Java Linked List Partition

Write java code for Given the head of a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.
You should preserve the original relative order of the nodes in each of the two partitions.
Input: head = [1,4,3,2,5,2], x = 3
Output: [1,2,2,4,3,5]
Input: head = [2,1], x = 2
Output: [1,2]
Use the method
public ListNode partition(ListNode head, int x) {

}

Here's a Java implementation of the `partition` method for a linked list based on the problem description:

java

```java
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }
}

public class Solution {
    public ListNode partition(ListNode head, int x) {
        // Create two dummy nodes for the two lists
        ListNode beforeHead = new ListNode(0);
        ListNode afterHead = new ListNode(0);

        // These will help us build the before and after lists
        ListNode before = beforeHead;
        ListNode after = afterHead;

        // Traverse the original list
        while (head != null) {
            if (head.val < x) {
                // Add to the 'before' list
                before.next = head;
                before = before.next;
            } else {
                // Add to the 'after' list
                after.next = head;
                after = after.next;
            }
            head = head.next;
        }

        // Prevent cycle by setting the next of the last node in 'after' list to null
        after.next = null;

        // Connect the two lists
        before.next = afterHead.next;

        // Return the head of the 'before' list, which is the new partitioned list
```

```
            return beforeHead.next;
        }
    }
```

Explanation:

1. **Two Lists Approach**:
    - We use two lists: one to store nodes with values less than `x` and another to store nodes with values greater than or equal to `x`.
    - We initialize two dummy nodes (`beforeHead` and `afterHead`) to easily manage the two partitions.
2. **Traversal and Partitioning**:
    - Traverse the original linked list, and depending on the node's value, append it to either the `before` list (if the value is less than `x`) or the `after` list.
3. **Concatenation**:
    - After traversing the entire list, we concatenate the `before` list with the `after` list and return the result.

Time Complexity:

- **O(n)**: Where `n` is the number of nodes in the linked list, as we are traversing the list only once.

This code ensures that the relative order of the nodes in each partition remains unchanged.