**Collection Interface Hierarchy**

The **Collections Framework** is divided into several interfaces and classes, with the **Collection interface** being at the core. The hierarchy is structured as follows:

Iterable

└── Collection

├── List

└── Set

├── SortedSet

└── NavigableSet

* **Iterable**: The root interface, allowing iteration over elements.
* **Collection**: Extends Iterable and provides basic operations like add(), remove(), contains(), and size().
* **List**: A collection that maintains insertion order and allows indexed access to elements (get(index), set(index, element)).
* **Set**: A collection that ensures no duplicate elements and does not support indexing.
  + **SortedSet**: A specialized Set that maintains elements in a sorted order.
  + **NavigableSet**: Extends SortedSet and adds navigation methods like reverse iteration and finding nearest elements.

This hierarchy helps manage and organize data in Java, offering flexibility for different needs such as maintaining order, uniqueness, or sorting.

**Iterable Interface - Summary**

* **Iterable Interface**: Not part of the Collections Framework but a super interface for Collection and all its subinterfaces.
* Introduced in **Java SE 5** (2004) alongside the **for-each loop**.
* Any object implementing Iterable can be iterated over using the enhanced for loop.

Example:

```

Collection<String> collection = ...;

for (String element : collection) {

// process each element

}

```

* To implement Iterable, you need to provide an Iterator to iterate over the elements.

**Storing Elements with the Collection Interface**

The **Collection interface** models common operations for containers of elements:

* **Basic Operations**:
  + Add or remove elements.
  + Test for the presence of an element.
  + Get the size of the collection or check if it is empty.
  + Clear the collection.
* **Set Operations**:
  + Test for inclusion of one set within another.
  + Union, intersection, and complement.
* **Accessing Elements**:
  + Iterate over elements using an iterator.
  + Create a stream (which can be parallel).

**Difference Between Collection, Set, and List**:

* **List**: Maintains order and supports indexed access.
* **Set**: Ensures no duplicate elements but does not support indexed access.

The Collection interface provides the foundation, and the List and Set interfaces extend it with specific behaviors.

**Extending Collection with List**

The **List interface** extends the **Collection interface** with the following differences:

* **Order**: A List maintains the order in which elements are added, unlike Collection or Set where the order is not guaranteed.
  + When iterating over a List, elements are accessed in the order they were added.
* **Indexing**: Elements in a List are indexed, allowing access to specific positions.
  + You can:
    - Get or remove an element at a specific index.
    - Insert or replace an element at a specific index.
    - Retrieve a range of elements between two indexes.

**Key takeaway**: A List offers order and indexing, which are not features of Collection or Set.

**Extending Collection with Set**

The **Set interface** extends the **Collection interface** with the following difference:

* **No Duplicates**: A Set does not allow duplicate elements, unlike a Collection where duplicates are permitted.
  + Adding a duplicate element to a Set may fail.
* **Indexing**: A Set does not provide indexing like a List. Some Set implementations may iterate over elements in a fixed order, but they do not have an index for element access.

**Key takeaway**: A Set ensures uniqueness but does not support indexing or additional operations beyond those provided by Collection.

**Sorting Elements in a Set with SortedSet and NavigableSet**

The **SortedSet** and **NavigableSet** interfaces extend **Set** with additional features:

* **SortedSet**:
  + Maintains elements in **ascending order**.
  + To sort elements, they must either implement Comparable (with compareTo()) or you can provide a Comparator.
  + Sorting vs. Ordering:
    - **Sorting**: Elements are ordered based on a comparison (e.g., ascending order).
    - **Ordering**: Elements remain in the order they were added (as in a List).

**Operations with SortedSet**:

* + Get the **lowest** and **largest** element.
  + Extract a **headSet** (elements less than a given element) and **tailSet** (elements greater than a given element).
  + Iteration goes from the **lowest to the greatest** element.

**Example**:

```

SortedSet<Integer> sortedSet = new TreeSet<>();

sortedSet.add(5);

sortedSet.add(3);

sortedSet.add(8);

for (Integer num : sortedSet) {

System.out.println(num); // Output: 3, 5, 8

}

```

* **NavigableSet**:
  + Extends SortedSet with additional functionality.
  + Allows **descending iteration** (reverse order).

**Example**:

```

NavigableSet<Integer> navigableSet = new TreeSet<>();

navigableSet.add(5);

navigableSet.add(3);

navigableSet.add(8);

// Iterating in descending order

for (Integer num : navigableSet.descendingSet()) {

System.out.println(num); // Output: 8, 5, 3

}

```

**Key takeaway**: SortedSet sorts elements, while NavigableSet extends this with additional features like descending iteration.