### \*\* Introduction to the Collections Framework\*\*



#### \*\*1.1 Introduction to Collections Framework\*\*

- \*\*What is a Collection?\*\*

- Definition: A collection is a group of objects, known as elements.

- Importance: Collections in Java provide a way to manage and organize groups of objects efficiently.

- \*\*Why Use the Collections Framework?\*\*

- Provides ready-made data structures and algorithms.

- Eliminates the need for developers to implement data structures manually.

- Enhances performance and simplifies the development process.

- \*\*Hierarchy of Collections Framework\*\*

- Overview of the core interfaces: `Collection`, `List`, `Set`, `Queue`, `Map` (brief mention of `Map` for context).

- Introduction to `Iterator`: An interface for iterating over collections.

#### \*\*1.2 Setting Up the Java Development Environment\*\*

- \*\*Installing Java Development Kit (JDK)\*\*

- JDK from Oracle’s official website.

- Set up the Java environment variable (`JAVA\_HOME`).

- \*\*Installing an Integrated Development Environment (IDE)\*\*

- Recommend popular IDEs like IntelliJ IDEA, Eclipse, or NetBeans.

\*\*Example Code:\*\*

```java

import java.util.\*;

public class CollectionExample {

public static void main(String[] args) {

Collection<String> collection = new ArrayList<>();

collection.add("Java");

collection.add("Python");

collection.add("C++");

System.out.println("Collection: " + collection);

}

}

```

\*\*Activities:\*\*

- Hands-on installation of JDK and IDE.

- Run the example code to ensure the environment is set up correctly.

### \*\* Collection Interfaces - List and Queue\*\*

#### \*\*2.1 Understanding the `List` Interface\*\*

- \*\*Overview of `List`:\*\*

- `List` is an ordered collection that allows duplicate elements.

- Common implementations: `ArrayList`, `LinkedList`.

- \*\*ArrayList\*\*

- Resizable array implementation of the `List` interface.

- Advantages: Fast random access, dynamic resizing.

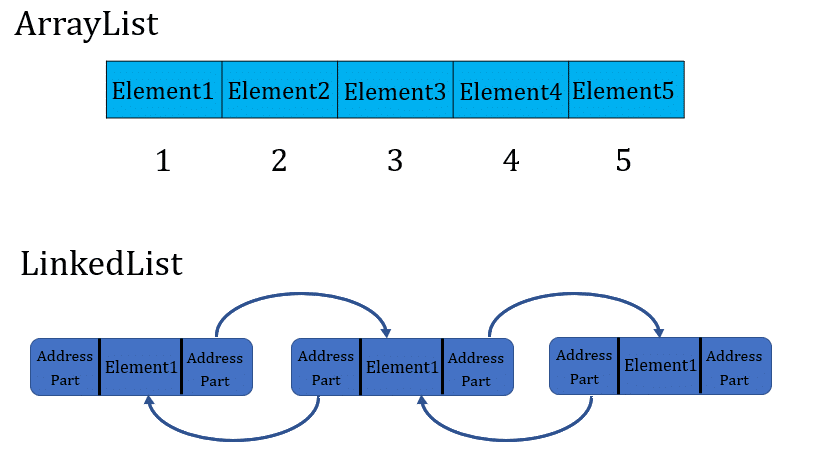
- Disadvantages: Slower at inserting and deleting elements in the middle of the list.

- \*\*LinkedList\*\*

- Doubly-linked list implementation of the `List` interface.

- Advantages: Efficient insertions and deletions.

- Disadvantages: Slower access time compared to `ArrayList`.



\*\*Example Code for `List`:\*\*

```java

// Java program to Illustrate Working of an ArrayList

// Importing required classes

import java.io.\*;

import java.util.\*;

// Main class

class ArraylistExample {

// Main driver method

public static void main(String[] args)

{

// Creating an ArrayList of Integer type

ArrayList<Integer> arrli

= new ArrayList<Integer>();

// Appending the new elements

// at the end of the list

// using add () method via for loops

for (int i = 1; i <= 5; i++)

arrli.add(i);

// Printing the ArrayList

System.out.println(arrli);

// Removing an element at index 3

// from the ArrayList

// using remove() method

arrli.remove(3);

// Printing the ArrayList after

// removing the element

System.out.println(arrli);

}

}

```  
  
LinkedList Example  
```  
// Java program to Demonstrate Working of a LinkedList

// Importing required classes

import java.util.\*;

// Main class

class LinkExample {

// main driver method

public static void main(String args[])

{

// Creating an object of the

// class linked list

LinkedList<String> object

= new LinkedList<String>();

// Adding the elements to the object created

// using add() and addLast() method

// Custom input elements

object.add("A");

object.add("B");

object.addLast("C");

// Print the current LinkedList

System.out.println(object);

// Removing elements from the List object

// using remove() and removeFirst() method

object.remove("B");

// object.removeFirst();

System.out.println("Linked list after "

+ "deletion: " + object);

}

}

```

#### \*\*2.2 Understanding the `Queue` Interface\*\*

- \*\*Overview of `Queue`:\*\*

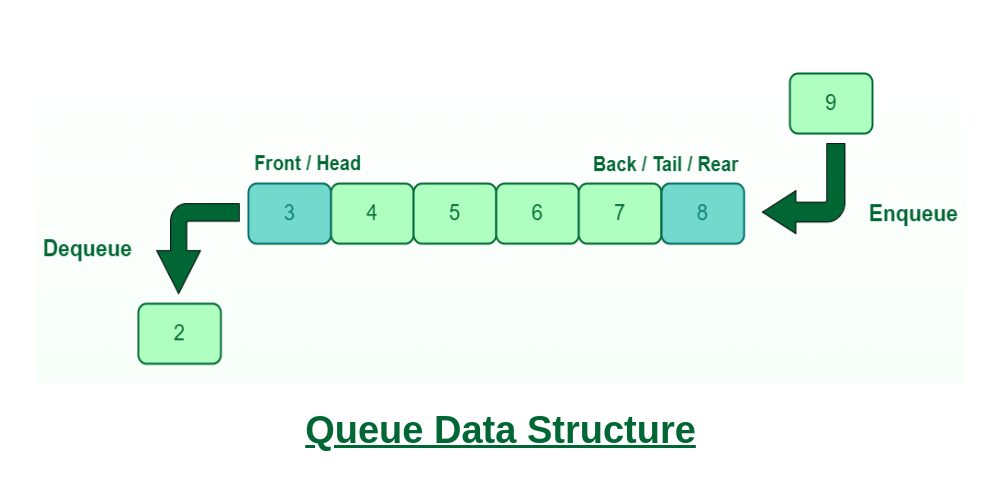
- A `Queue` is a collection used to hold multiple elements prior to processing.

- Follows FIFO (First-In-First-Out) order.

- \*\*Common Implementations:\*\*

- `LinkedList` (can act as both a list and a queue).

- `PriorityQueue` (orders elements based on their natural ordering or by a comparator).



\*\*Example Code for `Queue`:\*\*

```java

import java.util.\*;

public class QueueExample {

public static void main(String[] args) {

Queue<String> queue = new LinkedList<>();

queue.add("First");

queue.add("Second");

queue.add("Third");

System.out.println("Queue: " + queue);

System.out.println("Removed: " + queue.poll()); // Removes the head of the queue

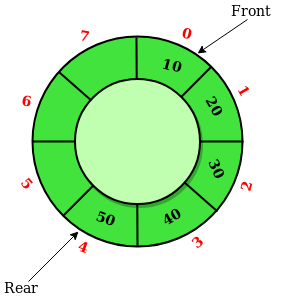
System.out.println("Queue after poll: " + queue);

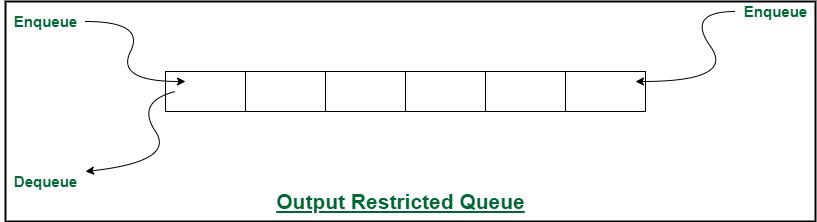
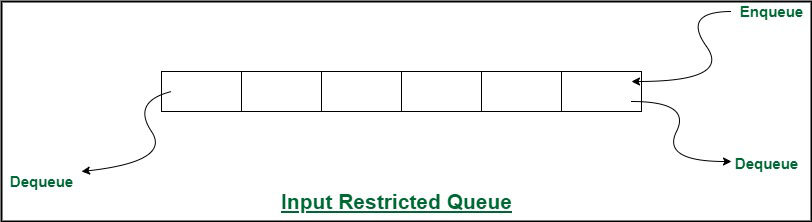
}

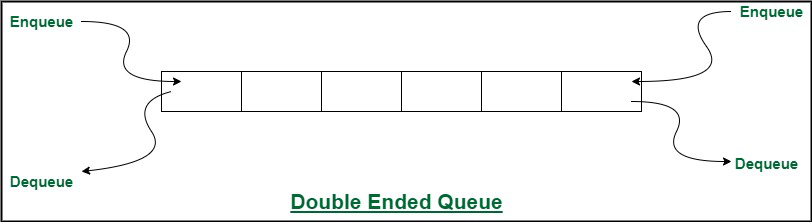
}

```

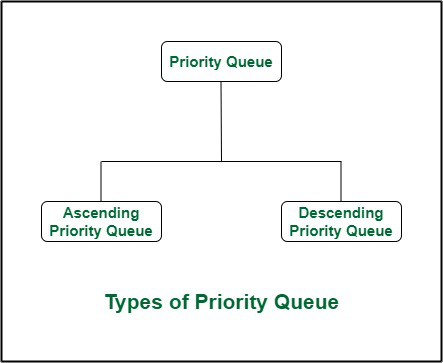
**Operations on Circular Queue:**

* **Front:** Get the front item from the queue.
* **Rear:** Get the last item from the queue.
* **enQueue(value)**This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at the rear position.
  + Check whether the queue is full – [i.e., the rear end is in just before the front end in a circular manner].
  + If it is full then display Queue is full.
    - If the queue is not full then,  insert an element at the end of the queue.
* **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position.
  + Check whether the queue is Empty.
  + If it is empty then display Queue is empty.
    - If the queue is not empty, then get the last element and remove it from the queue.
    - 





|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Time Complexity** |
| **push\_front()** | Inserts the element at the beginning. | **O(1)** |
| **push\_back()** | Adds element at the end. | **O(1)** |
| **pop\_front()** | Removes the first element from the deque. | **O(1)** |
| **pop\_back()** | Removes the last element from the deque. | **O(1)** |
| **front()** | Gets the front element from the deque. | **O(1)** |
| **back()** | Gets the last element from the deque. | **O(1)** |
| **empty()** | Checks whether the deque is empty or not. | **O(1)** |
| **size()** | Determines the number of elements in the deque. | **O(1)** |

[**Priority Queue**](https://www.geeksforgeeks.org/priority-queue-set-1-introduction/)**:** A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. There are two types of Priority Queues. They are:

1. **Ascending Priority Queue:** Element can be inserted arbitrarily but only smallest element can be removed. For example, suppose there is an array having elements 4, 2, 8 in the same order. So, while inserting the elements, the insertion will be in the same sequence but while deleting, the order will be 2, 4, 8.
2. **Descending priority Queue:** Element can be inserted arbitrarily but only the largest element can be removed first from the given Queue. For example, suppose there is an array having elements 4, 2, 8 in the same order. So, while inserting the elements, the insertion will be in the same sequence but while deleting, the order will be 8, 4, 2.

### \*\* Collections Interfaces - Set (Overview) and Collection Classes\*\*

#### \*\*3.1 Overview of the `Set` Interface\*\*

- \*\*Understanding `Set`:\*\*

- A `Set` is a collection that does not allow duplicate elements.

- Implementations: `HashSet`, `LinkedHashSet`, `TreeSet`.

- \*\*Common Set Implementations:\*\*

- `HashSet`: Fast access time, no guaranteed order of elements.

- `LinkedHashSet`: Maintains insertion order.

- `TreeSet`: Stores elements in a sorted order.

\*\*Example Code for `Set`:\*\*

```java

import java.util.\*;

public class SetExample {

public static void main(String[] args) {

Set<String> hashSet = new HashSet<>();

hashSet.add("Red");

hashSet.add("Green");

hashSet.add("Blue");

hashSet.add("Green"); // Duplicate, will not be added

System.out.println("HashSet: " + hashSet);

Set<String> treeSet = new TreeSet<>();

treeSet.add("Zebra");

treeSet.add("Apple");

treeSet.add("Mango");

System.out.println("TreeSet (sorted): " + treeSet);

}

}

```

#### \*\*3.2 Comparison between `ArrayList` and `LinkedList`\*\*

- \*\*When to Use `ArrayList`:\*\*

- `ArrayList` is ideal when you need fast, random access to elements via index.

- It provides constant-time access for retrieving elements, making it efficient for read-heavy operations.

- Best suited for scenarios where the list is more frequently accessed than modified (e.g., search operations).

- However, adding or removing elements (especially in the middle) can be slow due to shifting elements.

- \*\*When to Use `LinkedList`:\*\*

- `LinkedList` excels in scenarios requiring frequent insertions and deletions, as these operations are faster than in an `ArrayList`.

- It is a good choice for implementing data structures like queues or stacks where elements are frequently added or removed from the ends.

- Each element in a `LinkedList` is a node with pointers to its neighbors, allowing for efficient sequential access.

- However, it has slower access time compared to `ArrayList` for random access due to the need to traverse the list.