```java
import java.util.Random;
public class RandomizedQuickSort {

    // Main QuickSort function
    public void quickSort(int[] arr, int low, int high) {
        // Base case: only proceed if there are elements to sort
        if (low < high) {
            // Call the randomized partition to get a pivot index
            int pivotIndex = randomizedPartition(arr, low, high);

            // Recursively apply QuickSort to the left subarray
            quickSort(arr, low, pivotIndex - 1);

            // Recursively apply QuickSort to the right subarray
            quickSort(arr, pivotIndex + 1, high);
        }
    }

    // Function to randomly select a pivot and partition the array
    private int randomizedPartition(int[] arr, int low, int high) {
        // Select a random index between low and high as the pivot
        int pivotIndex = new Random().nextInt(high - low + 1) + low;

        // Move the random pivot to the end of the array (position 'high')
        swap(arr, pivotIndex, high);

        // Call the partition function and return the final pivot index
        return partition(arr, low, high);
    }

    // Standard partition function that arranges elements around the pivot
    private int partition(int[] arr, int low, int high) {
        // Use the last element as the pivot (after moving random pivot to end)
        int pivot = arr[high];

        // Pointer to keep track of the "smaller" section of the array
        int i = low - 1;

        // Iterate through the array, moving smaller elements to the left
        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;  // Move boundary for the "smaller" section
                swap(arr, i, j);  // Place current element in the "smaller"
section
            }
        }

        // Move the pivot to its correct sorted position
        swap(arr, i + 1, high);

        // Return the index of the pivot
        return i + 1;
    }
```

```
    // Utility function to swap two elements in the array
    private void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```

How we applied the example :

```
Initial Setup
Array: [8, 7, 1, 3, 5, 6, 4]
Random Pivot: 7 (element at index 1)
Steps
Randomized Partition:

Move the randomly chosen pivot (7) to the end of the array by swapping it with
the last element.
The array now becomes [8, 4, 1, 3, 5, 6, 7].Now we will use Lomuto’s partition
scheme with 7 as the pivot.
Partitioning:
Pivot: 7 (now the last element in the array)
Initialize Pointers: i = -1, j = 0Partition Process:

Step 1: j = 0, arr[j] = 8 (no swap since 8 > 7)
Step 2: j = 1, arr[j] = 4 (swap with arr[i + 1], increment i to 0 → Array: [4, 8,
1, 3, 5, 6, 7])
Step 3: j = 2, arr[j] = 1 (swap with arr[i + 1], increment i to 1 → Array: [4, 1,
8, 3, 5, 6, 7])
Step 4: j = 3, arr[j] = 3 (swap with arr[i + 1], increment i to 2 → Array: [4, 1,
3, 8, 5, 6, 7])
Step 5: j = 4, arr[j] = 5 (swap with arr[i + 1], increment i to 3 → Array: [4, 1,
3, 5, 8, 6, 7])
Step 6: j = 5, arr[j] = 6 (swap with arr[i + 1], increment i to 4 → Array: [4, 1,
3, 5, 6, 8, 7])After iterating through the array, i = 4.
Place the Pivot:

Swap the pivot 7 (at index high = 6) with arr[i + 1] (at index 5).
The array becomes [4, 1, 3, 5, 6, 7, 8].Result After Partition:
After partitioning, the array is [4, 1, 3, 5, 6, 7, 8], with 7 in its final
sorted position at index 5.Recursive QuickSort:
Now, QuickSort is recursively applied to the left and right sections:Left
Subarray: [4, 1, 3, 5, 6]Right Subarray: [8]
```

Exercise 2:

```
public class QuickSortLomuto {

    // Main QuickSort function
    public void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            // Partition the array and get the pivot index
            int pivotIndex = lomutoPartition(arr, low, high);

            // Recursively apply QuickSort to elements left of the pivot
            quickSort(arr, low, pivotIndex - 1);

            // Recursively apply QuickSort to elements right of the pivot
            quickSort(arr, pivotIndex + 1, high);
```

```
        }
    }

    // Lomuto partition function
    private int lomutoPartition(int[] arr, int low, int high) {
        int pivot = arr[high]; // Choose the last element as pivot
        int i = low - 1;       // Initialize pointer for smaller elements

        // Loop through the array to position elements around the pivot
        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {  // If element is smaller than pivot
                i++;                // Move the boundary of smaller section
                swap(arr, i, j);    // Place the element on the left side
            }
        }

        // Place the pivot in its correct sorted position
        swap(arr, i + 1, high);

        // Return the index of the pivot
        return i + 1;
    }

    // Utility function to swap two elements in the array
    private void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```

How we applied the example :

```
Initially :
  Array: [8, 7, 1, 3, 5, 6, 4]
  Pivot: 4 (last element)
  Initialize: i = -1, j = 0
Partition Steps1.Iterate through the array with j from 0 to high - 1 (i.e., up to
the second-to-last element).2.Compare each element to the pivot:
oIf arr[j] is less than or equal to the pivot (4), increment i and swap arr[i]
with arr[j].3.After the loop, place the pivot (4) in the correct position by
swapping arr[i + 1] with arr[high].
Detailed Partition Process:
  Step 1: j = 0, arr[j] = 8 (no swap since 8 > 4)
  Step 2: j = 1, arr[j] = 7 (no swap since 7 > 4)
  Step 3: j = 2, arr[j] = 1 (swap with arr[i + 1], i = 0 → Array: [1, 7, 8, 3, 5,
6, 4])
  Step 4: j = 3, arr[j] = 3 (swap with arr[i + 1], i = 1 → Array: [1, 3, 8, 7, 5,
6, 4])
  Step 5: j = 4, arr[j] = 5 (no swap since 5 > 4)
  Step 6: j = 5, arr[j] = 6 (no swap since 6 > 4)
  End of loop: Place the pivot 4 by swapping arr[i + 1] with arr[high], resulting
in [1, 3, 4, 7, 5, 6, 8].Final Array after Partition
  After partitioning, the array becomes [1, 3, 4, 7, 5, 6, 8], with 4 correctly
placed in its final sorted position.
```