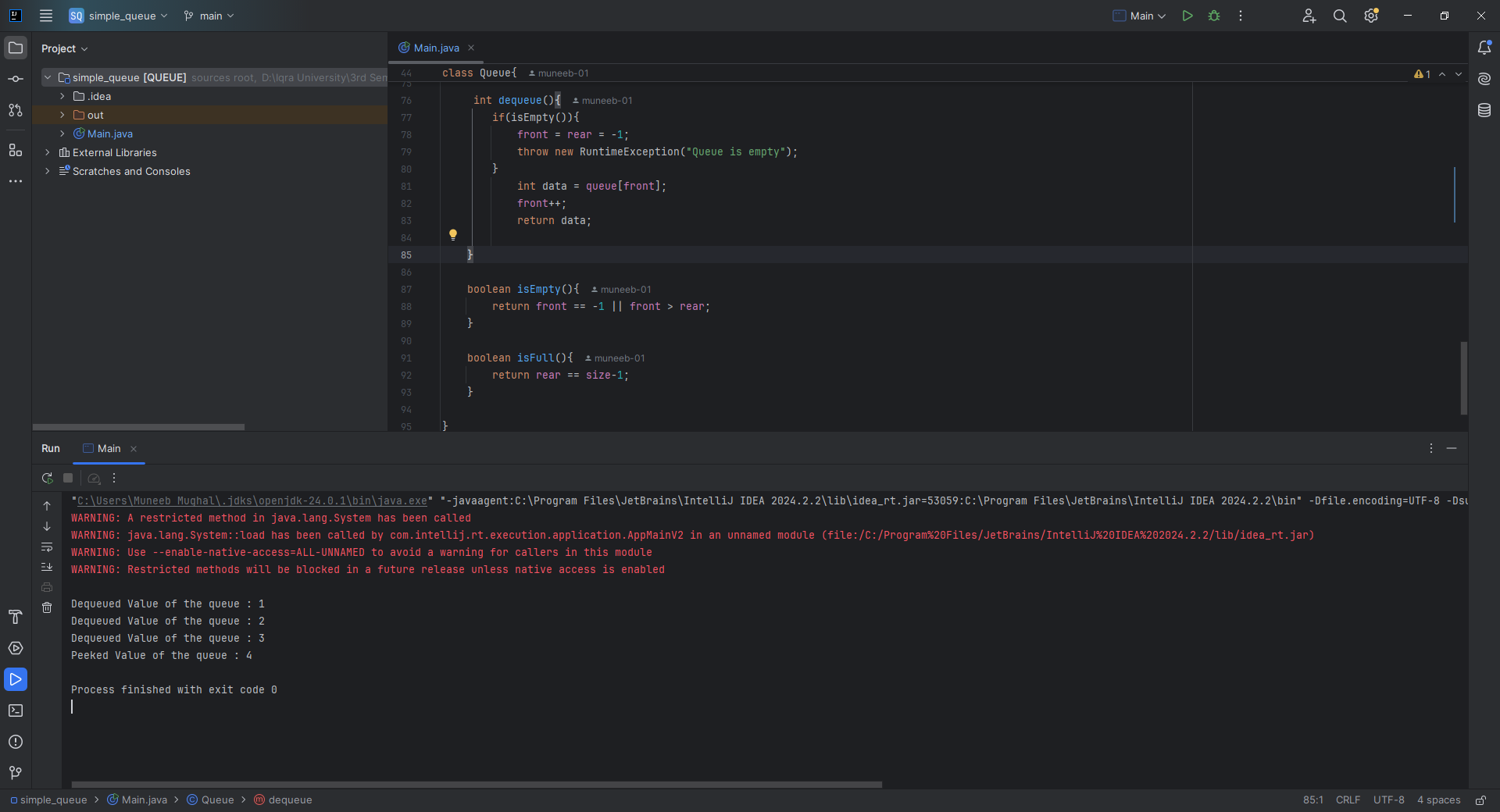
**CLASS TASK 1 & 2:**

**CODE:**  
class Queue{  
 int front = -1;  
 int rear = -1;  
 int[] queue;  
 int size;  
  
 Queue(int size){  
 queue = new int[size];  
 this.size = size;  
 }  
  
 void enqueue(int data){  
 if(isFull()){  
 System.*out*.println("Queue is full");  
 front = rear = -1;  
 }else{  
 if(isEmpty()){  
 front = 0;  
 }  
 rear++;  
 queue[rear] = data;  
 }  
 }  
  
 int peek(){  
 if(isEmpty()){  
 front = rear = -1;  
 throw new RuntimeException("Queue is empty");  
 }  
 return queue[front];  
 }  
  
 int dequeue(){  
 if(isEmpty()){  
 front = rear = -1;  
 throw new RuntimeException("Queue is empty");  
 }  
 int data = queue[front];  
 front++;  
 return data;  
  
 }  
  
 boolean isEmpty(){  
 return front == -1 || front > rear;  
 }  
  
 boolean isFull(){  
 return rear == size-1;  
 }  
  
}

public class Main {  
 public static void main(String[] args) {  
 Queue q = new Queue(5);  
 q.enqueue(1);  
 q.enqueue(2);  
 q.enqueue(3);  
 q.enqueue(4);  
 q.enqueue(5);  
  
 try{  
 System.*out*.println( "Dequeued Value of the queue : " +q.dequeue());  
 System.*out*.println( "Dequeued Value of the queue : " +q.dequeue());  
 System.*out*.println( "Dequeued Value of the queue : " +q.dequeue());  
 }  
 catch(RuntimeException e){  
 System.*out*.println(e.getMessage());  
 }  
  
 try {  
 System.*out*.println("Peeked Value of the queue : "+q.peek());  
 } catch (RuntimeException e) {  
 System.*out*.println(e.getMessage());  
 }  
 }

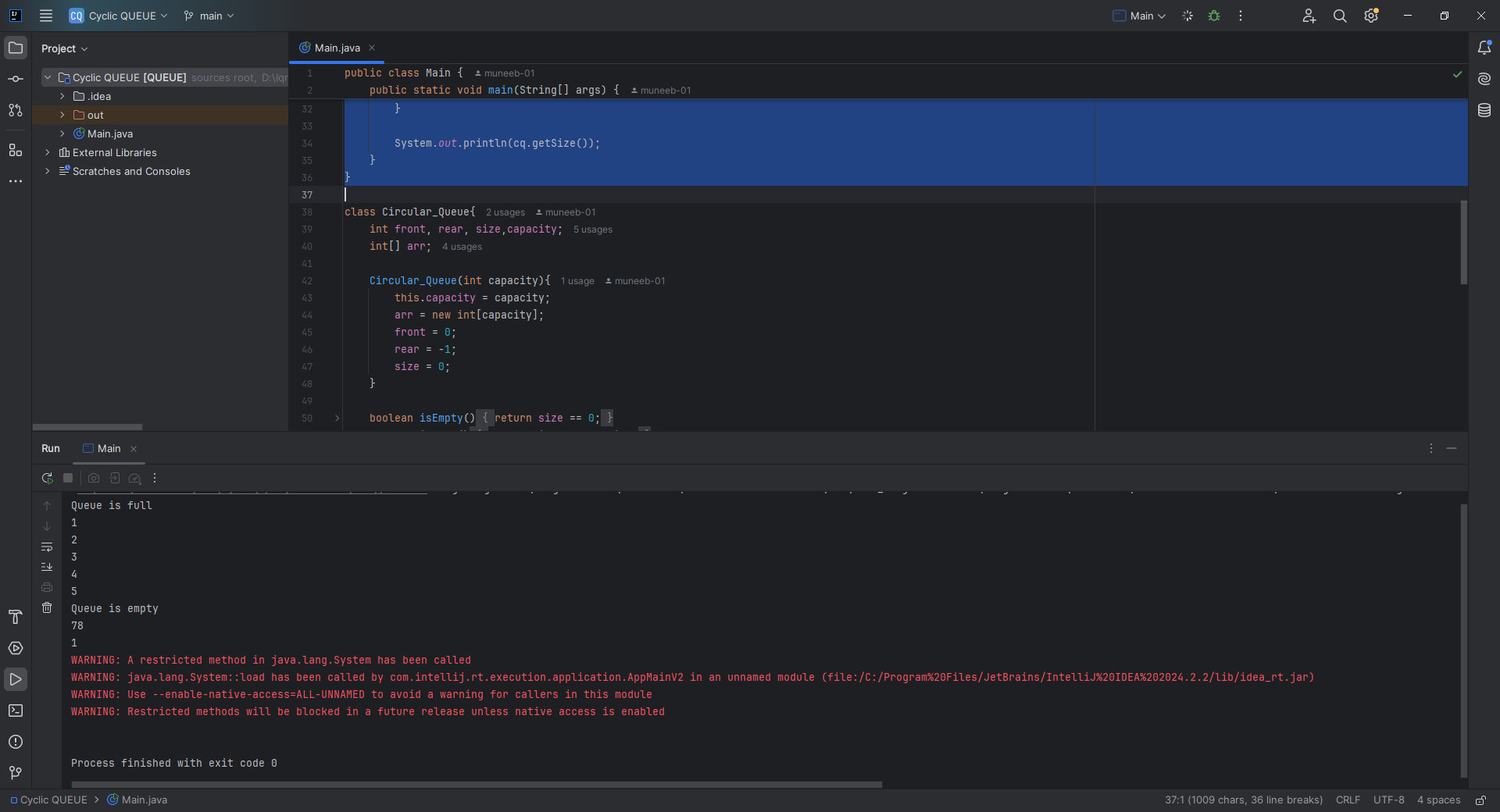


**CLASS TASK 3:**

**CODE:**

class Circular\_Queue{  
 int front, rear, size,capacity;  
 int[] arr;  
  
 Circular\_Queue(int capacity){  
 this.capacity = capacity;  
 arr = new int[capacity];  
 front = 0;  
 rear = -1;  
 size = 0;  
 }  
  
 boolean isEmpty(){  
 return size == 0;  
 }  
 boolean isFull(){  
 return size == capacity;  
 }  
 void enqueue(int value){  
 if(isFull()){  
 throw new RuntimeException("Queue is full");  
 }  
 else{  
 rear = (rear + 1) % capacity;  
 arr[rear] = value;  
 size++;  
 }  
 }  
 int dequeue(){  
 if(isEmpty()){  
 throw new RuntimeException("Queue is empty");  
 }  
 else{  
 int value = arr[front];  
 front = (front + 1) % capacity;  
 size--;  
 return value;  
 }  
 }  
 int peek(){  
 if(isEmpty()){  
 throw new RuntimeException("Queue is empty");  
 }  
 else{  
 return arr[front];  
 }  
 }  
 int getSize(){  
 return size;  
 }  
}

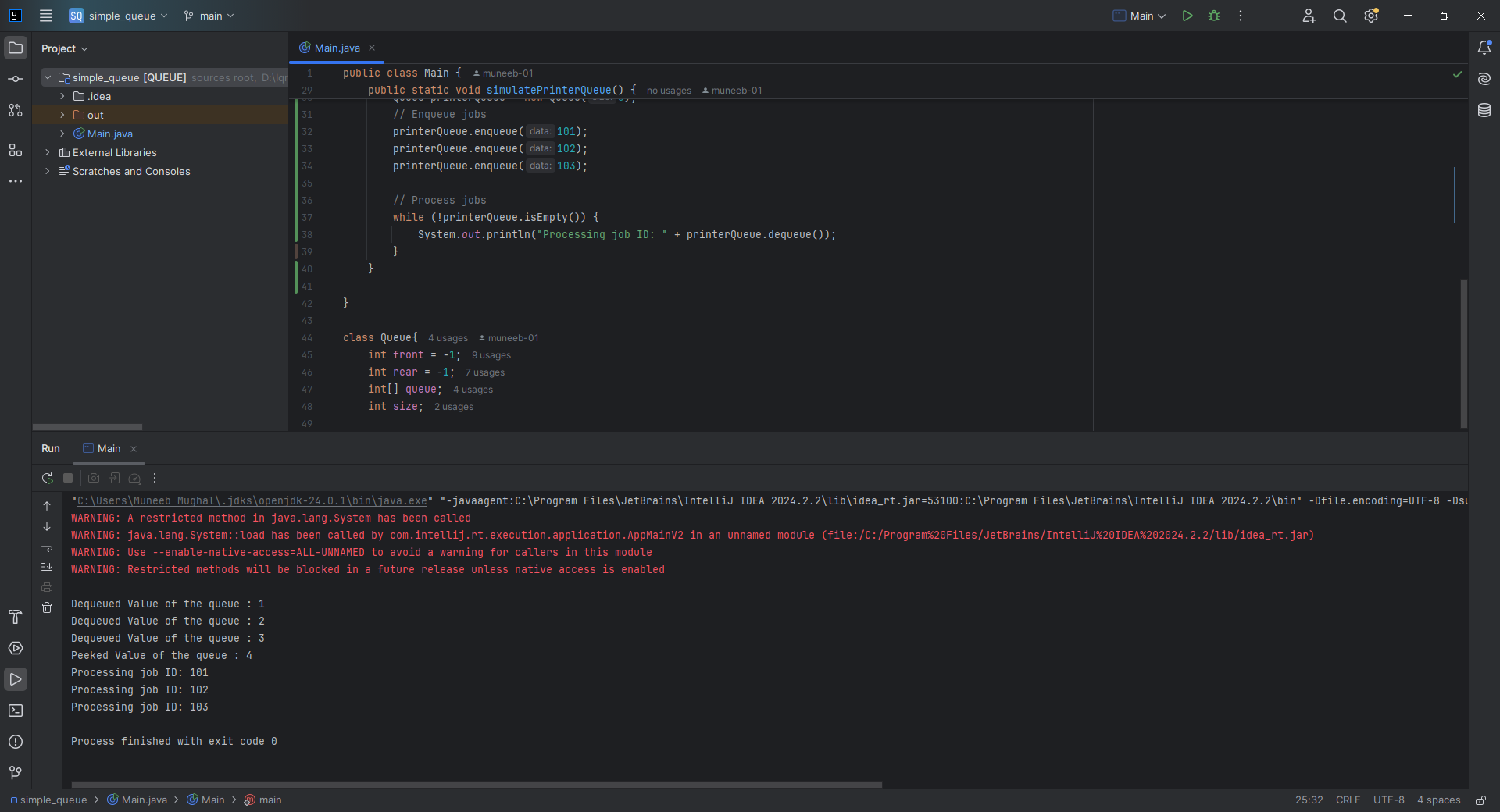
public class Main {  
 public static void main(String[] args) {  
 Circular\_Queue cq = new Circular\_Queue(5);  
 try{  
 cq.enqueue(1);  
 cq.enqueue(2);  
 cq.enqueue(3);  
 cq.enqueue(4);  
 cq.enqueue(5);  
 cq.enqueue(6);  
 }  
 catch (RuntimeException e){  
 System.*out*.println(e.getMessage());  
 }  
  
 try{  
 System.*out*.println(cq.dequeue());  
 System.*out*.println(cq.dequeue());  
 System.*out*.println(cq.dequeue());  
 System.*out*.println(cq.dequeue());  
 System.*out*.println(cq.dequeue());  
 System.*out*.println(cq.dequeue());  
 } catch (RuntimeException e) {  
 System.*out*.println(e.getMessage());  
 }  
  
 cq.enqueue(78);  
 try{  
 System.*out*.println(cq.peek());  
 }catch (RuntimeException e){  
 System.*out*.println(e.getMessage());  
 }  
  
 System.*out*.println(cq.getSize());  
 }  
}



**CLASS TASK 4**

**CODE:**

public static void simulatePrinterQueue() {  
 Queue printerQueue = new Queue(5);  
 // Enqueue jobs  
 printerQueue.enqueue(101);  
 printerQueue.enqueue(102);  
 printerQueue.enqueue(103);  
  
 // Process jobs  
 while (!printerQueue.isEmpty()) {  
 System.*out*.println("Processing job ID: " + printerQueue.dequeue());  
 }  
 }  
  
}

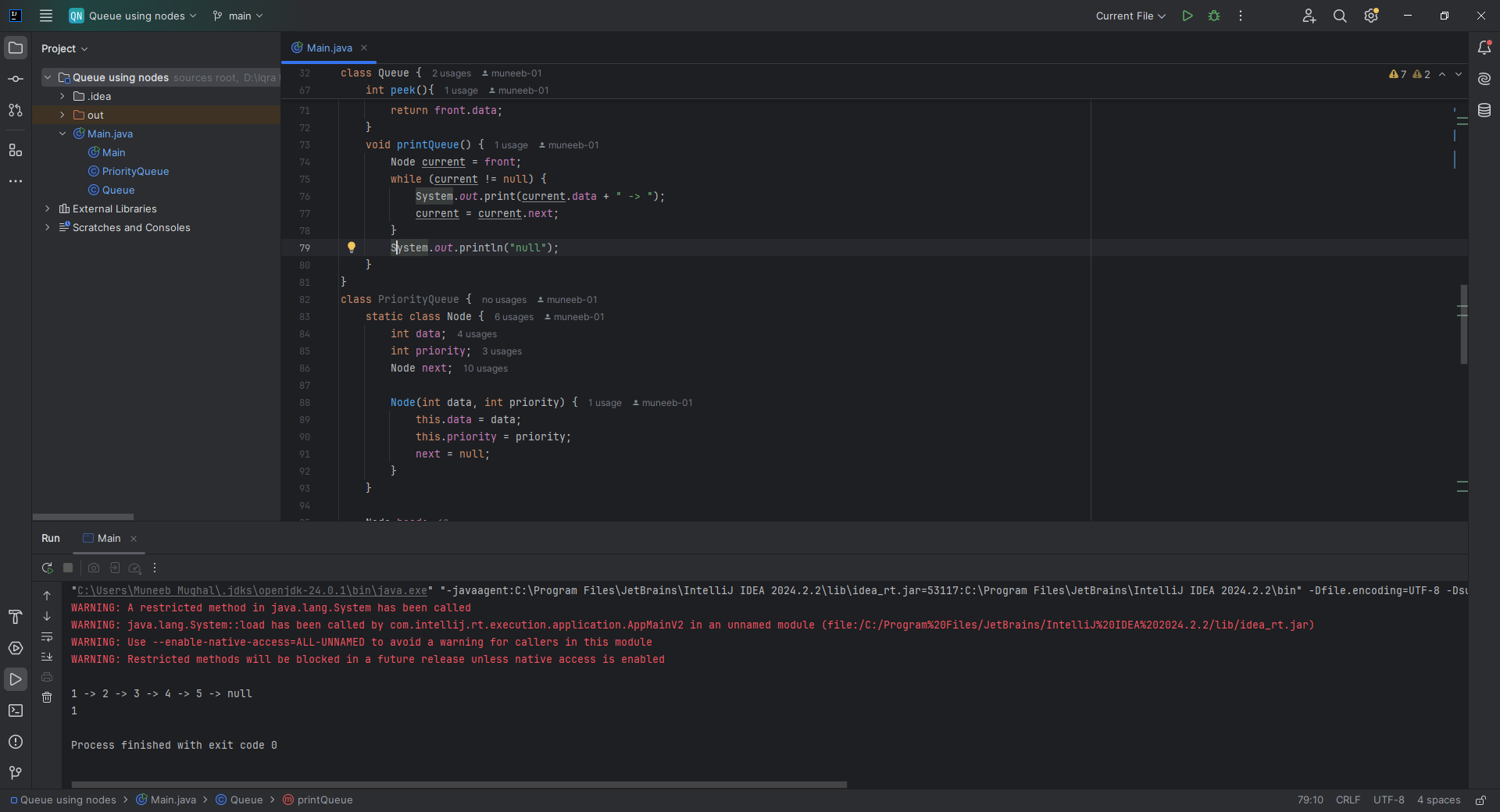


**HOME TASK 1:** Implementing a Queue Using Linked List

**CODE:**

public class Main {  
 public static void main(String[] args) {  
 Queue queue = new Queue();  
 queue.enqueue(1);  
 queue.enqueue(2);  
 queue.enqueue(3);  
 queue.enqueue(4);  
 queue.enqueue(5);  
 *reverseQueue*(queue);  
 queue.printQueue();  
 try{  
 System.*out*.println(queue.peek());  
 }catch(RuntimeException e){  
 System.*out*.println(e.getMessage());  
 }

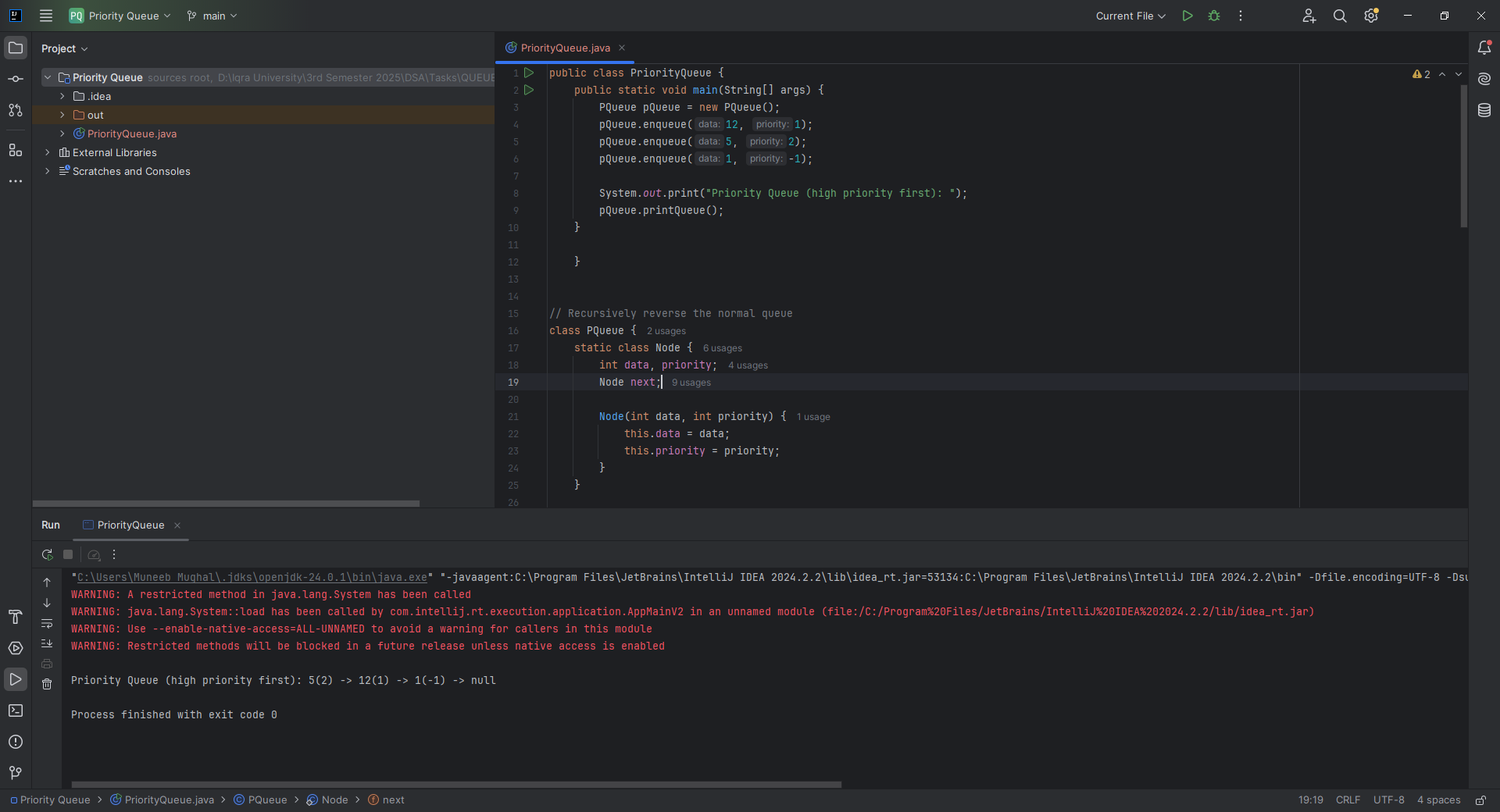
class Queue {  
 static class Node {  
 int data;  
 Node next;  
 Node(int data) {  
 this.data = data;  
 next = null;  
 }  
 }  
 Node front, rear;  
  
 Queue() {  
 front = rear = null;  
 }  
  
 void enqueue(int data) {  
 Node newNode = new Node(data);  
 if (front == null) {  
 front = rear = newNode;  
 }  
 else {  
 rear.next = newNode;  
 rear = newNode;  
 }  
 }  
  
 int dequeue() {  
 if (front == null) {  
 throw new RuntimeException("Queue is empty");  
 }  
 int value = front.data;  
 front = front.next;  
 return value;  
 }  
  
 int peek(){  
 if (front == null) {  
 throw new RuntimeException("Queue is empty");  
 }  
 return front.data;  
 }  
 void printQueue() {  
 Node current = front;  
 while (current != null) {  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
 }  
 System.*out*.println("null");  
 }  
}



**HOME TASK 2:** Implement a Priority Queue

**CODE:**

public class PriorityQueue {  
 public static void main(String[] args) {  
 PQueue pQueue = new PQueue();  
 pQueue.enqueue(12, 1);  
 pQueue.enqueue(5, 2);  
 pQueue.enqueue(1, -1);  
  
 System.*out*.print("Priority Queue (high priority first): ");  
 pQueue.printQueue();  
 }  
  
 }  
  
  
// Recursively reverse the normal queue  
class PQueue {  
 static class Node {  
 int data, priority;  
 Node next;  
  
 Node(int data, int priority) {  
 this.data = data;  
 this.priority = priority;  
 }  
 }  
  
 Node head;  
  
 void enqueue(int data, int priority) {  
 Node newNode = new Node(data, priority);  
  
 // Insert at the correct position based on priority  
 if (head == null || priority > head.priority) {  
 newNode.next = head;  
 head = newNode;  
 } else {  
 Node temp = head;  
 while (temp.next != null && temp.next.priority >= priority) {  
 temp = temp.next;  
 }  
 newNode.next = temp.next;  
 temp.next = newNode;  
 }  
 }  
  
 int dequeue() {  
 if (isEmpty()) throw new RuntimeException("Priority Queue is empty");  
 int value = head.data;  
 head = head.next;  
 return value;  
 }  
  
 int peek() {  
 if (isEmpty()) throw new RuntimeException("Priority Queue is empty");  
 return head.data;  
 }  
  
 boolean isEmpty() {  
 return head == null;  
 }  
  
 void printQueue() {  
 Node current = head;  
 while (current != null) {  
 System.*out*.print(current.data + "(" + current.priority + ") -> ");  
 current = current.next;  
 }  
 System.*out*.println("null");  
 }  
}



**Home Task 3:** Implement Recursive Queue Reversal

**CODE:**

static void reverseQueue(Queue q) {  
 if (q.front == null) return;  
  
 int frontData = q.dequeue();  
 *reverseQueue*(q); // Recursive call  
 q.enqueue(frontData); // Put dequeued item back at rear  
}

