www.kiran-khanal.com.np

BEGINNER'S GUIDE TO JAVA COLLECTIONS

Explore Java Collections — the foundation of flexible, efficient data handling in modern Java applications.

Swipe for more

Java Collection is a popular framework to store and manipulate the group of objects in Java. The Java Collection Framework (JCF) provides a set of interfaces and classes to implement various types of data structures and algorithms. It gives programmers access to prepackaged data structures and algorithms for manipulating them.

Collections are the one-stop solution for all sorts of data manipulation jobs, such as storing data, searching, sorting, insertion, updating, and deletion.

In simpler terms, the Java Collections Framework helps you manage a group of objects and perform operations on them efficiently. It is a predefined architecture capable of storing a group of elements and making them behave as a single entity, such as an object.

Before JCF, developers had to rely on custom data structures or limited APIs like Vector and Hashtable. JCF brought consistency, flexibility, and performance to the table.

🔗 Core Interfaces of Java Collections

🧱 LIST : An Ordered Collection

The List interface in Java represents an ordered collection that allows duplicate elements. It is a part of **java.util** package. The elements are stored in the order that they were added; hence, it maintains the **insertion order.**

List also provides indexed access, allowing developers to retrieve or modify elements by their **position** in the list.

Some of the most commonly used implementations of the List interface include **ArrayList, LinkedList,** and **Vector.** Each of these interfaces has its own performance characteristics and use cases.

```java
import java.util.*
public class ListExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        System.out.println(fruits);
    }
}
```

💡 Common Implementation of List

📦 ArrayList

ArrayList is a dynamic array-based implementation. It is fast for get and set operations but slower for insertion and deletion in the middle of the list.

🔗 LinkedList

LinkedList is an implementation of a doubly linked list. It excels in insertions and deletions but may be slower for access due to its non-contiguous memory structure, meaning that the elements are not stored next to each other. Instead, each element, called a node, contains actual data and a reference (pointer) to both the previous and the next node on the list.

🛡️ Vector

Vector is a class in Java that implements the List interface and works like an ArrayList, but with one key difference—it is synchronized. This means it is thread-safe and can be safely used in multi-threaded environments. However, because of this synchronization, Vector is usually slower than ArrayList in single-threaded situations.

🥞 Stack

Stack is a class in Java that represents a Last-In, First-Out (LIFO) data structure. It extends the Vector class and provides methods like push() to add elements, pop() to remove the top element, and peek() to look at the top element without removing it. Since it inherits from Vector, it's also synchronized by default.

Feature	ArrayList	LinkedList	Vector	Stack
Backed By	Dynamic Array	Doubly Linked List	Dynamic Array	Dynamic Array (via Vector)
Performance	Fast for access	Fast of insert/delete	Slower due to sync	LIFO operations
Order Maintained	Yes	Yes	Yes	Yes
Thread-Safe	No	No	Yes	Yes
Index Access	Fast	Slow	Fast	Not typically used

Core Methods of the List Interface 🛠️

➕ add(element)

The add() method adds an element to the list.

```
List<String> cities = new ArrayList<>();
cities.add("New York");
```

🔍 get(index)

The get() method retrieves the element at the specified index.

```
List<String> cities = new ArrayList<>();
cities.add("New York");
System.out.println("City at index 1: " + cities.get(0));
```

✏️ set(index, element)

The set() method replaces the element at the specified index.

```
List<String> cities = new ArrayList<>();
cities.add("New York");
cities.set(0, "Paris");
```

❌ remove(index or element)

The remove() method removes an element by index or by value.

```
List<String> cities = new ArrayList<>();
cities.add("Tokyo");
cities.add("Hongkong")
cities.remove("Tokyo");
cities.remove(1);
```

✅ contains(element)

The contains() method checks if the list contains the given element.

```
List<String> cities = new ArrayList<>();
cities.add("Tokyo");
cities.add("Hongkong");
cities.contains("Tokyo");
```

🔢 indexOf(element)

The indexOf() method returns the index of the first occurrence.

```
List<String> cities = new ArrayList<>();
cities.add("Tokyo");
System.out.println("Index of 'Tokyo': " + cities.indexOf("Tokyo"));
```

🧹 clear()

The clear() method removes all the elements in the list.

```
List<String> cities = new ArrayList<>();
cities.add("Tokyo");
cities.add("Kathmandu");
cities.clear();
```

📏 size()

The size() method returns the number of elements in the list.

```
List<String> cities = new ArrayList<>();
cities.add("Tokyo");
cities.add("Kathmandu");
System.out.println("Size:", cities.size());
```

🔗 Core Interfaces of Java Collections

🔁 SET : Unordered Collection

The Set interface in Java represents a collection of unique elements. Unlike a List, a Set does not allow duplicate values, making it ideal when you want to ensure that every item is distinct — like storing unique IDs, tags, or usernames.

Sets do not maintain any specific order by default, and they also don’t allow accessing elements by index. This is because Sets are designed to focus on the presence of elements, not their position.

Some of the most commonly used implementations of the Set interface include **HashSet, LinkedHashSet,** and **Treeset.**

```java
public class HashSetExample {
    public static void main(String[] args) {
        Set<String> fruits = new HashSet<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Orange");
        fruits.add("Apple"); // duplicate, won't be added
        System.out.println("Fruits Set: " + fruits);
    }
}
```

💡 Common Implementation of Set

⚡ HashSet

HashSet is the most commonly used Set implementation. It doesn't guarantee any specific order of elements — the items may appear in a different order each time you run your code. However, it offers constant-time performance for basic operations like add, remove, and contains.

📋 LinkedHashSet

LinkedHashSet extends HashSet and adds a twist — it maintains the insertion order of elements. This means when you iterate over the set, the elements will appear in the same order in which they were added. It's slightly slower than HashSet, but useful when you need both uniqueness and order.

🌲 TreeSet

TreeSet implements the NavigableSet interface and stores elements in a sorted (natural) order — alphabetically for strings, numerically for numbers. It uses a Red-Black tree under the hood, making operations a bit slower than HashSet, but perfect when you want the elements to stay sorted automatically.

Core Methods of the Set Interface 🛠️

➕ add(element)

The add() method adds an element to the set.

```
Set<String> fruits = new HashSet<>();
fruits.add("Apple");
```

❌ remove(Object element)

The remove() method removes the specified item from the set, if present.

```
Set<String> name = new HashSet<>();
name.add("Kiran");
name.remove("Kiran")
```

🔍 contains(Object element)

The contains() method checks if a specific element exists in the set.

```
Set<String> name = new HashSet<>();
name.add("Kiran");
set.contains("Kiran")
```

🔁 clear()

Removes all elements from the set. (same as List)

📏 size()

Returns the number of elements in the set.

📦 isEmpty()

Checks whether the set is empty.

🔁 forEach() / enhanced for loop

Used to iterate through the elements in the set.

```
Set<String> name = new HashSet<>();
name.add("John");
name.add("Jack");
name.add("Jina");
name.add("Jane");

for(String name : set) {
    System.out.println(name);
}
```

🗺️ Map Interface in Java

The Map interface in Java represents a collection of **key-value** pairs, where each key is unique, and each key maps to exactly one value. Unlike List or Set, a Map is not a subtype of the **Collection** interface, but it's an equally essential part of the **Java Collections Framework.**

In a Map, each key must be unique. However, values can be duplicated — multiple keys can point to the same value. If you try to insert a new value using an existing key, the old value gets replaced with the new one, keeping the key unique in the map.

Some of the most commonly used implementations of the Map interface include **HashMap, LinkedHashMap,** and **TreeMap.**

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Nepal", "Kathmandu");
capitals.put("India", "New Delhi");
capitals.put("Nepal", "Pokhara"); // Replaces "Kathmandu"
```

💡 Common Implementation of Map

⚡ HashMap

HashMap is the most commonly used implementation of the Map interface. It allows storing key-value pairs with no guaranteed order of keys — the elements may appear in a different sequence each time you access them. HashMap offers fast performance for basic operations like insertion, deletion, and lookup.

📋 LinkedHashMap

LinkedHashMap is a subclass of HashMap that maintains the insertion order of entries — meaning keys are returned in the order they were added. This makes it useful when the order of elements matters. However, due to the extra overhead of maintaining this order, it is slightly slower than a regular HashMap in terms of performance.

🌲 TreeMap

TreeMap is a sorted map implementation that stores its keys in natural order (like alphabetical for strings or ascending for numbers), or according to a custom comparator. It’s ideal when you need the data to be automatically sorted by key. The benefit is having a consistently ordered map without additional sorting logic.

Core Methods of the Map Interface 🛠️

🔑 put(key, value)

Adds a key-value pair. Replaces the value if the key already exists.

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Nepal", "Kathmandu");
capitals.put("India", "New Delhi");
capitals.put("Nepal", "Pokhara"); // Replaces "Kathmandu"
```

🔍 get(key)

Returns the value for the given key (or null if not found).

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Nepal", "Kathmandu");
capitals.put("India", "New Delhi");
 System.out.println("Capital of Nepal: " + capitals.get("Nepal")); //Kathmandu
```

❌ remove(key)

Removes the key and its associated value from the map.

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Nepal", "Kathmandu");
capitals.remove("Nepal");
```

✅ containsKey(key)

Checks if a key exists in the map.

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Nepal", "Kathmandu");
capitals.containsKey("Nepal");
```

🔎 containsValue(value)

Checks if a value exists in the map.

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Nepal", "Kathmandu");
capitals.containsValue("Kathmandu");
```

🗝️ keySet()

Returns a Set of all the keys.

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Nepal", "Kathmandu");
System.out.println("All keys: " + capitals.keySet()); // [Nepal]
```

📦 values()

Returns a collection of all the values.

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Nepal", "Kathmandu");
System.out.println("All values: " + capitals.values()); //[Kathmandu]
```

🧩 entrySet()

Returns a Set of all key-value pairs as Map.Entry.

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Nepal", "Kathmandu");
capitals.put("India","New Delhi");
for (Map.Entry<String, String> entry : capitals.entrySet()) {
  System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

📏 size()

Returns the number of key-value pairs.

📪 isEmpty()

Checks if the map is empty.

🧹 clear()

Removes all entries from the map.

🚪 putIfAbsent(key, value)

Adds only if the key isn’t already in the map.

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Nepal", "Kathmandu");
capitals.put("India","New Delhi");
capitals.putIfAbsent("France", "Paris");
```

🔁 replace(key, value)

Replaces the value only if the key already exists.

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Nepal", "Jhapa");
capitals.put("India","New Delhi");
capitals.replace("Nepal", "Jhapa", "Kathmandu");
```

🔄 forEach((k,v) -> {...})

Iterates over each key-value pair using a lambda/action.

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Nepal", "Kathmandu");
capitals.put("India","New Delhi");
 capitals.forEach((k, v) -> System.out.println(k + " -> " + v));
```

Kiran Khanal

Software Engineer

If you find this helpful, please like and share it with your friends

www.linkedin.com/kirankhanalleo