Java Collections Framework - Complete Guide

Overview

The Java Collections Framework is a unified architecture for representing and manipulating collections of objects. It provides a set of interfaces, implementations, and algorithms that make it easier to work with groups of objects.

Framework Hierarchy

```
Collection (Interface)
├── List (Interface)
│   ├── ArrayList
│   ├── LinkedList
│   └── Vector
├── Set (Interface)
│   ├── HashSet
│   ├── LinkedHashSet
│   └── TreeSet
└── Queue (Interface)
    ├── PriorityQueue
    ├── ArrayDeque
    └── LinkedList

Map (Interface) - Separate hierarchy
├── HashMap
├── LinkedHashMap
├── TreeMap
└── Hashtable
```

Core Interfaces

1. Collection Interface

The root interface of the collection hierarchy. Defines basic operations like add, remove, contains, size, etc.

2. List Interface

Ordered collection that allows duplicates and provides indexed access.

3. Set Interface

Collection that doesn't allow duplicates.

4. Queue Interface

Collection designed for holding elements prior to processing.

5. Map Interface

Object that maps keys to values, cannot contain duplicate keys.

LIST IMPLEMENTATIONS

ArrayList

Description: Resizable array implementation of the List interface.

Visual Representation:

```
ArrayList: [Element0][Element1][Element2][Element3][  null  ][  null  ]
Indices:     0        1        2        3        4        5
Capacity: 6, Size: 4
```

Key Characteristics:

- **Access Time**: O(1) - Direct index access
- **Insertion**: O(1) at end, O(n) at beginning/middle
- **Deletion**: O(1) at end, O(n) at beginning/middle
- **Memory**: Contiguous memory allocation
- **Thread Safety**: Not thread-safe

Best Use Cases:

- Frequent random access to elements
- More reads than writes
- When you need indexed access

Code Example:

```java
ArrayList<String> list = new ArrayList<>();
list.add("Apple");    // O(1)
list.add("Banana");   // O(1)
list.get(0);          // O(1) - returns "Apple"
list.set(1, "Orange"); // O(1)
```

LinkedList

Description: Doubly-linked list implementation of List and Deque interfaces.

Visual Representation:

```
LinkedList Structure:
[HEAD] ↔ [Node1: "A"] ↔ [Node2: "B"] ↔ [Node3: "C"] ↔ [TAIL]
         prev|next     prev|next     prev|next
```

Each Node Contains:

```
┌─────────────────┐
│  Previous Node  │ ← Reference to previous node
├─────────────────┤
│      Data       │ ← Actual data
├─────────────────┤
│    Next Node    │ ← Reference to next node
└─────────────────┘
```

Key Characteristics:

- **Access Time**: O(n) - Must traverse from head/tail
- **Insertion**: O(1) at beginning/end, O(n) in middle
- **Deletion**: O(1) if node reference available, O(n) by value
- **Memory**: Non-contiguous, extra memory for pointers
- **Thread Safety**: Not thread-safe

Best Use Cases:

- Frequent insertions/deletions at beginning/end
- When you don't need random access
- Implementing stacks, queues, deques

SET IMPLEMENTATIONS

HashSet

Description: Hash table implementation of Set interface.

Visual Representation:

```
HashSet Internal Structure (Hash Table):
Bucket Array:
[0] → null
[1] → "Apple" → null
[2] → "Banana" → "Orange" → null  (collision chain)
[3] → null
[4] → "Grape" → null
[5] → null
```

Hash Function Process:

```
"Apple" → hashCode() → 12345 → 12345 % 6 → bucket[3]
"Banana" → hashCode() → 67890 → 67890 % 6 → bucket[0]
```

Key Characteristics:

- **Access Time**: O(1) average, O(n) worst case
- **Insertion**: O(1) average
- **Deletion**: O(1) average
- **Order**: No guaranteed order
- **Null Values**: Allows one null value
- **Thread Safety**: Not thread-safe

LinkedHashSet

Description: Hash table with linked list to maintain insertion order.

Visual Representation:

```
LinkedHashSet:
Hash Table + Insertion Order Chain

Hash Buckets:       Insertion Order:
[0] → null          [HEAD] → "Apple" → "Banana" → "Orange" → [TAIL]
[1] → "Apple"                 ↓         ↓          ↓
[2] → "Banana"              (maintains order while hashing)
[3] → "Orange"
```

Key Characteristics:

- **Access Time**: O(1) average
- **Order**: Maintains insertion order
- **Memory**: Higher than HashSet due to linked list

- **Performance**: Slightly slower than HashSet

TreeSet

Description: Red-Black tree implementation of NavigableSet.

Visual Representation:

```
TreeSet (Red-Black Tree):
        "Dog"(B)
       /       \
   "Cat"(R)   "Horse"(R)
   /    \     /       \
"Ant"(B) "Cow"(B) "Elephant"(B) "Lion"(B)

(B) = Black node, (R) = Red node
```

Key Characteristics:

- **Access Time**: O(log n)
- **Insertion**: O(log n)
- **Deletion**: O(log n)
- **Order**: Sorted order (natural or custom comparator)
- **Null Values**: Not allowed
- **Thread Safety**: Not thread-safe

QUEUE IMPLEMENTATIONS

PriorityQueue

Description: Heap-based priority queue implementation.

Visual Representation:

```
PriorityQueue (Min-Heap):
Array: [1, 3, 2, 7, 5, 4, 6]
Index:  0  1  2  3  4  5  6

Tree Structure:
      1
    /  \
   3    2
  /\  /\
 7   5 4   6

Parent of index i: (i-1)/2
Left child of index i: 2*i+1
Right child of index i: 2*i+2
```

Key Characteristics:

- **Access Time**: O(1) for peek, O(log n) for poll
- **Insertion**: O(log n)
- **Order**: Priority-based (min-heap by default)
- **Null Values**: Not allowed

ArrayDeque

Description: Circular array implementation of Deque interface.

Visual Representation:

```
ArrayDeque (Circular Array):
Array: [  C  ][  D  ][ null ][ null ][  A  ][  B  ]
Index:   0     1      2       3       4      5
         ↑                    ↑
        tail                 head

addFirst() → decrements head
addLast() → increments tail
```

Key Characteristics:

- **Access Time**: O(1) at both ends
- **Insertion**: O(1) at both ends
- **Memory**: More efficient than LinkedList
- **Null Values**: Not allowed

MAP IMPLEMENTATIONS

HashMap

Description: Hash table implementation of Map interface.

Visual Representation:

```
HashMap Internal Structure:
Bucket Array:
[0] → null
[1] → (key1, value1) → null
[2] → (key2, value2) → (key3, value3) → null  (collision)
[3] → null
[4] → (key4, value4) → null

Key-Value Pair Structure:
┌─────────────┐
│    Key      │
├─────────────┤
│   Value     │
├─────────────┤
│ Next Node   │ ← For collision handling
└─────────────┘
```

Key Characteristics:

- **Access Time**: O(1) average, O(n) worst case
- **Insertion**: O(1) average
- **Order**: No guaranteed order
- **Null Values**: Allows one null key, multiple null values
- **Thread Safety**: Not thread-safe

LinkedHashMap

Description: Hash table with linked list to maintain insertion order.

Visual Representation:

```
LinkedHashMap:
HashMap + Doubly Linked List for order

Hash Table:        Insertion Order:
[0] → (A,1)       [HEAD] ↔ (A,1) ↔ (B,2) ↔ (C,3) ↔ [TAIL]
[1] → (B,2)
[2] → (C,3)
```

Key Characteristics:

- **Access Time**: O(1) average
- **Order**: Maintains insertion order or access order
- **Memory**: Higher overhead than HashMap
- **Use Case**: LRU cache implementation

TreeMap

Description: Red-Black tree implementation of NavigableMap.

Visual Representation:

```
TreeMap (Red-Black Tree):
       ("Dog",4)(B)
      /           \
  ("Cat",3)(R)   ("Horse",5)(R)
  /        \    /           \
("Ant",1)(B) ("Cow",2)(B) ("Elephant",6)(B) ("Lion",7)(B)
```

Key Characteristics:

- **Access Time**: O(log n)
- **Insertion**: O(log n)
- **Order**: Sorted by keys (natural or custom comparator)
- **Null Keys**: Not allowed
- **Thread Safety**: Not thread-safe

PERFORMANCE COMPARISON

Time Complexity Summary

Operation	ArrayList	LinkedList	HashSet	TreeSet	HashMap	TreeMap
Add	O(1)*	O(1)	O(1)	O(log n)	O(1)	O(log n)
Remove	O(n)	O(n)	O(1)	O(log n)	O(1)	O(log n)
Contains	O(n)	O(n)	O(1)	O(log n)	O(1)	O(log n)
Get	O(1)	O(n)	N/A	N/A	O(1)	O(log n)

*ArrayList add is O(n) when resizing is needed

Space Complexity

- **ArrayList**: O(n) + extra capacity
- **LinkedList**: O(n) + pointer overhead
- **HashSet**: O(n) + bucket array
- **TreeSet**: O(n) + tree structure overhead
- **HashMap**: O(n) + bucket array
- **TreeMap**: O(n) + tree structure overhead

THREAD SAFETY

Thread-Safe Alternatives

- **Vector**: Thread-safe version of ArrayList
- **Hashtable**: Thread-safe version of HashMap
- **Collections.synchronizedXxx()**: Wrapper methods
- **ConcurrentHashMap**: High-performance thread-safe Map
- **CopyOnWriteArrayList**: Thread-safe List for read-heavy scenarios

Synchronization Example

java

```java
// Creating thread-safe collections
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
Map<String, String> syncMap = Collections.synchronizedMap(new HashMap<>());
```

CHOOSING THE RIGHT COLLECTION

Use ArrayList when:

- You need fast random access to elements
- You do more reading than writing

- You need to access elements by index

Use LinkedList when:

- You frequently add/remove elements at the beginning
- You implement stacks, queues, or deques
- You don't need random access

Use HashSet when:

- You need fast lookups and don't care about order
- You want to ensure uniqueness
- You need best performance for basic operations

Use TreeSet when:

- You need elements in sorted order
- You need range operations (subSet, headSet, tailSet)
- You want to maintain elements in natural order

Use HashMap when:

- You need fast key-value lookups
- Order doesn't matter
- You want best performance for basic operations

Use TreeMap when:

- You need key-value pairs in sorted order
- You need range operations on keys
- You want to maintain keys in natural order

BEST PRACTICES

1. **Choose the right collection based on your use case**
2. **Specify initial capacity when size is known**
3. **Use generics for type safety**
4. **Consider thread safety requirements**
5. **Override equals() and hashCode() for custom objects in hash-based collections**
6. **Use enhanced for-loop or iterators for traversal**
7. **Consider using utility classes like Collections and Arrays**

Common Pitfalls

1. **Modifying collection during iteration** - Use Iterator.remove()
2. **Not overriding equals/hashCode** - Leads to unexpected behavior in hash-based collections
3. **Using raw types** - Causes ClassCastException
4. **Ignoring capacity settings** - Leads to unnecessary resizing
5. **Assuming thread safety** - Most collections are not thread-safe