Object Oriented Programming in Java

Collections

Suhel Hammoud

Extending Collection with List

Exploring the List Interface

The List interface adds:

- **Order preservation**: Elements are kept in insertion order.
- **Indexing**: Access, insert, and remove by position.

```
List<String> list = List.of("a", "b", "c");
System.out.println(list.get(0)); // "a"
```

▓▓ Choosing a List Implementation

Two main implementations:

- **ArrayList**: Backed by a dynamic array
- **LinkedList**: Doubly-linked nodes

Best choice: ArrayList for general use (faster access & iteration) Use LinkedList for LIFO/FIFO operations (stack/queue-like behavior)

▓▓ Accessing Elements by Index

Key methods:

- `add(index, element)`
- `get(index)`
- `set(index, element)`
- `remove(index)`
- `indexOf(element)` / `lastIndexOf(element)`

⚠️ Throws `IndexOutOfBoundsException` for invalid indices

▓▓ Sublist Views and Modifications

Use `subList(start, end)` to get a view (not a copy):

```
List<String> list = new ArrayList<>(List.of("0", "1", "2", "3", "4", "5"));
list.subList(2, 5).clear();
System.out.println(list); // [0, 1, 5]
```

Changes in sublist affect the main list.

▓▓ Inserting Collections at an Index

You can insert a whole collection at a specific index:

```
list.addAll(index, otherCollection);
```

This shifts existing elements accordingly.

▓▓ Sorting a List

Since Java 8:

```
list.sort(Comparator.naturalOrder());
```

- Requires elements to be `Comparable` or provide a `Comparator`
- Avoid passing `null` directly

Prior to Java 8: use `Collections.sort(list, comparator)`

▓▓ Iterating with ListIterator

Provides more control than Iterator:

- hasPrevious() / previous()
- nextIndex() / previousIndex()
- set(element) to update the last returned element

```
var list = new ArrayList<>(List.of("one", "two", "three"));
ListIterator<String> it = list.listIterator();
while (it.hasNext()) {
    if (it.next().equals("two")) {
        it.set("2");
    }
}
```

Output:

```
[one, 2, three]
```

Section Two

- Extending Collection with Set, SortedSet, and NavigableSet

▓▓ Exploring the Set Interface

- Set forbids **duplicates** but doesn't guarantee order.
- Main implementation: HashSet (internally uses HashMap).
- Order of iteration is **unpredictable**:

```
Set<String> set = new HashSet<>(List.of("one", "two", "three"));
set.forEach(System.out::println); // Unordered output
```

Avoid relying on insertion order with plain Set.

▓▓ Extending Set with SortedSet

- SortedSet keeps elements **sorted** by natural order or custom Comparator.
- Implementation: TreeSet.

Useful methods:

- first(), last()
- headSet(toElement), tailSet(fromElement)
- subSet(from, to) — inclusive/exclusive rules apply

```
SortedSet<String> set = new TreeSet<>(Set.of("a", "b", "c", "d", "e", "f"));
System.out.println(set.subSet("aa", "d")); // [b, c]
```

Subsets are **views**, not copies — changes are reflected both ways.

▓▓ Subset Constraints in SortedSet

- Subsets (`headSet`, `tailSet`, `subSet`) remember their boundaries.
- Illegal to add elements **outside** these bounds → `IllegalArgumentException`.

```
SortedSet<String> subset = set.subSet("b", "e");
subset.add("a"); // ❌ Illegal
```

▓▓ Extending SortedSet with NavigableSet

- NavigableSet (Java 6) extends SortedSet with more powerful methods.
- TreeSet implements both interfaces.

New methods:

- ceiling(e), floor(e)
- higher(e), lower(e)
- pollFirst(), pollLast()
- descendingIterator(), descendingSet()

```
NavigableSet<String> set = new TreeSet<>(Set.of("a", "b", "c", "d"));
System.out.println(set.descendingSet()); // [d, c, b, a]
```

▓▓ Recap

- `Set` ensures uniqueness, no order.
- `SortedSet` ensures sorted order, supports subset views.
- `NavigableSet` adds fine-grained control (inclusive/exclusive bounds, reverse iteration).

Prefer:

- `HashSet` for uniqueness without order.
- `TreeSet` for sorted, searchable sets with range queries.

Section Three

- Creating and Processing Data with Collections Factory Methods

▓▓ Creating Immutable Collections (Java 9+)

Use factory methods for immutable collections:

```
List<String> list = List.of("one", "two", "three");
Set<String> set = Set.of("one", "two", "three");
```

✅ Immutable ❌ No null values ❌ No duplicates (for Set) ⚠️ Internal implementation is not ArrayList/HashSet

▓▓ Getting an Immutable Copy (Java 10)

Use `copyOf()` to create immutable snapshots:

```
List<String> list = List.copyOf(strings);
Set<String> set = Set.copyOf(strings);
```

- Source must be non-null and contain no `null` elements
- `Set.copyOf()` removes duplicates
- Returned structure is immutable

▓▓ Wrapping an Array in a List

```
List<String> list = Arrays.asList("a", "b", "c");
```

- Fixed size: supports `set()` but **not** `add()` or `remove()`
- Backed by the original array
- Not a true immutable list

▓▓ Extracting Min/Max from a Collection

```
String min = Collections.min(list);
String max = Collections.max(list, comparator);
```

⚠️ Collection must not be empty ⚠️ Elements must be Comparable or a Comparator must be provided

▓▓ Searching for a Sublist

```
int index = Collections.indexOfSublist(source, target);
int last = Collections.lastIndexOfSublist(source, target);
```

Returns index of first/last occurrence of target in source.

▓▓ Reordering a List (using Collections class)

- `sort(comparator)` → in-place sort
- `shuffle()` → randomize order
- `rotate(list, distance)` → cyclic shift
- `reverse()` → reverse order
- `swap(list, i, j)` → swap elements

▓▓ Creating Immutable Wrappers

```
List<String> immutableList = Collections.unmodifiableList(list);
```

- ❌ Modifications through wrapper
- ✅ Changes to underlying list are reflected

✅ Use defensive copy for true immutability

▓▓ Creating Synchronized Wrappers

```
List<String> syncList = Collections.synchronizedList(list);
```

🔒 Synchronize manually when iterating:

```
synchronized (syncList) {
    for (String s : syncList) { ... }
}
```

⚠️ Prefer java.util.concurrent for thread-safe collections

▓▓ Section Summary

✅ `List.of()`, `Set.of()` for immutable creation ✅ `copyOf()` for immutable snapshot ✅ `Collections` class for:

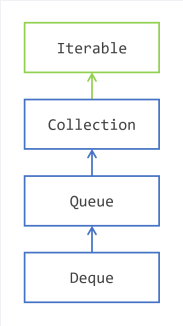
- min/max
- sublist search
- reordering
- wrappers (immutable/synchronized)

Use factory methods for safety, clarity, and reduced boilerplate.

Section Four

Storing Elements in Stacks and Queues

the following image shows it



▓▓ Queue and Deque in the Collections Framework

- Java SE 5: Queue interface added
- Java SE 6: Deque (double-ended queue) added
- Both extend Collection and support push/pop/poll/peek operations
- Deque = Stack + Queue

```
Queue<String> q = new LinkedList<>();
Deque<String> d = new ArrayDeque<>();
```

Stack vs Queue Behavior

- **Stack (LIFO)**: Last In, First Out
- **Queue (FIFO)**: First In, First Out

Common Operations:

- `push(e)` – Add element
- `pop()` / `poll()` – Remove element
- `peek()` – View next element (no removal)

▓▓ Queue Interface: Behavior on Capacity Limits

Operation	Method	On failure
Push	`add(e)`	Throws exception
Push	`offer(e)`	Returns `false`
Pop	`remove()`	Throws exception
Pop	`poll()`	Returns `null`
Peek	`element()`	Throws exception
Peek	`peek()`	Returns `null`

▓▓ Deque Interface: FIFO and LIFO Support

▒▒ FIFO-style (Queue behavior)

Operation	Method
Push	`addLast(e)` / `offerLast(e)`
Pop	`removeFirst()` / `pollFirst()`
Peek	`getFirst()` / `peekFirst()`

▒▒ LIFO-style (Stack behavior)

Operation	Method
Push	`addFirst(e)` / `offerFirst(e)`
Pop	`removeFirst()` / `pollFirst()`
Peek	`getFirst()` / `peekFirst()`

▓▓ Additional Deque Methods

- `push(e)` → alias for `addFirst()`
- `pop()` → alias for `removeFirst()`
- `poll()` → alias for `pollLast()`
- `peek()` → alias for `peekLast()`

If no elements are available: **returns** `null`

Implementations of Queue and Deque

- **ArrayDeque**:
  - Implements both Queue and Deque
  - Backed by array, dynamically resizes
  - Fast for stack/queue operations
- **LinkedList**:
  - Implements both
  - Fast first/last access due to linked structure
- **PriorityQueue**:
  - Implements only Queue
  - Maintains sorted order via Comparator or Comparable

▓▓ Avoiding Stack Class

- `Stack` extends `Vector` (synchronized, legacy)
- Avoid using `Stack` — use `Deque` or `ArrayDeque` instead
- For thread-safe needs, prefer `BlockingQueue` implementations

```
Deque<String> stack = new ArrayDeque<>();
stack.push("data");
```

▓▓ Summary

- Use `Queue` for FIFO, `Deque` for both FIFO/LIFO
- Prefer `ArrayDeque` or `LinkedList` over `Stack`
- Avoid legacy `Stack`/`Vector`
- Understand method behavior on full/empty collections
- Use `PriorityQueue` for sorted queue operations