SET FAMILY

1. **Unordered Collection:**
   * Elements are not stored in any particular order.
   * The order may depend on the specific implementation (e.g., HashSet is unordered, while LinkedHashSet maintains insertion order).
2. **No Duplicate Elements:**
   * A Set ensures that no two elements are equal as determined by their equals() and hashCode() methods.
3. **Supports Null Elements:**
   * Most implementations allow a single null element (e.g., HashSet), while some may not support null at all (e.g., TreeSet if a custom comparator does not handle null).
4. **Efficient Membership Testing:**
   * Provides efficient mechanisms to check for the presence of an element.
5. **Dynamic Resizing:**
   * Similar to Lists, Sets can dynamically grow or shrink as elements are added or removed.
6. **Supports Iteration:**
   * Provides Iterator and enhanced for-each loop to iterate over elements.
7. **Type-Safe Collections:**
   * Since Java 5, generics allow type-safe Sets, preventing ClassCastException at runtime.

**Core Methods of the Set Interface**

Basic Operations

1. boolean add(E e)
   * Adds the specified element to the set if it is not already present.
2. boolean remove(Object o)
   * Removes the specified element from the set if it exists.
3. boolean contains(Object o)
   * Returns true if the set contains the specified element.
4. void clear()
   * Removes all elements from the set.
5. int size()
   * Returns the number of elements in the set.
6. boolean isEmpty()
   * Returns true if the set is empty.
7. Object[] toArray()
   * Returns an array containing all the elements in the set.
8. <T> T[] toArray(T[] a)
   * Returns an array containing all the elements in the set in proper sequence.

Bulk Operations

9. boolean addAll(Collection<? extends E> c)

* Adds all the elements from the specified collection to the set if they are not already present.

10. boolean containsAll(Collection<?> c)

* Returns true if the set contains all the elements of the specified collection.

11. boolean removeAll(Collection<?> c)

* Removes from the set all its elements that are contained in the specified collection.

12. boolean retainAll(Collection<?> c)

* Retains only the elements in the set that are contained in the specified collection.

**1. HashSet**

**Description:**  
A hash table-based implementation of the Set interface, which stores elements using a hashing mechanism.

**Key Features:**

* **Performance:** Average time complexity of O(1) for basic operations (add, remove, contains) in ideal conditions.
* **No Order Guarantee:** Elements are stored in a manner determined by their hash codes, and insertion order is not maintained.
* **Null Handling:** Allows at most one null element.
* **Backed by HashMap:** Internally, HashSet uses a HashMap where the elements act as keys and a dummy value is used as the associated value.

**Limitations:**

* Performance can degrade to O(n) if many hash collisions occur due to poor hashCode implementation.

**2. LinkedHashSet**

**Description:**  
A subclass of HashSet that maintains the insertion order of elements.

**Key Features:**

* **Preserves Insertion Order:** Elements are maintained in the order they are added.
* **Null Handling:** Allows at most one null element.
* **Performance:** Slightly slower than HashSet due to the overhead of maintaining a linked list for order.
* **Internals:** Uses a doubly linked list to maintain order in addition to a hash table.

**Common Use Case:**  
When you need a set that maintains the order of elements.

**3. TreeSet**

**Description:**  
A NavigableSet implementation based on a Red-Black Tree, where elements are sorted either by their natural order or by a custom comparator.

**Key Features:**

* **Sorted Set:** Ensures elements are stored in sorted order.
* **No Null Values:** Does not allow null elements.
* **Performance:** O(log n) for add, remove, and contains.
* **Comparator Support:** You can define a custom sorting order using a Comparator.

**Limitations:**

* Slower compared to HashSet and LinkedHashSet due to tree-based operations.
* Does not allow heterogeneous elements unless a custom comparator handles them.

**4. CopyOnWriteArraySet**

**Description:**  
A thread-safe implementation of the Set interface, built on the CopyOnWriteArrayList.

**Key Features:**

* **Thread Safety:** No external synchronization needed for concurrent access.
* **Snapshot Iterators:** Iterators iterate over a snapshot of the set at the time of its creation, so they don’t throw ConcurrentModificationException.
* **Performance:** High memory and performance cost for modifications, as every change creates a new copy of the underlying array.
* **No Null Elements:** Does not allow null values.

**Limitations:**

* Best suited for scenarios with many reads and few writes.
* Memory overhead due to the creation of new arrays for every update.

**Usage Example:**

A screen shot of a computer program

Description automatically generated

**Common Use Case:**  
When you need a thread-safe set for mostly-read-heavy operations in multithreaded environments.

**5. EnumSet**

**Description:**  
A specialized implementation for enum types, which provides a highly efficient way to store enum constants.

**Key Features:**

* **Highly Efficient:** Uses a bit-vector internally for storage, making it compact and fast.
* **Range Support:** Supports operations like ranges (allOf, range, etc.).
* **No Null Elements:** Does not allow null values.
* **Type Restriction:** Works only with enum types.

**Limitations:**

* Can only store elements of a single enum type.
* Not thread-safe; requires manual synchronization for concurrent access.

**Usage Example:**

A computer screen with blue text

Description automatically generated

**Common Use Case:**  
When working with enum constants and need high performance.

================================================================

**Advanced Features**

1. **Custom Sorting in TreeSet:**
   * Use a Comparator for custom ordering.

A computer screen shot of a black screen

Description automatically generated

1. **Thread-Safe Variants:**
   * Use Collections.synchronizedSet to wrap a Set for thread-safe operations:

Set<String> synchronizedSet = Collections.synchronizedSet(new HashSet<>());

1. **Iterators:**
   * Use iterators for traversing sets safely.

A screen shot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generatedOverview

**Key Points:**

* **Space Complexity:** All three data structures have linear space complexity, meaning the space required grows linearly with the number of elements.
* **Time Complexity:**
  + **HashSet:** Offers constant-time average performance for basic operations due to its hash table implementation.
  + **TreeSet:** Provides logarithmic time complexity for operations, making it efficient for sorted operations and range queries.
  + **LinkedHashSet:** Combines the advantages of both HashSet and LinkedLists, providing constant-time average performance for basic operations while maintaining insertion order.

**Choosing the Right Set:**

* **HashSet:** Ideal for scenarios where you need fast lookup, insertion, and deletion, and don't care about the order of elements.
* **TreeSet:** Suitable for scenarios where you need elements to be sorted and efficiently retrieve elements within a specific range.
* **LinkedHashSet:** Useful when you need to maintain the insertion order of elements while still benefiting from efficient operations.

A screenshot of a computer

Description automatically generated