Java Collection Framework: Complete Guide (0 to 100x Developer)

Table of Contents

1. Introduction to Collection Framework
2. Collection Hierarchy
3. List Interface Implementations
4. Set Interface Implementations
5. Map Interface Implementations
6. Queue Interface Implementations
7. Time & Space Complexity Summary
8. Practice Questions
9. Advanced Implementation Details
10. Best Practices & Tips

---

1. Introduction to Collection Framework {#introduction}

The Java Collection Framework is a unified architecture for representing and manipulating collections of objects. It provides:

- **Interfaces**: Abstract data types that represent collections
- **Implementations**: Concrete implementations of collection interfaces
- **Algorithms**: Methods that perform useful computations on collections

Core Interfaces

```
Collection (root interface)
├── List (ordered, allows duplicates)
├── Set (no duplicates)
└── Queue (processing elements in specific order)

Map (separate hierarchy, key-value pairs)
```

---

2. Collection Hierarchy {#hierarchy}

```
Collection<E>
├── List<E>
│   ├── ArrayList<E>
│   ├── LinkedList<E>
│   └── Vector<E>
├── Set<E>
│   ├── HashSet<E>
│   ├── LinkedHashSet<E>
│   └── TreeSet<E>
└── Queue<E>
    ├── LinkedList<E>
    ├── PriorityQueue<E>
    └── ArrayDeque<E>

Map<K,V>
├── HashMap<K,V>
├── LinkedHashMap<K,V>
├── TreeMap<K,V>
└── Hashtable<K,V>
```

---

3. List Interface Implementations {#list-implementations}

ArrayList

Internal Structure: Dynamic array that resizes automatically

Key Characteristics:

- Maintains insertion order
- Allows duplicates and null values
- Not thread-safe
- Default initial capacity: 10
- Growth factor: 1.5x (new capacity = old capacity + old capacity/2)

Time Complexity:

- Access by index: O(1)
- Search: O(n)
- Insertion at end: O(1) amortized, O(n) worst case
- Insertion at specific index: O(n)
- Deletion: O(n)

Space Complexity: O(n)

When to Use:

- Frequent random access by index
- More reads than writes
- When you know approximate size

Example Implementation:

```java
public class SimpleArrayList<E> {
    private Object[] elementData;
    private int size = 0;
    private static final int DEFAULT_CAPACITY = 10;

    public SimpleArrayList() {
        elementData = new Object[DEFAULT_CAPACITY];
    }

    public boolean add(E e) {
        ensureCapacity();
        elementData[size++] = e;
        return true;
    }

    private void ensureCapacity() {
        if (size == elementData.length) {
            int newCapacity = elementData.length + (elementData.length >> 1); // 1.5x
            elementData = Arrays.copyOf(elementData, newCapacity);
        }
    }

    @SuppressWarnings("unchecked")
    public E get(int index) {
        if (index >= size || index < 0) {
            throw new IndexOutOfBoundsException();
        }
        return (E) elementData[index];
    }
}
```

LinkedList

Internal Structure: Doubly linked list

Key Characteristics:

- Maintains insertion order
- Allows duplicates and null values
- Not thread-safe
- Implements both List and Deque interfaces

Time Complexity:

- Access by index: O(n)
- Search: O(n)
- Insertion at beginning/end: O(1)
- Insertion at specific index: O(n)
- Deletion at beginning/end: O(1)
- Deletion at specific index: O(n)

Space Complexity: O(n) + extra memory for node pointers

When to Use:

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

Example Implementation:

java

```java
public class SimpleLinkedList<E> {
    private Node<E> first;
    private Node<E> last;
    private int size = 0;

    private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;

        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }

    public boolean add(E e) {
        linkLast(e);
        return true;
    }

    private void linkLast(E e) {
        final Node<E> l = last;
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode;
        if (l == null)
            first = newNode;
        else
            l.next = newNode;
        size++;
    }
}
```

Vector

Internal Structure: Similar to ArrayList but synchronized

Key Characteristics:

- Thread-safe (synchronized methods)
- Growth factor: 2x (doubles in size)
- Legacy class (use ArrayList instead)

Time Complexity: Same as ArrayList but with synchronization overhead **Space Complexity**: O(n)

---

4. Set Interface Implementations {#set-implementations}

HashSet

Internal Structure: Hash table (backed by HashMap)

Key Characteristics:

- No duplicates
- No guaranteed order
- Allows one null value
- Not thread-safe
- Default load factor: 0.75

Time Complexity:

- Add: O(1) average, O(n) worst case
- Remove: O(1) average, O(n) worst case
- Contains: O(1) average, O(n) worst case

Space Complexity: O(n)

When to Use:

- Need fast lookups
- Don't care about order
- Want to eliminate duplicates

Example Implementation:

java

```
public class SimpleHashSet<E> {
    private HashMap<E, Object> map;
    private static final Object PRESENT = new Object();

    public SimpleHashSet() {
        map = new HashMap<>();
    }

    public boolean add(E e) {
        return map.put(e, PRESENT) == null;
    }

    public boolean contains(Object o) {
        return map.containsKey(o);
    }

    public boolean remove(Object o) {
        return map.remove(o) == PRESENT;
    }
}
```

LinkedHashSet

Internal Structure: Hash table + doubly-linked list

Key Characteristics:

- Maintains insertion order
- No duplicates
- Slightly slower than HashSet due to linked list overhead

Time Complexity: Same as HashSet **Space Complexity**: O(n) + extra memory for maintaining order

TreeSet

Internal Structure: Red-Black Tree (self-balancing BST)

Key Characteristics:

- Sorted order (natural or custom comparator)
- No duplicates
- No null values
- Not thread-safe

Time Complexity:

- Add: O(log n)
- Remove: O(log n)
- Contains: O(log n)

Space Complexity: O(n)

When to Use:

- Need sorted data
- Range queries
- Want to find min/max efficiently

---

5. Map Interface Implementations {#map-implementations}

HashMap

Internal Structure: Array of buckets (each bucket is a linked list or red-black tree)

Key Characteristics:

- Key-value pairs
- No duplicate keys
- One null key allowed, multiple null values
- Not thread-safe
- Load factor: 0.75, Initial capacity: 16

Time Complexity:

- Get: O(1) average, O(n) worst case
- Put: O(1) average, O(n) worst case
- Remove: O(1) average, O(n) worst case

Space Complexity: O(n)

Internal Working:

1. Hash function calculates hash code for key
2. Index = hash & (capacity - 1)
3. If collision occurs, use chaining (linked list)
4. If chain length > 8 and capacity > 64, convert to red-black tree

Example Implementation:

java

```java
public class SimpleHashMap<K, V> {
    private Node<K, V>[] table;
    private int size;
    private int threshold;
    private static final int DEFAULT_CAPACITY = 16;
    private static final float LOAD_FACTOR = 0.75f;

    static class Node<K, V> {
        final int hash;
        final K key;
        V value;
        Node<K, V> next;

        Node(int hash, K key, V value, Node<K, V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

    @SuppressWarnings("unchecked")
    public SimpleHashMap() {
        table = new Node[DEFAULT_CAPACITY];
        threshold = (int) (DEFAULT_CAPACITY * LOAD_FACTOR);
    }

    public V put(K key, V value) {
        int hash = hash(key);
        int index = hash & (table.length - 1);

        // Check if key already exists
        Node<K, V> current = table[index];
        while (current != null) {
            if (current.hash == hash && Objects.equals(current.key, key)) {
                V oldValue = current.value;
                current.value = value;
                return oldValue;
            }
            current = current.next;
        }

        // Add new node
        Node<K, V> newNode = new Node<>(hash, key, value, table[index]);
        table[index] = newNode;
        size++;
        if (size > threshold) {
            resize();
        }

        return null;
    }

    public V get(K key) {
        int hash = hash(key);
        int index = hash & (table.length - 1);

        Node<K, V> current = table[index];
        while (current != null) {
            if (current.hash == hash && Objects.equals(current.key, key)) {
                return current.value;
            }
            current = current.next;
        }

        return null;
    }

    private int hash(K key) {
        if (key == null) return 0;
        int h = key.hashCode();
        return h ^ (h >>> 16); // XOR higher bits with lower bits
    }

    private void resize() {
        // Implementation for resizing hash table
        // Double the capacity and rehash all elements
    }
}
```

LinkedHashMap

Internal Structure: HashMap + doubly-linked list

Key Characteristics:

- Maintains insertion order (or access order)
- Useful for implementing LRU cache
- Slightly slower than HashMap

Time Complexity: Same as HashMap **Space Complexity**: O(n) + extra memory for maintaining order

TreeMap

Internal Structure: Red-Black Tree

Key Characteristics:

- Sorted by keys (natural or custom comparator)
- No null keys
- Not thread-safe

Time Complexity:

- Get: O(log n)
- Put: O(log n)
- Remove: O(log n)

Space Complexity: O(n)

---

6. Queue Interface Implementations {#queue-implementations}

PriorityQueue

Internal Structure: Binary heap (array-based)

Key Characteristics:

- Elements are ordered by priority (natural or custom comparator)
- Not thread-safe
- No null elements

Time Complexity:

- Add: O(log n)
- Poll/Remove: O(log n)
- Peek: O(1)

Space Complexity: O(n)

ArrayDeque

Internal Structure: Resizable circular array

Key Characteristics:

- Double-ended queue
- No capacity restrictions

- Not thread-safe
- No null elements

Time Complexity:

- Add/Remove at ends: O(1) amortized
- Random access: Not supported

Space Complexity: O(n)

---

7. Time & Space Complexity Summary {#complexity-summary}

Collection	Get/Access	Search	Insert	Delete	Space
ArrayList	O(1)	O(n)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)*	O(1)*	O(n)
HashSet	N/A	O(1)	O(1)	O(1)	O(n)
TreeSet	N/A	O(log n)	O(log n)	O(log n)	O(n)
HashMap	O(1)	O(1)	O(1)	O(1)	O(n)
TreeMap	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
PriorityQueue	O(1)**	O(n)	O(log n)	O(log n)	O(n)

*At known position
**Only peek operation

---

8. Practice Questions {#practice-questions}

Beginner Level

Q1: Which collection would you use to store unique student IDs and retrieve them quickly? Answer: HashSet - provides O(1) lookup time and automatically handles uniqueness.

Q2: You need to maintain a list of tasks and frequently add/remove from the beginning. Which collection is best? Answer: LinkedList - O(1) insertion/deletion at the beginning, unlike ArrayList which requires shifting elements (O(n)).

Q3: How does ArrayList handle capacity when it's full? Answer: It creates a new array with 1.5x capacity and copies all elements to the new array.

Intermediate Level

Q4: Explain why HashMap allows one null key but multiple null values. Answer: Keys must be unique, so only one null key is allowed. Values can be duplicated, so multiple null values are permitted.

Q5: When would TreeSet be preferred over HashSet? Answer: When you need sorted data, range operations (subSet, headSet, tailSet), or want to find min/max elements efficiently.

Q6: What happens when two objects have the same hashCode() in HashMap? Answer: A collision occurs. HashMap uses chaining - multiple key-value pairs are stored in the same bucket as a linked list (or red-black tree if chain length > 8).

Advanced Level

Q7: Design a LRU Cache using Java collections.

```java
public class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private final int capacity;

    public LRUCache(int capacity) {
        super(capacity, 0.75f, true); // access-order
        this.capacity = capacity;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > capacity;
    }
}
```

Q8: Why is String immutable beneficial for HashMap keys? Answer: Immutable strings ensure hashCode() remains constant, preventing keys from getting lost in HashMap after insertion. If keys were mutable and their hash changed, they'd become unreachable.

Q9: Implement a custom ArrayList that automatically removes duplicates.

```java
public class UniqueArrayList<E> extends ArrayList<E> {
    private Set<E> seen = new HashSet<>();

    @Override
    public boolean add(E e) {
        if (seen.add(e)) {
            return super.add(e);
        }
        return false;
    }
}
```

9. Advanced Implementation Details {#advanced-implementation}

HashMap Internal Mechanics

Hash Function: Java uses a sophisticated hash function to minimize collisions:

```java
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

Collision Resolution:

1. **Separate Chaining**: Elements with same hash are stored in linked lists
2. **Tree Conversion**: When chain length exceeds 8 and table size > 64, linked list converts to red-black tree for O(log n) worst-case performance

Load Factor Impact:

- Higher load factor: More memory efficient but more collisions
- Lower load factor: Fewer collisions but wastes memory
- Default 0.75 provides good balance

ArrayList Growth Strategy

Why 1.5x instead of 2x?

- 2x growth can lead to memory fragmentation
- 1.5x allows better memory reuse
- Mathematical analysis shows 1.5x provides optimal amortized performance

TreeSet/TreeMap Red-Black Tree Properties

Why Red-Black Tree over regular BST?

- Guarantees O(log n) operations even with sorted input
- Self-balancing prevents worst-case O(n) performance
- More efficient than AVL trees for insertions/deletions

Red-Black Tree Rules:

1. Every node is red or black
2. Root is black

3. Red nodes have black children
4. Every path from root to null has same number of black nodes

---

10. Best Practices & Tips {#best-practices}

Choosing the Right Collection

Use ArrayList when:

- Frequent random access by index
- More reads than writes
- Memory usage is important

Use LinkedList when:

- Frequent insertions/deletions at beginning/end
- Implementing stacks or queues
- Don't need random access

Use HashMap when:

- Need fast key-value lookups
- Keys are well-distributed (good hashCode())
- Don't need sorted order

Use TreeMap when:

- Need sorted keys
- Range operations required
- Want to iterate in sorted order

Performance Optimization Tips

1. **Initialize with appropriate capacity**:

```java
// Good: Avoid multiple resizing operations
List<String> list = new ArrayList<>(1000);

// Bad: Will resize multiple times
List<String> list = new ArrayList<>();
```

2. **Use appropriate collection for the use case**:

```java
// Good: For fast lookups
Set<String> uniqueItems = new HashSet<>();

// Bad: O(n) lookup time
List<String> uniqueItems = new ArrayList<>();
```

3. **Implement proper equals() and hashCode()**:

```java
public class Person {
    private String name;
    private int age;

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof Person)) return false;
        Person person = (Person) obj;
        return age == person.age && Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
```

4. **Use StringBuilder for string concatenation in loops**:

```java
// Good
StringBuilder sb = new StringBuilder();
for (String s : strings) {
    sb.append(s);
}

// Bad: Creates new String objects in each iteration
String result = "";
for (String s : strings) {
    result += s;
}
```

Thread Safety Considerations

For thread-safe collections, use:

- `Collections.synchronizedList(list)`
- `ConcurrentHashMap` instead of `HashMap`
- `Vector` instead of `ArrayList` (though less preferred)
- `CopyOnWriteArrayList` for read-heavy scenarios

Memory Considerations

1. **ArrayList vs LinkedList memory usage**:
   - ArrayList: 4 bytes per reference + array overhead
   - LinkedList: 24 bytes per node (object header + 3 references)
2. **HashMap memory optimization**:
   - Choose initial capacity wisely: `new HashMap<>(expectedSize / 0.75f + 1)`
   - Consider `HashMap<Integer, Integer>` vs `TIntIntHashMap` for primitive types

Common Pitfalls to Avoid

1. **Modifying collection while iterating**:

```java
// Wrong: ConcurrentModificationException
for (String item : list) {
    if (shouldRemove(item)) {
        list.remove(item); // Exception!
    }
}

// Correct: Use iterator
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    if (shouldRemove(it.next())) {
        it.remove();
    }
}
```

2. **Using raw types**:

```java
```

```
// Wrong: No type safety
List list = new ArrayList();

// Correct: Generic types
List<String> list = new ArrayList<>();
```

3. **Not overriding hashCode() when overriding equals()**:

```java
// Wrong: Will break HashMap/HashSet
public boolean equals(Object obj) { /* implementation */ }
// Missing hashCode() override

// Correct: Always override both
public boolean equals(Object obj) { /* implementation */ }
public int hashCode() { /* implementation */ }
```

Conclusion

Mastering Java Collections is crucial for becoming a proficient developer. Understanding the internal workings, time complexities, and appropriate use cases will help you write efficient and maintainable code. Practice implementing these collections from scratch to deepen your understanding, and always consider the specific requirements of your use case when choosing a collection type.

Remember: The best collection is the one that fits your specific use case in terms of performance, memory usage, and functionality requirements.