# Java Collection Framework

## 1. Introduction

- The **Java Collection Framework (JCF)** provides a set of classes and interfaces to store and manipulate groups of objects.
- It is present in **`java.util` package**.
- It provides **ready-to-use data structures** like lists, sets, queues, maps, etc.
- Helps in writing **reusable, efficient, and maintainable code**.

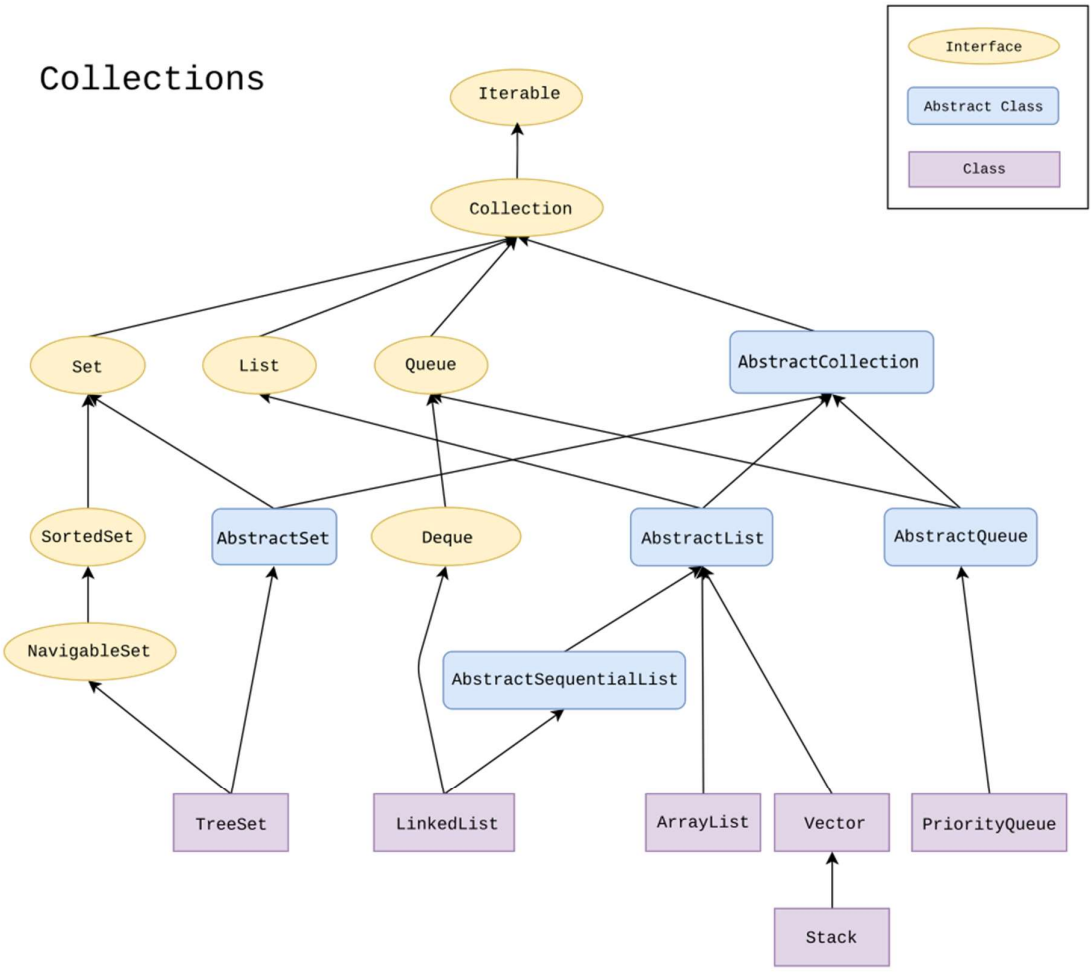
## 2. Key Features

- **Interfaces**: Provide abstract data types (List, Set, Queue, Map).
- **Implementations**: Concrete classes (ArrayList, HashSet, PriorityQueue, HashMap).
- **Algorithms**: Utility methods (sorting, searching) in `Collections` class.
- **Polymorphism**: You can use interface reference and change the implementation easily.

## 3. Core Interfaces of Collection Framework

- **Collection (root interface)**
  - Base interface for all collections (except Map).
  - Methods: `add()`, `remove()`, `size()`, `iterator()`.
- **List (extends Collection)**
  - Ordered, allows duplicate elements.
  - Implementations: `ArrayList`, `LinkedList`, `Vector`, `Stack`.
- **Set (extends Collection)**
  - No duplicates allowed.
  - Implementations: `HashSet`, `LinkedHashSet`, `TreeSet`.
- **Queue (extends Collection)**
  - Follows FIFO order (mostly).
  - Implementations: `PriorityQueue`, `ArrayDeque`, `LinkedList`.
- **Map (separate interface, not part of Collection)**
  - Stores key-value pairs, keys are unique.
  - Implementations: `HashMap`, `LinkedHashMap`, `TreeMap`, `Hashtable`.

## 4. Collection Framework Hierarchy




## 5. Important Classes

- **ArrayList**: Dynamic array, fast random access.
- **LinkedList**: Doubly linked list, efficient insert/delete.
- **Vector**: Legacy class, synchronized.
- **Stack**: LIFO stack (extends Vector).
- **HashSet**: No duplicates, unordered.
- **LinkedHashSet**: Maintains insertion order.
- **TreeSet**: Sorted set (ascending order by default).
- **HashMap**: Key-value pairs, no order.
- **LinkedHashMap**: Maintains insertion order.
- **TreeMap**: Sorted key-value pairs.
- **PriorityQueue**: Elements processed based on priority.
- **ArrayDeque**: Double-ended queue.

## 6. Utility Class

`Collections` class: Provides static methods for algorithms.

`sort(list)` – Sorts the list.
`reverse(list)` – Reverses the list.
`shuffle(list)` – Randomly shuffles elements.
`max(list),min(list)` – Finds maximum/minimum element.

# ArrayList in Java

- **ArrayList** is a **resizable (dynamic) array** implementation of the `List` interface.
- Belongs to **`java.util` package**.
- Unlike normal arrays, ArrayList can **grow or shrink dynamically**.
- Allows **duplicate elements** and maintains **insertion order**.
- Provides **random access** (fast element retrieval by index).

### Declaration:

```
ArrayList<Type> list = new ArrayList<>();
```

### 2. Constructors

- `ArrayList()` → Creates an empty ArrayList.
- `ArrayList(int capacity)` → Creates ArrayList with specified initial capacity.
- `ArrayList(Collection c)` → Creates ArrayList containing elements of another collection.

### 3. Commonly Used Methods

## Adding Elements

- `add(E e)` → Appends element at the end.
- `add(int index, E e)` → Inserts element at given index.

```
list.add("Apple");
list.add(1, "Mango");
```

## Accessing Elements

- `get(int index)` → Returns element at given index.

```
String fruit = list.get(0);
```

## Modifying Elements

- `set(int index, E e)` → Replaces element at index with new value.

```
list.set(1, "Banana");
```

## Removing Elements

- `remove(int index)` → Removes element at index.

- `remove(Object o)` → Removes first occurrence of given element.
- `clear()` → Removes all elements.

```
list.remove(0);
list.remove("Apple");
list.clear();
```

## Searching Elements

- `contains(Object o)` → Returns `true` if element exists.
- `indexOf(Object o)` → Returns index of first occurrence (or -1 if not found).
- `lastIndexOf(Object o)` → Returns index of last occurrence.

```
boolean check = list.contains("Mango");
int pos = list.indexOf("Banana");
```

## Size & Capacity

- `size()` → Returns number of elements.
- `isEmpty()` → Returns `true` if empty.
- `ensureCapacity(int minCapacity)` → Increases capacity if needed.
- `trimToSize()` → Trims capacity to current size.

```
int n = list.size();
boolean empty = list.isEmpty();
```

## Iteration

- `iterator()` → Returns iterator.
- `listIterator()` → Returns list iterator (can traverse both directions).
- `forEach()` → Traverses using lambda.

```
for(String fruit : list) {
    System.out.println(fruit);
}
```

## Conversion

- `toArray()` → Converts to array.

```
Object[] arr = list.toArray();
```

Example Code:

```
import java.util.*;

public class ArrayListDemo {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();

        // Adding elements
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Orange");
```

```
        // Access
        System.out.println(fruits.get(1)); // Mango

        // Modify
        fruits.set(1, "Banana");

        // Remove
        fruits.remove("Apple");

        // Search
        System.out.println(fruits.contains("Orange")); // true

        // Iteration
        for (String f : fruits) {
            System.out.println(f);
        }
    }
}
```

### Summary:

- **ArrayList** = resizable array, maintains order, allows duplicates.
- Key methods = `add()`, `get()`, `set()`, `remove()`, `contains()`, `size()`, `iterator()`.
- Good for **random access**, but insertion/deletion in middle is slower than `LinkedList`.

# LinkedList in Java

- **LinkedList** is a **doubly linked list** implementation of `List` and `Deque` interfaces.
- Belongs to **`java.util` package**.
- Maintains **insertion order**.
- Allows **duplicate elements**.
- Each element (called **Node**) contains:
    - **data**
    - **address of previous node**
    - **address of next node**

Unlike `ArrayList`, insertion and deletion are **faster**, but random access is **slower**.

### Declaration:

```
LinkedList<Type> list = new LinkedList<>();
```

## 2. Constructors

`LinkedList()` → Creates an empty list.
`LinkedList(Collection c)` → Creates list with elements of another collection.

## Commonly Used Methods

### Adding Elements

- `add(E e)` → Adds element at the end.

- `add(int index, E e)` → Inserts element at index.
- `addFirst(E e)` → Inserts element at beginning.
- `addLast(E e)` → Inserts element at end.

```
list.add("A");
list.addFirst("Start");
list.addLast("End");
```

## Accessing Elements

- `get(int index)` → Returns element at index.
- `getFirst()` → Returns first element.
- `getLast()` → Returns last element.

```
String s = list.get(1);
String first = list.getFirst();
```

## Modifying Elements

- `set(int index, E e)` → Replaces element at index.

```
list.set(1, "NewValue");
```

## Removing Elements

- `remove(int index)` → Removes element at index.
- `remove(Object o)` → Removes first occurrence.
- `removeFirst()` → Removes first element.
- `removeLast()` → Removes last element.
- `clear()` → Removes all elements.

```
list.remove("A");
list.removeFirst();
list.clear();
```

## Queue/Deque Operations (because LinkedList implements `Deque`)

- `offer(E e)` → Adds element at end.
- `offerFirst(E e)` → Adds at beginning.
- `offerLast(E e)` → Adds at end.
- `poll()` → Retrieves and removes head.
- `pollFirst()` / `pollLast()` → Removes first/last element.
- `peek()` → Returns head without removing.
- `peekFirst()` / `peekLast()` → Returns first/last element without removing.

## Size & Check

- `size()` → Returns number of elements.
- `isEmpty()` → Checks if empty.

## Iteration

- `iterator()` → Forward iteration.
- `descendingIterator()` → Reverse iteration.
- Enhanced for loop (`for-each`).

### Example Code

```
import java.util.*;

public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedList<String> names = new LinkedList<>();

        // Adding elements
        names.add("Neeraj");
        names.add("Rahul");
        names.addFirst("Amit");
        names.addLast("Suman");

        // Access
        System.out.println("First: " + names.getFirst());
        System.out.println("Last: " + names.getLast());

        // Modify
        names.set(1, "Updated");

        // Remove
        names.removeFirst();
        names.remove("Suman");

        // Queue operations
        names.offer("Extra");
        System.out.println("Peek: " + names.peek());

        // Iteration
        for (String n : names) {
            System.out.println(n);
        }
    }
}
```

## 5. Comparison: ArrayList vs LinkedList

| Feature | ArrayList | LinkedList |
|---|---|---|
| **Underlying DS** | Dynamic Array | Doubly Linked List |
| **Access (get/set)** | Fast (O(1)) | Slow (O(n)) |
| **Insert/Delete (middle)** | Slow (O(n)) | Fast (O(1)) if iterator known |
| **Insert/Delete (end)** | Fast | Fast |
| **Memory Usage** | Less | More (extra node pointers) |

---

Summary:

- **LinkedList** = good for **frequent insertions/deletions**.
- Implements **List + Deque** → works as **list + queue + stack**.
- Key methods: `addFirst()`, `addLast()`, `removeFirst()`, `removeLast()`, `peek()`, `poll()`.

# ArrayDeque in Java

- **ArrayDeque** = **Resizable array-based implementation of Deque** (Double Ended Queue).
- Belongs to **`java.util` package**.
- Faster than `Stack` and `LinkedList` for **stack** and **queue** operations.
- **No capacity restriction** (grows dynamically).
- **Null elements not allowed**.
- Can function as both:
    - **Queue (FIFO)**
    - **Stack (LIFO)**

**Declaration:**

```
ArrayDeque<Type> dq = new ArrayDeque<>();
```

## 2. Constructors

`ArrayDeque()` → Creates an empty deque.
`ArrayDeque(int numElements)` → Creates deque with specified capacity.
`ArrayDeque(Collection c)` → Creates deque containing elements of another collection.

## 3. Commonly Used Methods

### Insertion

- `addFirst(E e)` → Adds element at front.
- `addLast(E e)` → Adds element at rear.
- `offerFirst(E e)` → Adds at front, returns `true/false`.
- `offerLast(E e)` → Adds at rear, returns `true/false`.

```
dq.addFirst("A");
dq.addLast("B");
dq.offerFirst("X");
dq.offerLast("Y");
```

### Removal

- `removeFirst()` → Removes and returns first element (exception if empty).
- `removeLast()` → Removes and returns last element.
- `pollFirst()` → Removes and returns first element (null if empty).
- `pollLast()` → Removes and returns last element (null if empty).
- `pop()` → Removes first element (stack behavior).

```
dq.removeFirst();
dq.pollLast();
String val = dq.pop();
```

## Access (Peek/Check)

- `getFirst()` → Returns first element (exception if empty).
- `getLast()` → Returns last element.
- `peekFirst()` → Returns first element (null if empty).
- `peekLast()` → Returns last element (null if empty).

```
System.out.println(dq.getFirst());
System.out.println(dq.peekLast());
```

## Stack Operations (LIFO)

- `push(E e)` → Adds element at front.
- `pop()` → Removes and returns element from front.

```
dq.push("Item1");  // same as addFirst()
dq.pop();          // same as removeFirst()
```

## Other

- `size()` → Returns number of elements.
- `isEmpty()` → Checks if deque is empty.
- `iterator()` → Forward iteration.
- `descendingIterator()` → Reverse iteration.
- `clear()` → Removes all elements.

### Example Code:

```
import java.util.*;

public class ArrayDequeDemo {
    public static void main(String[] args) {
        ArrayDeque<String> dq = new ArrayDeque<>();

        // Insertion
        dq.add("Apple");
        dq.addFirst("Start");
        dq.addLast("End");

        // Access
        System.out.println("First: " + dq.peekFirst());
        System.out.println("Last: " + dq.peekLast());

        // Removal
        dq.removeFirst();
        dq.pollLast();

        // Stack operations
        dq.push("Banana");
        System.out.println("Popped: " + dq.pop());
```

```
        // Iteration
        for (String s : dq) {
            System.out.println(s);
        }
    }
}
```

### 5. Advantages of ArrayDeque

- **Faster** than `LinkedList` (cache-friendly, less overhead).
- **Better** than `Stack` (modern replacement).
- Can be used as **stack, queue, deque**.
- Dynamic resizing (no fixed capacity).

### Summary:

- **ArrayDeque** = best general-purpose implementation of **Deque**.
- Works as **Stack + Queue**.
- Key methods: `addFirst()`, `addLast()`, `removeFirst()`, `removeLast()`, `peekFirst()`, `peekLast()`, `push()`, `pop()`.
- More efficient than `Stack` and `LinkedList`.

# Hashing in Java

- Hashing is a technique used to **map data to a fixed-size value (hash code)** using a hash function.
- This hash code helps in **fast searching, insertion, and deletion** of elements.

### Key Points

1. **Hash Function** → Converts the input (like a string or number) into an integer (hash code).
2. **Hash Table** → Stores key-value pairs using the hash code as the index.
3. **Collision** → When two different inputs produce the same hash code.
    - Handled by **chaining** (linked list at same index) or **open addressing**.
4. **Advantages of Hashing**
    - Very **fast lookups** (on average O(1)).
    - Useful in sets, maps, and caching.

# HashSet in Java

- `HashSet` is a **collection class** in Java that implements the **Set interface**.
- It uses a **hash table (backed by HashMap)** for storage.
- It **does not allow duplicate elements**.

### Features of HashSet

- **No duplicates** → Each element is unique.
- **Unordered** → Does not maintain insertion order.
- **Null allowed** → Only **one null element** is allowed.

- **Backed by HashMap** → Internally uses hashing for fast operations.

### Exmaple

```
import java.util.*;

class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();

        // Adding elements
        set.add("Apple");
        set.add("Banana");
        set.add("Mango");
        set.add("Banana"); // Duplicate, will not be added

        // Display elements
        System.out.println(set);

        // Checking
        System.out.println("Contains Mango? " + set.contains("Mango"));

        // Removing
        set.remove("Apple");
        System.out.println("After removing Apple: " + set);
    }
}
```

## Important Methods of HashSet

- `add(E e)` → Adds element.
- `remove(Object o)` → Removes element.
- `contains(Object o)` → Checks if element exists.
- `size()` → Returns number of elements.
- `clear()` → Removes all elements.
- `isEmpty()` → Checks if set is empty.
- `iterator()` → Returns an iterator for traversal.

## Difference between HashSet and List

| Aspect | HashSet | List (ArrayList/LinkedList) |
|---|---|---|
| **Order** | No order maintained | Maintains insertion order |
| **Duplicates** | Not allowed | Allowed |
| **Nulls** | At most 1 null allowed | Multiple nulls allowed |
| **Performance** | Fast lookup (O(1) average) | Slower lookup (O(n)) |