PR1:-

List:

```
import java.util.ArrayList;
public class Main{
   public static void main(String[] args) {
      ArrayList<String> list = new ArrayList<>();

      // Adding elements to the ArrayList
      list.add("Apple");
      list.add("Banana");
      list.add("Orange");
      list.add("Mango");
      System.out.println("\nArrayList after insertion: " + list);

      // Accessing an element by index
      String fruit = list.get(2);
      System.out.println("\nElement at index 2: " + fruit);

      // Updating an element at a specific index
      list.set(1, "Grapes");
      System.out.println("\nArrayList after updating element at index 1: " + list);

      // Removing an element
      list.remove("Mango");
      System.out.println("\nArrayList after removing 'Mango': " + list);
```

```
        // Checking if the list contains a specific element
        boolean hasApple = list.contains("Apple");
        System.out.println("\nDoes the list contain 'Apple'? " + hasApple);

        // Getting the size of the ArrayList
        int size = list.size();
        System.out.println("\nSize of the ArrayList: " + size);

        // Clearing all elements from the ArrayList
        list.clear();
        System.out.println("\nArrayList after clearing all elements: " + list);
    }
}
```

A List is a collection that maintains the order of elements and allows duplicates. It is implemented through classes like ArrayList, LinkedList, and Vector. A List is useful when you want to preserve the sequence of data, such as maintaining a list of items in the order they were added.

The key feature of a List is its ability to access elements through their index. This makes it ideal for scenarios where you need to fetch or update elements frequently. For example, if you need to retrieve the third element from a collection, you can directly use the get(2) method.

In a List, duplicate values are allowed, which makes it different from a Set. This property is important for applications like maintaining an inventory list, where the same product might appear multiple times.

Lists can grow dynamically, meaning they can resize themselves as elements are added or removed. Operations like adding, removing, and updating elements are straightforward, making the List an essential component of Java's Collection Framework.

2. Set

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // Creating a Set using HashSet
        Set<String> set = new HashSet<>();

        // Adding elements to the Set
        set.add("Red");
        set.add("Blue");
        set.add("Green");
        set.add("Yellow");
        set.add("Red"); // Duplicate, will be ignored

        System.out.println("Set after insertion: " + set);

        // Checking if the Set contains a specific element
        boolean containsBlue = set.contains("Blue");
        System.out.println("\nDoes the Set contain 'Blue'? " + containsBlue);

        // Removing an element from the Set
        set.remove("Yellow");
        System.out.println("\nSet after removing 'Yellow': " + set);

        // Iterating over the Set
```

```
        System.out.println("\nIterating over the Set:");
        for (String color : set) {
            System.out.println(color);
        }

        // Getting the size of the Set
        int size = set.size();
        System.out.println("\nSize of the Set: " + size);

        // Clearing the Set
        set.clear();
        System.out.println("\nSet after clearing all elements: " + set);
    }
}
```

A Set is a collection that does not allow duplicate elements. It is implemented using classes like HashSet, LinkedHashSet, and TreeSet. A Set is particularly useful when you need to ensure that all elements in a collection are unique.

The HashSet is an unordered collection, meaning it does not maintain the order of elements. It is backed by a hash table and provides constant-time performance for basic operations like adding, removing, and checking if an element exists.

The LinkedHashSet is similar to HashSet but maintains the insertion order of elements. This is useful in scenarios where the order of elements is important but duplicates are not allowed.

The TreeSet, on the other hand, maintains elements in a sorted order. It is implemented as a Red-Black Tree, making it suitable for tasks where you need sorted data and uniqueness.

Sets are commonly used in applications like managing unique user IDs, storing configurations without duplicates, or finding the union or intersection of multiple data sets.

3. Queue:

```
import java.util.*;

public class Main {

    public static void main(String[] args) {
        System.out.println("\nQueue Example (LinkedList):");

        // Creating a Queue using LinkedList
        Queue<String> queue = new LinkedList<>();

        // Adding elements to the queue using offer() method
        queue.offer("Tom");
        queue.offer("Jerry");
        queue.offer("Mickey");
        queue.offer("Donald");

        // Display the current Queue
        System.out.println("Queue: " + queue);

        // Removing an element from the queue (FIFO: First In, First Out)
        System.out.println("Removing from Queue: " + queue.poll());

        // Display the Queue after removing an element
        System.out.println("Queue after poll: " + queue);
    }
```

}

In Java, a **Queue** is a collection used to store elements in a specific order. It follows the **First In, First Out (FIFO)** principle, meaning that the first element added to the queue will be the first one to be removed. It is similar to a real-life queue, such as people standing in line at a ticket counter, where the first person in line gets served first.

Java provides the Queue interface and several classes that implement it, such as LinkedList, PriorityQueue, and ArrayDeque. The Queue interface is part of the java.util package.

Key Concepts of Queue:

1. **FIFO Principle**:
    - The **first element added** to the queue is the **first element to be removed**.
    - It behaves like a line at a ticket counter or a customer service desk.
2. **Queue Operations**:
    - **offer(E e)**: Adds the element e to the queue. If the element is added successfully, it returns true; otherwise, it returns false (for bounded queues).
    - **poll()**: Removes and returns the head (first) element of the queue. If the queue is empty, it returns null.
    - **peek()**: Returns (but does not remove) the head of the queue. It returns null if the queue is empty.
    - **add(E e)**: Adds the element e to the queue. It throws an IllegalStateException if the queue cannot accommodate more elements (in the case of a bounded queue).
    - **remove()**: Removes and returns the head of the queue. It throws NoSuchElementException if the queue is empty.