index**.

- If another key hashes to the same index (collision), it is stored in a **linked list/tree at that index**.

7️⃣ Performance Analysis

Operation	HashMap	LinkedHashMap	TreeMap
`put(K, V)`	**O(1)**	**O(1)**	**O(log n)**
`get(K)`	**O(1)**	**O(1)**	**O(log n)**
`remove(K)`	**O(1)**	**O(1)**	**O(log n)**
`containsKey(K)`	**O(1)**	**O(1)**	**O(log n)**

📌 **Why is `HashMap` faster than `TreeMap`?**

- **`HashMap` uses direct indexing via hashing (`O(1)`)**.
- **`TreeMap` uses a Red-Black tree for sorting (`O(log n)`)**, which is slower.

8️⃣ When to Use Map<K, V>?

✔ **Use `HashMap` when:**
☑ Fast access (`O(1)`).
☑ No need to maintain order.
☑ Allows one `null` key.

✔ **Use `LinkedHashMap` when:**
☑ You need to maintain insertion order.
☑ Performance similar to `HashMap`.

✔ **Use `TreeMap` when:**
☑ You need sorted keys.
☑ `O(log n)` operations are acceptable.

✔ **Use `ConcurrentHashMap` when:**
☑ You need a thread-safe alternative.
☑ Better performance than `Hashtable`.

📌 Summary

✔ **Map stores key-value pairs where keys are unique.**
✔ **Different implementations serve different use cases (`HashMap`, `TreeMap`, etc.).**
✔ **Performance varies based on use case (`O(1)` vs `O(log n)`).**
✔ **Choosing the right `Map` depends on order, thread safety, and lookup speed.**

📌 Deep Dive into `HashMap<K, V>` (Easy & Detailed Explanation)

1️⃣ What is `HashMap<K, V>`?

A `HashMap<K, V>` is a **key-value-based** data structure in Java that stores unique keys and their associated values. It is **unordered** and allows for **fast retrieval** of values using keys.

✔ Key Features of `HashMap<K, V>`

- ☑ **Stores elements in key-value pairs (`K → V`).**
- ☑ **Keys must be unique, but values can be duplicate.**
- ☑ **Allows one `null` key and multiple `null` values.**
- ☑ **Unordered (does not maintain insertion order).**
- ☑ **Uses hashing to store data for fast access (`O(1)` time complexity).**
- ☑ **Not thread-safe (use `ConcurrentHashMap` for multi-threading).**

📌 Example:

Imagine a **phonebook** where names (keys) are mapped to phone numbers (values).

```
{ "Alice" → 9876543210, "Bob" → 8765432109, "Charlie" → 7654321098 }
```

Here, names are **keys** (unique) and phone numbers are **values**.

2️⃣ Why Use `HashMap` Instead of List or Array?

Feature	Array	List	HashMap
Stores Elements	☑ Yes	☑ Yes	🚫 No (Stores Key-Value)
Allows Duplicates	☑ Yes	☑ Yes	🚫 No (Keys are unique)
Ordered	☑ Yes (Array order)	☑ Yes (List order)	❌ No (Unordered)
Fast Lookup	❌ No (`O(n)`)	❌ No (`O(n)`)	☑ Yes (`O(1)`)
Key-Value Mapping	❌ No	❌ No	☑ Yes

📌 Use `HashMap` when:

✔ **You need fast lookups, insertions, and deletions (`O(1)`).**
✔ **You want a unique key for each value.**
✔ **You don't care about ordering.**

3️⃣ How `HashMap` Works Internally?

`HashMap` uses **hashing** to store key-value pairs. It converts a key into a **hashcode** and determines its storage location (bucket) in an **array of nodes**.

🔹 Steps of `put(K, V)` Method:

1. **Compute the Hash Code**
   - Converts the key into a **hashcode** (unique number).
   - Example: `"Apple".hashCode()` → `2536478`
2. **Find the Bucket (Index Calculation)**
   - Uses `hash % capacity` formula to find a storage index.
   - Example: `2536478 % 16 = 6` → Stored in bucket 6.
3. **Insert the Key-Value Pair**
   - If the bucket is empty, store the pair.
   - If a **collision** occurs (same bucket), use **Linked List** or **Balanced Tree** (from Java 8) to store multiple entries.

🔹 Steps of `get(K)` Method:

4. **Compute the Hash Code of the Key.**
5. **Find the Bucket Using Hashing Formula.**
6. **Search for the Key in That Bucket.**
7. **If Found, Return the Value; Otherwise, Return `null`.**

📌 **Visual Representation of `HashMap` Storage:**

```
Bucket | Key     | Value
```

---

0 | null | null
1 | "Bob" | 8765432109
2 | null | null
3 | "Alice" | 9876543210
4 | null | null
5 | "Charlie" | 7654321098

```
 🔹 **Collision Handling:**
If two keys produce the **same hash**, `HashMap` uses **Linked List or Balanced
Tree** at that bucket index.
```

---

4 `HashMap` Constructors

Constructor	Description
`HashMap()`	Creates an empty HashMap with default size (16).
`HashMap(int capacity)`	Creates HashMap with given capacity.
`HashMap(int capacity, float loadFactor)`	Creates HashMap with capacity and load factor (default = 0.75).
`HashMap(Map<K, V> m)`	Creates HashMap with elements from another map.

---

5 Important Methods of `HashMap`

Method	Description
`put(K key, V value)`	Adds a key-value pair to the map.
`get(K key)`	Retrieves the value for a key.
`remove(K key)`	Removes a key-value pair.
`containsKey(K key)`	Checks if a key exists.
`containsValue(V value)`	Checks if a value exists.
`keySet()`	Returns all keys as a `Set`.
`values()`	Returns all values as a `Collection`.
`entrySet()`	Returns all key-value pairs as a `Set`.
`size()`	Returns the number of key-value pairs.
`isEmpty()`	Checks if the map is empty.

---

6 `HashMap` Example Code

```
import java.util.*;

public class HashMapExample {
    public static void main(String[] args) {
        // Creating a HashMap
        HashMap<String, Integer> map = new HashMap<>();

        // Adding elements (put method)
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 28);

        // Retrieving values (get method)
        System.out.println("Age of Alice: " + map.get("Alice")); // 25
```

```
        // Checking if a key exists
        System.out.println("Contains 'Bob'? " + map.containsKey("Bob")); // true

        // Removing a key-value pair
        map.remove("Charlie");

        // Iterating through the HashMap
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " → " + entry.getValue());
        }
    }
}
```

✔ **Output:**

```
Age of Alice: 25
Contains 'Bob'? true
Alice → 25
Bob → 30
```

---

7 Performance Analysis

Operation	Time Complexity
`put(K, V)`	**O(1)** (Best case) / **O(n)** (Worst case, collisions)
`get(K)`	**O(1)** (Best case) / **O(n)** (Worst case)
`remove(K)`	**O(1)** (Best case) / **O(n)** (Worst case)
`containsKey(K)`	**O(1)**
`containsValue(V)`	**O(n)**

📌 **Why is `O(1)` lookup possible?**
Because `HashMap` directly accesses the **bucket index** using **hashing**.

📌 **When does `O(n)` happen?**
When **many keys have the same hashcode** (collisions), forcing a **linked list traversal**.

---

8 When to Use `HashMap<K, V>`?

✔ **When you need fast lookup (`O(1)`).**
✔ **When key order doesn't matter.**
✔ **When you want one `null` key and multiple `null` values.**

❌ **Avoid `HashMap` if:**

- You need **ordered keys** (`Use LinkedHashMap`).
- You need **sorted keys** (`Use TreeMap`).

- You need **thread safety** (`Use ConcurrentHashMap`).

📌 Summary

✔ **`HashMap<K, V>` stores key-value pairs using hashing.**
✔ **Keys must be unique, but values can be duplicated.**
✔ **Offers `O(1)` lookup time but may degrade to `O(n)` in case of collisions.**
✔ **Unordered (does not maintain insertion order).**
✔ **Used when fast retrieval of data is needed.**

📌 Deep Dive into `LinkedHashMap<K, V>` (Easy & Detailed Explanation)

1️⃣ What is `LinkedHashMap<K, V>`?

A `LinkedHashMap<K, V>` is a **key-value-based** data structure in Java that extends `HashMap<K, V>`, but **maintains insertion order**.

✔ **Key Features of `LinkedHashMap<K, V>`**

- ☑ **Stores elements in key-value pairs (`K → V`).**
- ☑ **Maintains insertion order (unlike `HashMap`).**
- ☑ **Uses a doubly linked list along with a hash table.**
- ☑ **Fast lookup and retrieval (`O(1)`).**
- ☑ **Allows one `null` key and multiple `null` values.**
- ☑ **Not thread-safe (use `Collections.synchronizedMap()` for thread safety).**

📌 **Example:**
Imagine an **attendance register** where names (keys) are mapped to attendance status (values).

```
{ "Alice" → Present, "Bob" → Absent, "Charlie" → Present }
```

Here, **insertion order is preserved**.

2️⃣ Difference Between `HashMap` and `LinkedHashMap`

Feature	HashMap	LinkedHashMap
Ordering	❌ No (Unordered)	☑ Yes (Insertion Order)

Performance	☑ Fast (O(1))	☑ Slightly Slower (O(1))
Memory Usage	☑ Less	❌ More (Extra Linked List)
Allows null Key	☑ Yes	☑ Yes
Thread-Safe	❌ No	❌ No
Use Case	Fast access, no order needed	Fast access, order matters

📌 **Use `LinkedHashMap` when:**
✔ **You need fast lookups but also maintain order.**
✔ **You need predictable iteration order.**
✔ **You want a cache with access-ordering (LRU Cache).**

3️⃣ How `LinkedHashMap` Works Internally?

`LinkedHashMap` is built on top of `HashMap`, but it **maintains insertion order** using a **doubly linked list**.

🔹 How Entries Are Stored?

- It maintains a **hash table** (like `HashMap`) for fast access.
- It also has a **doubly linked list** that keeps track of order.

📌 **Example:** Adding `"Alice" → 25`, `"Bob" → 30`, `"Charlie" → 28`

```
Hash Table (Buckets) → Fast Lookup
Bucket | Key     | Value  | Next (Linked List)
```

---

0 | null | null | null
1 | "Alice" | 25 | Bob → Charlie → null (Doubly Linked List)
2 | "Bob" | 30 | Charlie → null
3 | "Charlie" | 28 | null

```
✔ **Doubly Linked List ensures order is maintained!**
```

4️⃣ `LinkedHashMap` Constructors

Constructor	Description
`LinkedHashMap()`	Creates an empty LinkedHashMap with default size (16).
`LinkedHashMap(int capacity)`	Creates LinkedHashMap with given capacity.
`LinkedHashMap(int capacity, float loadFactor)`	Creates LinkedHashMap with capacity and

	load factor (default = 0.75).
`LinkedHashMap(int capacity, float loadFactor, boolean accessOrder)`	Creates LinkedHashMap with access-order (LRU cache).

---

5 Important Methods of `LinkedHashMap`

Method	Description
`put(K key, V value)`	Adds a key-value pair to the map.
`get(K key)`	Retrieves the value for a key.
`remove(K key)`	Removes a key-value pair.
`containsKey(K key)`	Checks if a key exists.
`containsValue(V value)`	Checks if a value exists.
`keySet()`	Returns all keys as a `Set`.
`values()`	Returns all values as a `Collection`.
`entrySet()`	Returns all key-value pairs as a `Set`.
`size()`	Returns the number of key-value pairs.
`isEmpty()`	Checks if the map is empty.

---

6 `LinkedHashMap` Example Code

```
import java.util.*;

public class LinkedHashMapExample {
    public static void main(String[] args) {
        // Creating a LinkedHashMap
        LinkedHashMap<String, Integer> map = new LinkedHashMap<>();

        // Adding elements (put method)
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 28);

        // Retrieving values (get method)
        System.out.println("Age of Alice: " + map.get("Alice")); // 25

        // Checking if a key exists
        System.out.println("Contains 'Bob'? " + map.containsKey("Bob")); // true

        // Removing a key-value pair
        map.remove("Charlie");
```

```
        // Iterating through the LinkedHashMap
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " → " + entry.getValue());
        }
    }
}
```

✔ **Output:**

```
Age of Alice: 25
Contains 'Bob'? true
Alice → 25
Bob → 30
```

✔ **Insertion order is maintained!**

---

7 Special Feature: Access Order (LRU Cache)

By default, `LinkedHashMap` maintains **insertion order**, but we can use **access order** for caching (Least Recently Used - LRU Cache).

📌 **LRU Cache Example:**

```
import java.util.*;

class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private final int capacity;

    public LRUCache(int capacity) {
        super(capacity, 0.75f, true); // Access Order = true
        this.capacity = capacity;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > capacity; // Remove oldest entry when full
    }
}

public class LRUExample {
    public static void main(String[] args) {
        LRUCache<Integer, String> cache = new LRUCache<>(3);

        cache.put(1, "A");
        cache.put(2, "B");
        cache.put(3, "C");

        // Access key 1, making it most recently used
        cache.get(1);

        // Adding new key, 2 should be removed (LRU policy)
        cache.put(4, "D");

        System.out.println(cache.keySet()); // Output: [3, 1, 4]
```

```
    }
}
```

✔ **LRU Cache removes least used items!**

---

8 Performance Analysis

Operation	Time Complexity
`put(K, V)`	**O(1)**
`get(K)`	**O(1)**
`remove(K)`	**O(1)**
`containsKey(K)`	**O(1)**
`containsValue(V)`	**O(n)**

📌 **Why does `O(1)` lookup happen?**
Because `LinkedHashMap` uses **hashing** like `HashMap`.

📌 **When does `O(n)` happen?**
When searching for a **specific value**, as all values must be checked.

---

9 When to Use `LinkedHashMap<K, V>`?

✔ **When you need insertion order.**
✔ **When you need fast lookup like `HashMap`.**
✔ **When you need an LRU cache.**

❌ **Avoid `LinkedHashMap` if:**

- You don't care about ordering (`Use HashMap`).
- You need sorted keys (`Use TreeMap`).
- You need thread safety (`Use ConcurrentHashMap`).

📌 Summary

✔ **`LinkedHashMap<K, V>` maintains insertion order.**
✔ **Fast lookup with `O(1)` complexity.**
✔ **Can be used as an LRU cache with access order.**
✔ **Uses extra memory for maintaining order.**

📌 Deep Dive into `TreeMap<K, V>` (Easy & Detailed Explanation)

1️⃣ What is `TreeMap<K, V>`?

A `TreeMap<K, V>` is a key-value collection in Java that **stores keys in sorted order** (ascending by default).

✔ **Key Features of `TreeMap<K, V>`**

- ☑ **Stores key-value pairs (`K → V`).**
- ☑ **Maintains keys in sorted order (Natural or Custom Comparator).**
- ☑ **Implements `NavigableMap<K, V>` and `SortedMap<K, V>`.**
- ☑ **Uses a Red-Black Tree for self-balancing.**
- ☑ **Search, Insert, Delete in `O(log n)`.**
- ☑ **Does NOT allow `null` keys (unlike `HashMap`).**
- ☑ **Thread-Unsafe (Use `Collections.synchronizedMap()` for thread safety).**

📌 **Example Use Case:**
Imagine a **student ranking system** where we store students' scores and want to retrieve them in **sorted order** automatically.

2️⃣ Difference Between `HashMap`, `LinkedHashMap`, and `TreeMap`

Feature	HashMap	LinkedHashMap	TreeMap
Ordering	❌ No order	☑ Insertion Order	☑ Sorted Order (Ascending by default)
Performance (Put, Get, Remove)	☑ `O(1)`	☑ `O(1)`	❌ `O(log n)`
Implementation	**Hash Table**	**Hash Table + Linked List**	**Red-Black Tree**
Allows `null` Key	☑ Yes	☑ Yes	❌ No
Memory Usage	☑ Low	❌ High (Linked List)	❌ High (Tree Structure)
Use Case	Fast lookup	Order-preserving	Sorted Data

📌 **Use `TreeMap<K, V>` when:**
✔ **You need keys to be sorted automatically.**
✔ **You need efficient range queries (`subMap`, `headMap`, `tailMap`).**
✔ **You need to maintain a priority-based ordering.**

3️⃣ How `TreeMap` Works Internally?

`TreeMap<K, V>` **uses a Red-Black Tree** for self-balancing.

🔹 How Data is Stored?

- Unlike `HashMap`, `TreeMap` **stores elements in a sorted tree structure**.
- Each node contains:
  - **Key (`K`)**
  - **Value (`V`)**
  - **Left Child (Smaller Keys)**
  - **Right Child (Larger Keys)**
- The tree is **balanced** using **Red-Black Tree** properties.

📌 **Example:**

```
TreeMap<Integer, String> map = new TreeMap<>();
map.put(50, "Alice");
map.put(30, "Bob");
map.put(70, "Charlie");
map.put(20, "David");
map.put(40, "Eve");
```

✔️ **Internally, the Red-Black Tree will arrange them as:**

```
        50
       /   \
     30    70
    /  \
  20   40
```

☑️ **Keys are always sorted in ascending order!**

4️⃣ `TreeMap` Constructors

Constructor	Description
`TreeMap()`	Creates an empty `TreeMap` with natural ordering.
`TreeMap(Comparator<? super K> comparator)`	Creates `TreeMap` with a custom sorting order.
`TreeMap(Map<? extends K, ? extends V> map)`	Creates a `TreeMap` with the same elements as an existing map.
`TreeMap(SortedMap<K, ? extends V> sortedMap)`	Creates a `TreeMap` from another sorted map.

5 Important Methods of `TreeMap<K, V>`

Method	Description
`put(K key, V value)`	Adds a key-value pair to the map.
`get(K key)`	Retrieves the value for a key.
`remove(K key)`	Removes a key-value pair.
`containsKey(K key)`	Checks if a key exists.
`containsValue(V value)`	Checks if a value exists.
`size()`	Returns the number of key-value pairs.
`isEmpty()`	Checks if the map is empty.
`keySet()`	Returns all keys as a `Set`.
`values()`	Returns all values as a `Collection`.
`entrySet()`	Returns all key-value pairs as a `Set`.
`firstKey()`	Returns the smallest key.
`lastKey()`	Returns the largest key.
`higherKey(K key)`	Returns the smallest key greater than `key`.
`lowerKey(K key)`	Returns the largest key less than `key`.
`subMap(K fromKey, K toKey)`	Returns a portion of the map between `fromKey` and `toKey`.

6 `TreeMap` Example Code

```
import java.util.*;

public class TreeMapExample {
    public static void main(String[] args) {
        // Creating a TreeMap
        TreeMap<Integer, String> map = new TreeMap<>();

        // Adding elements (put method)
        map.put(50, "Alice");
        map.put(30, "Bob");
        map.put(70, "Charlie");
        map.put(20, "David");
        map.put(40, "Eve");

        // Retrieving values (get method)
        System.out.println("Value of 50: " + map.get(50)); // Alice
```

```
        // Getting first and last key
        System.out.println("Smallest key: " + map.firstKey()); // 20
        System.out.println("Largest key: " + map.lastKey()); // 70

        // Iterating through TreeMap (Sorted Order)
        for (Map.Entry<Integer, String> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " → " + entry.getValue());
        }
    }
}
```

✔ **Output:**

```
Value of 50: Alice
Smallest key: 20
Largest key: 70
20 → David
30 → Bob
40 → Eve
50 → Alice
70 → Charlie
```

✔ **Keys are sorted automatically!**

---

7 Custom Sorting with `TreeMap`

We can pass a **custom comparator** to define our own sorting order.

📌 **Example: Sorting in Descending Order**

```
import java.util.*;

public class TreeMapDescending {
    public static void main(String[] args) {
        // Custom comparator for descending order
        TreeMap<Integer, String> map = new TreeMap<>(Comparator.reverseOrder());

        map.put(50, "Alice");
        map.put(30, "Bob");
        map.put(70, "Charlie");

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

✔ **Output:**

```
{70=Charlie, 50=Alice, 30=Bob}
```

✔ **Sorted in descending order!**

8️⃣ Performance Analysis

Operation	Time Complexity
`put(K, V)`	**O(log n)**
`get(K)`	**O(log n)**
`remove(K)`	**O(log n)**
`containsKey(K)`	**O(log n)**
`containsValue(V)`	**O(n)**

📌 **Why `O(log n)`?**
Because `TreeMap` is based on **Red-Black Tree**, a balanced tree structure.

9️⃣ When to Use `TreeMap<K, V>`?

✔ **When you need keys to be sorted automatically.**
✔ **When you need efficient range queries (`subMap`, `headMap`, `tailMap`).**
✔ **When maintaining order is crucial.**

❌ **Avoid `TreeMap` if:**

- You don't need sorted order (`Use HashMap`).
- Performance is a priority (`TreeMap` is slower than `HashMap`).
- You need thread safety (`Use ConcurrentSkipListMap`).

📌 Summary

✔ **`TreeMap<K, V>` maintains sorted order.**
✔ **Uses Red-Black Tree (`O(log n)` performance).**
✔ **Great for range queries (`subMap`, `headMap`, etc.).**
✔ **No `null` keys allowed!**

📌 Deep Dive into `Hashtable<K, V>` (Easy & Detailed Explanation)

1️⃣ What is `Hashtable<K, V>`?

A `Hashtable<K, V>` is a key-value data structure in Java that is **thread-safe** and does **not allow `null` keys or values**.

✔ **Key Features of `Hashtable<K, V>`**

- ☑ **Stores key-value pairs (`K → V`).**
- ☑ **Thread-Safe (Synchronized Methods).**
- ☑ **No `null` keys or values allowed.**
- ☑ **Uses a Hash Table for fast lookups (`O(1)` in most cases).**
- ☑ **Implemented using `synchronized` methods, making it slower than `HashMap`.**
- ☑ **Legacy class (Introduced in Java 1.0, before `HashMap`).**

📌 **Example Use Case:**
Imagine a **multi-threaded banking system** where we store customer account balances and need thread safety to avoid data corruption.

---

2 Difference Between `Hashtable`, `HashMap`, and `ConcurrentHashMap`

Feature	Hashtable	HashMap	ConcurrentHashMap
Thread-Safe?	☑ Yes	❌ No	☑ Yes (Better Performance)
Allows `null` Key?	❌ No	☑ Yes	❌ No
Allows `null` Values?	❌ No	☑ Yes	❌ No
Performance (Put, Get, Remove)	❌ Slow (Synchronized)	☑ Fast (`O(1)`)	☑ Fast (Lock-Free Reads)
Iteration	**Slow** (Locks Entire Table)	**Fast** (Uses `fail-fast` iterator)	**Fast** (Lock-Free Segments)
Usage	**Legacy, Avoid Using**	**Best for Single-Threaded Apps**	**Best for Multi-Threaded Apps**

📌 **Use `Hashtable<K, V>` when:**
✔ **You need thread safety in older Java versions.**
✔ **You are maintaining legacy Java code.**
✔ **You cannot use `ConcurrentHashMap` (for some reason).**

❌ **Avoid `Hashtable<K, V>` if:**

- You don't need thread safety (`Use HashMap`).
- You need better performance (`Use ConcurrentHashMap`).

---

3 How `Hashtable` Works Internally?

`Hashtable<K, V>` **uses a hash table with synchronization** to store key-value pairs.

🔹 How Data is Stored?

- Similar to `HashMap`, `Hashtable` **uses an array of "buckets"**.
- Each bucket stores **key-value pairs** using **linked lists** (to handle collisions).
- The **hash function** determines the bucket index for a key.
- If two keys have the same hash, they are stored in the same bucket **(chaining method)**.

📌 **Example:**

```
Hashtable<Integer, String> table = new Hashtable<>();
table.put(50, "Alice");
table.put(30, "Bob");
table.put(70, "Charlie");
table.put(20, "David");
table.put(40, "Eve");
```

✔ **Internally, the Hashtable might look like this:**

```
Bucket 0: (50 → Alice)
Bucket 1: (30 → Bob)
Bucket 2: (70 → Charlie)
Bucket 3: (20 → David)
Bucket 4: (40 → Eve)
```

☑ **Data is stored in hash buckets, ensuring fast lookup.**

4️⃣ `Hashtable` Constructors

Constructor	Description
`Hashtable()`	Creates an empty `Hashtable` with default capacity.
`Hashtable(int initialCapacity)`	Creates a `Hashtable` with a specific capacity.
`Hashtable(int initialCapacity, float loadFactor)`	Creates a `Hashtable` with capacity and load factor.
`Hashtable(Map<? extends K, ? extends V> map)`	Creates a `Hashtable` from another map.

5️⃣ Important Methods of `Hashtable<K, V>`

Method	Description
`put(K key, V value)`	Adds a key-value pair to the table.
`get(K key)`	Retrieves the value for a key.
`remove(K key)`	Removes a key-value pair.

`containsKey(K key)`	Checks if a key exists.
`containsValue(V value)`	Checks if a value exists.
`size()`	Returns the number of key-value pairs.
`isEmpty()`	Checks if the table is empty.
`keySet()`	Returns all keys as a `Set`.
`values()`	Returns all values as a `Collection`.
`entrySet()`	Returns all key-value pairs as a `Set`.
`clone()`	Creates a copy of the `Hashtable`.
`clear()`	Removes all elements from the `Hashtable`.

---

6 `Hashtable` Example Code

```
import java.util.*;

public class HashtableExample {
    public static void main(String[] args) {
        // Creating a Hashtable
        Hashtable<Integer, String> table = new Hashtable<>();

        // Adding elements (put method)
        table.put(50, "Alice");
        table.put(30, "Bob");
        table.put(70, "Charlie");
        table.put(20, "David");
        table.put(40, "Eve");

        // Retrieving values (get method)
        System.out.println("Value of 50: " + table.get(50)); // Alice

        // Checking if a key exists
        System.out.println("Contains key 30? " + table.containsKey(30)); // true

        // Iterating through Hashtable
        for (Map.Entry<Integer, String> entry : table.entrySet()) {
            System.out.println(entry.getKey() + " → " + entry.getValue());
        }
    }
}
```

✔ **Output:**

```
Value of 50: Alice
Contains key 30? true
20 → David
30 → Bob
40 → Eve
```

```
50 → Alice
70 → Charlie
```

✔ **Keys are stored in a hash table, not sorted!**

---

7 Thread-Safety in `Hashtable`

Since `Hashtable` methods are **synchronized**, only **one thread** can access them at a time.

📌 **Example (Multiple Threads Using `Hashtable`)**

```
import java.util.*;

public class HashtableThreadExample {
    public static void main(String[] args) {
        Hashtable<Integer, String> table = new Hashtable<>();

        // Thread 1 (Adding Data)
        Thread t1 = new Thread(() -> {
            table.put(1, "A");
            table.put(2, "B");
            table.put(3, "C");
        });

        // Thread 2 (Reading Data)
        Thread t2 = new Thread(() -> {
            System.out.println(table.get(1));
            System.out.println(table.get(2));
        });

        t1.start();
        t2.start();
    }
}
```

✔ `Hashtable` **prevents data corruption by synchronizing access.**

---

8 Performance Analysis

Operation	Time Complexity
`put(K, V)`	**O(1)** (best case), **O(n)** (worst case - collisions)
`get(K)`	**O(1)** (best case), **O(n)** (worst case - collisions)
`remove(K)`	**O(1)** (best case), **O(n)** (worst case - collisions)

📌 **Why Slower than `HashMap`?**

- **Every method is synchronized** → **More overhead** in multi-threaded environments.
- **Collisions can degrade performance** to `O(n)` in worst cases.

9️⃣ When to Use `Hashtable<K, V>`?

✔ **When you need thread safety in older Java versions.**
✔ **When working with legacy applications.**
✔ **When you don't need `null` keys/values.**

❌ **Avoid `Hashtable<K, V>` if:**

- You need better performance (`Use ConcurrentHashMap`).
- You need null keys or values (`Use HashMap`).

📌 Summary

✔ **`Hashtable<K, V>` is thread-safe (`synchronized`).**
✔ **Uses a hash table (`O(1)` lookups in most cases).**
✔ **No `null` keys or values allowed.**
✔ **Slower than `HashMap` due to synchronization.**

📌 Deep Dive into `ConcurrentHashMap<K, V>` (Easy & Detailed Explanation)

1️⃣ What is `ConcurrentHashMap<K, V>`?

A **`ConcurrentHashMap<K, V>`** is an advanced **thread-safe** version of `HashMap` that allows **multiple threads** to read and write without blocking the entire map.

✔ **Key Features of `ConcurrentHashMap<K, V>`**

- ☑ **Thread-Safe without using `synchronized` on the entire map.**
- ☑ **Faster than `Hashtable` (Uses Locking at Segment Level).**
- ☑ **No `null` keys or values allowed.**
- ☑ **Uses multiple "segments" (buckets) to allow concurrent operations.**
- ☑ **Best suited for multi-threaded environments.**
- ☑ **Improved performance over `Hashtable`.**

📌 **Example Use Case:**
Imagine a **real-time stock market system** where thousands of users update stock prices simultaneously. `ConcurrentHashMap` ensures efficient, thread-safe updates without performance bottlenecks.

2️⃣ Difference Between `HashMap`, `Hashtable`, and `ConcurrentHashMap`

Feature	`HashMap`	`Hashtable`	`ConcurrentHashMap`
Thread-Safe?	❌ No	☑ Yes (Slow)	☑ Yes (Faster)
Allows `null` Keys?	☑ Yes	❌ No	❌ No
Allows `null` Values?	☑ Yes	❌ No	❌ No
Performance (Put, Get, Remove)	☑ Fast (`O(1)`)	❌ Slow (Locks Entire Table)	☑ Fast (Segmented Locks)
Usage	Best for Single-Threaded Apps	Legacy (Avoid Using)	Best for Multi-Threaded Apps

📌 **When to Use `ConcurrentHashMap`?**
✔ **When you need high-performance thread-safe operations.**
✔ **When multiple threads need to read and write simultaneously.**
✔ **When `HashMap` is not safe but `Hashtable` is too slow.**

3️⃣ How `ConcurrentHashMap` Works Internally?

Instead of locking the entire map (like `Hashtable`), `ConcurrentHashMap` **divides the map into segments (buckets)** and locks only the affected segment during updates.

🔹 How Data is Stored?

- **Uses a bucket-based structure**, similar to `HashMap`.
- **Each bucket (segment) is locked separately**, allowing multiple threads to access different buckets concurrently.
- Uses **a special locking mechanism (CAS - Compare-And-Swap)** to ensure consistency without full table locking.

📌 **Example:**

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
map.put(1, "Alice");
map.put(2, "Bob");
map.put(3, "Charlie");
```

✔ **Internally, the map might look like this:**

```
Bucket 0: (1 → Alice)
Bucket 1: (2 → Bob)
Bucket 2: (3 → Charlie)
```

☑ **Each bucket (segment) is locked individually, allowing faster access.**

---

4 `ConcurrentHashMap` Constructors

Constructor	Description
`ConcurrentHashMap()`	Creates an empty map with default capacity.
`ConcurrentHashMap(int initialCapacity)`	Creates a map with a specific initial capacity.
`ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)`	Creates a map with defined concurrency level.
`ConcurrentHashMap(Map<? extends K, ? extends V> map)`	Creates a `ConcurrentHashMap` from another map.

---

5 Important Methods of `ConcurrentHashMap<K, V>`

Method	Description
`put(K key, V value)`	Adds a key-value pair to the map.
`get(K key)`	Retrieves the value for a key.
`remove(K key)`	Removes a key-value pair.
`containsKey(K key)`	Checks if a key exists.
`containsValue(V value)`	Checks if a value exists.
`size()`	Returns the number of key-value pairs.
`isEmpty()`	Checks if the map is empty.
`keySet()`	Returns all keys as a `Set`.
`values()`	Returns all values as a `Collection`.
`entrySet()`	Returns all key-value pairs as a `Set`.
`replace(K key, V oldValue, V newValue)`	Replaces a value if the current value matches.
`computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)`	Computes a value if the key is absent.
`computeIfPresent(K key, BiFunction<? super K, ? super V,`	Computes a new value if the

`? extends V> remappingFunction)`	key is present.

6️⃣ `ConcurrentHashMap` Example Code

```
import java.util.concurrent.*;

public class ConcurrentHashMapExample {
    public static void main(String[] args) {
        // Creating a ConcurrentHashMap
        ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();

        // Adding elements (put method)
        map.put(1, "Alice");
        map.put(2, "Bob");
        map.put(3, "Charlie");

        // Retrieving values (get method)
        System.out.println("Value of 1: " + map.get(1)); // Alice

        // Checking if a key exists
        System.out.println("Contains key 2? " + map.containsKey(2)); // true

        // Iterating through ConcurrentHashMap
        for (ConcurrentHashMap.Entry<Integer, String> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " → " + entry.getValue());
        }
    }
}
```

✔️ **Output:**

```
Value of 1: Alice
Contains key 2? true
1 → Alice
2 → Bob
3 → Charlie
```

☑️ **Supports fast, thread-safe operations without full map locking.**

7️⃣ Multi-Threading with `ConcurrentHashMap`

Unlike `Hashtable`, `ConcurrentHashMap` **does not block the entire map for every operation**. Multiple threads can update different segments at the same time.

📌 **Example (Multiple Threads Using `ConcurrentHashMap`)**

```
import java.util.concurrent.*;

public class ConcurrentHashMapThreadExample {
    public static void main(String[] args) {
        ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
```

```
        // Thread 1 (Adding Data)
        Thread t1 = new Thread(() -> {
            map.put(1, "A");
            map.put(2, "B");
            map.put(3, "C");
        });

        // Thread 2 (Reading Data)
        Thread t2 = new Thread(() -> {
            System.out.println(map.get(1));
            System.out.println(map.get(2));
        });

        t1.start();
        t2.start();
    }
}
```

✔ **Thread-safe operations without full map locking.**

---

8 Performance Analysis

Operation	Time Complexity
`put(K, V)`	**O(1)** (best case), **O(n)** (worst case - collisions)
`get(K)`	**O(1)** (best case), **O(n)** (worst case - collisions)
`remove(K)`	**O(1)** (best case), **O(n)** (worst case - collisions)

📌 **Why Faster than `Hashtable`?**

- **Does not lock the entire map.**
- **Uses fine-grained segment locks.**
- **Supports concurrent reads and writes.**

---

9 When to Use `ConcurrentHashMap<K, V>`?

✔ **When you need high-performance thread-safe operations.**
✔ **When multiple threads need to read and write concurrently.**
✔ **When `HashMap` is not safe but `Hashtable` is too slow.**

❌ **Avoid `ConcurrentHashMap<K, V>` if:**

- You need `null` keys or values (`Use HashMap`).
- You need strict synchronization (`Use Hashtable`).

📌 Summary

✔ `ConcurrentHashMap<K, V>` **is thread-safe (**`Segmented Locking`**).**
✔ **Uses hash buckets (**`O(1)` **lookups in most cases).**
✔ **No** `null` **keys or values allowed.**
✔ **Faster than** `Hashtable` **due to better concurrency.**

📌 Deep Dive into `WeakHashMap<K, V>` (Easy & Detailed Explanation)

1️⃣ What is `WeakHashMap<K, V>`?

A `WeakHashMap<K, V>` is a special type of `Map` that **automatically removes entries** when their keys are no longer **strongly referenced** anywhere in the application.

✔ Key Features of `WeakHashMap<K, V>`

- ☑ **Uses weak references for keys**, meaning **entries can be garbage collected (GC) automatically**.
- ☑ **Prevents memory leaks** by allowing garbage collection to remove unused keys.
- ☑ **Best suited for caching mechanisms** where keys can be discarded when not in use.
- ☑ **Works similarly to** `HashMap`**, but with weak keys.**

📌 **Example Use Case:**
Imagine a **cache system** that stores temporary data. If an object (key) is no longer needed in memory, it should be automatically removed from the cache. `WeakHashMap` helps in this case by removing the entry when the key is no longer referenced elsewhere.

2️⃣ Difference Between `HashMap`, `Hashtable`, `ConcurrentHashMap`, and `WeakHashMap`

Feature	`HashMap`	`Hashtable`	`ConcurrentHashMap`	`WeakHashMap`
Thread-Safe?	❌ No	☑ Yes (Synchronized)	☑ Yes (Segmented Locking)	❌ No
Garbage Collection Aware?	❌ No	❌ No	❌ No	☑ Yes (Removes Unused Keys)
Allows `null` **Keys?**	☑ Yes	❌ No	❌ No	☑ Yes
Allows `null` **Values?**	☑ Yes	❌ No	❌ No	☑ Yes

Performance	☑ Fast (O(1))	❌ Slow (Full Locking)	☑ Fast (Concurrent Access)	☑ Fast (O(1))
When to Use?	General Purpose Map	Thread-Safe, but slow	High-Performance Thread-Safe Map	Auto-removing keys (cache, temporary data)

📌 **When to Use `WeakHashMap`?**

✔ **For caching mechanisms** where objects should be automatically removed when no longer needed.

✔ **When preventing memory leaks** by ensuring unused keys do not remain in memory.

✔ **When you need a `Map<K, V>` but want automatic cleanup of unused keys.**

3️⃣ How `WeakHashMap` Works Internally?

Instead of using **strong references**, `WeakHashMap` **uses weak references for its keys**.

🔹 What is a Weak Reference?

- Normally, Java objects are referenced **strongly**—they remain in memory until no reference exists.
- **Weak references** allow objects to be garbage collected even when still in the `WeakHashMap`.

📌 **Example:**

```
import java.util.*;

public class WeakHashMapExample {
    public static void main(String[] args) {
        Map<Object, String> map = new WeakHashMap<>();

        Object key1 = new String("key1");  // Weak reference key
        Object key2 = new String("key2");

        map.put(key1, "Value 1");
        map.put(key2, "Value 2");

        System.out.println("Before GC: " + map);

        // Remove strong references to keys
        key1 = null;
        key2 = null;

        // Call garbage collector
        System.gc();

        // Wait for GC to complete
        try { Thread.sleep(2000); } catch (InterruptedException e) {}

        System.out.println("After GC: " + map);
    }
}
```

✔ **Output (Example, may vary depending on GC execution):**

```
Before GC: {key1=Value 1, key2=Value 2}
After GC: {}
```

☑ **The keys were garbage collected, so the map became empty!**

---

4 `WeakHashMap` Constructors

Constructor	Description
`WeakHashMap()`	Creates an empty `WeakHashMap`.
`WeakHashMap(int initialCapacity)`	Creates a `WeakHashMap` with a specified capacity.
`WeakHashMap(int initialCapacity, float loadFactor)`	Creates a `WeakHashMap` with a specified capacity and load factor.
`WeakHashMap(Map<? extends K, ? extends V> m)`	Creates a `WeakHashMap` from an existing map.

---

5 Important Methods of `WeakHashMap<K, V>`

Method	Description
`put(K key, V value)`	Adds a key-value pair to the map.
`get(K key)`	Retrieves the value for a key.
`remove(K key)`	Removes a key-value pair.
`containsKey(K key)`	Checks if a key exists.
`containsValue(V value)`	Checks if a value exists.
`size()`	Returns the number of key-value pairs.
`isEmpty()`	Checks if the map is empty.
`keySet()`	Returns all keys as a `Set`.
`values()`	Returns all values as a `Collection`.
`entrySet()`	Returns all key-value pairs as a `Set`.

---

6 `WeakHashMap` Example Code

```
import java.util.WeakHashMap;

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

```
        WeakHashMap<String, String> map = new WeakHashMap<>();

        String key1 = new String("User1");
        String key2 = new String("User2");

        map.put(key1, "Alice");
        map.put(key2, "Bob");

        System.out.println("Before GC: " + map);

        key1 = null; // Removing strong reference

        System.gc(); // Request Garbage Collection

        try { Thread.sleep(2000); } catch (InterruptedException e) {}

        System.out.println("After GC: " + map);
    }
}
```

✔ **Output (May vary depending on GC execution):**

```
Before GC: {User1=Alice, User2=Bob}
After GC: {User2=Bob}
```

☑ **Only `User1` was garbage collected because we removed its reference!**

---

7 Performance Analysis

Operation	Time Complexity
`put(K, V)`	**O(1)** (best case), **O(n)** (worst case - hash collisions)
`get(K)`	**O(1)** (best case), **O(n)** (worst case - hash collisions)
`remove(K)`	**O(1)** (best case), **O(n)** (worst case - hash collisions)

📌 **Why `WeakHashMap` is Useful?**

- Prevents **memory leaks** by automatically removing unused keys.
- Ideal for **caching and temporary storage**.

---

8 When to Use `WeakHashMap<K, V>`?

✔ **When you want automatic cleanup of unused keys.**
✔ **When storing temporary/cache data that should be removed when not needed.**
✔ **When you need a `Map<K, V>` that does not block garbage collection.**

❌ **Avoid `WeakHashMap<K, V>` if:**

- You need strong references (Use `HashMap` instead).

- You need a thread-safe map (`Use ConcurrentHashMap`).

📌 Summary

✔ **`WeakHashMap<K, V>` is a special `Map` that automatically removes unused keys.**
✔ **Uses weak references, so keys are garbage collected when not strongly referenced.**
✔ **Best for caching and temporary data storage.**
✔ **Not thread-safe, but faster than `Hashtable`.**

📌 Deep Dive into `IdentityHashMap<K, V>` (Easy & Detailed Explanation)

1️⃣ What is `IdentityHashMap<K, V>`?

`IdentityHashMap<K, V>` is a special type of `Map<K, V>` in Java that **compares keys using reference equality (`==`) instead of object equality (`equals()`).**

✔ Key Features of `IdentityHashMap<K, V>`

- ☑ **Uses `==` instead of `equals()` for comparing keys.**
- ☑ **Allows duplicate keys if they are different objects (`new String("A")` ≠ `new String("A")`).**
- ☑ **Not thread-safe (like `HashMap`).**
- ☑ **Does not maintain insertion order (like `HashMap`).**
- ☑ **Faster than `HashMap` because it avoids extra hash computations.**

📌 **Example Use Case:**
Imagine you need to **store unique objects based on memory references, not content**—for example, when handling proxies, caches, or serialization where object identity matters.

2️⃣ Difference Between `HashMap`, `WeakHashMap`, and `IdentityHashMap`

Feature	`HashMap<K, V>`	`WeakHashMap<K, V>`	`IdentityHashMap<K, V>`
Key Comparison	Uses `equals()`	Uses `equals()`	Uses `==` (Reference)
Garbage Collection Aware?	❌ No	☑ Yes	❌ No
Allows `null` Keys?	☑ Yes	☑ Yes	☑ Yes
Thread-Safe?	❌ No	❌ No	❌ No

Performance	**O(1)** (Best case)	**O(1)** (Best case)	**O(1)** (Best case)
When to Use?	General Purpose Map	Auto-removing keys (caching)	Object Identity-Based Mapping

📌 **When to Use `IdentityHashMap`?**

✔ **When you want different instances of the same object to be treated as different keys.**

✔ **When handling proxies, serialization, or tracking object identity.**

✔ **When performance is important (faster lookup due to `==` comparison).**

3️⃣ How `IdentityHashMap<K, V>` Works Internally?

Unlike `HashMap`, which uses **hash codes and `equals()`**, `IdentityHashMap` uses **memory references (`==`)** for key comparison.

📌 **Example:**

```
import java.util.IdentityHashMap;

public class IdentityHashMapExample {
    public static void main(String[] args) {
        IdentityHashMap<String, Integer> map = new IdentityHashMap<>();

        String key1 = new String("A"); // Different Object
        String key2 = new String("A"); // Different Object

        map.put(key1, 1);
        map.put(key2, 2); // Different object, so it will be added separately

        System.out.println("Map Size: " + map.size()); // Output: 2
        System.out.println("Map: " + map);
    }
}
```

☑ **Output:**

```
Map Size: 2
Map: {A=1, A=2}
```

✔ Unlike `HashMap`, **both `"A"` keys are treated as different because they are different objects in memory.**

4️⃣ `IdentityHashMap` Constructors

Constructor	Description
`IdentityHashMap()`	Creates an empty `IdentityHashMap`.
`IdentityHashMap(int expectedSize)`	Creates an `IdentityHashMap` with an expected size.

`IdentityHashMap(Map<? extends K, ? extends V> m)`	Creates an `IdentityHashMap` from an existing map.

---

5 Important Methods of `IdentityHashMap<K, V>`

Method	Description
`put(K key, V value)`	Adds a key-value pair to the map.
`get(K key)`	Retrieves the value for a key.
`remove(K key)`	Removes a key-value pair.
`containsKey(K key)`	Checks if a key exists.
`containsValue(V value)`	Checks if a value exists.
`size()`	Returns the number of key-value pairs.
`isEmpty()`	Checks if the map is empty.
`keySet()`	Returns all keys as a `Set`.
`values()`	Returns all values as a `Collection`.
`entrySet()`	Returns all key-value pairs as a `Set`.

---

6 `IdentityHashMap` Example Code

```
import java.util.IdentityHashMap;

public class IdentityHashMapDemo {
    public static void main(String[] args) {
        IdentityHashMap<Integer, String> map = new IdentityHashMap<>();

        Integer key1 = new Integer(10);
        Integer key2 = new Integer(10);

        map.put(key1, "Value 1");
        map.put(key2, "Value 2"); // Treated as different keys

        System.out.println("Map Size: " + map.size());
        System.out.println("Map: " + map);
    }
}
```

✔ **Output:**

```
Map Size: 2
Map: {10=Value 1, 10=Value 2}
```

☑ **Both `10` keys are treated as different objects because they are different instances.**

7️⃣ Performance Analysis

Operation	Time Complexity
`put(K, V)`	**O(1)** (best case), **O(n)** (worst case - hash collisions)
`get(K)`	**O(1)** (best case), **O(n)** (worst case - hash collisions)
`remove(K)`	**O(1)** (best case), **O(n)** (worst case - hash collisions)

📌 **Why `IdentityHashMap` is Useful?**

- 🚀 **Faster than `HashMap` because it avoids hash computation overhead.**
- 🚀 **Useful when object identity matters instead of content comparison.**

8️⃣ When to Use `IdentityHashMap<K, V>`?

✔ **When object identity (`==`) matters, not content comparison (`equals()`).**
✔ **When you need to distinguish between different object instances of the same value.**
✔ **When performance is critical, and avoiding hash computations speeds up the program.**

❌ **Avoid `IdentityHashMap<K, V>` if:**

- You need keys to be compared based on content (`equals()`).
- You need a thread-safe map (`Use ConcurrentHashMap`).

📌 Summary

✔ **`IdentityHashMap<K, V>` is a `Map` that compares keys using reference equality (`==`) instead of `equals()`.**
✔ **Allows duplicate-looking keys if they are different objects in memory.**
✔ **Faster than `HashMap` for specific use cases.**
✔ **Not thread-safe, does not maintain insertion order.**
✔ **Useful for object identity tracking, serialization, caching.**

📌 Chapter 7: Comparators and Sorting in Collections (Easy & Deep Explanation)

Sorting is a crucial part of working with collections in Java. Java provides two key interfaces to handle sorting:

1️⃣ **`Comparable<T>`** (Natural Sorting)
2️⃣ **`Comparator<T>`** (Custom Sorting)

1️⃣ Why Do We Need Sorting in Java Collections?

Sorting helps in:

✔️ **Quickly searching elements** in a large dataset.
✔️ **Efficient data processing** by ordering records logically.
✔️ **Enhancing performance** in searching algorithms like binary search.
✔️ **Organizing user data** (e.g., sorting students by marks, sorting products by price).

💡 Java provides two main ways to sort collections:

- **Natural Sorting** (`Comparable<T>`)
- **Custom Sorting** (`Comparator<T>`)

2️⃣ How Sorting Works in Java Collections?

Java collections can be sorted using:

1️⃣ **`Collections.sort(list)`** → Sorts a `List` using natural ordering (must implement `Comparable<T>`).
2️⃣ **`Collections.sort(list, comparator)`** → Sorts a `List` using a `Comparator<T>` for custom ordering.
3️⃣ **`TreeSet<T>` and `TreeMap<K, V>`** → Automatically sort elements based on natural ordering or a custom comparator.

3️⃣ Understanding `Comparable<T>` and `Comparator<T>` (Key Differences)

Feature	`Comparable<T>`	`Comparator<T>`
Purpose	Defines **natural sorting order** of an object.	Defines **custom sorting order** for objects.
Method Used	`compareTo(T o)`	`compare(T o1, T o2)`
Where to Implement?	Implemented **inside the class** being sorted.	Implemented in a **separate class** or using lambda functions.
Modifies Original Class?	☑ Yes, class must implement `Comparable<T>`.	❌ No, sorting logic is external.
Sorts By	Single field (e.g., sorting students by marks).	Multiple fields (e.g., sorting students by name and then marks).
Used In	`TreeSet`, `TreeMap`, `Collections.sort()`.	`Collections.sort()`, `TreeSet`, `TreeMap`.

☑ **Use** `Comparable<T>` when the class has a **single natural sorting order** (e.g., sorting employees by salary).
☑ **Use** `Comparator<T>` when sorting should be **flexible** (e.g., sorting employees by salary or by age).

4️⃣ Sorting Lists, Sets, and Maps in Java

Sorting can be applied to different collections:

☑ Sorting a `List<T>`

- `Collections.sort(List<T>)` → Uses `Comparable<T>`
- `Collections.sort(List<T>, Comparator<T>)` → Uses `Comparator<T>`

☑ Sorting a `Set<T>`

- `TreeSet<T>` automatically sorts elements based on `Comparable<T>` or `Comparator<T>`.

☑ Sorting a `Map<K, V>`

- `TreeMap<K, V>` automatically sorts based on `Comparable<K>` or `Comparator<K>`.
- `LinkedHashMap<K, V>` maintains **insertion order**, not sorting.
- `HashMap<K, V>` does **not** sort keys or values.

📌 Deep Dive into `Comparable<T>` (Natural Sorting) (Easy Explanation)

1️⃣ What is `Comparable<T>`?

☑ `Comparable<T>` is an interface in Java used for **natural sorting** of objects.
☑ It allows a class to define its own **default sorting order**.
☑ It provides the `compareTo(T o)` method to define sorting logic.

2️⃣ Why Do We Need `Comparable<T>`?

Imagine we have a list of **students**, and we want to sort them by their **marks**.
Without `Comparable<T>`, Java **does not know** how to compare student objects.
By implementing `Comparable<T>`, we can **tell Java** how to compare them (e.g., highest marks first).

3️⃣ How to Use `Comparable<T>`?

Steps to Implement `Comparable<T>`:
1️⃣ **Make the class implement `Comparable<T>`.**

2️⃣ **Override the `compareTo(T o)` method.**
3️⃣ **Define sorting logic inside `compareTo`.**
4️⃣ **Use `Collections.sort(List<T>)` to sort the list.**

4️⃣ Example: Sorting Students by Marks (Ascending Order)

```
import java.util.*;

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

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

    // Implement compareTo method (Natural Sorting)
    @Override
    public int compareTo(Student other) {
        return this.marks - other.marks; // Sorting by marks (Ascending Order)
    }

    // Display method
    public String toString() {
        return "Student{" + "ID=" + id + ", Name='" + name + "', Marks=" + marks +
'}';
    }
}

public class ComparableExample {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(101, "Alice", 85));
        students.add(new Student(102, "Bob", 72));
        students.add(new Student(103, "Charlie", 90));

        System.out.println("Before Sorting:");
        System.out.println(students);

        // Sorting using Collections.sort() (Natural Order)
        Collections.sort(students);

        System.out.println("\nAfter Sorting:");
        System.out.println(students);
    }
}
```

🔹 Output:

```
Before Sorting:
[Student{ID=101, Name='Alice', Marks=85}, Student{ID=102, Name='Bob', Marks=72},
Student{ID=103, Name='Charlie', Marks=90}]

After Sorting:
[Student{ID=102, Name='Bob', Marks=72}, Student{ID=101, Name='Alice', Marks=85},
Student{ID=103, Name='Charlie', Marks=90}]
```

✔ **Explanation:**

- We implemented `Comparable<Student>`.
- The `compareTo` method **sorts students by marks in ascending order**.
- `Collections.sort(students)` **sorts the list based on `compareTo` method**.

5️⃣ Changing Sorting Order (Descending Order)

By default, `compareTo` sorts in **ascending order**.
To sort in **descending order**, modify `compareTo`:

```
@Override
public int compareTo(Student other) {
    return other.marks - this.marks; // Sorting by marks (Descending Order)
}
```

Now, the highest marks will come first.

6️⃣ Sorting Objects with Multiple Fields

We can modify `compareTo` to sort by **multiple criteria**.

Example: Sorting by Marks, then by Name (if Marks are Equal)

```
@Override
public int compareTo(Student other) {
    if (this.marks == other.marks) {
        return this.name.compareTo(other.name); // Sort by Name (Alphabetical Order)
    }
    return other.marks - this.marks; // Sort by Marks (Descending Order)
}
```

✔ **Now students with the same marks will be sorted alphabetically.**

7️⃣ Key Points About `Comparable<T>`

✔ Used for **natural sorting** (default order).
✔ We **must modify the original class** (implements `Comparable<T>`).

✔ Sorting logic is defined in **`compareTo(T o)` method**.
✔ Used in `TreeSet<T>`, `TreeMap<K, V>`, and `Collections.sort(List<T>)`.
✔ **Only one sorting order is possible per class.**

📌 Deep Dive into `Comparator<T>` (Custom Sorting) (Easy Explanation)

1️⃣ What is `Comparator<T>`?

☑ `Comparator<T>` is an interface used to **define custom sorting logic** for objects.
☑ It allows **multiple sorting orders** without modifying the original class.
☑ It provides the **`compare(T o1, T o2)`** method to define sorting logic.

2️⃣ Why Do We Need `Comparator<T>`?

Imagine we have a list of **students** and we want to sort them in different ways:
✔ By **marks** (highest to lowest).
✔ By **name** (alphabetical order).
✔ By **ID** (ascending order).

If we use `Comparable<T>`, we can **only define one sorting order** inside the class.
But with `Comparator<T>`, we can define **multiple sorting orders externally**.

3️⃣ How to Use `Comparator<T>`?

Steps to Implement `Comparator<T>`:
1️⃣ **Create a separate class** that implements `Comparator<T>`.
2️⃣ **Override the `compare(T o1, T o2)` method**.
3️⃣ **Use `Collections.sort(List<T>, Comparator<T>)`** to sort the list.

4️⃣ Example: Sorting Students by Marks (Descending Order)

```
import java.util.*;

class Student {
    int id;
    String name;
    int marks;

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

```
    }

    public String toString() {
        return "Student{" + "ID=" + id + ", Name='" + name + "', Marks=" + marks +
'}';
    }
}

// Custom Comparator for sorting by Marks (Descending Order)
class SortByMarks implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s2.marks - s1.marks; // Highest marks first
    }
}

public class ComparatorExample {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(101, "Alice", 85));
        students.add(new Student(102, "Bob", 72));
        students.add(new Student(103, "Charlie", 90));

        System.out.println("Before Sorting:");
        System.out.println(students);

        // Sorting using Comparator
        Collections.sort(students, new SortByMarks());

        System.out.println("\nAfter Sorting (By Marks Descending):");
        System.out.println(students);
    }
}
```

🔹 Output:

```
Before Sorting:
[Student{ID=101, Name='Alice', Marks=85}, Student{ID=102, Name='Bob', Marks=72},
Student{ID=103, Name='Charlie', Marks=90}]

After Sorting (By Marks Descending):
[Student{ID=103, Name='Charlie', Marks=90}, Student{ID=101, Name='Alice', Marks=85},
Student{ID=102, Name='Bob', Marks=72}]
```

✔ **Explanation:**

- We created a **separate class** `SortByMarks` that implements `Comparator<Student>`.
- The `compare` method sorts students **by marks in descending order**.
- We passed `new SortByMarks()` to `Collections.sort()` for sorting.

5️⃣ Sorting Students by Name (Alphabetical Order)

We can create another **custom comparator** for sorting by name.

```
// Custom Comparator for sorting by Name (Alphabetical Order)
class SortByName implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name); // A to Z order
    }
}
```

Now, we can sort by name:

```
Collections.sort(students, new SortByName());
```

✔️ Now students will be sorted in **alphabetical order by name**.

6️⃣ Sorting by Multiple Fields

What if **marks are equal**?
We can **first sort by marks**, and if they are the same, **sort by name**.

```
class SortByMarksThenName implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        if (s1.marks == s2.marks) {
            return s1.name.compareTo(s2.name); // Sort by Name (Alphabetical)
        }
        return s2.marks - s1.marks; // Sort by Marks (Descending)
    }
}
```

Now, we can sort students:

```
Collections.sort(students, new SortByMarksThenName());
```

✔️ If two students have the **same marks**, they will be sorted **alphabetically by name**.

7️⃣ Using Lambda Expressions for Comparator

Instead of creating separate classes, we can use **lambda expressions**.

Sorting by Marks (Descending) Using Lambda

```
Collections.sort(students, (s1, s2) -> s2.marks - s1.marks);
```

Sorting by Name (Alphabetical) Using Lambda

```
Collections.sort(students, (s1, s2) -> s1.name.compareTo(s2.name));
```

💡 **Lambda makes sorting more readable and concise**.

8️⃣ Key Differences: `Comparable<T>` vs `Comparator<T>`

Feature	`Comparable<T>`	`Comparator<T>`
Purpose	Defines **natural sorting order** of an object.	Defines **custom sorting order** for objects.
Method Used	`compareTo(T o)`	`compare(T o1, T o2)`
Where to Implement?	Implemented **inside the class** being sorted.	Implemented in a **separate class** or using lambda functions.
Modifies Original Class?	☑ Yes, class must implement `Comparable<T>`.	❌ No, sorting logic is external.
Sorts By	Single field (e.g., sorting students by marks).	Multiple fields (e.g., sorting students by name and then marks).
Used In	`TreeSet`, `TreeMap`, `Collections.sort()`.	`Collections.sort()`, `TreeSet`, `TreeMap`.

☑ **Use `Comparable<T>`** when the class has a **single natural sorting order**.
☑ **Use `Comparator<T>`** when sorting should be **flexible**.

📌 Sorting in Sets and Maps Using Comparator<T> (Easy Explanation)

Sorting **Lists** is easy with `Comparator<T>`, but **how do we sort Sets and Maps**? 🤔
Let's explore **sorting techniques for Sets and Maps in Java**.

1️⃣ Sorting `Set<T>` (TreeSet, HashSet, LinkedHashSet)

☑ **By Default:**

- `TreeSet<T>` sorts elements **automatically in ascending order**.
- `HashSet<T>` and `LinkedHashSet<T>` **do NOT maintain sorting order**.

☑ **How to Sort Sets?**
Since `HashSet<T>` and `LinkedHashSet<T>` don't support sorting, we must:
✔ Convert them into a **List**.
✔ Sort the list using `Comparator<T>`.
✔ Convert the list **back into a Set**.

Sorting `TreeSet<T>` Using `Comparator<T>`

- `TreeSet<T>` allows custom sorting using a **Comparator**.

```
import java.util.*;

class SortTreeSetExample {
    public static void main(String[] args) {
        // TreeSet with custom sorting (Descending Order)
        TreeSet<Integer> numbers = new TreeSet<>(Comparator.reverseOrder());

        numbers.add(10);
        numbers.add(50);
        numbers.add(30);
        numbers.add(20);

        System.out.println("Sorted TreeSet (Descending Order): " + numbers);
    }
}
```

🔹 Output:

```
Sorted TreeSet (Descending Order): [50, 30, 20, 10]
```

✔ We passed `Comparator.reverseOrder()` to sort the TreeSet **in descending order**.

Sorting `HashSet<T>` Using `Comparator<T>`

🔹 `HashSet<T>` does not maintain sorting order, so we need to convert it to a `List<T>`, sort it, and convert it back.

```
import java.util.*;

class SortHashSetExample {
    public static void main(String[] args) {
        HashSet<String> names = new HashSet<>();
        names.add("Charlie");
        names.add("Alice");
        names.add("Bob");

        // Convert HashSet to List
        List<String> sortedList = new ArrayList<>(names);

        // Sort List using Comparator (Alphabetical Order)
        sortedList.sort(Comparator.naturalOrder());

        // Convert List back to Set
        LinkedHashSet<String> sortedSet = new LinkedHashSet<>(sortedList);

        System.out.println("Sorted HashSet: " + sortedSet);
    }
}
```

🔹 Output:

```
Sorted HashSet: [Alice, Bob, Charlie]
```

✔ We used `Comparator.naturalOrder()` to sort names **in alphabetical order**.

2️⃣ Sorting `Map<K, V>` (HashMap, LinkedHashMap, TreeMap)

☑ **By Default:**

- `TreeMap<K, V>` **sorts keys in natural order**.
- `HashMap<K, V>` and `LinkedHashMap<K, V>` **do NOT maintain sorting order**.

☑ **How to Sort Maps?**

✔ We can sort **by keys** or **by values** using `Comparator<T>`.

Sorting `TreeMap<K, V>` By Custom Order

- By default, `TreeMap<K, V>` sorts **by key in ascending order**.
- We can **customize the sorting order**.

```
import java.util.*;

class SortTreeMapExample {
    public static void main(String[] args) {
        // TreeMap sorted in reverse order of keys
        TreeMap<Integer, String> treeMap = new TreeMap<>(Comparator.reverseOrder());

        treeMap.put(1, "Apple");
        treeMap.put(3, "Banana");
        treeMap.put(2, "Cherry");

        System.out.println("Sorted TreeMap (By Key Descending): " + treeMap);
    }
}
```

🔹 Output:

```
Sorted TreeMap (By Key Descending): {3=Banana, 2=Cherry, 1=Apple}
```

✔ We used `Comparator.reverseOrder()` to **sort keys in descending order**.

Sorting `HashMap<K, V>` By Keys

- Since `HashMap<K, V>` is **unordered**, we:

✔ Convert it into a `List<Map.Entry<K, V>>`.

✔ Sort it using a `Comparator<K>`.

✔ Insert it into a `LinkedHashMap<K, V>`.

```
import java.util.*;

class SortHashMapByKeyExample {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();
```

```
        map.put(3, "Banana");
        map.put(1, "Apple");
        map.put(2, "Cherry");

        // Convert to List
        List<Map.Entry<Integer, String>> entryList = new ArrayList<>(map.entrySet());

        // Sort by Key (Ascending)
        entryList.sort(Map.Entry.comparingByKey());

        // Convert back to LinkedHashMap
        LinkedHashMap<Integer, String> sortedMap = new LinkedHashMap<>();
        for (Map.Entry<Integer, String> entry : entryList) {
            sortedMap.put(entry.getKey(), entry.getValue());
        }

        System.out.println("Sorted HashMap (By Key Ascending): " + sortedMap);
    }
}
```

- **Output:**

```
Sorted HashMap (By Key Ascending): {1=Apple, 2=Cherry, 3=Banana}
```

✔ We used `Map.Entry.comparingByKey()` to **sort by key**.

Sorting `HashMap<K, V>` By Values

- If we want to **sort by values** instead of keys:

```
import java.util.*;

class SortHashMapByValueExample {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();
        map.put(3, "Banana");
        map.put(1, "Apple");
        map.put(2, "Cherry");

        // Convert to List
        List<Map.Entry<Integer, String>> entryList = new ArrayList<>(map.entrySet());

        // Sort by Value (Alphabetical Order)
        entryList.sort(Map.Entry.comparingByValue());

        // Convert back to LinkedHashMap
        LinkedHashMap<Integer, String> sortedMap = new LinkedHashMap<>();
        for (Map.Entry<Integer, String> entry : entryList) {
            sortedMap.put(entry.getKey(), entry.getValue());
        }

        System.out.println("Sorted HashMap (By Value): " + sortedMap);
    }
}
```

🔹 Output:

```
Sorted HashMap (By Value): {1=Apple, 3=Banana, 2=Cherry}
```

✔ We used `Map.Entry.comparingByValue()` to **sort by value alphabetically**.

🔹 Summary: Sorting Techniques for Collections

Collection	Sorting Strategy
`ArrayList<T>`	Use `Collections.sort(list, comparator)`
`TreeSet<T>`	Use `new TreeSet<>(comparator)`
`HashSet<T>`	Convert to `List<T>`, sort, convert back
`TreeMap<K, V>`	Use `new TreeMap<>(comparator)`
`HashMap<K, V>`	Convert to `List<Map.Entry<K, V>>`, sort, convert back

☑ **Use `Comparator<T>` for custom sorting** in Lists, Sets, and Maps.
☑ **Convert HashSet/HashMap to a List** if they don't support sorting directly.

📌 Chapter 8: Collections Utility Class (Helper Methods) – Deep Explanation

The **Collections Utility Class** in Java provides several static methods to operate on **lists, sets, and maps** easily. It includes methods for:

☑ **Sorting** (e.g., `Collections.sort()`)
☑ **Searching** (e.g., `Collections.binarySearch()`)
☑ **Making Collections Immutable** (e.g., `Collections.unmodifiableList()`)
☑ **Creating Thread-Safe Collections** (e.g., `Collections.synchronizedList()`)

🔹 What is the `Collections` Utility Class?

The `Collections` class is a **final class** in Java's `java.util` package.

- It **cannot be instantiated** because it only contains **static methods**.
- It **enhances** how we work with collections by providing **common utility functions**.

📌 Key Features of `Collections` Class

Feature	Description
Sorting	Sorts a `List<T>` using natural or custom ordering.
Searching	Searches for an element in a sorted list using binary search.

Thread-Safe Collections	Converts collections into thread-safe versions.
Immutable Collections	Creates **unmodifiable** collections that cannot be changed.
Shuffling	Randomizes the order of elements.
Reversing	Reverses the order of elements in a list.
Filling	Replaces all elements in a list with a specified value.
Copying	Copies elements from one list to another.
Finding Min/Max	Finds the smallest or largest element in a collection.

🔹 Why Use `Collections` Utility Class?

☑ Without `Collections` Class (Manual Sorting Example)

```
import java.util.*;

class WithoutCollections {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(5, 2, 8, 1));

        // Manual sorting using loops
        for (int i = 0; i < numbers.size() - 1; i++) {
            for (int j = i + 1; j < numbers.size(); j++) {
                if (numbers.get(i) > numbers.get(j)) {
                    // Swap elements
                    int temp = numbers.get(i);
                    numbers.set(i, numbers.get(j));
                    numbers.set(j, temp);
                }
            }
        }

        System.out.println("Sorted List: " + numbers);
    }
}
```

🔹 Output:

```
Sorted List: [1, 2, 5, 8]
```

✔ Here, we had to **write a lot of code** just to sort a list.

☑ With `Collections` Class (Easy Sorting)

```
import java.util.*;

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

```
        List<Integer> numbers = new ArrayList<>(Arrays.asList(5, 2, 8, 1));

        // Using Collections.sort()
        Collections.sort(numbers);

        System.out.println("Sorted List: " + numbers);
    }
}
```

🔹 Output:

```
Sorted List: [1, 2, 5, 8]
```

✔️ Just **one line of code** using `Collections.sort()`!

🔹 List of Important Methods in `Collections` Class

Method	Description
`Collections.sort(List<T>)`	Sorts a list in natural order.
`Collections.sort(List<T>, Comparator<T>)`	Sorts a list using a custom comparator.
`Collections.binarySearch(List<T>, key)`	Searches for an element in a sorted list using binary search.
`Collections.unmodifiableList(List<T>)`	Creates an **immutable list**.
`Collections.synchronizedList(List<T>)`	Makes a list **thread-safe**.
`Collections.reverse(List<T>)`	Reverses the order of elements in a list.
`Collections.shuffle(List<T>)`	Randomizes the order of elements.
`Collections.fill(List<T>, value)`	Replaces all elements in a list with a specified value.
`Collections.copy(List<T>, List<T>)`	Copies elements from one list to another.
`Collections.min(Collection<T>)`	Finds the smallest element.
`Collections.max(Collection<T>)`	Finds the largest element.

Deep Dive into First Three Methods of `Collections` Class

In this section, we'll **deeply understand** the first three methods of the `Collections` utility class:

☑ `Collections.sort(List<T>)`
☑ `Collections.sort(List<T>, Comparator<T>)`
☑ `Collections.binarySearch(List<T>, key)`

We'll cover:

- **What the method does**

- **How it works internally**
- **Code examples**
- **Time complexity**
- **Real-world use cases**

1️⃣ `Collections.sort(List<T>)` (Natural Sorting)

📌 What It Does?

- This method **sorts a List** in **ascending order** using **natural ordering**.
- It works with elements that implement the **`Comparable` interface** (like `Integer`, `String`, `Double`, etc.).
- Sorting is done using **Timsort**, which is a combination of **Merge Sort** and **Insertion Sort**.

📌 Syntax

```
Collections.sort(List<T> list);
```

- This method **modifies the original list** by sorting it.

📌 Internal Working (How It Works?)

1. **Checks if the list implements `RandomAccess`** (i.e., if it's an `ArrayList`).
    - If `true` → Uses **Dual-Pivot QuickSort** (fastest for arrays).
    - If `false` → Uses **Merge Sort** (better for linked lists).
2. **Calls `TimSort.sort()` method** for sorting the list.
3. **Rearranges elements in ascending order**.

📌 Example: Sorting a List of Numbers

```
import java.util.*;

class SortExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(9, 3, 7, 1, 5));

        // Sorting in ascending order
        Collections.sort(numbers);

        System.out.println("Sorted List: " + numbers);
    }
}
```

🔹 Output:

```
Sorted List: [1, 3, 5, 7, 9]
```

📌 Example: Sorting a List of Strings

```
import java.util.*;

class StringSortExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>(Arrays.asList("John", "Alice", "Bob"));

        // Sorting alphabetically
        Collections.sort(names);

        System.out.println("Sorted Names: " + names);
    }
}
```

🔹 Output:

```
Sorted Names: [Alice, Bob, John]
```

📌 Time Complexity

- **Worst Case:** `O(n log n)`
- **Best Case (Already Sorted):** `O(n)`
- **Average Case:** `O(n log n)`

✔ **Fast and efficient** for large datasets.

2️⃣ `Collections.sort(List<T>, Comparator<T>)` (Custom Sorting)

📌 What It Does?

- This method **sorts a list** using a **custom sorting logic** defined by a `Comparator<T>`.
- Used when elements **do not have natural ordering** (e.g., sorting objects, sorting in descending order).

📌 Syntax

```
Collections.sort(List<T> list, Comparator<T> comparator);
```

- The **comparator** defines **how elements should be sorted**.

📌 Example: Sorting in Descending Order

```
import java.util.*;

class DescendingSort {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(9, 3, 7, 1, 5));

        // Sorting in descending order
        Collections.sort(numbers, (a, b) -> b - a);

        System.out.println("Sorted Descending: " + numbers);
    }
}
```

🔹 Output:

```
Sorted Descending: [9, 7, 5, 3, 1]
```

📌 Example: Sorting a List of Objects

```
import java.util.*;

class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

class AgeComparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age; // Sort by age (ascending)
    }
}

class ObjectSortingExample {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>(Arrays.asList(
            new Person("Alice", 25),
            new Person("Bob", 22),
            new Person("Charlie", 28)
        ));

        // Sorting by age
        Collections.sort(people, new AgeComparator());
```

```
        System.out.println("Sorted by Age: " + people);
    }
}
```

🔹 Output:

```
Sorted by Age: [Bob (22), Alice (25), Charlie (28)]
```

📌 Time Complexity

- Same as `Collections.sort(List<T>)`, i.e., `O(n log n)`.
  - ✔ **More flexible**, as it allows **custom sorting**.

---

3 `Collections.binarySearch(List<T>, key)` (Efficient Searching)

📌 What It Does?

- **Searches for an element in a sorted list** using **Binary Search**.
- It is **faster than linear search** (`O(log n)` instead of `O(n)`).
- The list **must be sorted** before using `binarySearch()`.

📌 Syntax

```
int index = Collections.binarySearch(List<T> list, T key);
```

- Returns the **index** of `key` if found.
- Returns **negative index** (`-(insertion point) - 1`) if **not found**.

📌 Example: Searching in a Sorted List

```
import java.util.*;

class BinarySearchExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 3, 5, 7, 9));

        int index = Collections.binarySearch(numbers, 5);
        System.out.println("Index of 5: " + index);
    }
}
```

🔹 Output:

```
Index of 5: 2
```

✔ `5` is found at **index 2**.

📌 Example: Searching for an Element Not in the List

```
import java.util.*;

class BinarySearchNotFound {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 3, 5, 7, 9));

        int index = Collections.binarySearch(numbers, 6);
        System.out.println("Index of 6: " + index);
    }
}
```

🔹 Output:

```
Index of 6: -4
```

✔ **Explanation:**

- `6` is **not in the list**.
- `-(insertion point) - 1` = `-(3) - 1 = -4`.
- The insertion point is `index 3`.

📌 Time Complexity

- **Best Case:** `O(1)` (if the element is at the middle).
- **Worst/Average Case:** `O(log n)`.
  ✔ **Faster** than a simple linear search.

📌 Summary Table

Method	Purpose	Time Complexity
`Collections.sort(List<T>)`	Sorts a list in natural order (ascending).	`O(n log n)`
`Collections.sort(List<T>, Comparator<T>)`	Sorts a list using custom order.	`O(n log n)`
`Collections.binarySearch(List<T>, key)`	Searches for an element in a sorted list.	`O(log n)`

Deep Dive into Next Three Methods of `Collections` Class

☑ `Collections.unmodifiableList(List<T>)`
☑ `Collections.synchronizedList(List<T>)`
☑ `Collections.reverse(List<T>)`

We'll cover:

- **What the method does**
- **How it works internally**
- **Code examples**
- **Time complexity**
- **Real-world use cases**

---

1 `Collections.unmodifiableList(List<T>)` (Immutable List)

📌 What It Does?

- Creates a **read-only (immutable) version** of a list.
- Any attempt to modify the list (add, remove, update) **throws an exception**.
- Used when you **want to protect a list** from accidental modification.

📌 Syntax

```
List<T> immutableList = Collections.unmodifiableList(List<T> list);
```

- **Returns an unmodifiable view** of the list.

📌 Internal Working (How It Works?)

1. **Wraps the original list** inside an unmodifiable wrapper.
2. **Allows only read operations** (like `get()`, `contains()`).
3. **Throws `UnsupportedOperationException`** for any modification.

📌 Example: Creating an Unmodifiable List

```
import java.util.*;

class UnmodifiableListExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>(Arrays.asList("Alice", "Bob",
"Charlie"));

        // Creating an unmodifiable list
        List<String> immutableNames = Collections.unmodifiableList(names);
```

```
        System.out.println("Immutable List: " + immutableNames);

        // Trying to modify the list (will throw an exception)
        immutableNames.add("David");  // Throws UnsupportedOperationException
    }
}
```

🔹 Output:

```
Exception in thread "main" java.lang.UnsupportedOperationException
```

✔ The program crashes when trying to modify the list.

📌 Real-World Use Case

- **Used in APIs** to return **safe lists** that clients cannot modify.
- Example: `List.of()` in Java 9+ creates immutable lists directly.

---

2 `Collections.synchronizedList(List<T>)` (Thread-Safe List)

📌 What It Does?

- Converts a **normal list** into a **thread-safe list**.
- Allows **multiple threads** to access the list **without conflicts**.
- **Synchronizes all methods** (`add()`, `remove()`, `get()`, etc.).

📌 Syntax

```
List<T> syncList = Collections.synchronizedList(List<T> list);
```

- **Returns a synchronized version** of the list.

📌 Internal Working (How It Works?)

1. **Wraps the list inside a synchronized wrapper**.
2. **Every method is synchronized** (ensures only one thread modifies at a time).
3. **Iterating still requires external synchronization** (`synchronized(syncList)`).

📌 Example: Making a List Thread-Safe

```
import java.util.*;

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

```
        List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3));

        // Creating a thread-safe list
        List<Integer> syncNumbers = Collections.synchronizedList(numbers);

        // Thread-safe modification
        syncNumbers.add(4);
        System.out.println("Synchronized List: " + syncNumbers);
    }
}
```

✔ **This prevents concurrency issues** in a multi-threaded environment.

📌 Important: Synchronizing Iteration

Even though the list is synchronized, **iteration must be synchronized externally**:

```
synchronized(syncNumbers) {
    for (Integer num : syncNumbers) {
        System.out.println(num);
    }
}
```

✔ Without this, **ConcurrentModificationException** may occur.

📌 Real-World Use Case

- Used in **multi-threaded applications** where lists are shared across threads.
- **Alternative:** `CopyOnWriteArrayList` (better performance for concurrent reads).

3️⃣ `Collections.reverse(List<T>)` (Reverse Order)

📌 What It Does?

- **Reverses the order** of elements in a list.
- Modifies the **original list**.
- Works on any `List<T>` implementation (`ArrayList`, `LinkedList`, etc.).

📌 Syntax

```
Collections.reverse(List<T> list);
```

- **Directly modifies** the input list.

📌 Internal Working (How It Works?)

1. **Swaps first and last elements**, second and second-last, and so on.
2. **Uses `O(n)` time complexity** (single pass).

📌 Example: Reversing a List

```
import java.util.*;

class ReverseExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

        // Reversing the list
        Collections.reverse(numbers);

        System.out.println("Reversed List: " + numbers);
    }
}
```

🔹 Output:

```
Reversed List: [5, 4, 3, 2, 1]
```

📌 Example: Reversing a List of Strings

```
import java.util.*;

class ReverseStringList {
    public static void main(String[] args) {
        List<String> words = new ArrayList<>(Arrays.asList("Hello", "World",
"Java"));

        // Reversing the list
        Collections.reverse(words);

        System.out.println("Reversed Words: " + words);
    }
}
```

🔹 Output:

```
Reversed Words: [Java, World, Hello]
```

📌 Time Complexity

- `O(n)`, as it swaps **half** the elements in a **single pass**.

📌 Summary Table

Method	Purpose	Time Complexity
`Collections.unmodifiableList(List<T>)`	Creates a read-only list (modification not allowed).	`O(1)`
`Collections.synchronizedList(List<T>)`	Creates a thread-safe list.	`O(1)` (method calls may take extra time)
`Collections.reverse(List<T>)`	Reverses the order of elements in a list.	`O(n)`

Deep Dive into Next Three Methods of `Collections` Class

☑ `Collections.fill(List<T>, T value)`
☑ `Collections.copy(List<T> dest, List<T> src)`
☑ `Collections.replaceAll(List<T> list, T oldValue, T newValue)`

We'll cover:

- **What the method does**
- **How it works internally**
- **Code examples**
- **Time complexity**
- **Real-world use cases**

1️⃣ `Collections.fill(List<T>, T value)` (Fill List with a Single Value)

📌 What It Does?

- Replaces **all elements** of a list with a **single value**.
- **Modifies the original list**.
- Works only with **modifiable lists** (not immutable lists).

📌 Syntax

```
Collections.fill(List<T> list, T value);
```

- Takes a **list** and a **value**, then replaces **every element** with that value.

📌 Internal Working (How It Works?)

1. Iterates through **each index of the list**.
2. Replaces the **existing value** with the new value.

3. **Time Complexity: O(n)** (as it modifies every element).

📌 Example: Filling a List

```
import java.util.*;

class FillExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

        // Filling the list with 0
        Collections.fill(numbers, 0);

        System.out.println("Filled List: " + numbers);
    }
}
```

🔹 Output:

```
Filled List: [0, 0, 0, 0, 0]
```

✔ **All elements are replaced with `0`.**

📌 Example: Filling a List of Strings

```
import java.util.*;

class FillStringList {
    public static void main(String[] args) {
        List<String> words = new ArrayList<>(Arrays.asList("Java", "Python", "C++"));

        // Filling the list with "Default"
        Collections.fill(words, "Default");

        System.out.println("Filled List: " + words);
    }
}
```

🔹 Output:

```
Filled List: [Default, Default, Default]
```

📌 Real-World Use Case

- Used to **reset a list** with a default value.
- Used in **game development** to **initialize** a list with default scores.

2️⃣ `Collections.copy(List<T> dest, List<T> src)` (Copy Elements from One List to Another)

📌 What It Does?

- Copies **all elements** from the **source list** (`src`) to the **destination list** (`dest`).
- **Destination list must have the same or larger size** as the source list.
- **Modifies the destination list**.

📌 Syntax

```
Collections.copy(List<T> dest, List<T> src);
```

- `dest` should be **at least the same size** as `src`, otherwise it throws `IndexOutOfBoundsException`.

📌 Internal Working (How It Works?)

1. Iterates through the **source list** and copies **each element** to the destination list.
2. **Overwrites** existing elements in `dest`.
3. **Time Complexity: O(n)** (as it processes every element).

📌 Example: Copying One List to Another

```
import java.util.*;

class CopyExample {
    public static void main(String[] args) {
        List<String> src = Arrays.asList("Apple", "Banana", "Cherry");
        List<String> dest = new ArrayList<>(Arrays.asList("X", "Y", "Z"));

        // Copying src to dest
        Collections.copy(dest, src);

        System.out.println("Destination List After Copy: " + dest);
    }
}
```

🔹 Output:

```
Destination List After Copy: [Apple, Banana, Cherry]
```

✔️ The elements in `dest` are **replaced** by elements from `src`.

📌 Important Notes

❌ This **will not work** if `dest` is smaller than `src`:

```
List<String> dest = new ArrayList<>(Arrays.asList("X", "Y")); // Smaller size
Collections.copy(dest, src); // Throws IndexOutOfBoundsException
```

✔ **Solution:** Ensure `dest` has the **same or larger size** before copying.

📌 Real-World Use Case

- Used to **backup lists** before modification.
- Used in **undo-redo features** to maintain copies of lists.

---

3 `Collections.replaceAll(List<T> list, T oldValue, T newValue)` (Replace a Specific Value in a List)

📌 What It Does?

- **Finds and replaces** all occurrences of `oldValue` in a list with `newValue`.
- **Modifies the original list**.

📌 Syntax

```
Collections.replaceAll(List<T> list, T oldValue, T newValue);
```

- Searches for **`oldValue`** and replaces it with **`newValue`** in the list.

📌 Internal Working (How It Works?)

1. Iterates through **each element** in the list.
2. If an element **matches `oldValue`**, it is replaced with `newValue`.
3. **Time Complexity: O(n)** (as it checks every element).

📌 Example: Replacing Elements in a List

```
import java.util.*;

class ReplaceAllExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>(Arrays.asList("Apple", "Banana",
"Apple", "Cherry"));

        // Replacing all "Apple" with "Mango"
        Collections.replaceAll(fruits, "Apple", "Mango");
```

```
        System.out.println("List After Replace: " + fruits);
    }
}
```

🔹 Output:

```
List After Replace: [Mango, Banana, Mango, Cherry]
```

✔ **All occurrences of "Apple" are replaced with "Mango".**

📌 Example: Replacing Numbers in a List

```
import java.util.*;

class ReplaceNumbers {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3, 2, 4));

        // Replacing all 2s with 99
        Collections.replaceAll(numbers, 2, 99);

        System.out.println("List After Replace: " + numbers);
    }
}
```

🔹 Output:

```
List After Replace: [1, 99, 3, 99, 4]
```

📌 Real-World Use Case

- Used to **update outdated values** in lists.
- Used to **change labels or categories** dynamically in applications.

📌 Summary Table

Method	Purpose	Time Complexity
`Collections.fill(List<T>, T value)`	Replaces all elements with a single value.	`O(n)`
`Collections.copy(List<T> dest, List<T> src)`	Copies elements from one list to another.	`O(n)`
`Collections.replaceAll(List<T>, T oldValue, T newValue)`	Replaces occurrences of `oldValue` with `newValue`.	`O(n)`

Deep Dive into Next Three Methods of `Collections` Class

☑ `Collections.shuffle(List<T> list)`
☑ `Collections.rotate(List<T> list, int distance)`
☑ `Collections.swap(List<T> list, int i, int j)`

We'll cover:

- **What the method does**
- **How it works internally**
- **Code examples**
- **Time complexity**
- **Real-world use cases**

1️⃣ `Collections.shuffle(List<T> list)` (Randomly Rearrange Elements in a List)

📌 What It Does?

- **Randomly shuffles** the elements of the list.
- **Changes the order each time** it is called.
- Uses **Java's Random class** internally to generate random indices.

📌 Syntax

```
Collections.shuffle(List<T> list);
```

- Modifies the **original list**.

📌 Internal Working (How It Works?)

1. Uses the **Fisher-Yates algorithm** (modern version of **Knuth shuffle**).
2. **Swaps each element** with a randomly chosen element after it.
3. Uses `java.util.Random` to generate **random indices**.
4. **Time Complexity: O(n)** (as each element is processed once).

📌 Example: Shuffling a List

```
import java.util.*;

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

```
        List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8,
9));

        // Shuffling the list
        Collections.shuffle(numbers);

        System.out.println("Shuffled List: " + numbers);
    }
}
```

🔹 Output (changes each time):

```
Shuffled List: [4, 9, 1, 7, 3, 5, 2, 8, 6]
```

✔ **The order of elements is randomized**.

📌 Example: Shuffling a List of Strings

```
import java.util.*;

class ShuffleStringList {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>(Arrays.asList("Alice", "Bob", "Charlie",
"David"));

        Collections.shuffle(names);

        System.out.println("Shuffled Names: " + names);
    }
}
```

🔹 Output (changes each time):

```
Shuffled Names: [Charlie, Alice, David, Bob]
```

📌 Real-World Use Case

- **Randomizing quiz questions** in an exam application.
- **Shuffling a deck of cards** in a card game.

2️⃣ `Collections.rotate(List<T> list, int distance)` (Rotate Elements in a List)

📌 What It Does?

- Rotates the list by moving elements **rightward** (positive distance) or **leftward** (negative distance).

📌 Syntax

```
Collections.rotate(List<T> list, int distance);
```

- **Positive distance → Right rotation**.
- **Negative distance → Left rotation**.

📌 Internal Working (How It Works?)

1. Uses **modulo arithmetic** (`distance % list.size()`) to optimize movement.
2. **Rightward rotation** shifts elements to the right.
3. **Leftward rotation** (when distance is negative) shifts elements to the left.
4. **Time Complexity: O(n)** (as every element is moved once).

📌 Example: Rotating Right by 2 Places

```
import java.util.*;

class RotateExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

        // Rotating right by 2
        Collections.rotate(numbers, 2);

        System.out.println("Rotated List: " + numbers);
    }
}
```

🔹 Output:

```
Rotated List: [4, 5, 1, 2, 3]
```

✔ **Last two elements move to the front.**

📌 Example: Rotating Left by 2 Places

```
import java.util.*;

class RotateLeftExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

        // Rotating left by 2 (equivalent to rotating right by -2)
        Collections.rotate(numbers, -2);

        System.out.println("Rotated List: " + numbers);
```

```
    }
}
```

🔹 Output:

```
Rotated List: [3, 4, 5, 1, 2]
```

✔ **First two elements move to the end.**

📌 Real-World Use Case

- **Shifting circular queue elements** in task scheduling.
- **Rotating images** in a slideshow.

3️⃣ `Collections.swap(List<T> list, int i, int j)` (Swap Two Elements in a List)

📌 What It Does?

- **Swaps two elements** in a list at given indices `i` and `j`.

📌 Syntax

```
Collections.swap(List<T> list, int i, int j);
```

- Works for **both mutable and immutable lists**.

📌 Internal Working (How It Works?)

1. **Stores the element at index `i` in a temp variable**.
2. **Replaces the element at `i` with the element at `j`**.
3. **Puts the temp value into `j`**.
4. **Time Complexity: O(1)** (only two assignments).

📌 Example: Swapping Two Elements

```
import java.util.*;

class SwapExample {
    public static void main(String[] args) {
        List<String> colors = new ArrayList<>(Arrays.asList("Red", "Green", "Blue",
"Yellow"));

        // Swapping "Green" and "Blue"
```

```
        Collections.swap(colors, 1, 2);

        System.out.println("List After Swap: " + colors);
    }
}
```

🔹 Output:

```
List After Swap: [Red, Blue, Green, Yellow]
```

✔️ **The elements at index 1 and 2 are swapped.**

📌 Real-World Use Case

- **Swapping players in a game leaderboard**.
- **Swapping elements in sorting algorithms (like Bubble Sort)**.

📌 Summary Table

Method	Purpose	Time Complexity
`Collections.shuffle(List<T> list)`	Randomly shuffles elements in a list.	`O(n)`
`Collections.rotate(List<T> list, int distance)`	Moves elements left or right in a list.	`O(n)`
`Collections.swap(List<T> list, int i, int j)`	Swaps two elements in a list.	`O(1)`

Deep Dive into Next Three Methods of `Collections` Class

☑ `Collections.min(Collection<T> coll)`
☑ `Collections.max(Collection<T> coll)`
☑ `Collections.frequency(Collection<T> coll, Object obj)`

We'll cover:

- **What the method does**
- **How it works internally**
- **Code examples**
- **Time complexity**
- **Real-world use cases**

1️⃣ `Collections.min(Collection<T> coll)` (Find the Minimum Element)

📌 What It Does?

- Finds the **smallest element** in a given collection.
- Uses **natural ordering** (`Comparable<T>`).
- Can also work with a **custom comparator**.

📌 Syntax

```
Collections.min(Collection<T> coll);
Collections.min(Collection<T> coll, Comparator<T> comp);
```

- **First version** → Uses **natural sorting order** (`Comparable<T>`).
- **Second version** → Uses a **custom comparator** (`Comparator<T>`).

📌 Internal Working (How It Works?)

1. **Iterates through all elements** of the collection.
2. **Compares elements** using the `compareTo()` method (for `Comparable<T>`) or `compare()` method (for `Comparator<T>`).
3. **Returns the smallest element**.
4. **Time Complexity: O(n)** (since every element is checked once).

📌 Example: Finding Minimum in a List

```
import java.util.*;

class MinExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(45, 12, 78, 34, 23);

        int minNumber = Collections.min(numbers);
        System.out.println("Minimum Number: " + minNumber);
    }
}
```

🔹 Output:

```
Minimum Number: 12
```

✔ **Finds the smallest number from the list.**

📌 Example: Finding Minimum with a Custom Comparator

```
import java.util.*;

class MinCustomExample {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "grape", "mango");

        // Finding the shortest word using a custom comparator
        String minWord = Collections.min(words,
Comparator.comparing(String::length));

        System.out.println("Shortest Word: " + minWord);
    }
}
```

🔹 Output:

```
Shortest Word: grape
```

✔ **Finds the shortest word using a custom comparator.**

📌 Real-World Use Case

- **Finding the lowest price in an e-commerce product list.**
- **Finding the youngest student in a list based on age.**

2️⃣ `Collections.max(Collection<T> coll)` (Find the Maximum Element)

📌 What It Does?

- Finds the **largest element** in a given collection.
- Uses **natural ordering** (`Comparable<T>`).
- Can also work with a **custom comparator**.

📌 Syntax

```
Collections.max(Collection<T> coll);
Collections.max(Collection<T> coll, Comparator<T> comp);
```

- **First version** → Uses **natural sorting order** (`Comparable<T>`).
- **Second version** → Uses a **custom comparator** (`Comparator<T>`).

📌 Internal Working (How It Works?)

1. **Iterates through all elements** of the collection.

2. **Compares elements** using the `compareTo()` method (for `Comparable<T>`) or `compare()` method (for `Comparator<T>`).
3. **Returns the largest element**.
4. **Time Complexity: O(n)** (since every element is checked once).

📌 Example: Finding Maximum in a List

```
import java.util.*;

class MaxExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(45, 12, 78, 34, 23);

        int maxNumber = Collections.max(numbers);
        System.out.println("Maximum Number: " + maxNumber);
    }
}
```

🔹 Output:

```
Maximum Number: 78
```

✔ **Finds the largest number from the list.**

📌 Example: Finding Maximum with a Custom Comparator

```
import java.util.*;

class MaxCustomExample {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "grape", "mango");

        // Finding the longest word using a custom comparator
        String maxWord = Collections.max(words,
Comparator.comparing(String::length));

        System.out.println("Longest Word: " + maxWord);
    }
}
```

🔹 Output:

```
Longest Word: banana
```

✔ **Finds the longest word using a custom comparator.**

📌 Real-World Use Case

- **Finding the highest salary from an employee list.**

- **Finding the most expensive product in an inventory.**

3️⃣ `Collections.frequency(Collection<T> coll, Object obj)` (Count Occurrences of an Element)

📌 What It Does?

- Counts how many times an element **appears** in a collection.

📌 Syntax

```
Collections.frequency(Collection<T> coll, Object obj);
```

- Returns an **integer** representing the **count** of `obj` in `coll`.

📌 Internal Working (How It Works?)

1. **Iterates through the entire collection**.
2. **Compares each element** with `obj` using `.equals()` method.
3. **Increments count** for each match.
4. **Returns total count**.
5. **Time Complexity: O(n)** (since every element is checked once).

📌 Example: Counting Frequency of a Number

```
import java.util.*;

class FrequencyExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 2, 4, 2, 5, 2);

        int count = Collections.frequency(numbers, 2);
        System.out.println("Frequency of 2: " + count);
    }
}
```

🔹 Output:

```
Frequency of 2: 4
```

✔ **Counts how many times `2` appears in the list.**

📌 Example: Counting Frequency of a String

```
import java.util.*;

class FrequencyStringExample {
    public static void main(String[] args) {
        List<String> colors = Arrays.asList("red", "blue", "green", "red", "yellow",
"red");

        int count = Collections.frequency(colors, "red");
        System.out.println("Frequency of 'red': " + count);
    }
}
```

🔹 Output:

```
Frequency of 'red': 3
```

✔ **Counts occurrences of the string `"red"`.**

📌 Real-World Use Case

- **Counting the number of times a word appears in a text file.**
- **Finding the most frequently purchased product in an e-commerce application.**

📌 Summary Table

Method	Purpose	Time Complexity
`Collections.min(Collection<T> coll)`	Finds the smallest element.	`O(n)`
`Collections.max(Collection<T> coll)`	Finds the largest element.	`O(n)`
`Collections.frequency(Collection<T> coll, Object obj)`	Counts occurrences of an element.	`O(n)`

Deep Dive into Next Three Methods of `Collections` Class

☑ `Collections.fill(List<T> list, T obj)`
☑ `Collections.replaceAll(List<T> list, T oldVal, T newVal)`
☑ `Collections.copy(List<T> dest, List<T> src)`

We will go step by step:

- **What the method does**
- **How it works internally**
- **Code examples**

- **Time complexity**
- **Real-world use cases**

1️⃣ `Collections.fill(List<T> list, T obj)` (Replace All Elements with One Value)

📌 What It Does?

- Replaces **all elements** of the list with the **same value**.
- Useful when we want to **reset** or **initialize** a list with a default value.

📌 Syntax

```
Collections.fill(List<T> list, T obj);
```

- `list` → The list to be modified.
- `obj` → The object to set in all positions.

🚨 **Important:** The `list` **must be mutable** (modifiable), otherwise it throws an exception!

📌 Internal Working (How It Works?)

1. **Iterates through each index** of the list.
2. **Replaces each element** with `obj`.
3. **Returns nothing** (modifies the list directly).
4. **Time Complexity: O(n)** (since every element is updated once).

📌 Example: Filling a List with a Default Value

```
import java.util.*;

class FillExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

        Collections.fill(numbers, 0);
        System.out.println("List after fill: " + numbers);
    }
}
```

🔹 Output:

```
List after fill: [0, 0, 0, 0, 0]
```

✔️ **All elements are replaced with `0`.**

📌 Real-World Use Case

- **Resetting a list** (e.g., clearing marks in a survey).
- **Initializing a list** (e.g., filling a list with `null` values in a cache).

2️⃣ `Collections.replaceAll(List<T> list, T oldVal, T newVal)` (Replace Specific Elements)

📌 What It Does?

- **Finds all occurrences** of `oldVal` in the list and **replaces them** with `newVal`.
- **Does NOT change** elements that do not match `oldVal`.

📌 Syntax

```
Collections.replaceAll(List<T> list, T oldVal, T newVal);
```

- `list` → The list to modify.
- `oldVal` → The value to replace.
- `newVal` → The new value to set.

📌 Internal Working (How It Works?)

1. **Iterates through the list** to find occurrences of `oldVal`.
2. **If a match is found**, it **replaces** it with `newVal`.
3. **Returns nothing** (modifies the list directly).
4. **Time Complexity: O(n)** (each element is checked once).

📌 Example: Replacing All `2`s with `99`

```
import java.util.*;

class ReplaceAllExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3, 2, 4, 2, 5));

        Collections.replaceAll(numbers, 2, 99);
        System.out.println("List after replaceAll: " + numbers);
    }
}
```

🔹 Output:

```
List after replaceAll: [1, 99, 3, 99, 4, 99, 5]
```

✔️ **All 2s are replaced with 99.**

📌 Example: Replacing Words in a List

```
import java.util.*;

class ReplaceAllStringExample {
    public static void main(String[] args) {
        List<String> words = new ArrayList<>(Arrays.asList("apple", "banana",
"apple", "grape"));

        Collections.replaceAll(words, "apple", "mango");
        System.out.println("List after replaceAll: " + words);
    }
}
```

🔹 Output:

```
List after replaceAll: [mango, banana, mango, grape]
```

✔️ **All `"apple"` entries are replaced with `"mango"`.**

📌 Real-World Use Case

- **Replacing censored words** in a list of comments.
- **Updating incorrect data** (e.g., replacing a misspelled name).

3️⃣ `Collections.copy(List<T> dest, List<T> src)` (Copy One List into Another)

📌 What It Does?

- Copies **all elements** from `src` (source) to `dest` (destination).
- 🚨 **The destination list (`dest`) must have the same size or larger than the source (`src`)!**

📌 Syntax

```
Collections.copy(List<T> dest, List<T> src);
```

- `dest` → The list that will receive the copied elements.
- `src` → The list from which elements are copied.

📌 Internal Working (How It Works?)

1. **Checks that `dest` has enough space** (throws an exception if not).
2. **Iterates through `src`** and copies each element to `dest`.
3. **Modifies `dest` in-place** (returns nothing).
4. **Time Complexity: O(n)** (since each element is copied once).

📌 Example: Copying One List into Another

```
import java.util.*;

class CopyExample {
    public static void main(String[] args) {
        List<Integer> src = Arrays.asList(1, 2, 3, 4, 5);
        List<Integer> dest = new ArrayList<>(Arrays.asList(0, 0, 0, 0, 0));

        Collections.copy(dest, src);
        System.out.println("Destination List after copy: " + dest);
    }
}
```

🔹 Output:

```
Destination List after copy: [1, 2, 3, 4, 5]
```

✔️ **`dest` now contains all elements from `src`.**

🚨 **Important:**
If `dest` has fewer elements than `src`, you will get an `IndexOutOfBoundsException`.
So, always ensure `dest` has **at least the same size** as `src`.

📌 Real-World Use Case

- **Copying user settings** from one list to another.
- **Backing up a list before making changes.**

📌 Summary Table

Method	Purpose	Time Complexity
`Collections.fill(List<T>, T)`	Replaces all elements with a single value.	`O(n)`
`Collections.replaceAll(List<T>, T, T)`	Replaces all occurrences of a value.	`O(n)`

`Collections.copy(List<T>, List<T>)`	Copies one list into another.	`O(n)`

📌 Chapter 9: Thread-Safety in Java Collections

🔹 What is Thread-Safety in Java Collections?

- **Thread-Safety** means that **multiple threads** can access a collection **without causing data inconsistency** or unexpected behavior.
- In Java, **normal collections like `ArrayList`, `HashSet`, and `HashMap` are NOT thread-safe** because multiple threads can modify them at the same time, leading to **race conditions.**
- Java provides **two solutions** for thread-safe collections:
    1. **Synchronized Collections** (Older Approach)
    2. **Concurrent Collections** (Modern Approach)

🔹 1️⃣ Synchronized Collections (Old Approach)

Java provides synchronized versions of collections using **`Collections.synchronizedXXX()`** methods.

📌 Example: Synchronized List

```
import java.util.*;

class SynchronizedListExample {
    public static void main(String[] args) {
        List<Integer> list = Collections.synchronizedList(new ArrayList<>());

        list.add(1);
        list.add(2);
        list.add(3);

        synchronized (list) {  // Required for safe iteration
            for (int num : list) {
                System.out.println(num);
            }
        }
    }
}
```

✔ **Problems with Synchronized Collections:**

- **Slow Performance:** Because it locks the entire collection.
- **Manual Synchronization Required:** You must manually synchronize while iterating (`synchronized` block).
- **Better Alternative? ☑ Use Concurrent Collections!**

🔹 2️⃣ Concurrent Collections (Modern Approach)

Java introduced the `java.util.concurrent` package to provide **faster and better thread-safe collections.**

🚀 **Key Concurrent Collections:**

Collection	Type	Feature
`CopyOnWriteArrayList`	**List**	**Thread-Safe ArrayList** (No Manual Synchronization Needed)
`CopyOnWriteArraySet`	**Set**	**Thread-Safe HashSet** (Works Like CopyOnWriteArrayList)
`ConcurrentHashMap`	**Map**	**Thread-Safe HashMap** (Uses Lock Stripes)
`ConcurrentSkipListSet`	**Set**	**Thread-Safe Sorted Set**
`ConcurrentSkipListMap`	**Map**	**Thread-Safe Sorted Map**

🔹 3️⃣ CopyOnWriteArrayList (Thread-Safe ArrayList)

📌 **What is it?**

- It is a **thread-safe version** of `ArrayList` that allows multiple threads to read the list **without locking**.
- **Whenever you modify it (add, remove, update), it creates a new copy of the list!**

📌 **When to Use?**

- When **reads are more frequent** than writes (e.g., a list of online users in a chat app).

Example: CopyOnWriteArrayList

```
import java.util.concurrent.CopyOnWriteArrayList;

class CopyOnWriteArrayListExample {
    public static void main(String[] args) {
        CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>();

        list.add(1);
        list.add(2);
        list.add(3);

        for (Integer num : list) {  // No need to manually synchronize
            System.out.println(num);
        }
    }
}
```

✔️ **Advantages:**

- **Thread-Safe without Locks** (Multiple threads can read at the same time).
- **No ConcurrentModificationException** (Unlike `ArrayList`, which throws errors during modification).

✔ **Disadvantages:**

- **Memory Overhead** (Creates a new copy every time you modify it).

🔹 4️⃣ CopyOnWriteArraySet (Thread-Safe HashSet)

- It is a **thread-safe version of** `HashSet` and works **just like** `CopyOnWriteArrayList`**.**
- **Each write operation (add/remove) creates a new copy of the set.**

📌 **Example:**

```
import java.util.concurrent.CopyOnWriteArraySet;

class CopyOnWriteArraySetExample {
    public static void main(String[] args) {
        CopyOnWriteArraySet<Integer> set = new CopyOnWriteArraySet<>();

        set.add(10);
        set.add(20);
        set.add(30);

        for (Integer num : set) {
            System.out.println(num);
        }
    }
}
```

✔ **Advantage:** No need for manual synchronization.
✔ **Disadvantage: Slower writes** due to copy creation.

🔹 5️⃣ ConcurrentHashMap (Thread-Safe HashMap)

📌 **What is it?**

- A **thread-safe version of** `HashMap` that allows **fast reads and writes using lock stripping.**
- Instead of locking the **entire map**, it locks **only specific parts (buckets).**

📌 **Example:**

```
import java.util.concurrent.ConcurrentHashMap;

class ConcurrentHashMapExample {
    public static void main(String[] args) {
        ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();

        map.put(1, "A");
```

```
        map.put(2, "B");
        map.put(3, "C");

        for (Integer key : map.keySet()) {
            System.out.println(key + " -> " + map.get(key));
        }
    }
}
```

✔ **Advantages:**

- **Faster than** `Hashtable` (does not lock entire map).
- **No ConcurrentModificationException** (safe for multi-threading).

🔹 6️⃣ ConcurrentSkipListSet (Thread-Safe Sorted Set)

- A **thread-safe version of** `TreeSet` (keeps elements sorted).
- Uses a **Skip List data structure** instead of a Red-Black tree.

📌 **Example:**

```
import java.util.concurrent.ConcurrentSkipListSet;

class ConcurrentSkipListSetExample {
    public static void main(String[] args) {
        ConcurrentSkipListSet<Integer> set = new ConcurrentSkipListSet<>();

        set.add(30);
        set.add(10);
        set.add(20);

        for (Integer num : set) {
            System.out.println(num);
        }
    }
}
```

✔ **Advantage:** Automatically **keeps elements sorted** while being **thread-safe.**

🔹 7️⃣ Performance Comparison: Synchronized vs Concurrent Collections

Collection	Thread-Safety Type	Performance
`Collections.synchronizedList()`	Full Locking	🚨 **Slow (Locks Entire Collection)**
`CopyOnWriteArrayList`	No Lock for Read	☑ **Fast Reads, Slow Writes**
`ConcurrentHashMap`	Partial Locking	☑ **Fast Read & Write**

`Hashtable`	Full Locking	🚨 **Slow (Locks Whole Table)**

✔ **Best Choice?**

- **Use `ConcurrentHashMap` instead of `Hashtable`** for better performance.
- **Use `CopyOnWriteArrayList` for thread-safe lists with frequent reads.**

📌 Summary

Collection	Type	Thread-Safe?	Best For
`CopyOnWriteArrayList`	**List**	☑ Yes	**Frequent Reads, Rare Writes**
`CopyOnWriteArraySet`	**Set**	☑ Yes	**Frequent Reads, Rare Writes**
`ConcurrentHashMap`	**Map**	☑ Yes	**High-Performance Thread-Safe Map**
`ConcurrentSkipListSet`	**Set**	☑ Yes	**Sorted Set in Multi-threading**

🚀 **Conclusion:**

- Use **Concurrent Collections** instead of `synchronized` collections for **better performance.**
- Choose **`CopyOnWriteArrayList` for frequent reads** and **`ConcurrentHashMap` for multi-threaded key-value storage.**

📌 Chapter 10: Best Practices and Performance Optimization

🔹 What is Best Practices and Performance Optimization in Collections?

Java Collections Framework provides a **wide range of data structures** to store and manipulate data efficiently. However, **using them correctly** is **crucial** for writing efficient, maintainable, and high-performance code.

📌 **Best practices** help you avoid common pitfalls, reduce errors, and make your code **clean and maintainable.**
📌 **Performance optimization** ensures your collections work **efficiently**, using the least memory and CPU power.

🔹 Why is Performance Optimization Important?

- Collections are **used everywhere** in Java applications (e.g., lists, sets, maps).
- **Poor choice of collection** can **slow down your application** significantly.
- **Incorrect usage** can lead to **memory leaks, unnecessary CPU usage, and crashes.**

- **Choosing the right collection** improves speed and reduces resource usage.

🔹 Best Practices for Java Collections

1️⃣ Choose the Right Collection for the Right Use Case

☑ **Use `ArrayList` when you need fast retrieval and random access.**
☑ **Use `LinkedList` when you need frequent insertions/deletions.**
☑ **Use `HashSet` when unique elements are required and order doesn't matter.**
☑ **Use `TreeSet` when unique elements are needed in sorted order.**
☑ **Use `HashMap` for fast key-value lookups.**
☑ **Use `Concurrent Collections` for multi-threading instead of synchronized collections.**

2️⃣ Prefer Immutable Collections When Possible

- If a collection **does not need to change**, use **unmodifiable collections** to prevent accidental modifications.
- Java provides `Collections.unmodifiableList()`, `Collections.unmodifiableSet()`, and `Collections.unmodifiableMap()`.

```
import java.util.*;

class ImmutableCollectionExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
        List<String> immutableList = Collections.unmodifiableList(list);

        immutableList.add("D"); // This will throw UnsupportedOperationException
    }
}
```

✔ **Advantage:** Prevents accidental modification, making the code more **secure** and **predictable**.

3️⃣ Minimize Unnecessary Autoboxing and Unboxing

- Java automatically **converts primitives** (int, double, etc.) into their wrapper classes (`Integer`, `Double`), which **causes performance overhead.**
- **Always prefer primitive collections** like `int[]` over `List<Integer>` if boxing/unboxing is unnecessary.

```
List<Integer> list = new ArrayList<>();  // Slower, due to autoboxing
list.add(10); // Converts int to Integer

int num = list.get(0); // Unboxes Integer to int
```

✔ **Solution:** Use `IntStream` or `Arrays` for primitive values instead of collections.

4️⃣ Avoid Memory Leaks with Collections

- **Problem:** If you keep adding elements but **never remove them**, memory usage will increase indefinitely.
- **Solution:** Always **clear large collections** when they are no longer needed.

```
List<String> list = new ArrayList<>();
list.add("data1");
list.add("data2");

// Clear collection when not needed
list.clear();
```

✔ **Advantage:** Reduces memory footprint and avoids **OutOfMemoryError**.

5️⃣ Use `computeIfAbsent()` for Efficient Map Updates

Instead of checking for null manually, use `computeIfAbsent()` to optimize adding values to a `Map`.

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

class ComputeIfAbsentExample {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();

        // Instead of checking manually, use computeIfAbsent
        map.computeIfAbsent("A", key -> 10);
        map.computeIfAbsent("B", key -> 20);

        System.out.println(map); // {A=10, B=20}
    }
}
```

✔ **Advantage:** Reduces redundant `if-else` checks and improves readability.

6️⃣ Use Streams and Parallel Processing for Large Collections

- Instead of **looping manually**, use **Java Streams API** for better performance.
- **Parallel Streams** can be used to **process large datasets faster** using multiple CPU cores.

```
import java.util.*;
import java.util.stream.Collectors;

class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Convert all numbers to square using Streams
```

```
        List<Integer> squares = numbers.stream()
                                       .map(n -> n * n)
                                       .collect(Collectors.toList());

        System.out.println(squares); // [1, 4, 9, 16, 25]
    }
}
```

✔ **Advantage:** Faster, more readable, and concise compared to traditional loops.

7️⃣ Use `containsKey()` Instead of `get()` for Maps

- Using `map.get(key) != null` can be **slower** than directly checking with `containsKey()`.
- **Best practice: Use `containsKey()` before calling `get()`.**

```
Map<String, Integer> map = new HashMap<>();

if (map.containsKey("A")) {
    System.out.println(map.get("A"));
}
```

✔ **Advantage:** Improves performance in large maps.

📌 When to Use Which Collection? (Deep Explanation)

Choosing the right **Java Collection** is crucial for building **efficient** and **high-performing** applications. The **wrong choice** can lead to **slow performance, memory issues, and unnecessary complexity**.

In this section, we will **deeply analyze** when to use each **List, Set, Queue, and Map** based on different use cases.

🔹 List Interface: When to Use?

A **List** is an **ordered collection** that **allows duplicate elements**.
Use a `List<T>` when:

✔ You need to maintain **insertion order**.
✔ You need **indexed access** (access elements by position).
✔ You need to allow **duplicates**.

1️⃣ `ArrayList<T>` – Fast Retrieval, Slow Insert/Delete

- **Best for: Read-heavy applications where elements are accessed frequently.**
- **Avoid if:** You need frequent **insertions/deletions in the middle**.

Operation	Time Complexity

Access (get(index))	O(1) ☑ (Super fast)
Insert (add at end)	O(1) ☑
Insert/Delete in middle	O(n) ✗ (Slow shifting required)

📌 **When to Use?**

✔ When you need **fast random access** to elements using indexes.

✔ When the **insertion order** should be maintained.

✔ Example: **Reading customer reviews, fetching product lists in an e-commerce website.**

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
names.add("Charlie");
System.out.println(names.get(1)); // Output: Bob
```

---

2 `LinkedList<T>` – Fast Insert/Delete, Slow Access

🔹 **Best for: Insert/delete-heavy applications.**

🔹 **Avoid if:** You need frequent **random access (get(index))**.

Operation	Time Complexity
Access (get(index))	O(n) ✗ (Slow, must traverse the list)
Insert/Delete in middle	O(1) ☑ (Just update pointers)

📌 **When to Use?**

✔ When you frequently **insert/delete elements in the middle**.

✔ Example: **Implementing undo/redo feature, task schedulers.**

```
List<String> tasks = new LinkedList<>();
tasks.add("Task 1");
tasks.add("Task 2");
tasks.add(1, "New Task in between");
System.out.println(tasks);
```

---

3 `Vector<T>` – Thread-Safe, But Rarely Used

🔹 **Best for: Thread-safe operations (legacy, use Concurrent collections instead).**

🔹 **Avoid if:** You don't need synchronization.

Operation	Time Complexity
Access (get(index))	O(1) ☑
Insert/Delete in middle	O(n) ✗ (Slow shifting required)

Thread-Safety	Yes ☑

📌 **When to Use?**

✔ When you need a **synchronized** version of an `ArrayList`.

✔ Example: **Multi-threaded application needing synchronized list.**

4️⃣ `Stack<T>` – Last-In-First-Out (LIFO)

- **Best for: Undo/Redo, Backtracking, Expression Evaluation.**
- **Avoid if:** You need **random access** to elements.

Operation	Time Complexity
Push (add element)	O(1) ☑
Pop (remove last element)	O(1) ☑

📌 **When to Use?**

✔ When you need **LIFO behavior**.

✔ Example: **Undo feature in text editors, evaluating expressions.**

```
Stack<Integer> stack = new Stack<>();
stack.push(10);
stack.push(20);
System.out.println(stack.pop()); // Output: 20 (LIFO)
```

🔹 Set Interface: When to Use?

A **Set** is a collection that **does not allow duplicate elements**.

Use a `Set<T>` when:

✔ You need **unique elements only**.

✔ You don't care about **insertion order** (except `LinkedHashSet`).

✔ You need **fast lookups**.

1️⃣ `HashSet<T>` – Fastest Set for Unordered Unique Elements

- **Best for: High-performance unique element storage.**
- **Avoid if:** You need to maintain order.

Operation	Time Complexity
Insert/Delete/Search	O(1) ☑

📌 **When to Use?**

✔ When you need **unique elements with fast performance**.

✔ Example: **Removing duplicate usernames in a system.**

```
Set<String> users = new HashSet<>();
users.add("Alice");
users.add("Bob");
users.add("Alice"); // Duplicate ignored
System.out.println(users); // Output: [Alice, Bob]
```

---

2 `LinkedHashSet<T>` – Maintains Insertion Order

- **Best for: Unique elements + maintaining order.**
- **Avoid if:** Order doesn't matter.

Operation	Time Complexity
Insert/Delete/Search	O(1) ☑

📌 **When to Use?**

✔ When you need **unique elements but order matters**.

✔ Example: **Maintaining a unique list of visited pages in a browser.**

```
Set<String> pages = new LinkedHashSet<>();
pages.add("Home");
pages.add("About");
pages.add("Contact");
System.out.println(pages); // Output: [Home, About, Contact]
```

---

3 `TreeSet<T>` – Sorted Unique Elements

- **Best for: Sorted unique elements (ascending order by default).**
- **Avoid if:** You don't need sorting (Use `HashSet` instead).

Operation	Time Complexity
Insert/Delete/Search	O(log n) ✖ (Slower than HashSet)

📌 **When to Use?**

✔ When you need **unique elements in sorted order**.

✔ Example: **Storing sorted employee IDs.**

```
Set<Integer> ids = new TreeSet<>();
ids.add(3);
ids.add(1);
ids.add(2);
System.out.println(ids); // Output: [1, 2, 3]
```

🔹 Map Interface: When to Use?

A **Map** stores **key-value pairs** for fast lookups.
Use a `Map<K, V>` when:

✔ You need to **map unique keys to values**.
✔ You need **fast lookups by key**.

1 `HashMap<K, V>` – Fastest Key-Value Lookup (Unordered)

- **Best for: Fast key-value storage.**
- **Avoid if:** You need sorted order.

Operation	Time Complexity
Insert/Delete/Search	O(1) ☑

```
Map<String, Integer> map = new HashMap<>();
map.put("Alice", 25);
map.put("Bob", 30);
System.out.println(map.get("Alice")); // Output: 25
```

---

2 `TreeMap<K, V>` – Sorted Key-Value Mapping

- **Best for: Sorted key-value pairs.**
- **Avoid if:** Sorting is unnecessary.

Operation	Time Complexity
Insert/Delete/Search	O(log n) ✗ (Slower than HashMap)

```
Map<Integer, String> treeMap = new TreeMap<>();
treeMap.put(2, "B");
treeMap.put(1, "A");
System.out.println(treeMap); // Output: {1=A, 2=B}
```

◆ Conclusion

Collection Type	Best For
`ArrayList<T>`	Fast access, slow insert/delete
`LinkedList<T>`	Fast insert/delete, slow access
`HashSet<T>`	Fast unique elements (unordered)
`TreeSet<T>`	Unique sorted elements
`HashMap<K,V>`	Fastest key-value storage
`TreeMap<K,V>`	Sorted key-value pairs

📌 Performance Considerations for Different Data Structures (Deep Explanation)

Choosing the **right data structure** is not just about functionality—it's also about **performance**.
Each **Collection** has different **strengths and weaknesses** depending on **time complexity, memory usage, and threading support**.

In this section, we will analyze the **performance of Lists, Sets, Queues, and Maps** in **depth** and compare their operations.

🔹 Understanding Performance Factors

The performance of a data structure depends on:

✔ **Time Complexity** - How fast the operations (insert, search, delete) are.
✔ **Memory Usage** - How much space the data structure consumes.
✔ **Thread-Safety** - Whether it supports multi-threading.
✔ **Sorting Needs** - Whether elements are sorted automatically.

🔹 Performance Analysis of List Implementations

1️⃣ `ArrayList<T>` – Fast Random Access, Slow Insert/Delete

✔ **Best For: Fast read-heavy operations**
❌ **Avoid If: Frequent insert/delete in the middle**

Operation	Time Complexity	Explanation
Access (get(i))	O(1) ☑	Direct index-based lookup.
Insert at End	O(1) ☑	If capacity allows, it's instant.
Insert in Middle	O(n) ❌	All elements after must shift.
Remove by Index	O(n) ❌	Elements shift left to fill gap.
Memory Usage	Medium	Uses contiguous memory.

📌 **Performance Tip:** Use **ArrayList** when you need **fast lookups** and **less insertion/deletion**.

2️⃣ `LinkedList<T>` – Fast Insert/Delete, Slow Access

✔ **Best For: Insert/delete-heavy operations**
❌ **Avoid If: Frequent random access needed**

Operation	Time Complexity	Explanation

Access (get(i))	O(n) ❌	Must traverse nodes one by one.
Insert at End	O(1) ☑	Just update last node's pointer.
Insert in Middle	O(1) ☑	If node reference is known.
Remove by Index	O(1) ☑	Just update pointers.
Memory Usage	High ❌	Stores extra pointers (next/prev).

📌 **Performance Tip:** Use **LinkedList** when you need **frequent insertions/deletions** and **don't need fast random access**.

3️⃣ `Vector<T>` – Thread-Safe but Slower than ArrayList

✔ **Best For: Multi-threaded applications requiring a List**
❌ **Avoid If: Single-threaded applications (use ArrayList instead)**

Operation	Time Complexity	Explanation
Access (get(i))	O(1) ☑	Same as ArrayList.
Insert at End	O(1) ☑	Same as ArrayList.
Insert in Middle	O(n) ❌	Shifting needed.
Thread-Safety	Yes ☑	Uses synchronization (slower).

📌 **Performance Tip:** Use **Vector** only if **synchronization is needed**, otherwise prefer **ArrayList**.

4️⃣ `Stack<T>` – LIFO Performance

✔ **Best For: Last-In-First-Out (LIFO) operations**
❌ **Avoid If: Random access is needed**

Operation	Time Complexity	Explanation
Push (add)	O(1) ☑	Just add at the top.
Pop (remove top)	O(1) ☑	Remove top element only.
Search (contains)	O(n) ❌	Must check each element.

📌 **Performance Tip:** Use **Stack** only for **LIFO-based operations** like **undo/redo**.

🔹 Performance Analysis of Set Implementations

1️⃣ `HashSet<T>` – Fastest Unique Element Storage

✔ **Best For: Fast unique element storage**
❌ **Avoid If: Sorting is required**

Operation	Time Complexity	Explanation
Insert/Delete	O(1) ☑	Uses **hashing** for quick access.
Search (contains)	O(1) ☑	Hash lookup is very fast.
Sorting	Not Supported ❌	No order maintained.

📌 **Performance Tip:** Use **HashSet** when you need **unique elements with fast lookups**.

2️⃣ `TreeSet<T>` – Unique + Sorted

✔ **Best For: Maintaining unique elements in sorted order**
❌ **Avoid If: You don't need sorting**

Operation	Time Complexity	Explanation
Insert/Delete	O(log n) ❌	Uses **Red-Black Tree** for sorting.
Search (contains)	O(log n) ❌	Must traverse the tree.
Sorting	Yes ☑	Elements are always sorted.

📌 **Performance Tip:** Use **TreeSet** when you need **sorting** but can accept **slightly slower performance**.

🔹 Performance Analysis of Queue Implementations

1️⃣ `PriorityQueue<T>` – Min-Heap Implementation

✔ **Best For: Processing elements based on priority**
❌ **Avoid If: You need FIFO behavior**

Operation	Time Complexity	Explanation
Insert (add)	O(log n) ❌	Maintains heap property.
Remove (poll)	O(log n) ❌	Heap must be restructured.
Peek (min element)	O(1) ☑	Fast access to smallest element.

📌 **Performance Tip:** Use **PriorityQueue** for **task scheduling, job processing, etc.**

🔹 Performance Analysis of Map Implementations

1️⃣ `HashMap<K, V>` – Fastest Key-Value Storage

✔ **Best For: Fast key-value lookup**
❌ **Avoid If: Sorting is needed**

Operation	Time Complexity	Explanation
Insert/Delete	O(1) ☑	Uses **hashing** for fast access.
Search (containsKey)	O(1) ☑	Direct hash lookup.
Sorting	Not Supported ❌	Unordered storage.

📌 **Performance Tip:** Use **HashMap** for **fast key-based lookups**.

2️⃣ `TreeMap<K, V>` – Sorted Key-Value Mapping

✔ **Best For: Maintaining sorted keys**
❌ **Avoid If: Sorting is unnecessary**

Operation	Time Complexity	Explanation
Insert/Delete	O(log n) ❌	Uses **Red-Black Tree**.
Search (containsKey)	O(log n) ❌	Tree traversal needed.
Sorting	Yes ☑	Always sorted.

📌 **Performance Tip:** Use **TreeMap** when you need **sorted key-value pairs**.

🔹 Conclusion: Choosing the Best Data Structure

Requirement	Best Choice
Fast Read (index-based access)	`ArrayList` ☑
Frequent Insert/Delete	`LinkedList` ☑
Unique Elements (Fast Access)	`HashSet` ☑
Unique Elements (Sorted)	`TreeSet` ☑
Fast Key-Value Storage	`HashMap` ☑
Sorted Key-Value Mapping	`TreeMap` ☑

FIFO Processing	`Queue` ☑
LIFO Processing	`Stack` ☑

📌 Avoiding NullPointerException in Collections (Deep and Easy Explanation)

A **NullPointerException (NPE)** occurs when you try to **access a method or property of a `null` object**.
In **Java Collections**, NPEs often happen when:

- ✔ You try to **add null values** into a collection that **doesn't support nulls** (e.g., `TreeSet`, `TreeMap`).
- ✔ You try to **access an element from a null collection**.
- ✔ You forget to **initialize a collection before using it**.
- ✔ You remove elements without checking if the collection is empty.

🔹 Common Scenarios Where NullPointerException Happens in Collections

1️⃣ Using a Null Collection Reference

📌 **Problem:** Trying to access or modify a collection that is not initialized.

```
List<String> list = null;
list.add("Hello"); // ❌ NullPointerException! list is null
```

☑ **Solution:** Always initialize collections before use.

```
List<String> list = new ArrayList<>(); // ☑ Safe initialization
list.add("Hello");
```

2️⃣ Adding Null Values into a Collection that Doesn't Allow Nulls

📌 **Problem:** Some collections do **not** allow `null` values.

```
Set<String> treeSet = new TreeSet<>();
treeSet.add(null); // ❌ NullPointerException! TreeSet does not allow nulls
```

☑ **Solution:** Use `HashSet` or `ArrayList` if `null` values are needed.

```
Set<String> hashSet = new HashSet<>();
hashSet.add(null); // ☑ Allowed in HashSet
```

3️⃣ Accessing a Null Element in a Collection

📌 **Problem:** Getting an element that is `null` and then calling a method on it.

```
List<String> names = new ArrayList<>();
names.add(null);

System.out.println(names.get(0).length()); // ❌ NullPointerException!
```

☑ **Solution:** Always check for `null` before using an element.

```
if (names.get(0) != null) {
    System.out.println(names.get(0).length()); // ☑ Safe
}
```

4️ Forgetting to Handle Null Return Values

📌 **Problem:** Some map methods return `null` if the key is not found.

```
Map<String, String> map = new HashMap<>();
String value = map.get("key"); // May return null

System.out.println(value.length()); // ❌ NullPointerException!
```

☑ **Solution:** Use `getOrDefault()` or check for `null`.

```
String value = map.getOrDefault("key", "Default");
System.out.println(value.length()); // ☑ Safe

// OR
if (value != null) {
    System.out.println(value.length());
}
```

5️ Using an Empty Collection Instead of Null

📌 **Problem:** Returning `null` from a method instead of an empty collection.

```
public List<String> getNames() {
    return null; // ❌ Bad practice
}

List<String> names = getNames();
System.out.println(names.size()); // ❌ NullPointerException!
```

☑ **Solution:** Return an **empty collection** instead of `null`.

```
public List<String> getNames() {
    return new ArrayList<>(); // ☑ Good practice
}

List<String> names = getNames();
System.out.println(names.size()); // ☑ Works fine (prints 0)
```

6️⃣ Checking for Null Before Removing Elements

📌 **Problem:** Trying to remove elements from a `null` collection.

```
List<String> list = null;
list.remove("Hello"); // ❌ NullPointerException!
```

☑️ **Solution:** Check for `null` before removing elements.

```
if (list != null) {
    list.remove("Hello"); // ☑️ Safe
}
```

🔹 Best Practices to Avoid NullPointerException in Collections

✔️ **Always initialize collections before use** (`new ArrayList<>();`).
✔️ **Use `getOrDefault()` for Maps** instead of directly using `get()`.
✔️ **Check for `null` before accessing or modifying collections**.
✔️ **Return empty collections instead of `null`** in methods.
✔️ **Prefer `Optional<T>` for return values that may be `null`**.

📌 Optimizing Memory and CPU Usage in Collections (Deep & Easy Explanation)

Java collections are powerful, but if **not used efficiently**, they can consume **more memory and CPU** than necessary.
To improve **performance**, follow these **best practices** to optimize **memory usage and processing speed**.

🔹 1️⃣ Choose the Right Collection Type

Using the **wrong collection type** leads to **high memory usage** and **slow performance**.

💡 Example: Using `ArrayList` vs. `LinkedList`

📌 If **more searching is needed**, use `ArrayList` because it supports **fast index-based access**.
📌 If **frequent insertions/deletions** happen, use `LinkedList`, because it avoids shifting elements.

```
List<Integer> arrayList = new ArrayList<>(); // ☑️ Best for fast retrieval
List<Integer> linkedList = new LinkedList<>(); // ☑️ Best for frequent
insertions/deletions
```

🔹 2️⃣ Avoid Unnecessary Memory Allocation

Some collections **resize dynamically**, which can cause **performance overhead**.

💡 Example: Setting Initial Capacity for Lists

📌 By default, `ArrayList` starts with **10 elements** and resizes when full.
📌 If you **already know the required size**, set the **initial capacity** to **avoid resizing overhead**.

```
List<Integer> list = new ArrayList<>(100); // ☑ Optimized for 100 elements
```

💡 Example: Using `HashMap` with Proper Capacity

📌 `HashMap` has a **default capacity of 16** and **grows when 75% full**.
📌 If you know you'll store **1000 elements**, set capacity properly:

```
Map<String, Integer> map = new HashMap<>(1000, 0.75f); // ☑ Prevents unnecessary resizing
```

🔹 3️⃣ Use `Collections.unmodifiableList()` to Save Memory

If a collection **doesn't need modification**, **use immutable collections** to **save memory** and **avoid accidental changes**.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<String> unmodifiableNames = Collections.unmodifiableList(names); // ☑ More efficient
```

🔹 4️⃣ Use Primitive Arrays Instead of Collections (If Possible)

Collections store **objects**, which take **more memory**.
If dealing with **only numbers**, use **primitive arrays (`int[]`)** instead of **`ArrayList<Integer>`**.

```
int[] numbers = new int[1000]; // ☑ Uses less memory than ArrayList<Integer>
```

🔹 5️⃣ Remove Unused Elements to Free Up Memory

If a collection **grows dynamically** and elements are removed, it may still hold **extra memory**.

💡 Example: Trim `ArrayList` After Removing Elements

```
ArrayList<Integer> list = new ArrayList<>(100);
list.add(10);
list.add(20);
list.remove(1);

list.trimToSize(); // ☑ Shrinks the ArrayList to free memory
```

🔹 6️⃣ Use `WeakHashMap` for Temporary Caching

A **regular `HashMap` keeps objects in memory forever**, even if they're no longer needed.
A `WeakHashMap` automatically **removes unused keys**, helping reduce memory usage.

```
Map<String, Integer> cache = new WeakHashMap<>();
```

🔹 7️⃣ Use `Concurrent Collections` for Multi-threading

If **multiple threads** access a collection, **avoid using `synchronized` manually**.
Use **thread-safe collections** like `ConcurrentHashMap` instead of manually locking a `HashMap`.

```
Map<String, Integer> concurrentMap = new ConcurrentHashMap<>(); // ☑ Faster than synchronized HashMap
```

🔹 8️⃣ Prefer `for-each` Instead of Traditional Loops

A **for-each loop** is **faster and uses less memory** than manually iterating with an **Iterator**.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// ☑ Better performance
for (String name : names) {
    System.out.println(name);
}

// ❌ Slower due to extra iterator object
Iterator<String> it = names.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

🔹 9️⃣ Avoid Auto-Boxing in Collections

Collections store **only objects**, so primitive types (**`int`**, **`double`**) are **converted into objects** (`Integer`, `Double`).
This is called **auto-boxing** and consumes **more memory**.

💡 Example: Using `int` vs. `Integer`

```
List<Integer> list = new ArrayList<>();
list.add(10);  // ❌ Auto-boxing happens, uses more memory

int num = list.get(0); // ❌ Auto-unboxing happens
```

☑ **Solution:** If dealing with **large numeric data**, consider **primitive arrays (`int[]`)**.

☑ Final Summary: Best Ways to Optimize Collections

☑ Use **the right collection** for the right task.
☑ Set **initial capacity** to avoid resizing overhead.
☑ Use **immutable collections** when modification is not needed.
☑ Use **primitive arrays (`int[]`)** instead of `ArrayList<Integer>` when possible.
☑ Use **`trimToSize()`** to free up unused memory in `ArrayList`.
☑ Use **`WeakHashMap`** for temporary caching.
☑ Use **`ConcurrentHashMap`** instead of manually synchronizing a `HashMap`.
☑ Use **for-each loops** instead of manually iterating.
☑ Avoid **auto-boxing** where possible.

📌 Chapter 11: Summary of Java Collection Framework (Final Revision Guide)

This chapter summarizes **everything we've learned** about **Java Collections Framework (JCF)** in an **easy-to-read, deep, and structured format**. 📜

🔹 1️⃣ What is the Java Collection Framework?

The **Java Collection Framework (JCF)** is a set of **predefined classes and interfaces** for handling **data structures** like **Lists, Sets, Queues, and Maps** efficiently.

☑ **Benefits of JCF:**
✔ **Reusable** – No need to create custom data structures.
✔ **Optimized Performance** – Built-in implementations are **highly optimized**.
✔ **Flexible & Scalable** – Collections can grow dynamically.
✔ **Thread-Safe Options** – Supports **concurrent programming**.
✔ **Sorting & Searching Support** – Utility methods like `Collections.sort()` and `binarySearch()`.

🔹 2️⃣ Collection Framework Hierarchy (Main Interfaces & Implementations)

The **Java Collections Framework** consists of **4 main interfaces**:

Interface	Description	Common Implementations
List	Ordered collection (allows duplicates)	`ArrayList`, `LinkedList`, `Vector`, `Stack`
Set	Unordered collection (unique elements only)	`HashSet`, `LinkedHashSet`, `TreeSet`

Queue	Follows **FIFO (First In, First Out)**	`LinkedList`, `PriorityQueue`, `ArrayDeque`
Map	Stores **key-value pairs** (unique keys)	`HashMap`, `LinkedHashMap`, `TreeMap`, `Hashtable`

🔹 3️⃣ Deep Dive into Collection Interfaces

📌 List Interface (`List<T>`) – Ordered Collection

A **List** maintains **insertion order** and allows **duplicate elements**.

✔ **Fast Retrieval** → `ArrayList`
✔ **Fast Insert/Delete** → `LinkedList`
✔ **Thread-Safe** → `Vector`, `CopyOnWriteArrayList`
✔ **LIFO** (Last-In, First-Out) → `Stack`

📌 Set Interface (`Set<T>`) – Unique Elements

A **Set** does **not allow duplicate elements**.

✔ **Fastest Search (Unordered)** → `HashSet`
✔ **Maintains Insertion Order** → `LinkedHashSet`
✔ **Sorted Elements** → `TreeSet`
✔ **Thread-Safe** → `CopyOnWriteArraySet`

📌 Queue Interface (`Queue<T>`) – FIFO Data Structure

A **Queue** follows **First In, First Out (FIFO)**.

✔ **Standard Queue** → `LinkedList`
✔ **Priority-Based Queue** → `PriorityQueue`
✔ **Double-Ended Queue** → `ArrayDeque`
✔ **Thread-Safe Queue** → `ConcurrentLinkedQueue`, `BlockingQueue`

📌 Map Interface (`Map<K, V>`) – Key-Value Pair Collection

A **Map** stores **key-value pairs** (keys must be unique).

✔ **Fastest Search (Unordered)** → `HashMap`
✔ **Maintains Insertion Order** → `LinkedHashMap`
✔ **Sorted by Key** → `TreeMap`
✔ **Thread-Safe** → `ConcurrentHashMap`

🔹 4️⃣ Sorting & Searching in Collections

☑ Sorting Collections

✔ `Collections.sort(list)` – Sorts a list **naturally**.
✔ `Collections.sort(list, comparator)` – Sorts a list **using custom logic**.
✔ `TreeSet` and `TreeMap` automatically maintain **sorted order**.

☑ Searching Collections

✔ `Collections.binarySearch(list, key)` – **Fastest search** on sorted lists.
✔ `HashMap.get(key)` – **Constant-time retrieval** for maps.
✔ `contains()` in **Set** is **faster than** `contains()` **in List**.

🔹 5️⃣ Thread-Safety in Java Collections

✔ **Thread-Safe Lists & Sets** → `CopyOnWriteArrayList`, `CopyOnWriteArraySet`
✔ **Thread-Safe Maps** → `ConcurrentHashMap`, `ConcurrentSkipListMap`
✔ **Blocking Queues for Multi-threading** → `ArrayBlockingQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`

🔹 6️⃣ Performance Optimization & Best Practices

🔹 **Use the Right Collection for the Task** (e.g., `HashMap` for fast lookup, `ArrayList` for fast read, etc.)
🔹 **Set Initial Capacity** to avoid resizing overhead (`new ArrayList<>(100)`, `new HashMap<>(100, 0.75f)`).
🔹 **Use Immutable Collections** (`Collections.unmodifiableList()`) when modification is **not needed**.
🔹 **Use WeakHashMap for Caching** (removes unused entries automatically).
🔹 **Avoid Auto-Boxing** (`int[]` is more memory-efficient than `ArrayList<Integer>`).
🔹 **Use** `Concurrent Collections` **Instead of Synchronized Wrappers** (`ConcurrentHashMap` > `synchronizedMap`).

🔹 7️⃣ When to Use Which Collection? (Quick Reference)

Scenario	Best Collection to Use
Fast retrieval (index-based search)	`ArrayList`
Frequent insertions/deletions	`LinkedList`
Unique unordered elements	`HashSet`
Unique ordered elements	`LinkedHashSet`
Sorted elements	`TreeSet`, `TreeMap`

Key-value mapping (fast lookup)	`HashMap`
Multi-threaded key-value storage	`ConcurrentHashMap`
FIFO queue (First In, First Out)	`LinkedList` (as Queue), `ArrayDeque`
Priority-based processing	`PriorityQueue`
LIFO stack (Last In, First Out)	`Stack`, `ArrayDeque`
Blocking queues (multi-threading)	`ArrayBlockingQueue`, `LinkedBlockingQueue`

🎯 Final Conclusion

The **Java Collection Framework** provides a **powerful and flexible** way to manage data structures efficiently.
By understanding the **different types of collections**, **sorting & searching methods**, and **thread-safety mechanisms**, you can **write optimized, high-performance Java applications**. 🚀

Got it! Here's a **Quick Reference Chapter** covering:

- **Key Methods** of each Collection (10+ per type)
- **Internal Working** of Data Structure
- **Time Complexity** (Big-O Notation)
- **When to Use** (Short One-Liner)

📌 Chapter 12: Quick Reference Guide for Java Collections

🔹 List Implementations (Ordered Collection, Allows Duplicates)

1️⃣ ArrayList (Dynamic Array, Fast Read)

☑ Key Methods:

1. `add(E e)` – Adds element at the end
2. `add(int index, E e)` – Inserts element at index
3. `get(int index)` – Retrieves element at index
4. `set(int index, E e)` – Replaces element at index
5. `remove(int index)` – Removes element at index
6. `indexOf(Object o)` – Returns first index of element
7. `lastIndexOf(Object o)` – Returns last index of element

8. `contains(Object o)` – Checks if element exists
9. `size()` – Returns number of elements
10. `clear()` – Removes all elements

☑ **Internal Working:** Uses a **resizable array** (grows dynamically).
☑ **Time Complexity:** `O(1)` for get, `O(n)` for insert/remove in the middle.
☑ **When to Use:** When **fast read access** is needed.

2️⃣ LinkedList (Doubly Linked List, Fast Insert/Delete)

☑ **Key Methods:**

1. `addFirst(E e)` – Adds element at the beginning
2. `addLast(E e)` – Adds element at the end
3. `removeFirst()` – Removes first element
4. `removeLast()` – Removes last element
5. `getFirst()` – Retrieves first element
6. `getLast()` – Retrieves last element
7. `offer(E e)` – Adds element (like `add()`)
8. `poll()` – Removes and returns first element
9. `peek()` – Retrieves but does not remove first element
10. `size()` – Returns number of elements

☑ **Internal Working:** Uses **nodes** (each node contains data + two pointers).
☑ **Time Complexity:** `O(1)` for insert/remove at ends, `O(n)` for search.
☑ **When to Use:** When **frequent insertions/deletions** are needed.

3️⃣ Stack (LIFO – Last In, First Out)

☑ **Key Methods:**

1. `push(E e)` – Pushes element onto stack
2. `pop()` – Removes and returns top element
3. `peek()` – Returns top element without removing
4. `empty()` – Checks if stack is empty
5. `search(Object o)` – Finds position of element

☑ **Internal Working:** Uses **ArrayList internally**.
☑ **Time Complexity:** `O(1)` for push/pop.
☑ **When to Use:** When **LIFO (Last-In, First-Out) behavior** is needed.

4️⃣ Vector (Thread-Safe, Legacy)

☑ **Key Methods:** *(Same as `ArrayList`, but synchronized)*
☑ **Internal Working:** Uses **synchronized resizable array**.
☑ **Time Complexity:** Similar to `ArrayList`.
☑ **When to Use:** When **thread-safe dynamic array** is needed.

🔹 Set Implementations (Unique Elements, No Duplicates)

5️⃣ HashSet (Unordered, Unique Elements)

☑ **Key Methods:**

1. `add(E e)` – Adds element
2. `remove(Object o)` – Removes element
3. `contains(Object o)` – Checks if element exists
4. `size()` – Returns number of elements
5. `clear()` – Removes all elements

☑ **Internal Working:** Uses **HashMap internally** (each element is a key).
☑ **Time Complexity:** `O(1)` for add/remove/search (average).
☑ **When to Use:** When **unique elements + fast lookup** are needed.

6️⃣ TreeSet (Sorted Unique Elements)

☑ **Key Methods:** *(Same as `HashSet` + sorting features)*
☑ **Internal Working:** Uses **Red-Black Tree (Self-Balancing BST)**.
☑ **Time Complexity:** `O(log n)` for add/remove/search.
☑ **When to Use:** When **sorted unique elements** are needed.

🔹 Queue Implementations (FIFO – First In, First Out)

7️⃣ PriorityQueue (Elements with Priority)

☑ **Key Methods:**

1. `offer(E e)` – Inserts element with priority
2. `poll()` – Retrieves and removes highest-priority element
3. `peek()` – Retrieves highest-priority element without removing

☑ **Internal Working:** Uses **Min-Heap (Binary Heap)**.
☑ **Time Complexity:** `O(log n)` for insertion/removal.
☑ **When to Use:** When **priority-based processing** is needed.

8️⃣ ArrayDeque (Double-Ended Queue)

☑ **Key Methods:** *(Combination of `Queue` & `Stack` methods)*
☑ **Internal Working:** Uses **circular array** for better performance.
☑ **Time Complexity:** `O(1)` for add/remove at both ends.
☑ **When to Use:** When **deque operations (both ends)** are needed.

🔹 Map Implementations (Key-Value Pairs)

9️⃣ HashMap (Unordered Key-Value Storage)

☑ **Key Methods:**

1. `put(K key, V value)` – Inserts key-value pair
2. `get(Object key)` – Retrieves value by key
3. `remove(Object key)` – Removes key-value pair
4. `containsKey(Object key)` – Checks if key exists
5. `containsValue(Object value)` – Checks if value exists
6. `size()` – Returns number of key-value pairs
7. `clear()` – Removes all entries

☑ **Internal Working:** Uses **Hashing (Bucket + LinkedList/Tree structure)**.
☑ **Time Complexity:** `O(1)` for get/put/remove (average), `O(n)` (worst case).
☑ **When to Use:** When **fast key-based lookup** is needed.

🔟 TreeMap (Sorted Key-Value Storage)

☑ **Key Methods:** *(Same as `HashMap`, but sorted)*
☑ **Internal Working:** Uses **Red-Black Tree (Self-Balancing BST)**.
☑ **Time Complexity:** `O(log n)` for get/put/remove.
☑ **When to Use:** When **sorted key-value pairs** are needed.

⏳ Complexity Summary

Data Structure	Best Case	Worst Case
ArrayList (get)	`O(1)`	`O(1)`
ArrayList (add/remove at end)	`O(1)`	`O(n)`
LinkedList (add/remove)	`O(1)`	`O(n)`
Stack (push/pop)	`O(1)`	`O(1)`
HashSet (search/add/remove)	`O(1)`	`O(n)`
TreeSet (search/add/remove)	`O(log n)`	`O(log n)`
PriorityQueue (insert/remove)	`O(log n)`	`O(log n)`
HashMap (search/add/remove)	`O(1)`	`O(n)`
TreeMap (search/add/remove)	`O(log n)`	`O(log n)`

🎯 Conclusion

This **Quick Reference Guide** helps you **choose the right data structure** based on:

- 🔹 **Operations Needed (Insertion, Deletion, Lookup, Sorting)**
- 🔹 **Performance Considerations (Time Complexity, Internal Working)**
- 🔹 **Thread-Safety & Usage Scenarios**

🔥 **That's it! Your Java Collection Framework Guide is COMPLETE!**