**How to Use the Java Collections Framework – A Guide for Developers**

In your Java applications, you’ll typically work with various types of objects. And you might want to perform operations like sorting, searching, and iterating on these objects.

Prior to the introduction of the Collections framework in JDK 1.2, you would’ve used Arrays and Vectors to store and manage a group of objects. But they had their own share of drawbacks.

The Java Collections Framework aims to overcome these issues by providing high-performance implementations of common data structures. These allow you to focus on writing the application logic instead of focusing on low-level operations.

Then, the introduction of Generics in JDK 1.5 significantly improved the Java Collections Framework. Generics let you enforce type safety for objects stored in a collection, which enhances the robustness of your applications. You can read more about Java Generics [here](https://www.freecodecamp.org/news/generics-in-java/).

In this article, I will guide you through how to use the Java Collections Framework. We’ll discuss the different types of collections, such as Lists, Sets, Queues, and Maps. I’ll also provide a brief explanation of their key characteristics such as:

* Internal mechanisms
* Handling of duplicates
* Support for null values
* Ordering
* Synchronization
* Performance
* Key methods
* Common implementations

We’ll also walk through some code examples for better understanding, and I’ll touch on the Collections utility class and its usage.

**Table of Contents:**

1. [Understanding the Java Collections Framework](https://www.freecodecamp.org/news/java-collections-framework-reference-guide/#heading-understanding-the-java-collections-framework)
   * [Collection Interfaces](https://www.freecodecamp.org/news/java-collections-framework-reference-guide/#heading-collection-interfaces)
   * [Decoding the Java Collections Framework Hierarchy](https://www.freecodecamp.org/news/java-collections-framework-reference-guide/#heading-decoding-the-java-collections-framework-hierarchy)
   * [A Closer Look at Collection Interfaces](https://www.freecodecamp.org/news/java-collections-framework-reference-guide/#heading-a-closer-look-at-collection-interfaces)
2. [Java Collection Interfaces](https://www.freecodecamp.org/news/java-collections-framework-reference-guide/#heading-java-collection-interfaces)
   * [Lists](https://www.freecodecamp.org/news/java-collections-framework-reference-guide/#heading-lists)
   * [Sets](https://www.freecodecamp.org/news/java-collections-framework-reference-guide/#heading-sets)
   * [Queues](https://www.freecodecamp.org/news/java-collections-framework-reference-guide/#heading-queues)
   * [Maps](https://www.freecodecamp.org/news/java-collections-framework-reference-guide/#heading-maps)
3. [Collections Utility Class](https://www.freecodecamp.org/news/java-collections-framework-reference-guide/#heading-collections-utility-class)
4. [Conclusion](https://www.freecodecamp.org/news/java-collections-framework-reference-guide/#heading-conclusion)

**Understanding the Java Collections Framework**

According to the Java [documentation](https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html), “*A collection is an object that represents a group of objects. A collections framework is a unified architecture for representing and manipulating collections*.”

In simple terms, the Java Collections Framework helps you manage a group of objects and perform operations on them efficiently and in an organized way. It makes it easier to develop applications by offering various methods to handle groups of objects. You can add, remove, search, and sort objects effectively using the Java Collections Framework.

**Collection Interfaces**

In Java, an interface specifies a contract that must be fulfilled by any class that implements it. This means the implementing class must provide concrete implementations for all the methods declared in the interface.

In the Java Collections Framework, various collection interfaces like Set, List, and Queue extend the Collection interface, and they must adhere to the contract defined by the Collection interface.

**Decoding the Java Collections Framework Hierarchy**

Check out this neat diagram from this [article](https://medium.com/@mbanaee61/mastering-the-java-collections-framework-hierarchy-with-java-code-and-junit-testing-ab2eb87746ed) that illustrates the Java Collection Hierarchy:

We’ll start from the top and work down so you can understand what this diagram is showing:

1. At the root of the Java Collections Framework is the Iterable interface, which lets you iterate over the elements of a collection.
2. The Collection interface extends the Iterable interface. This means it inherits the properties and behavior of the Iterable interface and adds its own behavior for adding, removing, and retrieving elements.
3. Specific interfaces such as List, Set, and Queue further extend the Collection interface. Each of these interfaces has other classes implementing their methods. For example, ArrayList is a popular implementation of the List interface, HashSet implements the Set interface, and so on.
4. The Map interface is part of the Java Collections Framework, but it does not extend the Collection interface, unlike the others mentioned above.
5. All the interfaces and classes in this framework are part of the java.util package.

Note: A common source of confusion in the Java Collections Framework revolves around the difference between Collection and Collections. Collection is an interface in the framework, while Collections is a utility class. The Collections class provides static methods that perform operations on the elements of a collection.

**Java Collection Interfaces**

By now, you’re familiar with the different types of collections that form the foundation of the collections framework. Now we’ll take a closer look at the List, Set, Queue, and Map interfaces.

In this section, we'll discuss each of these interfaces while exploring their internal mechanisms. We'll examine how they handle duplicate elements and whether they support the insertion of null values. We'll also understand the ordering of elements during insertion and their support for synchronization, which deals with the concept of thread safety. Then we’ll walk through a few key methods of these interfaces and conclude by reviewing common implementations and their performance for various operations.

Before we begin, let's talk briefly about Synchronization and Performance.

* Synchronization controls access to shared objects by multiple threads, ensuring their integrity and preventing conflicts. This is crucial for maintaining thread safety.
* When choosing a collection type, one important factor is its performance during common operations like insertion, deletion, and retrieval. Performance is usually expressed using Big-O notation. You can learn more about it [here](https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/).

**Lists**

A List is an ordered or sequential collection of elements. It follows zero-based indexing, allowing the elements to be inserted, removed, or accessed using their index position.

1. **Internal mechanism**: A List is internally supported by either an array or a linked list, depending on the type of implementation. For example, an ArrayList uses an array, while a LinkedList uses a linked list internally. You can read more about LinkedList [here](https://www.freecodecamp.org/news/how-linked-lists-work/). A List dynamically resizes itself upon the addition or removal of elements. The indexing-based retrieval makes it a very efficient type of collection.
2. **Duplicates**: Duplicate elements are allowed in a List, which means there can be more than one element in a List with the same value. Any value can be retrieved based on the index at which it is stored.
3. **Null**: Null values are also allowed in a List. Since duplicates are permitted, you can also have multiple null elements.
4. **Ordering**: A List maintains insertion order, meaning the elements are stored in the same order they are added. This is helpful when you want to retrieve elements in the exact order they were inserted.
5. **Synchronization**: A List is not synchronized by default, which means it doesn't have a built-in way to handle access by multiple threads at the same time.
6. **Key methods**: Here are some key methods of a List interface: add(E element), get(int index), set(int index, E element), remove(int index), and size(). Let's look at how to use these methods with an example program.
7. import java.util.ArrayList;
8. import java.util.List;
9. public class ListExample {
10. public static void main(String[] args) {
11. // Create a list
12. List<String> list = new ArrayList<>();
13. // add(E element)
14. list.add("Apple");
15. list.add("Banana");
16. list.add("Cherry");
17. // get(int index)
18. String secondElement = list.get(1); // "Banana"
19. // set(int index, E element)
20. list.set(1, "Blueberry");
21. // remove(int index)
22. list.remove(0); // Removes "Apple"
23. // size()
24. int size = list.size(); // 2
25. // Print the list
26. System.out.println(list); // Output: [Blueberry, Cherry]
27. // Print the size of the list
28. System.out.println(size); // Output: 2
29. }
30. }
31. **Common implementations**: ArrayList, LinkedList, Vector, Stack
32. **Performance**: Typically, insert and delete operations are fast in both ArrayList and LinkedList. But fetching elements can be slow because you have to traverse through the nodes.

| **Operation** | **ArrayList** | **LinkedList** |
| --- | --- | --- |
| Insertion | Fast at the end - O(1) amortized, slow at the beginning or middle- O(n) | Fast at the beginning or middle - O(1), slow at the end - O(n) |
| Deletion | Fast at the end - O(1) amortized, slow at the beginning or middle- O(n) | Fast - O(1) if position is known |
| Retrieval | Fast - O(1) for random access | Slow - O(n) for random access, as it involves traversing |

**Sets**

A Set is a type of collection that does not allow duplicate elements and represents the concept of a mathematical set.

1. **Internal mechanism**: A Set is internally backed by a HashMap. Depending on the implementation type, it is supported by either a HashMap, LinkedHashMap, or a TreeMap. I have written a detailed article about how HashMap works internally [here](https://www.freecodecamp.org/news/how-java-hashmaps-work-internal-mechanics-explained). Be sure to check it out.
2. **Duplicates**: Since a Set represents the concept of a mathematical set, duplicate elements are not allowed. This ensures that all elements are unique, maintaining the integrity of the collection.
3. **Null**: A maximum of one null value is allowed in a Set because duplicates are not permitted. But this does not apply to the TreeSet implementation, where null values are not allowed at all.
4. **Ordering**: Ordering of elements in a Set depends on the type of implementation.
   * HashSet: Order is not guaranteed, and elements can be placed in any position.
   * LinkedHashSet: This implementation maintains the insertion order, so you can retrieve the elements in the same order they were inserted.
   * TreeSet: Elements are inserted based on their natural order. Alternatively, you can control the insertion order by specifying a custom comparator.
5. **Synchronization**: A Set is not synchronized, meaning you might encounter concurrency issues, like race conditions, which can affect data integrity if two or more threads try to access a Set object simultaneously
6. **Key methods**: Here are some key methods of a Set interface: add(E element), remove(Object o), contains(Object o), and size(). Let's look at how to use these methods with an example program.
7. import java.util.HashSet;
8. import java.util.Set;
9. public class SetExample {
10. public static void main(String[] args) {
11. // Create a set
12. Set<String> set = new HashSet<>();
13. // Add elements to the set
14. set.add("Apple");
15. set.add("Banana");
16. set.add("Cherry");
17. // Remove an element from the set
18. set.remove("Banana");
19. // Check if the set contains an element
20. boolean containsApple = set.contains("Apple");
21. System.out.println("Contains Apple: " + containsApple);
22. // Get the size of the set
23. int size = set.size();
24. System.out.println("Size of the set: " + size);
25. }
26. }
27. **Common implementations**: HashSet, LinkedHashSet, TreeSet
28. **Performance**: Set implementations offer fast performance for basic operations, except for a TreeSet, where the performance can be relatively slower because the internal data structure involves sorting the elements during these operations.

| **Operation** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| --- | --- | --- | --- |
| Insertion | Fast - O(1) | Fast - O(1) | Slower - O(log n) |
| Deletion | Fast - O(1) | Fast - O(1) | Slower - O(log n) |
| Retrieval | Fast - O(1) | Fast - O(1) | Slower - O(log n) |

**Queues**

A Queue is a linear collection of elements used to hold multiple items before processing, usually following the FIFO (first-in-first-out) order. This means elements are added at one end and removed from the other, so the first element added to the queue is the first one removed.

1. **Internal mechanism**: The internal workings of a Queue can differ based on its specific implementation.
   * LinkedList – uses a doubly-linked list to store elements, which means you can traverse both forward and backward, allowing for flexible operations.
   * PriorityQueue – is internally backed by a binary heap, which is very efficient for retrieval operations.
   * ArrayDeque – is implemented using an array that expands or shrinks as elements are added or removed. Here, elements can be added or removed from both ends of the queue.
2. **Duplicates**: In a Queue, duplicate elements are permitted, allowing multiple instances of the same value to be inserted
3. **Null**: You cannot insert a null value into a Queue because, by design, some methods of a Queue return null to indicate that it is empty. To avoid confusion, null values are not allowed.
4. **Ordering**: Elements are inserted based on their natural order. Alternatively, you can control the insertion order by specifying a custom comparator.
5. **Synchronization**: A Queue is not synchronized by default. But, you can use a ConcurrentLinkedQueue or a BlockingQueue implementation for achieving thread safety.
6. **Key methods**: Here are some key methods of a Queue interface: add(E element), offer(E element), poll(), and peek(). Let's look at how to use these methods with an example program.
7. import java.util.LinkedList;
8. import java.util.Queue;
9. public class QueueExample {
10. public static void main(String[] args) {
11. // Create a queue using LinkedList
12. Queue<String> queue = new LinkedList<>();
13. // Use add method to insert elements, throws exception if insertion fails
14. queue.add("Element1");
15. queue.add("Element2");
16. queue.add("Element3");
17. // Use offer method to insert elements, returns false if insertion fails
18. queue.offer("Element4");
19. // Display queue
20. System.out.println("Queue: " + queue);
21. // Peek at the first element (does not remove it)
22. String firstElement = queue.peek();
23. System.out.println("Peek: " + firstElement); // outputs "Element1"
24. // Poll the first element (retrieves and removes it)
25. String polledElement = queue.poll();
26. System.out.println("Poll: " + polledElement); // outputs "Element1"
27. // Display queue after poll
28. System.out.println("Queue after poll: " + queue);
29. }
30. }
31. **Common implementations**: LinkedList, PriorityQueue, ArrayDeque
32. **Performance**: Implementations like LinkedList and ArrayDeque are usually quick for adding and removing items. The PriorityQueue is a bit slower because it inserts items based on the set priority order.

| **Operation** | **LinkedList** | **PriorityQueue** | **ArrayDeque** |
| --- | --- | --- | --- |
| Insertion | Fast at the beginning or middle - O(1), slow at the end - O(n) | Slower - O(log n) | Fast - O(1), Slow - O(n), if it involves resizing of the internal array |
| Deletion | Fast - O(1) if position is known | Slower - O(log n) | Fast - O(1), Slow - O(n), if it involves resizing of the internal array |
| Retrieval | Slow - O(n) for random access, as it involves traversing | Fast - O(1) | Fast - O(1) |

**Maps**

A Map represents a collection of key-value pairs, with each key mapping to a single value. Although Map is part of the Java Collection framework, it does not extend the java.util.Collection interface.

1. **Internal mechanism**: A Map works internally using a HashTable based on the concept of hashing. I have written a detailed [article](https://www.freecodecamp.org/news/how-java-hashmaps-work-internal-mechanics-explained) on this topic, so give it a read for a deeper understanding.
2. **Duplicates**: A Map stores data as key-value pairs. Here, each key is unique, so duplicate keys are not allowed. But duplicate values are permitted.
3. **Null**: Since duplicate keys are not allowed, a Map can have only one null key. As duplicate values are permitted, it can have multiple null values. In the TreeMap implementation, keys cannot be null because it sorts the elements based on the keys. However, null values are allowed.
4. **Ordering**: Insertion order of a Map varies on the implementation:
   * HashMap – the insertion order is not guaranteed as they are determined based on the concept of hashing.
   * LinkedHashMap – the insertion order is preserved and you can retrieve the elements back in the same order that they were added into the collection.
   * TreeMap – Elements are inserted based on their natural order. Alternatively, you can control the insertion order by specifying a custom comparator.
5. **Synchronization**: A Map is not synchronized by default. But you can use Collections.synchronizedMap() or ConcurrentHashMap implementations for achieving thread safety.
6. **Key methods**: Here are some key methods of a Map interface: put(K key, V value), get(Object key), remove(Object key), containsKey(Object key), and keySet(). Let's look at how to use these methods with an example program.
7. import java.util.HashMap;
8. import java.util.Map;
9. import java.util.Set;
10. public class MapMethodsExample {
11. public static void main(String[] args) {
12. // Create a new HashMap
13. Map<String, Integer> map = new HashMap<>();
14. // put(K key, V value) - Inserts key-value pairs into the map
15. map.put("Apple", 1);
16. map.put("Banana", 2);
17. map.put("Orange", 3);
18. // get(Object key) - Returns the value associated with the key
19. Integer value = map.get("Banana");
20. System.out.println("Value for 'Banana': " + value);
21. // remove(Object key) - Removes the key-value pair for the specified key
22. map.remove("Orange");
23. // containsKey(Object key) - Checks if the map contains the specified key
24. boolean hasApple = map.containsKey("Apple");
25. System.out.println("Contains 'Apple': " + hasApple);
26. // keySet() - Returns a set view of the keys contained in the map
27. Set<String> keys = map.keySet();
28. System.out.println("Keys in map: " + keys);
29. }
30. }
31. **Common implementations**: HashMap, LinkedHashMap, TreeMap, Hashtable, ConcurrentHashMap
32. **Performance**: HashMap implementation is widely used mainly due to its efficient performance characteristics depicted in the below table.

| **Operation** | **HashMap** | **LinkedHashMap** | **TreeMap** |
| --- | --- | --- | --- |
| Insertion | Fast - O(1) | Fast - O(1) | Slower - O(log n) |
| Deletion | Fast - O(1) | Fast - O(1) | Slower - O(log n) |
| Retrieval | Fast - O(1) | Fast - O(1) | Slower - O(log n) |

**Collections Utility Class**

As highlighted at the beginning of this article, the Collections utility class has several useful static methods that let you perform commonly used operations on the elements of a collection. These methods help you reduce the boilerplate code in your application and lets you focus on the business logic.

Here are some key features and methods, along with what they do, listed briefly:

1. **Sorting:** Collections.sort(List<T>) – this method is used to sort the elements of a list in ascending order.
2. **Searching:** Collections.binarySearch(List<T>, key) – this method is used to search for a specific element in a sorted list and return its index.
3. **Reverse order:** Collections.reverse(List<T>) – this method is used to reverse the order of elements in a list.
4. **Min/Max Operations:** Collections.min(Collection<T>) and Collections.max(Collection<T>) – these methods are used to find the minimum and maximum elements in a collection, respectively.
5. **Synchronization:** Collections.synchronizedList(List<T>) – this method is used to make a list thread-safe by synchronizing it.
6. **Unmodifiable Collections:** Collections.unmodifiableList(List<T>) – this method is used to create a read-only view of a list, preventing modifications.

Here's a sample Java program that demonstrates various functionalities of the Collections utility class:

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class CollectionsExample {

public static void main(String[] args) {

List<Integer> numbers = new ArrayList<>();

numbers.add(5);

numbers.add(3);

numbers.add(8);

numbers.add(1);

// Sorting

Collections.sort(numbers);

System.out.println("Sorted List: " + numbers);

// Searching

int index = Collections.binarySearch(numbers, 3);

System.out.println("Index of 3: " + index);

// Reverse Order

Collections.reverse(numbers);

System.out.println("Reversed List: " + numbers);

// Min/Max Operations

int min = Collections.min(numbers);

int max = Collections.max(numbers);

System.out.println("Min: " + min + ", Max: " + max);

// Synchronization

List<Integer> synchronizedList = Collections.synchronizedList(numbers);

System.out.println("Synchronized List: " + synchronizedList);

// Unmodifiable Collections

List<Integer> unmodifiableList = Collections.unmodifiableList(numbers);

System.out.println("Unmodifiable List: " + unmodifiableList);

}

}

This program demonstrates sorting, searching, reversing, finding minimum and maximum values, synchronizing, and creating an unmodifiable list using the Collections utility class.