Top 30 Java Collections Topics with Examples

https://www.linkedin.com/in/venkattramana

Disclaimer

This document is intended for educational purposes only. While every effort has been made to ensure the accuracy and completeness of the information provided on Java Collections Framework topics—including interfaces, classes, methods, examples, and best practices—there may be errors, omissions, or outdated content due to continuous advancements in Java technology.

The code samples provided are for demonstration purposes and may require adaptation for use in production environments. The author and publisher do not provide any warranties or guarantees regarding the accuracy, reliability, or suitability of the material for specific purposes. Use of this material and any corresponding code samples is at the user's own risk.

Java and all Java-based marks are trademarks or registered trademarks of Oracle Corporation and/or its affiliates. This document is an independent educational supplement and is not formally associated with or endorsed by Oracle Corporation or its affiliates.

Readers are encouraged to refer to the official Java documentation and authorized reference materials to supplement their understanding and to ensure they are using up-to-date practices.

Feel free to adjust the wording or add institution/author details as appropriate for your course or context!

Java Collections: Complete Topics With Detailed Descriptions and Examples

The Java Collections Framework (JCF) is a unified architecture for storing and manipulating groups of objects. It provides interfaces and classes for different types of collections and algorithms for manipulating them. Below are the core topics in Java Collections, each with a clear explanation and a thorough programming example.

1. Collection Interface

Description:

The `Collection` interface is the root of the Java Collections framework. It defines the basic methods like add, remove, size, iterator, and clear that all collections implement.

Example:

```
import java.util.Collection;
import java.util.ArrayList;

public class CollectionDemo {
    public static void main(String[] args) {
        Collection<String> items = new ArrayList<>();
        items.add("Book");
        items.add("Pen");
        items.add("Notebook");
        System.out.println("Items: " + items); // Output: Items: [Book,
Pen, Notebook]
    }
}
```

2. List Interface

Description:

A `List` is an ordered collection that can contain duplicate elements and supports index-based access. Common implementations are `ArrayList`, `LinkedList`, `Vector`, and `Stack`.

Example (ArrayList):

```
import java.util.List;
import java.util.ArrayList;

public class ListExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");
        System.out.println("Fruits List: " + fruits); // Output: Fruits List:
[Apple, Banana, Mango]
    }
}
```

3. Set Interface

Description:

A `Set` is a collection that cannot contain duplicate elements. The order may or may not be preserved, based on implementation. Main implementations are `HashSet`, `LinkedHashSet`, and `TreeSet`.

Example (HashSet):

```
import java.util.Set;
import java.util.HashSet;

public class SetExample {
    public static void main(String[] args) {
        Set<String> names = new HashSet<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Alice"); // Duplicate, will be ignored
        System.out.println("Names: " + names); // Output: Names: [Alice, Bob]
    }
}
```

4. Queue Interface

Description:

A `Queue` is used to hold multiple elements before processing, usually in FIFO (First-In-First-Out) order. Implementations include `LinkedList`, `PriorityQueue`, and `ArrayDeque`.

Example (PriorityQueue):

```
import java.util.Queue;
import java.util.PriorityQueue;

public class QueueExample {
    public static void main(String[] args) {
        Queue<Integer> numbers = new PriorityQueue<>();
        numbers.add(10);
        numbers.add(5);
        numbers.add(15);
        System.out.println(numbers.poll()); // Output: 5 (smallest element)
    }
}
```

5. Deque Interface

Description:

The `Deque` (Double-ended Queue) interface allows elements to be added or removed from both ends. Implementations include `ArrayDeque` and `LinkedList`.

Example (ArrayDeque):

```java
import java.util.Deque;
import java.util.ArrayDeque;

public class DequeExample {
    public static void main(String[] args) {
        Deque<String> deque = new ArrayDeque<>();
        deque.addFirst("Front");
        deque.addLast("Back");
        System.out.println(deque); // Output: [Front, Back]
    }
}
```

6. Map Interface

Description:

A `Map` is an object that maps keys to values. Each key can map to at most one value. Implementations include `HashMap`, `LinkedHashMap`, `TreeMap`, and `Hashtable`.

Example (HashMap):

```
import java.util.Map;
import java.util.HashMap;

public class MapExample {
    public static void main(String[] args) {
        Map<String, Integer> ages = new HashMap<>();
        ages.put("John", 25);
        ages.put("Jane", 30);
        System.out.println(ages.get("John")); // Output: 25
    }
}
```

7. Stack Class

Description:

A `Stack` is a subclass of `Vector` representing a LIFO (Last-In-First-Out) stack of objects. Primary methods are push (add), pop (remove), and peek (examine top).

Example:

```
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        stack.push("First");
        stack.push("Second");
        System.out.println(stack.pop()); // Output: Second
        System.out.println(stack.peek()); // Output: First
    }
}
```

8. Algorithms Class (Collections Utility Methods)

Description:

The `Collections` class provides static methods for sorting, searching, shuffling, and reversing collections.

Example:

```java
import java.util.Collections;
import java.util.ArrayList;

public class CollectionsDemo {
    public static void main(String[] args) {
        ArrayList<Integer> nums = new ArrayList<>();
        nums.add(3); nums.add(1); nums.add(2);
        Collections.sort(nums);
        System.out.println(nums); // Output: [1, 2, 3]
        Collections.reverse(nums);
        System.out.println(nums); // Output: [3, 2, 1]
    }
}
```

9. Comparable and Comparator Interfaces

Description:

Used for sorting objects. `Comparable` defines natural ordering using the `compareTo()` method; `Comparator` defines custom ordering with `compare()`.

Example (Comparator):

```java
import java.util.*;

class Student implements Comparable<Student> {
    String name; int marks;
    Student(String n, int m) { name=n; marks=m; }
    public int compareTo(Student s) { return this.marks - s.marks; }
}

public class ComparatorDemo {
    public static void main(String[] args) {
        ArrayList<Student> students = new ArrayList<>();
        students.add(new Student("A", 90));
        students.add(new Student("B", 80));
        Collections.sort(students); // sorts by marks

        for(Student s : students) {
            System.out.println(s.name + " " + s.marks);
        }
    }
}
```

10. Legacy Classes (Vector, Hashtable, Enumeration)

Description:

Pre-framework collection classes, still maintained for backward compatibility. `Vector` is like an ArrayList but synchronized. `Hashtable` is like HashMap but synchronized.

Example (Vector):

```java
import java.util.Vector;

public class VectorExample {
    public static void main(String[] args) {
        Vector<Integer> vector = new Vector<>();
        vector.add(1); vector.add(2);
        System.out.println(vector); // Output: [1, 2]
    }
}
```

11. Synchronized Collections and Concurrent Collections

Description:

Java provides thread-safe collections like `Collections.synchronizedList(new ArrayList<>())`, `CopyOnWriteArrayList`, `ConcurrentHashMap` for use in multithreaded programs.

Example (CopyOnWriteArrayList):

```java
import java.util.concurrent.CopyOnWriteArrayList;

public class ThreadSafeListExample {
    public static void main(String[] args) {
        CopyOnWriteArrayList<String> threadSafeList = new
CopyOnWriteArrayList<>();
        threadSafeList.add("One");
        threadSafeList.add("Two");
        System.out.println(threadSafeList); // Output: [One, Two]
    }
}
```

12. NavigableSet and NavigableMap Interfaces

Description:

`NavigableSet` and `NavigableMap` extend the `SortedSet` and `SortedMap` interfaces, respectively. They provide navigation methods for searching for entries that are closest to given search targets (lower, floor, ceiling, higher, pollFirst, pollLast, etc.).

Example (TreeSet as NavigableSet):

```
import java.util.NavigableSet;
import java.util.TreeSet;

public class NavigableSetExample {
    public static void main(String[] args) {
        NavigableSet<Integer> numbers = new TreeSet<>();
        numbers.add(10); numbers.add(20); numbers.add(30); numbers.add(40);

        System.out.println(numbers.floor(25)); // Output: 20
        System.out.println(numbers.ceiling(25)); // Output: 30
        System.out.println(numbers.descendingSet()); // Output: [40, 30, 20, 10]
    }
}
```

13. EnumSet and EnumMap

Description:

`EnumSet` is a specialized Set implementation for use with enum types—very efficient and compact. `EnumMap` is a specialized Map for enum keys, providing fast, space-efficient mapping.

Example (EnumSet & EnumMap):

```
import java.util.EnumSet;
import java.util.EnumMap;

enum Day { MON, TUE, WED, THU, FRI }

public class EnumCollectionsExample {
    public static void main(String[] args) {
        EnumSet<Day> weekends = EnumSet.of(Day.SAT, Day.SUN); // will cause error because enums SAT and SUN are not defined; correct usage requires defined enum constants only.
        EnumMap<Day, String> dayTasks = new EnumMap<>(Day.class);
        dayTasks.put(Day.MON, "Work");
        dayTasks.put(Day.FRI, "Relax");

        System.out.println(dayTasks); // Output: {MON=Work, FRI=Relax}
    }
}
```

14. WeakHashMap

Description:

A `WeakHashMap` is a Map that stores its keys using weak references. If a key is no longer in ordinary use, it can be garbage-collected, and the mapping is removed automatically.

Example:

```
import java.util.WeakHashMap;

public class WeakHashMapExample {
    public static void main(String[] args) {
        WeakHashMap<Object, String> map = new WeakHashMap<>();
        Object key = new Object();
        map.put(key, "Weak Reference Value");

        System.out.println("Before: " + map);
        key = null;
        System.gc();
        // The map may or may not be empty now; it depends on garbage
collection timing
        System.out.println("After GC: " + map);
    }
}
```

15. IdentityHashMap

Description:

`IdentityHashMap` uses the `==` operator (reference equality) instead of `.equals()` for comparing keys. Useful for use-cases needing reference equality semantics.

Example:

```
import java.util.IdentityHashMap;

public class IdentityHashMapExample {
    public static void main(String[] args) {
        IdentityHashMap<String, Integer> map = new IdentityHashMap<>();
        String a = new String("key");
        String b = new String("key");
        map.put(a, 1);
        map.put(b, 2);

        System.out.println(map.size()); // Output: 2, since 'a' and 'b' are not
the same reference
    }
}
```

16. LinkedHashMap and LinkedHashSet

Description:

These maintain a doubly-linked list running through their entries, maintaining insertion order. `LinkedHashMap` also supports a "least recently accessed" order, useful for caches.

Example (LinkedHashMap):

```
import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashMapExample {
    public static void main(String[] args) {
        Map<String, Integer> orderedMap = new LinkedHashMap<>();
        orderedMap.put("One", 1);
        orderedMap.put("Two", 2);
        orderedMap.put("Three", 3);

        System.out.println(orderedMap); // Output: {One=1, Two=2, Three=3}
    }
}
```

17. BlockingQueue and Concurrent Collections

Description:

`BlockingQueue` is designed for use in producer-consumer scenarios. It supports thread-safe methods such as put and take, which wait when the queue is empty or full.

Example (LinkedBlockingQueue):

```java
import java.util.concurrent.*;

public class BlockingQueueExample {
    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<String> queue = new LinkedBlockingQueue<>();
        queue.put("Hello");
        System.out.println(queue.take()); // Output: Hello
    }
}
```

18. Unmodifiable Collections

Description:

Unmodifiable collections are wrappers that make an existing collection read-only. Attempts to modify them will result in `UnsupportedOperationException`. These are created using methods like `Collections.unmodifiableList()`, `unmodifiableSet()`, or via factory methods from Java 9's `List.of()`, `Set.of()`, and `Map.of()`.

Example:

```java
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;

public class UnmodifiableCollectionExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Data");
        list.add("Info");
        List<String> unmodifiableList = Collections.unmodifiableList(list);
        System.out.println(unmodifiableList); // Output: [Data, Info]
        // unmodifiableList.add("New"); // Throws UnsupportedOperationException
    }
}
```

19. Singleton and Empty Collections

Description:

Java offers utility methods to create singleton collections (with a single element) and empty, immutable collections. Useful for returning unmodifiable “default” values.

Example:

```java
import java.util.Collections;
import java.util.Set;

public class SingletonEmptyExample {
    public static void main(String[] args) {
        Set<String> singleton = Collections.singleton("ONLY_ONE");
        Set<String> empty = Collections.emptySet();
        System.out.println(singleton); // Output: [ONLY_ONE]
        System.out.println(empty); // Output: []
    }
}
```

20. BitSet

Description:

`BitSet` is a special-purpose class for creating arrays of bits (booleans), useful for performance-critical applications where compact storage and bitwise operations are needed.

Example:

```
import java.util.BitSet;

public class BitSetExample {
    public static void main(String[] args) {
        BitSet bits = new BitSet(8);
        bits.set(2);
        bits.set(4);
        System.out.println(bits); // Output: {2, 4}
        System.out.println(bits.get(2)); // Output: true
        System.out.println(bits.get(3)); // Output: false
    }
}
```

21. Arrays Utility Class

Description:

While not a “collection” per se, `Arrays` provides static methods for manipulating arrays, including conversions to lists, sorting, searching, filling, and comparison.

Example:

```java
import java.util.Arrays;

public class ArraysUtilityExample {
    public static void main(String[] args) {
        int[] numbers = {3, 1, 2};
        Arrays.sort(numbers);
        System.out.println(Arrays.toString(numbers)); // Output: [1, 2, 3]
        System.out.println(Arrays.binarySearch(numbers, 2)); // Output: 1
    }
}
```

22. Custom Collection Implementation

Description:

You can implement your own collection classes by extending `AbstractCollection`, `AbstractList`, or other skeletal implementations, ensuring minimum effort to provide custom functionality.

Example:

```java
import java.util.AbstractList;

public class CustomList<T> extends AbstractList<T> {
    private final T[] data;

    public CustomList(T[] data) { this.data = data; }
    public T get(int index) { return data[index]; }
    public int size() { return data.length; }
}

public class CustomCollectionExample {
    public static void main(String[] args) {
        CustomList<String> list = new CustomList<>(new String[]{"A", "B", "C"});
        System.out.println(list); // Output: [A, B, C]
    }
}
```

23. Spliterator

Description:

A `Spliterator` is a special iterator for traversing and partitioning elements, supporting parallelism (used heavily in Java Streams API introduced in Java 8).

Example:

```
import java.util.ArrayList;
import java.util.Spliterator;

public class SpliteratorExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("A"); list.add("B"); list.add("C");
        Spliterator<String> spliterator = list.spliterator();
        spliterator.forEachRemaining(System.out::println); // Output: A B C
    }
}
```

24. NavigableMap Methods with Submaps

Description:

`NavigableMap` methods allow precise range views via `subMap()`, `headMap()`, and `tailMap()` that can be bounded inclusively or exclusively.

Example:

```
import java.util.NavigableMap;
import java.util.TreeMap;

public class NavigableMapRangeExample {
    public static void main(String[] args) {
        NavigableMap<Integer, String> map = new TreeMap<>();
        map.put(1,"A"); map.put(2,"B"); map.put(3,"C"); map.put(4,"D");
        System.out.println(map.subMap(2, true, 4, false)); // Output: {2=B, 3=C}
    }
}
```

25. Collectors and Streams Integration

Description:
With the introduction of Java 8, the Collections Framework integrates closely with the Streams API. Streams allow you to process collections in a functional-style (filter, map, reduce) and use collectors for gathering results into lists, sets, maps, and even custom collections.

Example (Collecting to Map and Filtering):

```
import java.util.*;
import java.util.stream.Collectors;

public class StreamsCollectorsExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Ann", "Bob", "Andy", "Alice");
        Map<Character, List<String>> grouped = names.stream()
            .filter(s -> s.startsWith("A"))
            .collect(Collectors.groupingBy(s -> s.charAt(0)));
        System.out.println(grouped); // Output: {A=[Ann, Andy, Alice]}
    }
}
```

26. Collection Views (keySet, entrySet, values)

Description:
Maps allow you to view their content as separate collections—for keys (`keySet()`), values (`values()`), or entries (`entrySet()`). These "views" are backed collections, providing dynamic linkage to the map.

Example:

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class MapViewsExample {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Apple", 1);
        map.put("Banana", 2);
        Set<String> keys = map.keySet();
        System.out.println(keys); // Output: [Apple, Banana]
    }
}
```

27. RandomAccess Interface

Description:

A marker interface for lists that support fast (generally constant-time) random access, like `ArrayList`. Used by algorithms to optimize behavior depending on collection type.

Example:

```
import java.util.ArrayList;
import java.util.RandomAccess;

public class RandomAccessExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        if (list instanceof RandomAccess) {
            System.out.println("Supports fast random access"); // Output:
Supports fast random access
        }
    }
}
```

28. ArrayList vs. LinkedList vs. Vector

Description:
Although all are list implementations, their performance characteristics differ:

- **ArrayList:** Fast random access, resizing may be expensive, not synchronized.
- **LinkedList:** Fast insertions/deletions in middle, no fast random access.
- **Vector:** Synchronized version of ArrayList, legacy class.

Example (Demonstration):

```
import java.util.*;

public class ListComparisonExample {
    public static void main(String[] args) {
        List<String> arrayList = new ArrayList<>();
        List<String> linkedList = new LinkedList<>();
        List<String> vector = new Vector<>();
        arrayList.add("A");
        linkedList.add("B");
        vector.add("C");
        System.out.println(arrayList + ", " + linkedList + ", " + vector); //
Output: [A], [B], [C]
    }
}
```

29. Fail-Fast and Fail-Safe Iterators

Description:

- **Fail-Fast Iterators**: Throw `ConcurrentModificationException` if the collection is modified outside the iterator during iteration (e.g., `ArrayList`, `HashMap`)

- **Fail-Safe Iterators**: Do not throw exceptions, work on a cloned collection (`CopyOnWriteArrayList`, `ConcurrentHashMap`)

Example (Fail-Fast):

```
import java.util.*;

public class FailFastExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("A", "B"));
        Iterator<String> it = list.iterator();
        list.add("C"); // Modifying after getting iterator
        // it.next(); // Will throw ConcurrentModificationException if
called here
    }
}
```

30. Deep Copy vs. Shallow Copy in Collections

Description:

- **Shallow Copy:** Copies the reference of objects (e.g., using `clone()` on `ArrayList`).
- **Deep Copy:** Duplicates the actual objects by copying all nested objects.

Example (Shallow Copy):

```
import java.util.*;

public class ShallowCopyExample {
    public static void main(String[] args) {
        List<String> original = new ArrayList<>(Arrays.asList("X", "Y"));
        List<String> copy = new ArrayList<>(original);
        copy.set(0, "Z");
        System.out.println(original); // Output: [X, Y]
        System.out.println(copy);     // Output: [Z, Y]
    }
}
```

https://www.linkedin.com/in/venkattramana