1.

```
public ListNode reverseList(ListNode head) {
    if (head == null || head.next == null) return head;
    ListNode p = reverseList(head.next);
    head.next.next = head;
    head.next = null;
    return p;
}


public ListNode reverseList(ListNode head) {
    ListNode prev = null;
    ListNode curr = head;
    while (curr != null) {
        ListNode nextTemp = curr.next;
        curr.next = prev;
        prev = curr;
        curr = nextTemp;
    }
    return prev;
}

public void deleteNode(ListNode node) {
    node.val=node.next.val;
    node.next=node.next.next;
}

Public ListNode findNthNode(ListNode head, int n) {
          ListNode result = head;
          While (n- 1 > 0){
                    result = result.next;
          }
          Return result;

}


public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode first = dummy;
    ListNode second = dummy;
    // Advances first pointer so that the gap between first and second is n nodes apart
```

```
    for (int i = 1; i <= n + 1; i++) {
        first = first.next;
    }
    // Move first to the end, maintaining the gap
    while (first != null) {
        first = first.next;
        second = second.next;
    }
    second.next = second.next.next;
    return dummy.next;
}


public boolean hasCycle(ListNode head) {
        if (head == null) {
            return false;
        }

        ListNode slow = head;
        ListNode fast = head.next;
        while (slow != fast) {
            if (fast == null || fast.next == null) {
                return false;
            }
            slow = slow.next;
            fast = fast.next.next;
        }
        return true;
}
```

2.

```
final class Student{


    private final String name;
    private final int id;
    private final Map<String, String> metadata;

    // Constructor of immutable class
    // Parameterized constructor
    public Student(String name, int ID,
                   Map<String, String> metadata)
    {

        this.name = name;
        this.id = id;

        Map<String, String> tempMap = new HashMap<>();

        for (Map.Entry<String, String> entry :
             metadata.entrySet()) {
            tempMap.put(entry.getKey(), entry.getValue());
        }

        this.metadata = tempMap;
    }

    public String getName() { return name; }

    public int getid() { return id; }

    public Map<String, String> getMetadata()
    {

        // Creating Map with HashMap reference
        Map<String, String> tempMap = new HashMap<>();

        for (Map.Entry<String, String> entry :
             this.metadata.entrySet()) {
            tempMap.put(entry.getKey(), entry.getValue());
        }
        return tempMap;
    }
}
```

3.

```
public final class ClassSingleton {

    private static ClassSingleton INSTANCE;
    private String info = "Initial info class";

    private ClassSingleton() {
    }

    public static ClassSingleton getInstance() {
        if(INSTANCE == null) {
            INSTANCE = new ClassSingleton();
        }

        return INSTANCE;
    }

    // getters and setters
}

public enum EnumSingleton {

    INSTANCE("Initial class info");

    private String info;

    private EnumSingleton(String info) {
        this.info = info;
    }

    public EnumSingleton getInstance() {
        return INSTANCE;
    }

    // getters and setters
}
```

4.

5.

```
public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) {
            return result;
        }
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            int size = queue.size();
            List<Integer> level = new ArrayList<>();
            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                level.add(node.val);
                if (node.left != null) {
                    queue.offer(node.left);
                }
                if (node.right != null) {
                    queue.offer(node.right);
                }
            }
            result.add(level);
        }
        return result;
    }
```

```
public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        levelHelper(res, root, 0);
        return res;
    }

    public void levelHelper(List<List<Integer>> res, TreeNode root, int height) {
        if (root == null) return;
        if (height >= res.size()) {
            res.add(new LinkedList<Integer>());
        }
        res.get(height).add(root.val);
```

```
    levelHelper(res, root.left, height+1);
    levelHelper(res, root.right, height+1);
}
```