**CLASS TASK 1: Max Heap**

**CODE:**

class MaxHeap {  
 private int[] heapArray;  
 private int maxSize;  
 private int size;  
  
 // Constructor to initialize the heap  
 public MaxHeap(int maxSize) {  
 this.maxSize = maxSize;  
 this.size = 0;  
 this.heapArray = new int[maxSize];  
 }  
  
 // Get index of parent, left child, and right child  
 private int parent(int i) { return (i - 1) / 2; }  
 private int leftChild(int i) { return 2 \* i + 1; }  
 private int rightChild(int i) { return 2 \* i + 2; }

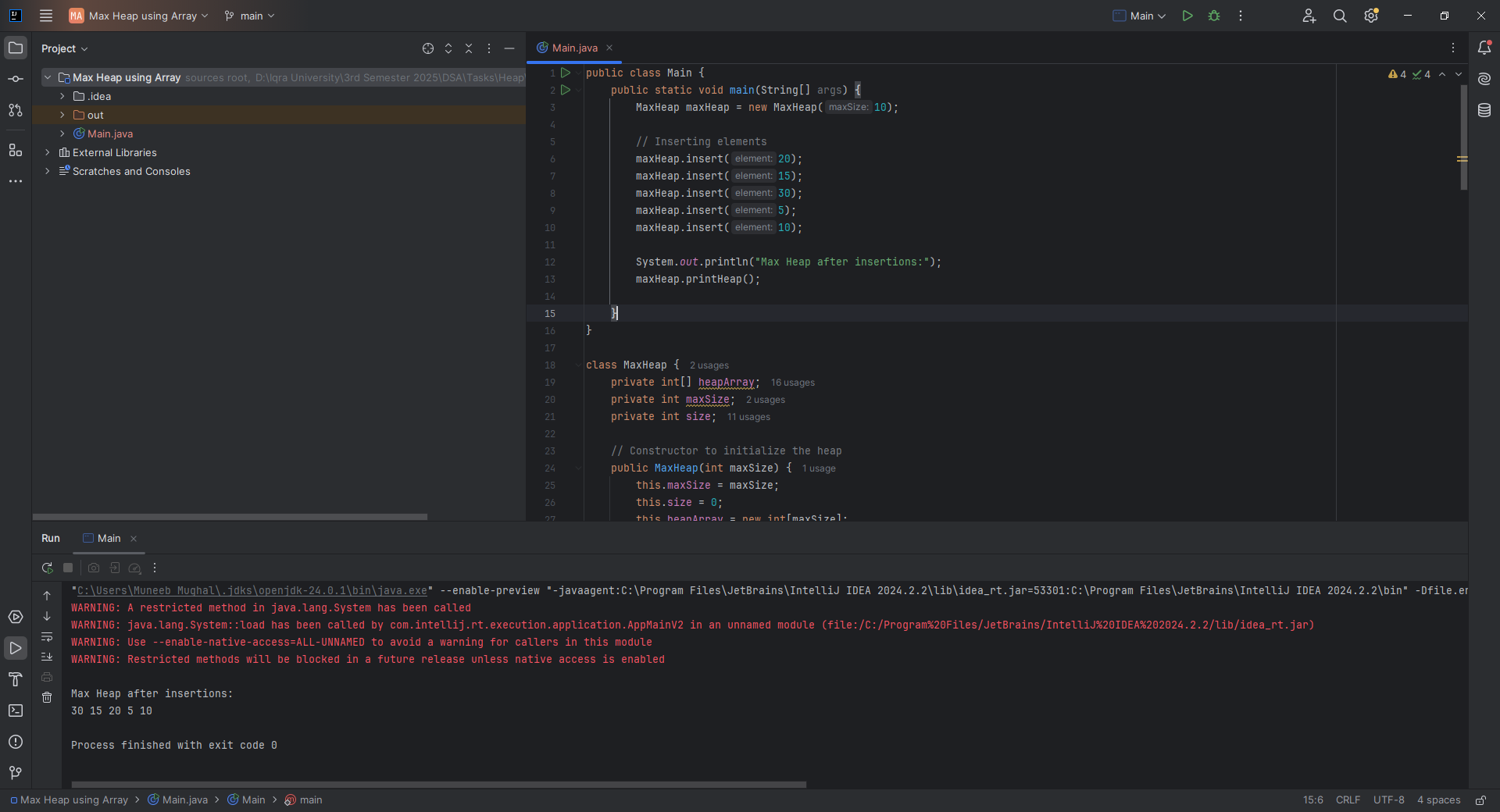
}

NO OUTPUT:

**CLASS TASK 2**

**CODE:**

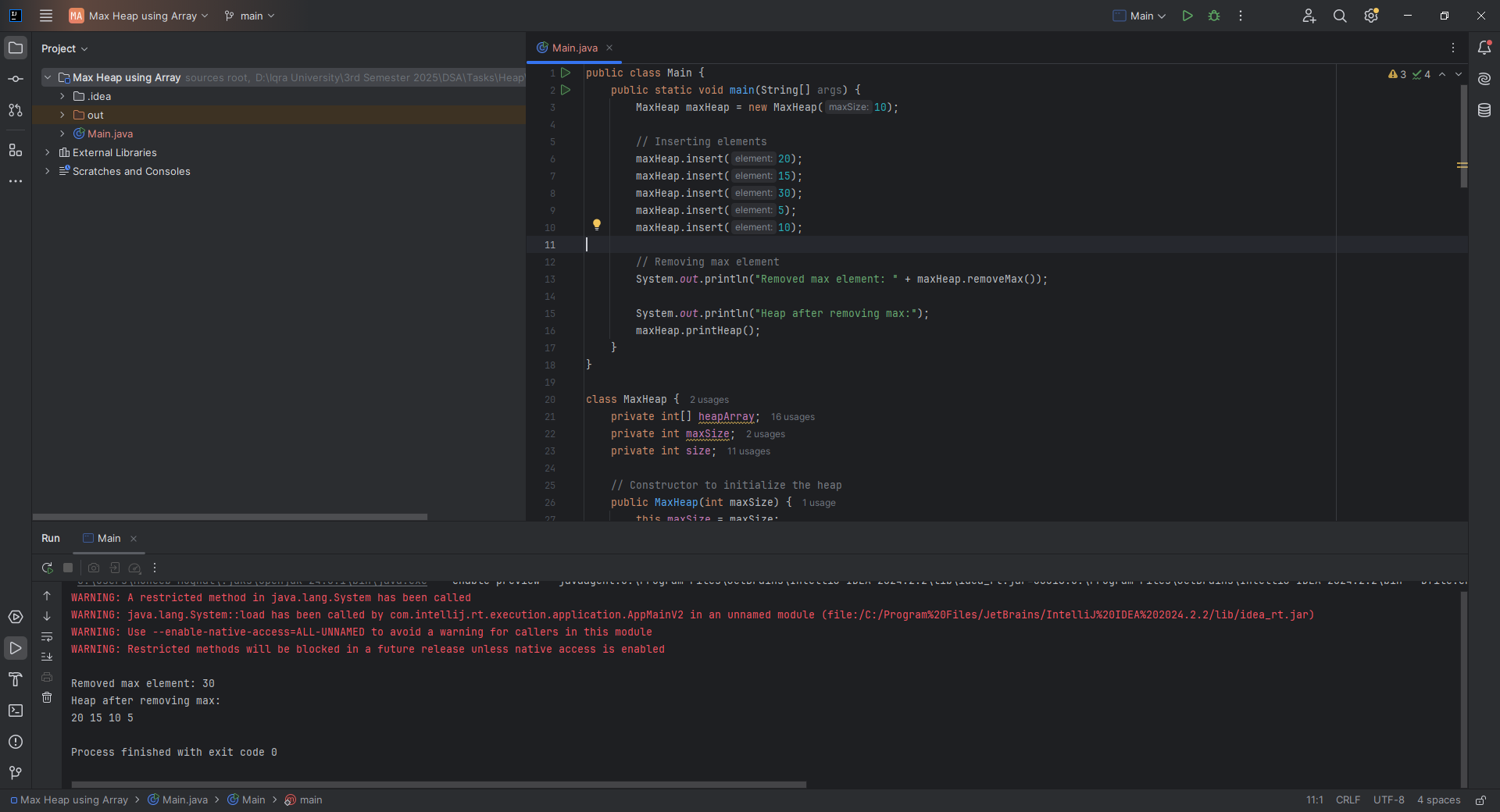
public class Main {  
 public static void main(String[] args) {  
 MaxHeap maxHeap = new MaxHeap(10);  
  
 // Inserting elements  
 maxHeap.insert(20);  
 maxHeap.insert(15);  
 maxHeap.insert(30);  
 maxHeap.insert(5);  
 maxHeap.insert(10);  
  
 System.*out*.println("Max Heap after insertions:");  
 maxHeap.printHeap();  
  
 }  
}  
  
class MaxHeap {  
 private int[] heapArray;  
 private int maxSize;  
 private int size;  
  
 // Constructor to initialize the heap  
 public MaxHeap(int maxSize) {  
 this.maxSize = maxSize;  
 this.size = 0;  
 this.heapArray = new int[maxSize];  
 }  
  
 // Get index of parent, left child, and right child  
 private int parent(int i) { return (i - 1) / 2; }  
 private int leftChild(int i) { return 2 \* i + 1; }  
 private int rightChild(int i) { return 2 \* i + 2; }  
  
 // Insert an element into the heap  
 public void insert(int element) {  
 if (size >= maxSize) {  
 System.*out*.println("Heap is full. Cannot insert.");  
 return;  
 }  
  
 heapArray[size] = element;  
 int current = size;  
 size++;  
  
 // Heapify-up  
 while (current > 0 && heapArray[current] > heapArray[parent(current)]) {  
 swap(current, parent(current));  
 current = parent(current);  
 }  
 }  
  
 // Swap elements  
 private void swap(int i, int j) {  
 int temp = heapArray[i];  
 heapArray[i] = heapArray[j];  
 heapArray[j] = temp;  
 }  
  
 // Print the heap (for testing)  
 public void printHeap() {  
 for (int i = 0; i < size; i++) {  
 System.*out*.print(heapArray[i] + " ");  
 }  
 System.*out*.println();  
 }  
}



**CLASS TASK 3: Removing a max element**

**CODE:**

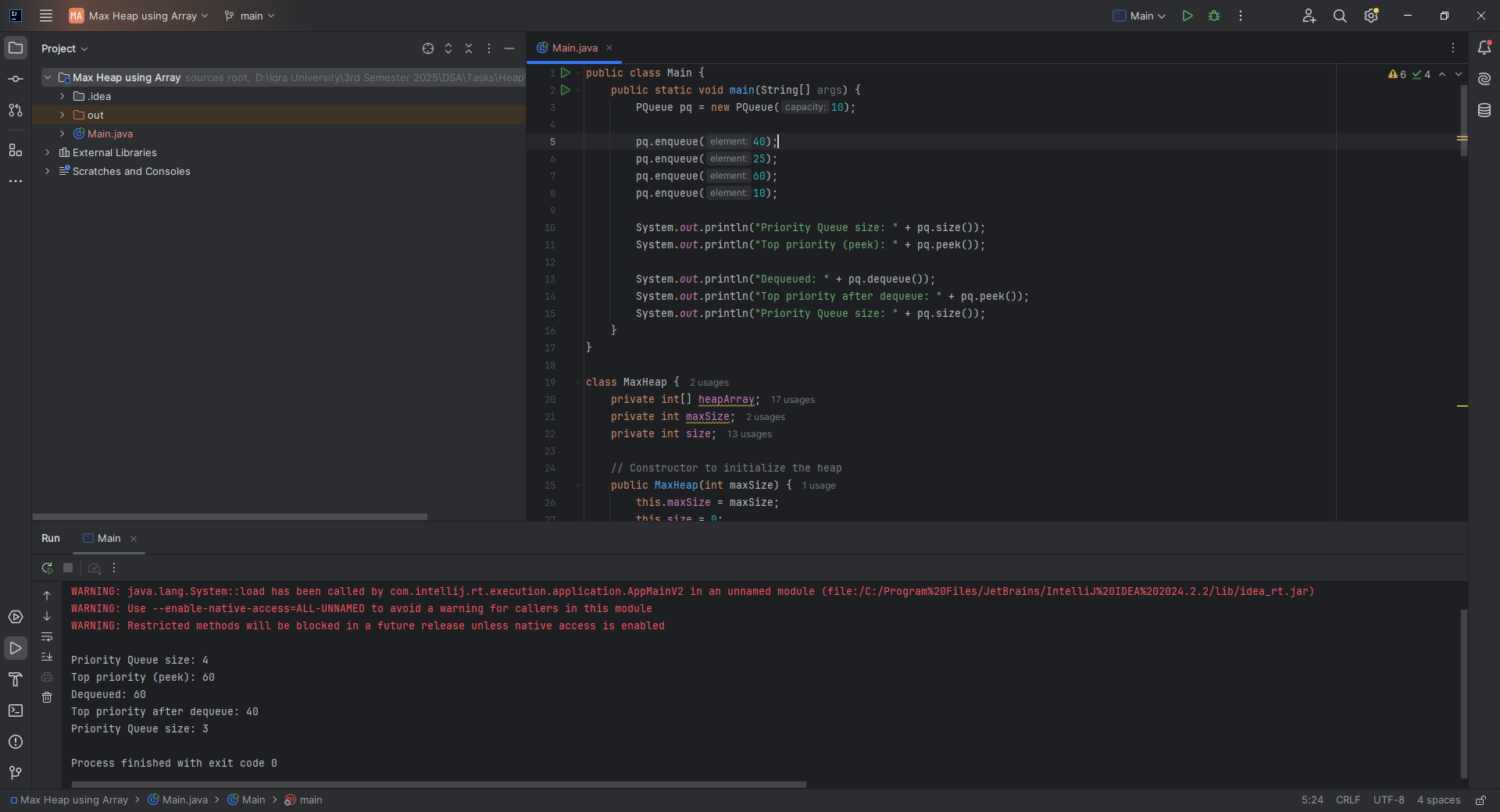
public int removeMax() {  
 if (size == 0) {  
 throw new IllegalStateException("Heap is empty");  
 }  
  
 int max = heapArray[0];  
 heapArray[0] = heapArray[size - 1];  
 size--;  
  
 // Heapify-down  
 heapify(0);  
  
 return max;  
}  
  
// Heapify-down method  
private void heapify(int i) {  
 int largest = i;  
 int left = leftChild(i);  
 int right = rightChild(i);  
  
 if (left < size && heapArray[left] > heapArray[largest]) {  
 largest = left;  
 }  
  
 if (right < size && heapArray[right] > heapArray[largest]) {  
 largest = right;  
 }  
  
 if (largest != i) {  
 swap(i, largest);  
 heapify(largest);  
 }  
}  
  
// Swap elements  
private void swap(int i, int j) {  
 int temp = heapArray[i];  
 heapArray[i] = heapArray[j];  
 heapArray[j] = temp;  
}



**CLASS TASK 5:** Implementing a Priority Queue with Max Heap

**CODE:**

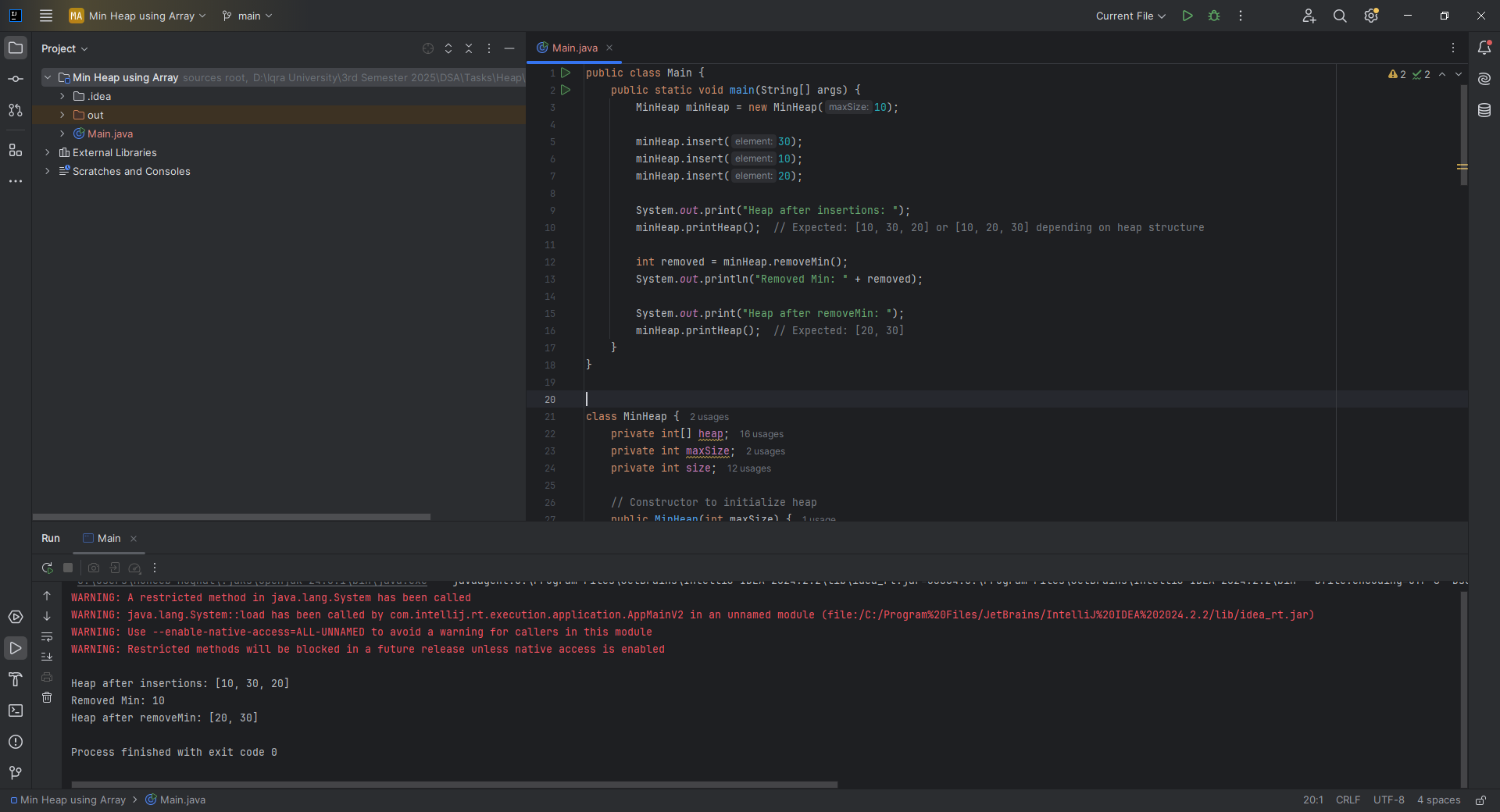
class PQueue {  
 private MaxHeap heap;  
  
 // Constructor: initialize PQueue with a max size  
 public PQueue(int capacity) {  
 this.heap = new MaxHeap(capacity);  
 }  
  
 // Insert an element into the priority queue  
 public void enqueue(int element) {  
 heap.insert(element);  
 }  
  
 // Remove and return the highest priority element  
 public int dequeue() {  
 return heap.removeMax();  
 }  
  
 // Peek at the highest priority element without removing  
 public int peek() {  
 if (heap.getSize() == 0) {  
 throw new IllegalStateException("Priority Queue is empty");  
 }  
 return heap.getMax();  
 }  
  
 // Check if the priority queue is empty  
 public boolean isEmpty() {  
 return heap.getSize() == 0;  
 }  
  
 // Get current size of the priority queue  
 public int size() {  
 return heap.getSize();  
 }  
}



**HOME TASK 1:**

**CODE:**

public class Main {  
 public static void main(String[] args) {  
 MinHeap minHeap = new MinHeap(10);  
  
 minHeap.insert(30);  
 minHeap.insert(10);  
 minHeap.insert(20);  
  
 System.*out*.print("Heap after insertions: ");  
 minHeap.printHeap(); // Expected: [10, 30, 20] or [10, 20, 30] depending on heap structure  
  
 int removed = minHeap.removeMin();  
 System.*out*.println("Removed Min: " + removed);  
  
 System.*out*.print("Heap after removeMin: ");  
 minHeap.printHeap(); // Expected: [20, 30]  
 }  
}  
  
  
class MinHeap {  
 private int[] heap;  
 private int maxSize;  
 private int size;  
  
 // Constructor to initialize heap  
 public MinHeap(int maxSize) {  
 this.maxSize = maxSize;  
 this.size = 0;  
 this.heap = new int[maxSize];  
 }  
  
 // Index helpers  
 private int parent(int i) { return (i - 1) / 2; }  
 private int leftChild(int i) { return 2 \* i + 1; }  
 private int rightChild(int i) { return 2 \* i + 2; }  
  
 // Swap elements  
 private void swap(int i, int j) {  
 int tmp = heap[i];  
 heap[i] = heap[j];  
 heap[j] = tmp;  
 }  
  
 // Insert a new element into the heap  
 public void insert(int element) {  
 if (size >= maxSize) {  
 System.*out*.println("Heap is full. Cannot insert.");  
 return;  
 }  
  
 heap[size] = element;  
 heapifyUp(size);  
 size++;  
 }  
  
 // Remove and return the minimum element (root)  
 public int removeMin() {  
 if (size == 0) {  
 throw new IllegalStateException("Heap is empty");  
 }  
  
 int min = heap[0];  
 heap[0] = heap[size - 1];  
 size--;  
  
 heapifyDown(0);  
 return min;  
 }  
  
 // Maintain min-heap by moving the element up  
 private void heapifyUp(int index) {  
 while (index > 0 && heap[index] < heap[parent(index)]) {  
 swap(index, parent(index));  
 index = parent(index);  
 }  
 }  
  
 // Maintain min-heap by moving the element down  
 private void heapifyDown(int index) {  
 int smallest = index;  
 int left = leftChild(index);  
 int right = rightChild(index);  
  
 if (left < size && heap[left] < heap[smallest]) {  
 smallest = left;  
 }  
  
 if (right < size && heap[right] < heap[smallest]) {  
 smallest = right;  
 }  
  
 if (smallest != index) {  
 swap(index, smallest);  
 heapifyDown(smallest);  
 }  
 }  
  
 // Print the heap  
 public void printHeap() {  
 System.*out*.print("[");  
 for (int i = 0; i < size; i++) {  
 System.*out*.print(heap[i]);  
 if (i < size - 1) System.*out*.print(", ");  
 }  
 System.*out*.println("]");  
 }  
}



**HOME TASK 2**

**CODE:**

public class Main {  
 public static void main(String[] args) {  
 int[] arr = {12, 11, 13, 5, 6, 7};  
  
 System.*out*.print("Input Array : ");  
 *print\_arr*(arr);  
  
 *heapSort*(arr);  
 System.*out*.print("Sorted Array : ");  
 *print\_arr*(arr);  
  
 }  
  
 static void print\_arr(int[] arr){  
 for(int i=0; i<arr.length; i++){  
 System.*out*.print(arr[i] + " ");  
 }  
 System.*out*.println();  
 }  
  
 static void heapSort(int[] arr) {  
 int n = arr.length;  
  
 // Build max heap  
 for (int i = n / 2 - 1; i >= 0; i--) *heapify*(arr, n, i);  
  
 // Extract elements  
 for (int i = n - 1; i > 0; i--) {  
 int temp = arr[0]; arr[0] = arr[i]; arr[i] = temp;  
 *heapify*(arr, i, 0);  
 }  
 }  
  
 static void heapify(int[] arr, int heapSize, int root) {  
 int largest = root;  
 int left = 2 \* root + 1;  
 int right = 2 \* root + 2;  
  
 if (left < heapSize && arr[left] > arr[largest]) largest = left;  
 if (right < heapSize && arr[right] > arr[largest]) largest = right;  
  
 if (largest != root) {  
 int temp = arr[root];  
 arr[root] = arr[largest]; arr[largest] = temp;  
 *heapify*(arr, heapSize, largest);  
 }  
 }  
}

