To implement the clone() method for the ArrayQueue class, we need to create a new queue that has the same elements as the original queue, preserving the order and maintaining the correct front and rear positions.

**Steps:**

1. **Create a New ArrayQueue**: We'll create a new queue with the same capacity as the original queue.
2. **Copy the Elements**: We'll copy the elements from the original queue to the new queue, considering the circular nature of the queue. We need to handle the case where the queue might wrap around.
3. **Copy the Queue State**: We'll also need to copy the front, rear, and size values to ensure that the cloned queue behaves the same as the original.

**Code Implementation:**

public class ArrayQueue<T> implements Cloneable {

private T[] data;

private int front; // Index of the front element

private int rear; // Index of the rear element

private int size; // Current size of the queue

private int capacity;

public ArrayQueue(int capacity) {

this.capacity = capacity;

this.data = (T[]) new Object[capacity];

this.front = 0;

this.rear = 0;

this.size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public boolean isFull() {

return size == capacity;

}

public void enqueue(T item) {

if (isFull()) throw new IllegalStateException("Queue is full");

data[rear] = item;

rear = (rear + 1) % capacity;

size++;

}

public T dequeue() {

if (isEmpty()) throw new IllegalStateException("Queue is empty");

T item = data[front];

front = (front + 1) % capacity;

size--;

return item;

}

public void rotate() {

if (isEmpty()) return;

front = (front + 1) % capacity;

rear = (rear + 1) % capacity;

}

public T peek() {

if (isEmpty()) throw new IllegalStateException("Queue is empty");

return data[front];

}

public int size() {

return size;

}

// Clone method: creates a new ArrayQueue with the same elements

@Override

public ArrayQueue<T> clone() {

try {

@SuppressWarnings("unchecked")

ArrayQueue<T> clone = (ArrayQueue<T>) super.clone(); // Clone the base object

clone.data = (T[]) new Object[capacity]; // Create a new array for the cloned queue

clone.front = this.front;

clone.rear = this.rear;

clone.size = this.size;

// Copy elements in the same order considering circular nature

for (int i = 0; i < size; i++) {

clone.data[(clone.front + i) % capacity] = this.data[(this.front + i) % capacity];

}

return clone;

} catch (CloneNotSupportedException e) {

throw new AssertionError("Clone not supported", e);

}

}

}

**Explanation of clone() method:**

* **super.clone()**: First, we call super.clone() to create a shallow copy of the current object (this handles the object structure and primitive types like integers and the capacity).
* **New Array Creation**: Since we want a deep copy of the queue, we create a new array data to hold the queue elements in the cloned queue.
* **Copying the State**: We copy the front, rear, and size values from the original queue to the cloned queue. These values define the state of the queue and help maintain its functionality.
* **Copying the Elements**: We iterate through the elements in the original queue, considering the circular structure, and copy them to the new array in the same order.
  + We use the formula (front + i) % capacity to handle the circular indexing correctly.

**Usage:**

The clone() method allows you to create an exact duplicate of the queue. Here's an example usage:

ArrayQueue<Integer> originalQueue = new ArrayQueue<>(5);

originalQueue.enqueue(1);

originalQueue.enqueue(2);

originalQueue.enqueue(3);

ArrayQueue<Integer> clonedQueue = originalQueue.clone();

System.out.println("Original Queue size: " + originalQueue.size()); // 3

System.out.println("Cloned Queue size: " + clonedQueue.size()); // 3

**Efficiency:**

* **Time Complexity**: The time complexity for the clone() method is O(n), where n is the size of the queue, because we copy each element once.
* **Space Complexity**: The space complexity is O(n) due to the need for a new array to hold the cloned elements.