Week 6. Collection Framework Java Generics, List & Queue

# Custom Generic Data Structure with List and Queue Integration

**Objective:** Create a data structure that combines the features of both List and Queue and focus on balancing the operations and ensuring efficient performance for both random access and queue operations.

# Details:

* 1. **Class Design:** Implement a class ListQueue<T> that combines both List and Queue functionalities.

# Operations:

* + - add(T element): Adds an element to the end of the queue (similar to enqueue in a Queue).
    - remove(): Removes and returns the element from the front of the queue (similar to dequeue in a Queue).
    - get(int index): Retrieves the element at the specified index (similar to get in a List).
    - set(int index, T element): Replaces the element at the specified index (similar to set in a List).
  1. **Internal Storage:** Use an internal List<T> to manage elements. Implement logic to support both Queue and List operations.

# Analyze and address the following challenging points:

* + - Balancing the operations of List and Queue within a single class**.**
    - Ensuring efficient performance for both random access and queue operations.

**Code : Class ListQueue**

import java.util.ArrayList;

import java.util.List;

public class ListQueue<T> {

private final List<T> storage;

private int front;

public ListQueue() {

this.storage = new ArrayList<>();

this.front = 0;

}

public void add(T element) {

storage.add(element);

}

public T remove() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

T element = storage.get(front);

front++;

if (front > storage.size() / 2) {

compact();

}

return element;

}

public T get(int index) {

checkIndex(index);

return storage.get(front + index);

}

public void set(int index, T element) {

checkIndex(index);

storage.set(front + index, element);

}

public int size() {

return storage.size() - front;

}

public boolean isEmpty() {

return front >= storage.size();

}

private void compact() {

storage.subList(0, front).clear();

front = 0;

}

private void checkIndex(int index) {

if (index < 0 || index >= size()) {

throw new IndexOutOfBoundsException("Index out of bounds: " + index);

}

}

@Override

public String toString() {

return storage.subList(front, storage.size()).toString();

}

public static void main(String[] args) {

ListQueue<Integer> listQueue = new ListQueue<>();

listQueue.add(10);

listQueue.add(20);

listQueue.add(30);

System.out.println("After adding elements: " + listQueue);

System.out.println("Removed element: " + listQueue.remove());

System.out.println("After removing: " + listQueue);

System.out.println("Element at index 1: " + listQueue.get(1));

listQueue.set(1, 50);

System.out.println("After updating index 1: " + listQueue);

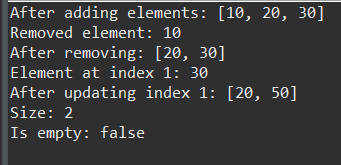
System.out.println("Size: " + listQueue.size());

System.out.println("Is empty: " + listQueue.isEmpty());

}

}

**Output**



# Type-safe Queue with Priorities

**Objective:** Create a priority queue that orders elements based on their priority and ensure correct element ordering.

# Details:

* 1. **Class Design:** Implement a generic class PriorityQueue<T> where T must implement Comparable<T>.

# Operations:

* + - enqueue(T element): Adds an element to the queue, maintaining the order based on the natural ordering of T.
    - dequeue(): Removes and returns the highest-priority element (the smallest element according to its Comparable implementation). If the queue is empty, throw a NoSuchElementException.
    - peek(): Returns the highest-priority element without removing it. If the queue is empty, throw a NoSuchElementException.
  1. **Internal Storage:** Use a List<T> to store elements. Maintain the order by sorting the list or using a priority queue mechanism.

# Analyze and address the following challenging points:

* + - Implementing efficient priority management.
    - Ensuring elements are added and removed according to their priority.

**Code : Class PriorityQueue**

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

import java.util.NoSuchElementException;

public class PriorityQueue<T extends Comparable<T>> {

private final List<T> elements;

public PriorityQueue() {

elements = new ArrayList<>();

}

public void enqueue(T element) {

if (element == null) {

throw new IllegalArgumentException("Element cannot be null.");

}

elements.add(element);

Collections.sort(elements);

}

public T dequeue() {

if (elements.isEmpty()) {

throw new NoSuchElementException("Queue is empty.");

}

return elements.remove(0);

}

public T peek() {

if (elements.isEmpty()) {

throw new NoSuchElementException("Queue is empty.");

}

return elements.get(0);

}

public int size() {

return elements.size();

}

public boolean isEmpty() {

return elements.isEmpty();

}

public static void main(String[] args) {

PriorityQueue<Integer> queue = new PriorityQueue<>();

queue.enqueue(5);

queue.enqueue(1);

queue.enqueue(3);

System.out.println("Peek: " + queue.peek());

System.out.println("Dequeue: " + queue.dequeue());

System.out.println("Dequeue: " + queue.dequeue());

System.out.println("Dequeue: " + queue.dequeue());

try {

System.out.println(queue.peek());

} catch (NoSuchElementException e) {

System.out.println("Exception: " + e.getMessage());

}

}

}

**Output**

A screenshot of a computer

Description automatically generated