⭐ What is Collection Framework?

The **Collection Framework** in Java is a standardized architecture for storing and manipulating groups of objects.
It provides:

- **Interfaces** (Collection, List, Set, Queue, Deque, Map)
- **Classes** (ArrayList, LinkedList, Vector, Stack, HashSet, LinkedHashSet, TreeSet, PriorityQueue, etc.)
- Utility Classes (Collections, Arrays)
- **Algorithms** (Sorting, Searching, Shuffling, Reversing)

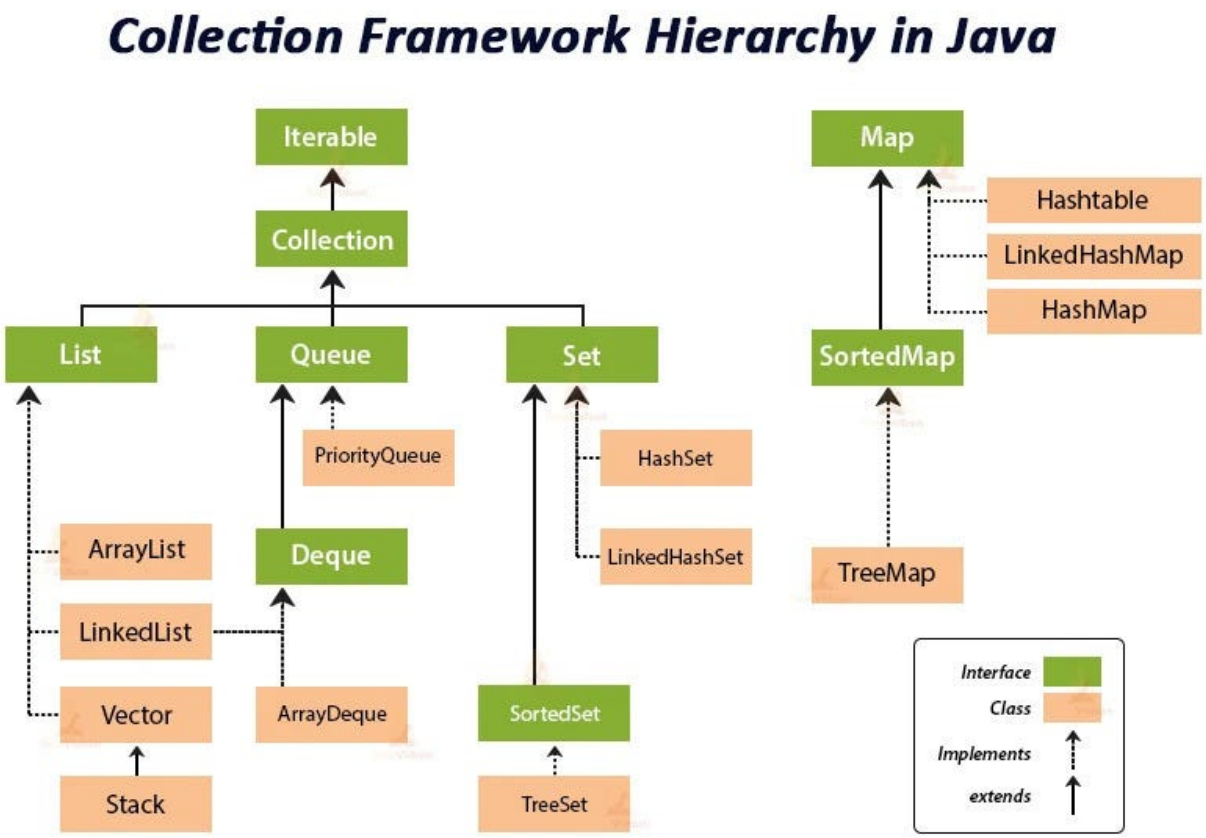
The framework supports operations like insertion, deletion, searching, sorting, updating, iteration, and manipulation of data.

⭐ Difference between Array and ArrayList

Array	ArrayList / Collection
Static size (fixed-length).	Dynamic size (grows/shrinks automatically).
Can store **primitive types** and objects.	Stores **objects only** (wrapper classes for primitives).
Fast and memory efficient for fixed-size data.	More flexible but slightly higher overhead.
Uses direct indexing with a fixed structure.	Built on top of a **dynamic internal array** (`Object[]`).
No built-in methods for manipulation.	Provides many built-in methods (add, remove, contains, etc.).
Supports multi-dimensional arrays.	One-dimensional; can nest lists to simulate multi-dimension.

⭐ Difference between Collection and Collections

Collection	Collections
It is a **interface** in the Collection Framework.	It is a **utility class** in `java.util`.
Represents a group of objects as a single unit.	Provides **static methods** like `sort()`, `reverse()`, `min()`, `max()`, `synchronizedList()`, etc.
Parent of List, Set, Queue.	Works *on* Collection objects.



⭐ List Interface

Definition:

List is a child interface of Collection.

It represents an **ordered collection** that preserves insertion order and allows duplicate elements.

✔ Key Features of List

1. **Maintains Order:** Insertion order is preserved.
2. **Duplicates Allowed:** Multiple identical elements permitted.
3. Index-Based Access: Supports add(index), get(index), set(index), remove(index).
4. **Null Allowed:** List implementations allow null values.
5. Supports Iterators:
    - Iterator (forward)
    - ListIterator (forward + backward)

✔ Implementations of List

1. ArrayList
    - Backed by dynamic array.
    - Fast random access (O(1)).
    - Slower for insert/delete in the middle (O(n)).
    - Not synchronized.

2. LinkedList
    - Backed by doubly linked list.
    - Fast insert/delete operations (O(1) at ends).
    - Slow random access (O(n)).
    - Also implements Queue/Deque.

3. **Vector** (Legacy)
    - Similar to ArrayList but synchronized.
    - Slower due to synchronization.
    - Rarely used today.

4. **Stack** (Legacy)
    - Extends Vector.
    - LIFO structure (push, pop, peek).
    - Modern replacement: ArrayDeque.

⭐1. ArrayList

Definition:
ArrayList is a **dynamic array implementation** of the List interface.
It can grow or shrink automatically as elements are added or removed.

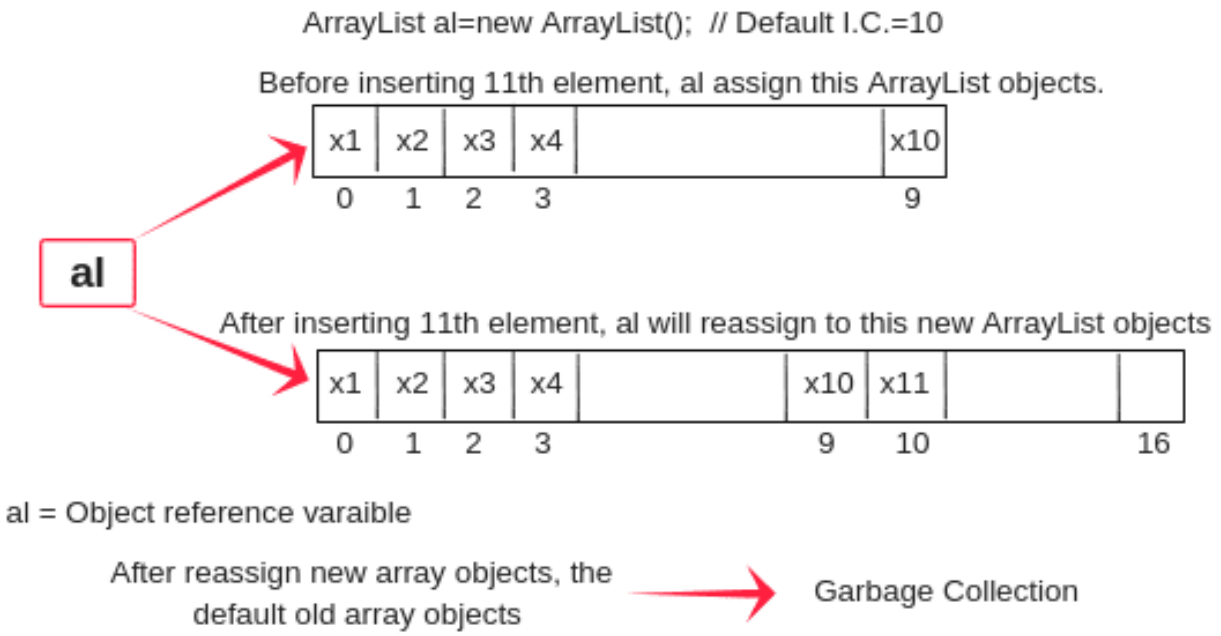
✔ Key Features of ArrayList

1. **Dynamic Array:** Automatically resizes.
2. Maintains Insertion Order: Index-based access.
3. **Allows Duplicates:** Same values can be inserted.
4. **Allows Null Values:** Can store one or more nulls.
5. Fast Random Access: get(index) is O(1).
6. **Not Synchronized:** Not thread-safe by default.
    - Thread-safe options:
        - Collections.synchronizedList(list)
        - CopyOnWriteArrayList
7. **Default Capacity:** 10 (auto-expands by 1.5×).
8. Resizable Underlying Array: Uses Object[] internally.

✔ Common Methods in ArrayList

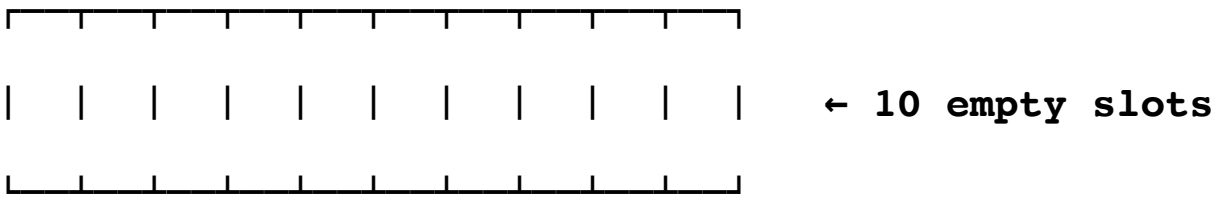
- add(E e)
- add(int index, E e)
- get(int index)

- set(int index, E e)
- remove(int index) / remove(Object o)
- contains(Object o)
- size()
- clear()
- indexOf() / lastIndexOf()
- iterator() / listIterator()
- addAll(Collection c)



Initially, when we declare `ArrayList<Integer> al = new ArrayList<>();`

`al`



`Index: 0 1 2 3 4 5 6 7 8 9`

After first add → `al.add(10);`

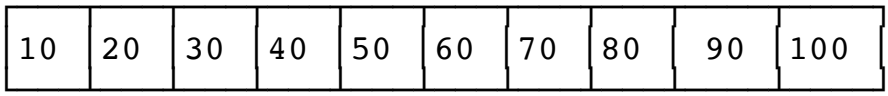
```
al
```


```
 Index: 0   1   2   3   4   5   6   7   8   9
```

```
After adding all initial elements (10 elements total)
```

```
al.add(10);
al.add(20);
al.add(30);
al.add(40);
al.add(50);
al.add(60);
al.add(70);
al.add(80);
al.add(90);
al.add(100);
```

```
al   (Initial capacity full)
```



Now add a new element → `al.add(110);`
➡ Capacity is full → ArrayList **resizes by 50%**
➡ New capacity = **15**

```
al     (Resized by 50%)
```


```
 Capacity: 15
 Size: 11
```

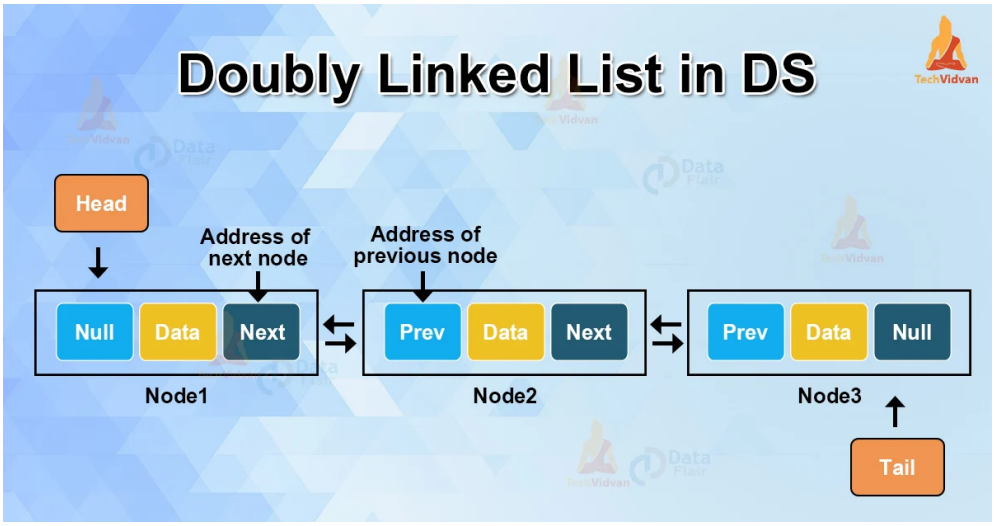
✔ Default Capacity = 10
✔ Resize happens by 1.5× (50%)
✔ Array grows only when full
✔ After removing elements, capacity does NOT shrink automatically

2. LinkedList

The LinkedList class in Java is part of the Java Collection Framework and implements the List, Deque, and Queue interfaces.
It represents a doubly-linked list internally and provides efficient insertion and deletion operations compared to ArrayList.

⭐ Key Features of LinkedList

1. **Doubly Linked List:**
   Each element (node) contains references to both its **previous** and **next** nodes.
2. **Efficient Insertions and Deletions:**
   Adding or removing elements—especially at the **beginning** or **middle**—is faster than ArrayList
   because no shifting of elements is required.
3. **Maintains Insertion Order:**
   Preserves the order in which elements are added.
4. **Allows Duplicates and Nulls:**
   Supports duplicate elements and can store null values.
5. Implements Queue and Deque Interfaces:
   Can be used as a Queue (FIFO) or as a Deque (Double-Ended Queue) using methods like offer(), poll(), offerFirst(), offerLast(), etc.



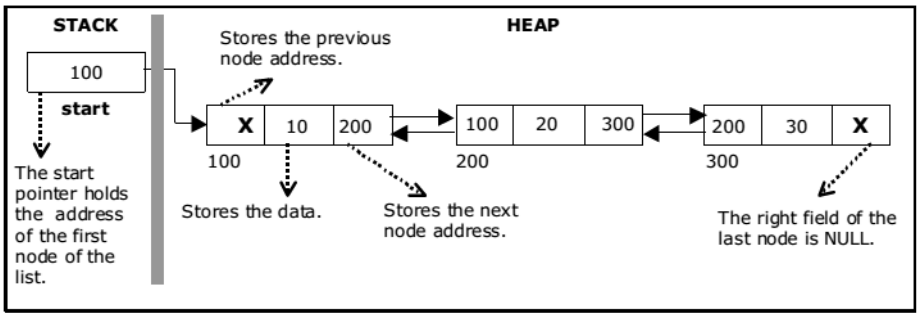


Figure 3.3.1. Double Linked List

🎯 When to Use LinkedList?

Use LinkedList when:
✔ You perform **frequent insertion/deletion**
✔ You need **Queue/Deque** implementation
✔ Order of elements matters
✔ You don't need fast random access

When NOT to Use LinkedList?

Avoid LinkedList when:
You frequently access elements by index
You need fast data reading
Memory is constrained (because nodes take more memory)

Why LinkedList is Faster for Insert/Delete?
Because only **pointers** change — no shifting of elements.

Why LinkedList is Slow for Accessing Elements?

Diff b/w arrayList And LinkedList

Aspect	ArrayList	LinkedList
Data Structure	Dynamic Array (contiguous memory)	Doubly Linked List (nodes: data + next + prev)
Random Access Performance	**Fast (O(1))** → direct index access	**Slow (O(n))** → must traverse nodes
Insertion/Deletion Performance	**Slow (O(n))** → shifting required	**Fast (O(1))** at beginning/end
Memory Usage	Less memory	More memory (extra pointers)
Thread-Safety	Not synchronized	Not synchronized
Growth	Increases capacity by **50%** (old * 1.5)	Adds nodes dynamically (**no fixed capacity**)
Best Use Case	Frequent read/search operations	Frequent add/delete operations
Allows Null	Yes	Yes
Maintains Order	Yes	Yes
Duplicates Allowed	Yes	Yes
Access by Index	Fast (O(1))	Slow (O(n))
Resizing	Automatic	Automatic

✅ 3. Vector

The **Vector** class in Java is part of the **Java Collection Framework** and is located in the java.util package. It implements the **List** interface and stores elements in a **dynamic, resizable array**.
Unlike ArrayList, **Vector is synchronized**, which makes it **thread-safe** for multi-threaded environments.

⭐ Key Features of Vector

1. **Resizable Array:**
   Automatically grows or shrinks as elements are added or removed.
2. **Synchronized:**
   All methods in Vector are synchronized, making it thread-safe but slightly slower than ArrayList.
3. **Maintains Insertion Order:**
   Elements remain in the order they are added.
4. **Allows Duplicates:**
   Duplicate elements are permitted.
5. Allows Null Elements:
   Can store one or more null values.
6. **Legacy Class:**
   Introduced in **JDK 1.0** and later retrofitted to implement the **List** interface.

Diff b/w arrayList and vector

Aspect	ArrayList	Vector
Thread-Safety	Not synchronized	**Synchronized (thread-safe)**
Performance	Faster	Slower (synchronization overhead)
Synchronization	Must be done manually	All methods synchronized
Growth	Grows by **50%** (old * 1.5)	Grows by **100%** (doubles capacity)
Memory Overhead	Less	More (sync + internal overhead)
Access Time	Fast	Slower
Resize Behavior	Automatic	Automatic (double size)
Null Elements Allowed	Yes	Yes
Insertion/Deletion Performance	Slower (shifting)	Slower than LinkedList but OK
Legacy Class	JDK 1.2	**JDK 1.0 (Legacy)**
Initial Capacity	10	10
Duplicates Allowed	Yes	Yes

⭐ 2. Set Interface

Set interface represents a collection of unique elements where duplicates are not allowed.
It extends the Collection interface and is implemented by HashSet, LinkedHashSet, and TreeSet.
Set does not maintain index, and ordering depends on the implementation class.

✔ Key Features of the Set Interface

1. **No Duplicate Elements:**
   A Set does not allow duplicate values. If you try to add a duplicate element, the Set simply ignores it.
2. **Unordered Collection:**
   A Set does not guarantee any specific order of elements.
   (However, specific implementations like **LinkedHashSet** maintain insertion order, and **TreeSet** maintains sorted order.)
3. **Extends Collection Interface:**
   Since Set extends the Collection interface, it inherits all common methods such as add(), remove(), contains(), size(), etc.

⭐IMPLEMENTATION CLASSES OF SET

The Set Interface has 3 main implementation classes:

1. HashSet
2. LinkedHashSet
3. TreeSet

⭐ 1. WHY SET CANNOT STORE DUPLICATES?

Set does NOT allow duplicate values because:

✔ Set uses hashing
Every element is converted into a **hash code** and stored in **buckets**.
✔ If a duplicate element is added:

- Same hashCode
- Same bucket
- Set checks using equals() method
- If values are equal → element is **NOT added**

👉 Set checks equality using:

1. hashCode()
2. equals()

⭐ HashSet in Java

The **HashSet** class (in java.util) implements the Set interface.
It stores elements in a **hash table** and ensures that the collection contains **only unique elements**.

✔ Key Features of HashSet

1. **No Duplicates:**
   HashSet does not allow duplicate values.
   When a duplicate is added, HashSet simply ignores it.
2. **Unordered:**
   HashSet does **not** maintain any specific order of elements.
   The output order depends on the **hash value** of each element.
3. Allows Null:
   HashSet allows only one null value.
4. **Not Synchronized:**
   HashSet is **not thread-safe**.
   If multiple threads access it simultaneously, external synchronization is required.

3. How HashSet Stores an Element?

✔ Step 1 → hashCode() of the element is generated
"Java".hashCode() → 23123

Step 2 → Bucket location calculated
bucketIndex = hashCode % capacity

✔ Step 3 → Check if bucket is empty

- If empty → store element
- If not empty (collision) → LinkedList/Tree structure used

✔ Step 4 → equals() check
If hashCode matches → equals() confirms if same element.
✔ Step 5 → If equals() = true → duplicate ignored
✔ If equals() = false → stored in same bucket (collision chain)

4. Default Capacity of HashSet
HashSet hs = new HashSet();

✔ Default Capacity = 16 buckets
(Because HashSet uses internal HashMap → HashMap capacity = 16)

5. Load Factor (VERY IMPORTANT)
Load Factor = 0.75 (Default)
Load Factor decides when to increase capacity.
✔ Load Factor Meaning:
When 75% of HashSet buckets are filled → it grows (rehashing).

6. Capacity Growth Formula
New Capacity = Old Capacity × 2

When 12 elements are added → HashSet grows its size to **32**

⭐ LinkedHashSet

LinkedHashSet stores unique elements while maintaining insertion order.It uses a combination of HashTable and LinkedList internally. Fast search, preserves order, and allows one null value.

LinkedHashSet is a **Set** implementation that:
✔ **Does NOT allow duplicates**
✔ **Maintains insertion order**
✔ Uses **LinkedList + HashTable** internally
✔ Allows **only one null**
✔ Faster than TreeSet
✔ Slightly slower than HashSet

2. Internal Structure (Very Important)
LinkedHashSet uses 2 data structures together:

1. **HashTable** → For hashing + fast search
2. **Doubly Linked List** → To maintain insertion order

. How LinkedHashSet Works Internally?
Step 1 → hashCode() generates bucket index
Step 2 → equals() checks duplicates
Step 3 → Node stored in bucket + a LinkedList chain
Step 4 → LinkedList preserves insertion order

3. TREESET

TreeSet is a **sorted Set** implementation in Java that:
✔ Stores **unique values only**
✔ Maintains **ascending sorted order**
✔ Does **NOT** allow null
✔ Uses **Red-Black Tree** internally
✔ Is slower than HashSet & LinkedHashSet
✔ Allows **range operations** (higher, lower, ceiling, floor)

3. Sorting Behavior
TreeSet maintains data in **ascending order** by default:

Input : 30, 10, 40, 20
Output: [10, 20, 30, 40]

4. Why TreeSet Does NOT Allow null?
Because null cannot be compared using comparison logic (compareTo()).

⭐ HashSet vs LinkedHashSet vs TreeSet — Comparison Table

Feature	HashSet	LinkedHashSet	TreeSet
Order	❌ No order (random)	✔Maintains **insertion order**	✔Maintains **sorted ✔(ascending) order**
Internal Structure	Hash Table	Hash Table + LinkedList	Red-Black Tree (Self-balancing BST)
Duplicates Allowed?	❌ No	❌ No	❌ No
Null Allowed?	✔ One null allowed	✔ One null allowed	❌ No null allowed
Performance (Speed)	⭐ Fastest	⭐ Fast	❌ Slowest
Time Complexity	O(1) search, add, delete	O(1) but slightly slower	O(log n)
When to Use?	When you only need unique elements, no order	When you want unique + maintain insertion order	When you want unique + automatically sorted values
Sorting	❌ Not supported automatically	❌ Not supported automatically	✔ Always sorted
Memory Usage	Least	Medium	High
Best Use Case	Fast lookup	Keep order + uniqueness	Sorted unique data (numbers, names)
Implement	Set	Set	NavigableSet (SortedSet)
Suitable for	Searching operations	Caching, predictable iteration	Range operations (higher(), lower(), ceiling())
Thread-safe?	❌ No	❌ No	❌ No

Difference between List vs Set

Aspect	List	Set
Order	Maintains **ordered elements** (insertion order).	Does **not guarantee any order** (unordered or sorted, depends on implementation).
Duplicates	Allows **duplicate elements**.	Does **not allow duplicate elements**.
Index-Based Access	Supports **index-based access** (like arrays).	❌ No index-based access; elements are stored without positions.
Insertion Order	Always preserves **insertion order**.	May or may not preserve insertion order (depends on implementation).
Structure	Internally based on **dynamic array** (ArrayList) or **linked list** (LinkedList).	Internally based on **hash table** (HashSet, LinkedHashSet) or **tree structure** (TreeSet).
Synchronized	Some implementations like **Vector** are synchronized. Others are not.	Generally **not synchronized**, unless manually synchronized.
Performance	Slower for frequent insert/delete operations (due to shifting or linked traversal). Searching is fast in ArrayList but slower in LinkedList.	Faster for insertions, deletions, and lookups because of **hash-based** or **tree-based** structure.
Method Support	Supports methods like `get()`, `set()`, `add()`, `remove()`, `size()`, `contains()`.	Supports methods like `add()`, `remove()`, `size()`, `contains()`, but ❌ **no index-based methods**.
Use Case	Best when you need **ordered data**, **duplicates**, and **index-based access**.	Best when you need **unique values**, **faster lookup**, and order does not matter (or sorted order using TreeSet).