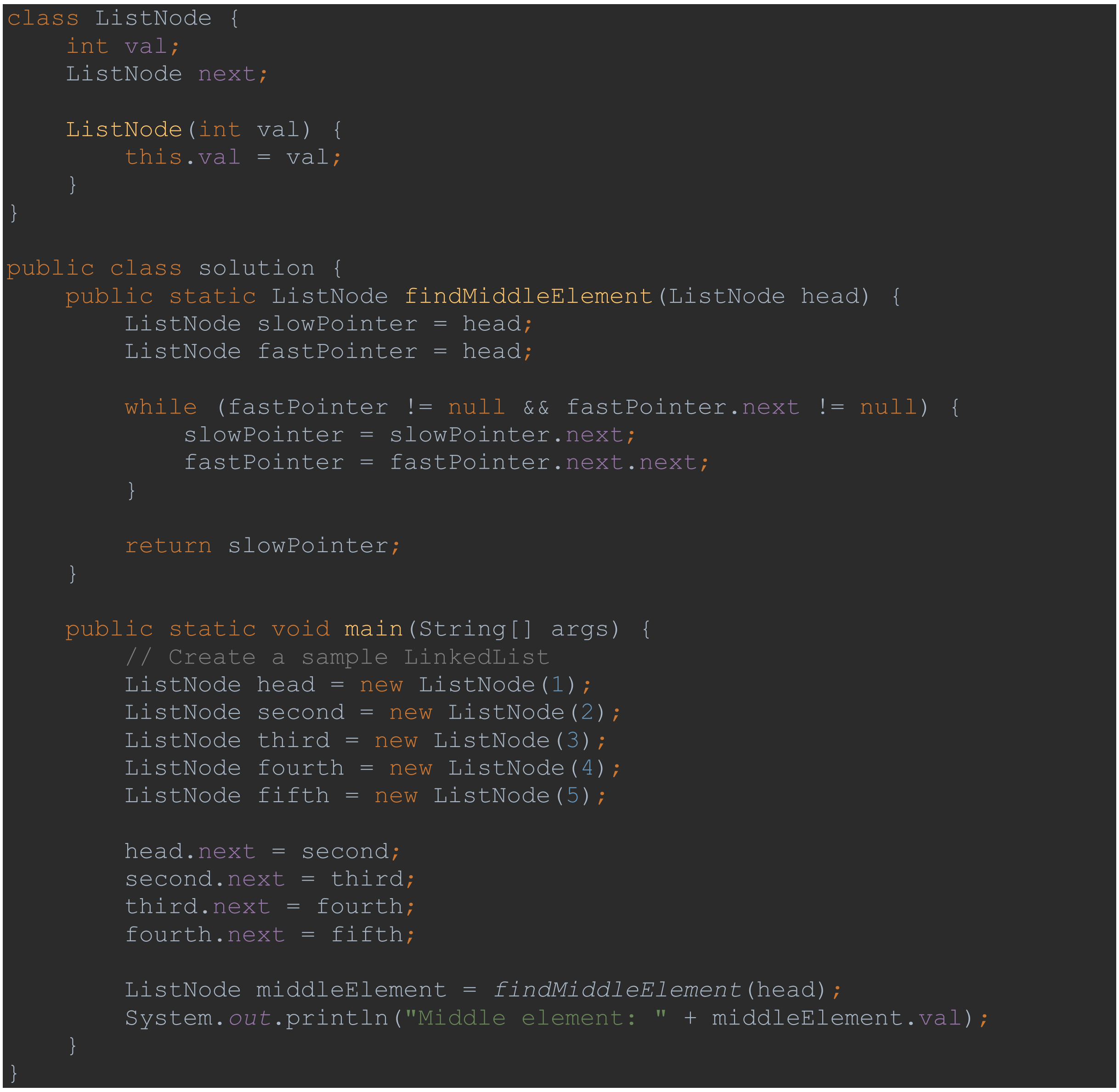
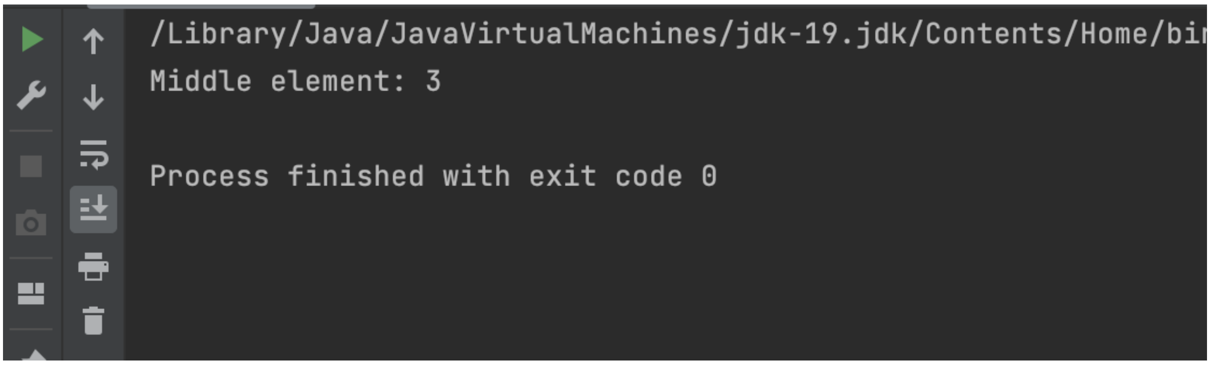
1. Find Middle Element in LinkedList.



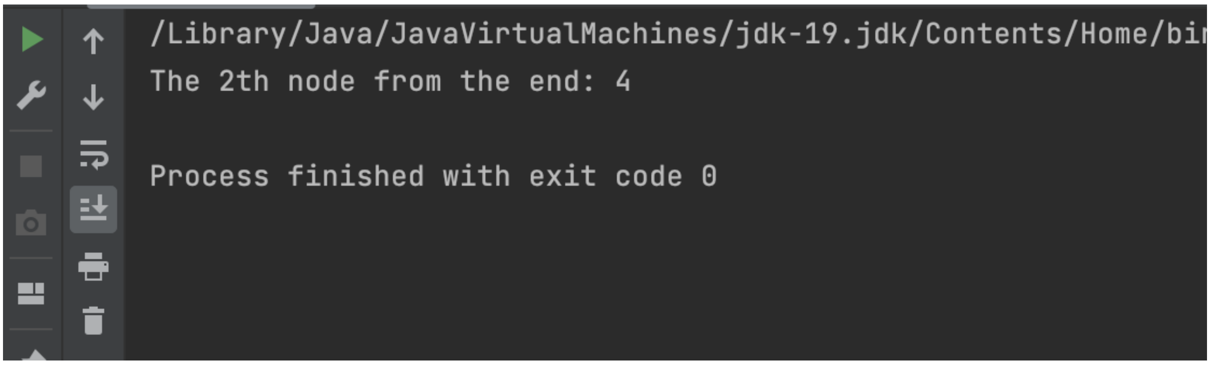
```java
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
    }
}

public class solution {
    public static ListNode findMiddleElement(ListNode head) {
        ListNode slowPointer = head;
        ListNode fastPointer = head;

        while (fastPointer != null && fastPointer.next != null) {
            slowPointer = slowPointer.next;
            fastPointer = fastPointer.next.next;
        }

        return slowPointer;
    }

    public static void main(String[] args) {
        // Create a sample LinkedList
        ListNode head = new ListNode(1);
        ListNode second = new ListNode(2);
        ListNode third = new ListNode(3);
        ListNode fourth = new ListNode(4);
        ListNode fifth = new ListNode(5);

        head.next = second;
        second.next = third;
        third.next = fourth;
        fourth.next = fifth;

        ListNode middleElement = findMiddleElement(head);
        System.out.println("Middle element: " + middleElement.val);
    }
}
```



```
/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bi
Middle element: 3

Process finished with exit code 0
```

2. Find the nth Node from the End of the Singly LinkedList

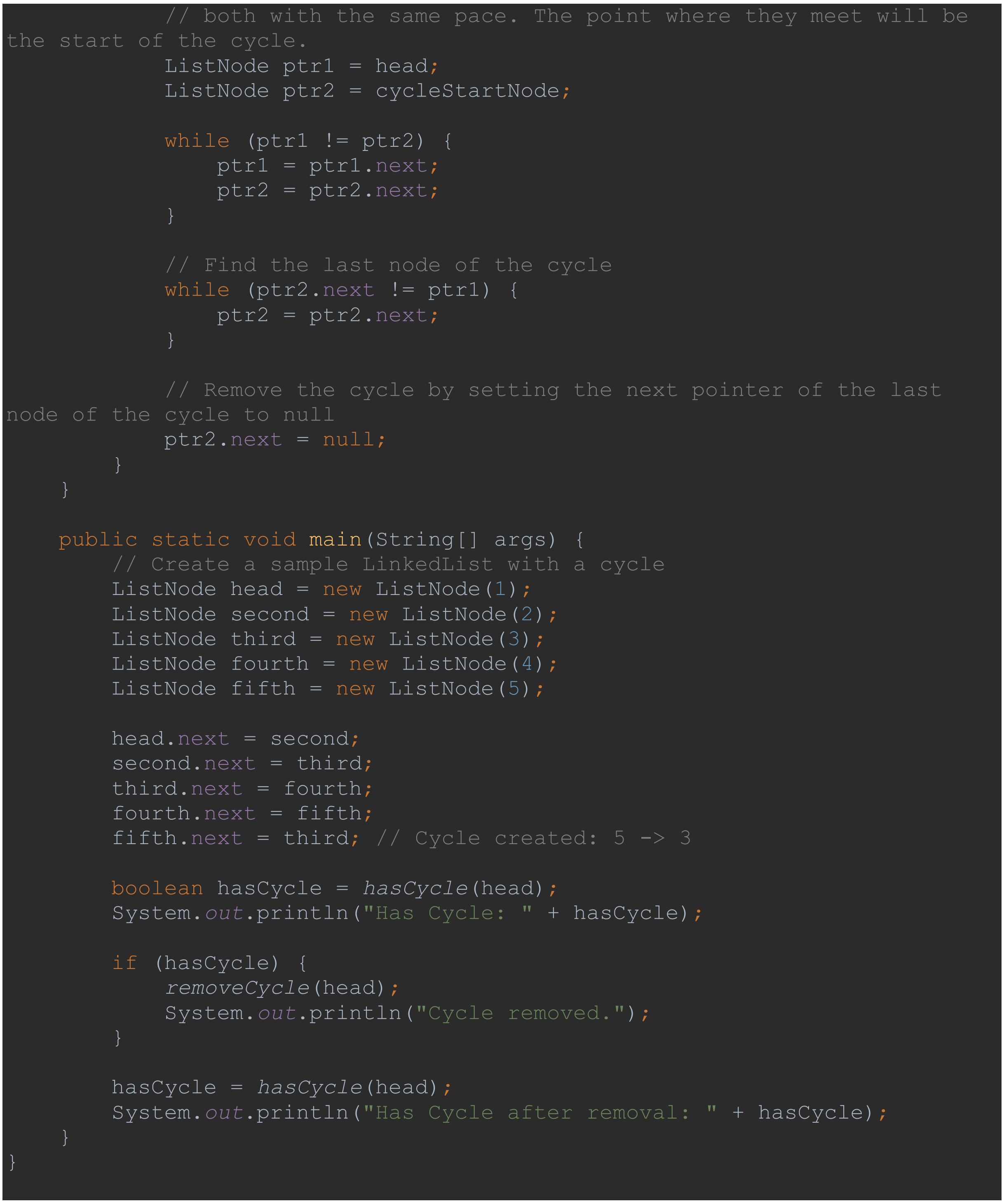
```
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
    }
}

public class solution {
    public static ListNode findNthNodeFromEnd(ListNode head, int n) {
        ListNode fastPointer = head;
        ListNode slowPointer = head;

        // Move the fast pointer 'n' nodes ahead
        for (int i = 0; i < n; i++) {
            if (fastPointer == null) {
                throw new IllegalArgumentException("The LinkedList
does not have " + n + " nodes.");
            }
            fastPointer = fastPointer.next;
        }

        // Move both pointers simultaneously until the fast pointer
reaches the end
        while (fastPointer != null) {
            fastPointer = fastPointer.next;
            slowPointer = slowPointer.next;
        }

        return slowPointer;
    }

    public static void main(String[] args) {
        // Create a sample LinkedList
        ListNode head = new ListNode(1);
        ListNode second = new ListNode(2);
        ListNode third = new ListNode(3);
        ListNode fourth = new ListNode(4);
        ListNode fifth = new ListNode(5);

        head.next = second;
        second.next = third;
        third.next = fourth;
        fourth.next = fifth;

        int n = 2; // Find the 2nd node from the end

        try {
            ListNode nthNodeFromEnd = findNthNodeFromEnd(head, n);
            System.out.println("The " + n + "th node from the end: "
+ nthNodeFromEnd.val);
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
    }
}
```



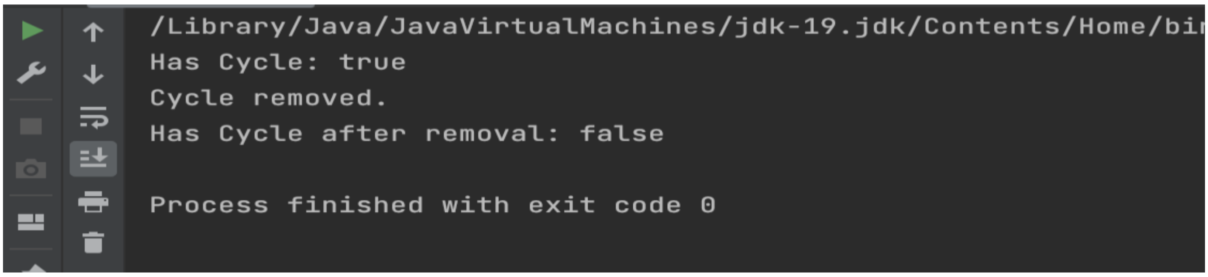
```
/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bi
The 2th node from the end: 4

Process finished with exit code 0
```

3. Detect Cycle and Remove Cycle in LinkedList

```
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
    }
}
public class solution {
    public static boolean hasCycle(ListNode head) {
        ListNode tortoise = head;
        ListNode hare = head;

        while (hare != null && hare.next != null) {
            tortoise = tortoise.next;
            hare = hare.next.next;

            if (tortoise == hare) {
                return true; // Cycle detected
            }
        }
        return false; // No cycle detected
    }
public static void removeCycle(ListNode head) {
        ListNode tortoise = head;
        ListNode hare = head;
        ListNode cycleStartNode = null;

        // Find the meeting point of the tortoise and hare (if cycle
exists)
        while (hare != null && hare.next != null) {
            tortoise = tortoise.next;
            hare = hare.next.next;

            if (tortoise == hare) {
                cycleStartNode = tortoise;
                break;
            }
        }

        if (cycleStartNode != null) {
            // Move one pointer to the head and another pointer from the
cycle start node,
```



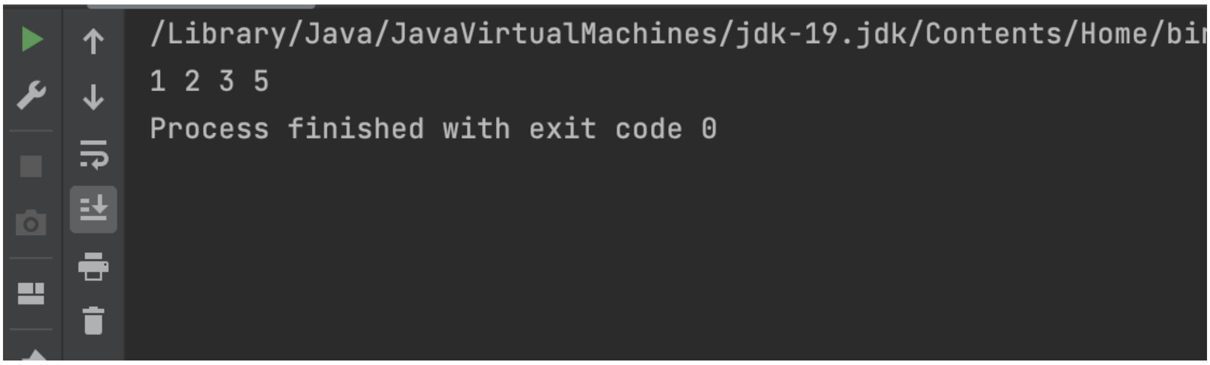
```
            // both with the same pace. The point where they meet will be
the start of the cycle.
            ListNode ptr1 = head;
            ListNode ptr2 = cycleStartNode;

            while (ptr1 != ptr2) {
                ptr1 = ptr1.next;
                ptr2 = ptr2.next;
            }

            // Find the last node of the cycle
            while (ptr2.next != ptr1) {
                ptr2 = ptr2.next;
            }

            // Remove the cycle by setting the next pointer of the last
node of the cycle to null
            ptr2.next = null;
        }
    }

    public static void main(String[] args) {
        // Create a sample LinkedList with a cycle
        ListNode head = new ListNode(1);
        ListNode second = new ListNode(2);
        ListNode third = new ListNode(3);
        ListNode fourth = new ListNode(4);
        ListNode fifth = new ListNode(5);

        head.next = second;
        second.next = third;
        third.next = fourth;
        fourth.next = fifth;
        fifth.next = third; // Cycle created: 5 -> 3

        boolean hasCycle = hasCycle(head);
        System.out.println("Has Cycle: " + hasCycle);

        if (hasCycle) {
            removeCycle(head);
            System.out.println("Cycle removed.");
        }

        hasCycle = hasCycle(head);
        System.out.println("Has Cycle after removal: " + hasCycle);
    }
}
```



```
/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bi
Has Cycle: true
Cycle removed.
Has Cycle after removal: false

Process finished with exit code 0
```

4. Remove the Nth node from the End of the Singly LinkedList

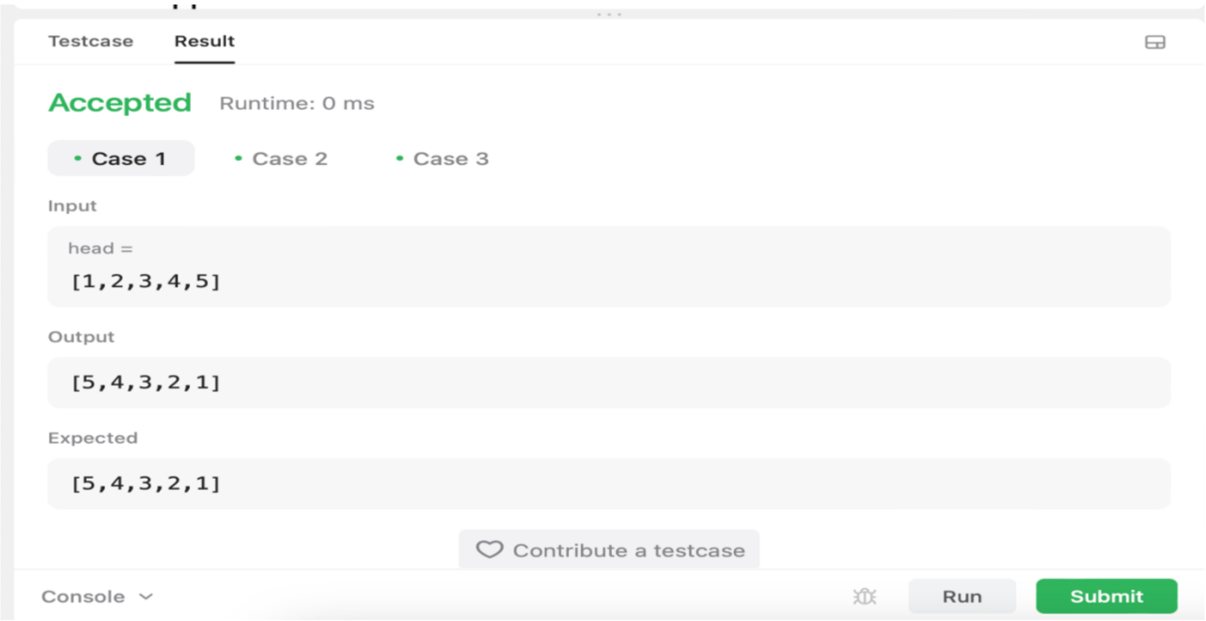
```
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
    }
}

public class solution {
    public static ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;

        ListNode slow = dummy;
        ListNode fast = dummy;

        // Move the fast pointer 'n+1' nodes ahead
        for (int i = 0; i <= n; i++) {
            fast = fast.next;
        }

                while (fast != null) {
            slow = slow.next;
            fast = fast.next;
        }

                slow.next = slow.next.next;

        return dummy.next;
    }

    public static void main(String[] args) {
        // Create a sample LinkedList
        ListNode head = new ListNode(1);
        ListNode second = new ListNode(2);
        ListNode third = new ListNode(3);
        ListNode fourth = new ListNode(4);
        ListNode fifth = new ListNode(5);

        head.next = second;
        second.next = third;
        third.next = fourth;
        fourth.next = fifth;

        int n = 2; // Remove the 2nd node from the end

        ListNode newHead = removeNthFromEnd(head, n);

        // Print the updated LinkedList
        ListNode current = newHead;
        while (current != null) {
            System.out.print(current.val + " ");
            current = current.next;
        }
    }
}
```



```
/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bi
1 2 3 5
Process finished with exit code 0
```

5. Reverse (https://leetcode.com/problems/reverse-linked-list/)

```
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode current = head;

        while (current != null) {
            ListNode nextNode = current.next;
            current.next = prev;
            prev = current;
            current = nextNode;
        }

        return prev;
    }
}
```



Accepted Runtime: 0 ms

Input

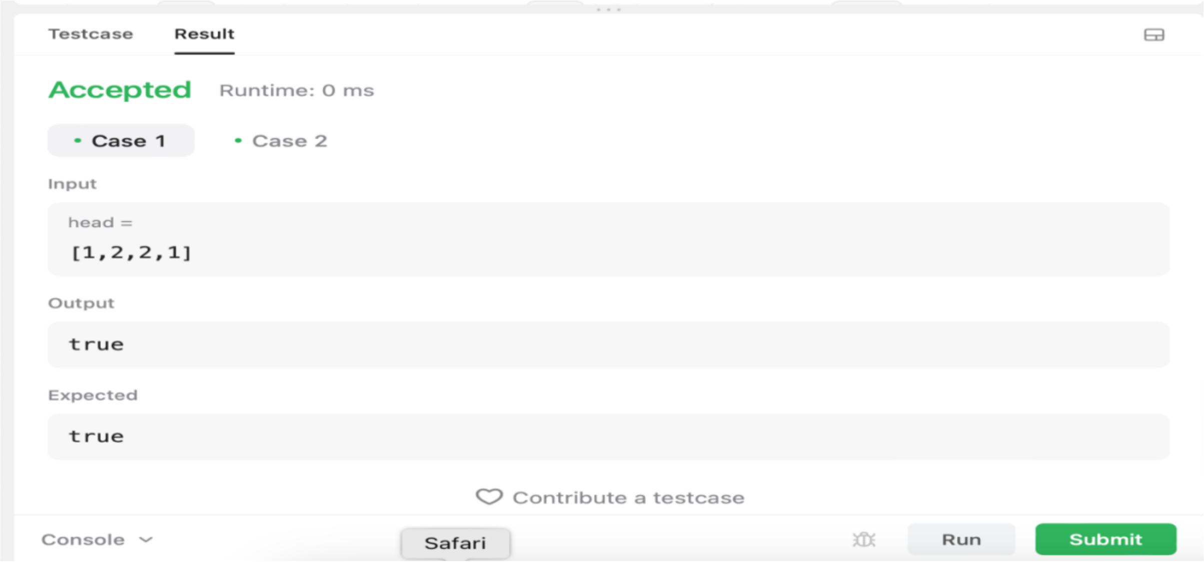
head =
```
[1,2,3,4,5]
```

Output
```
[5,4,3,2,1]
```

Expected
```
[5,4,3,2,1]
```

6. Palindrome in LinkedList (https://leetcode.com/problems/palindrome-linked-list/)

```
class Solution {
    public boolean isPalindrome(ListNode head) {
        List<Integer> values = new ArrayList<>();

        ListNode current = head;
        while (current != null) {
            values.add(current.val);
            current = current.next;
        }

        int start = 0;
        int end = values.size() - 1;
        while (start < end) {
            if (!values.get(start).equals(values.get(end))) {
                return false;
            }
            start++;
            end--;
        }

        return true;
    }
}
```



Accepted Runtime: 0 ms

Input

```
head =
[1,2,2,1]
```

Output

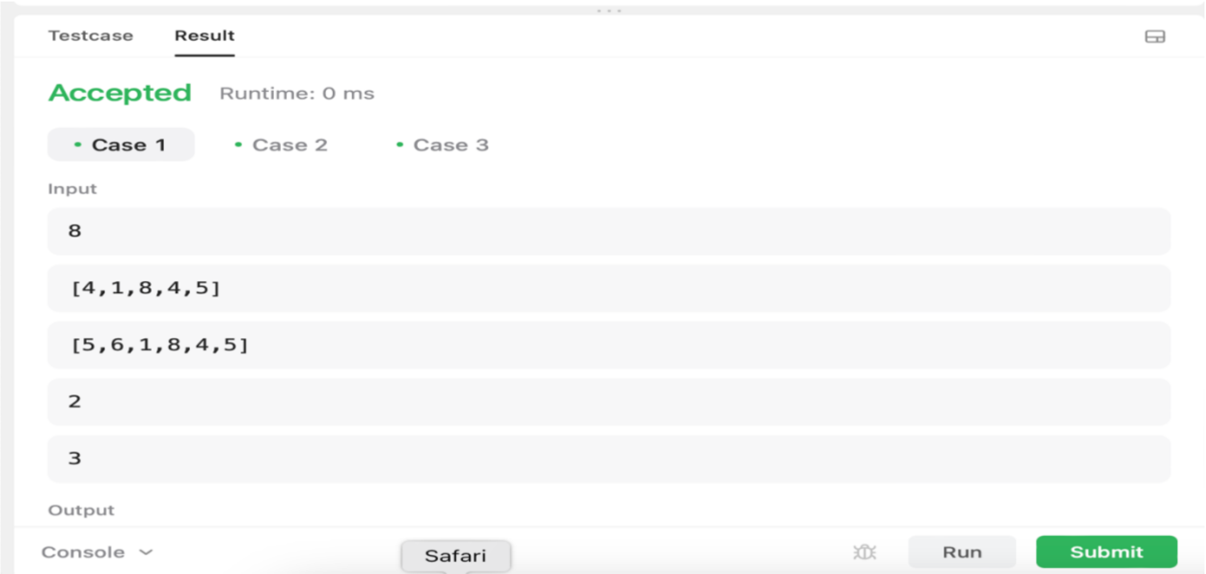
```
true
```

Expected

```
true
```

7. Intersection of 2 LinkedList (https://leetcode.com/problems/intersection-of-two-linked-lists/)

```
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        ListNode pointerA = headA;
        ListNode pointerB = headB;

        while (pointerA != pointerB) {
            // Move pointerA to the next node in listA
            if (pointerA == null)
                pointerA = headB;
            else
                pointerA = pointerA.next;
            // Move pointerB to the next node in listB
            if (pointerB == null)
                pointerB = headA;
            else
                pointerB = pointerB.next;
        }

        return pointerA; // Return the intersecting node or null
    }
}
```



Testcase Result

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

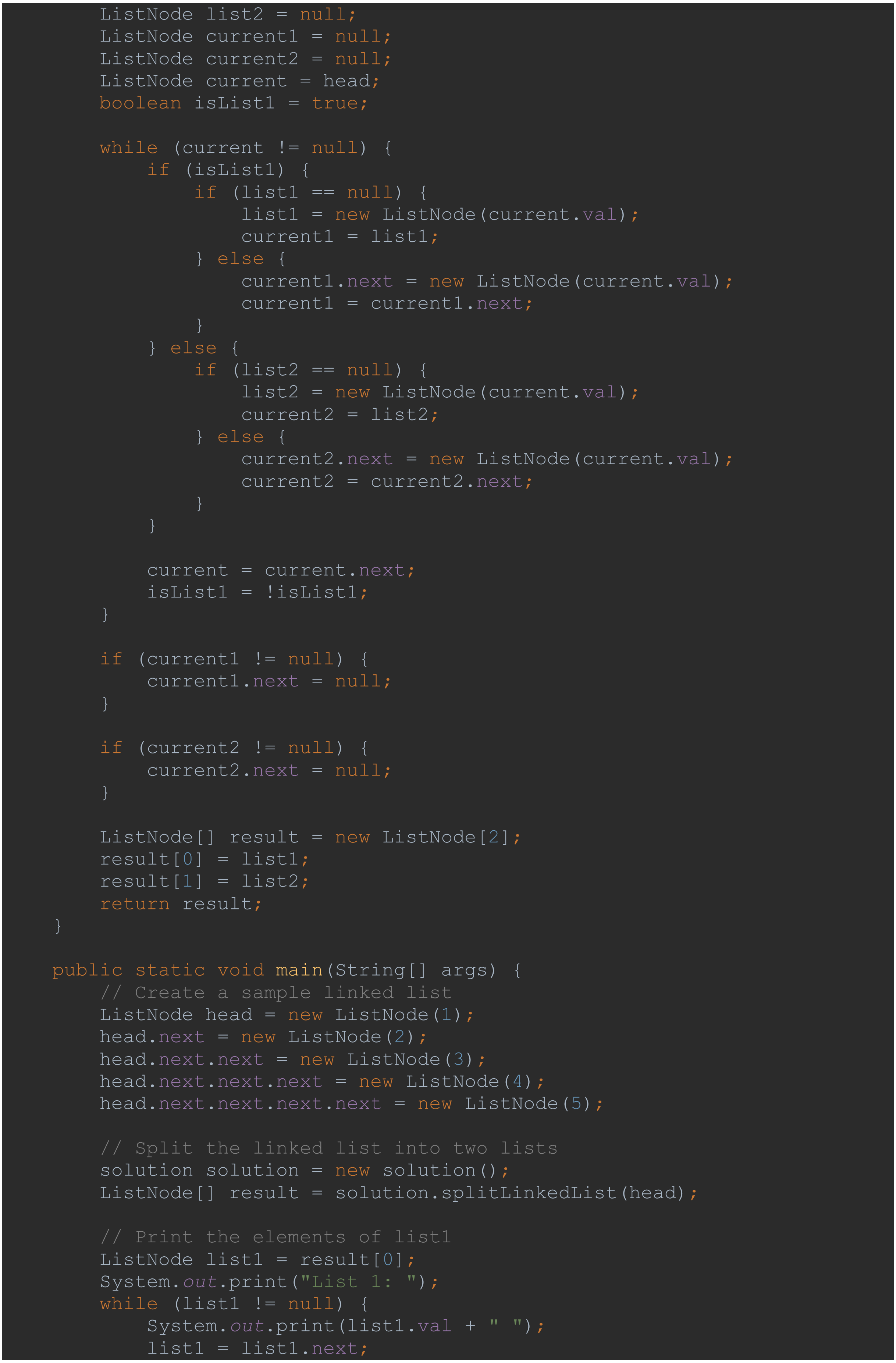
Input

```
8
[4,1,8,4,5]
[5,6,1,8,4,5]
2
3
```

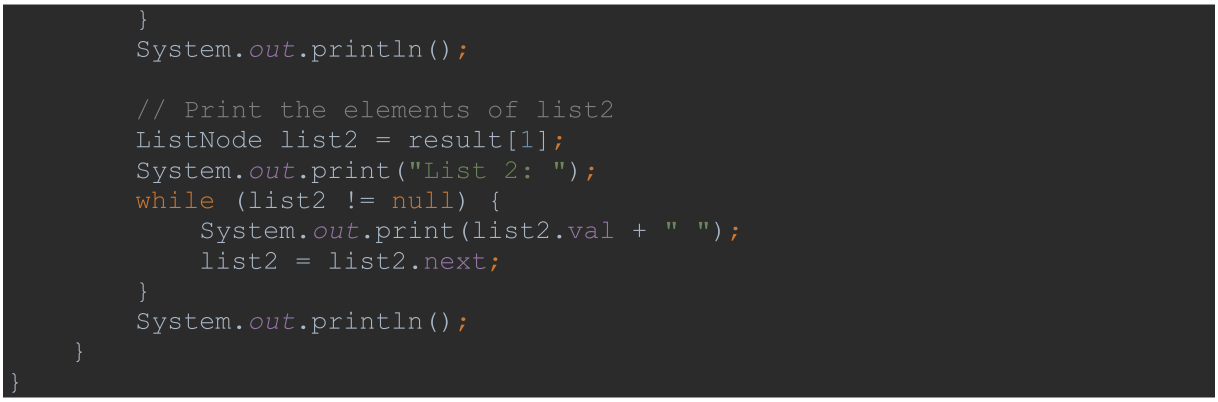
Output

8. Split a LinkedList into 2 Singly LinkedList in Alternative Fashion

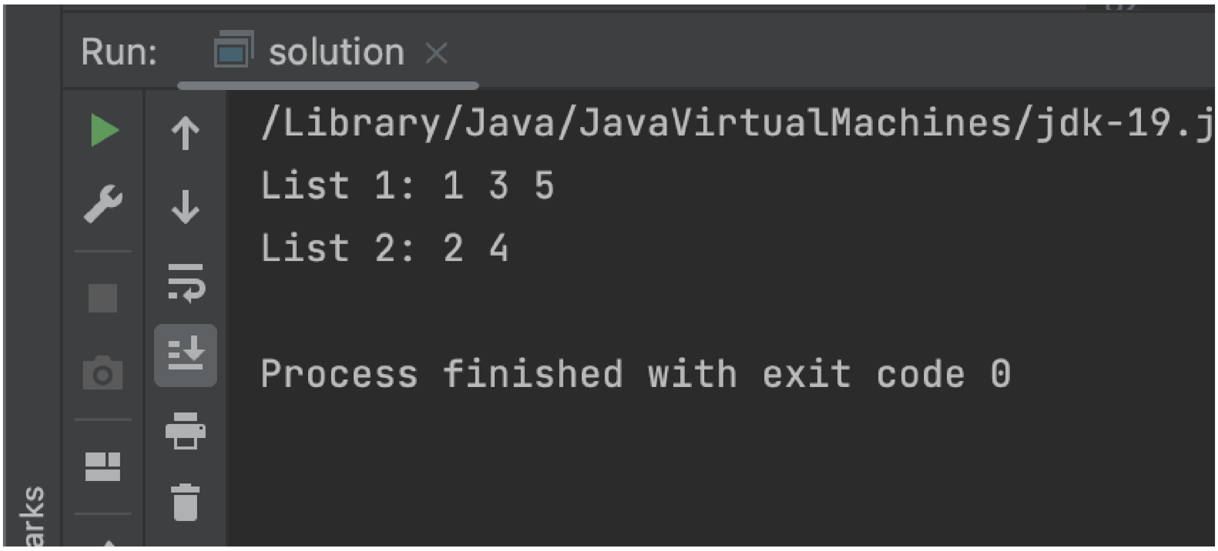
```
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}

public class solution {
    public ListNode[] splitLinkedList(ListNode head) {
        ListNode list1 = null;
```



```
        ListNode list2 = null;
        ListNode current1 = null;
        ListNode current2 = null;
        ListNode current = head;
        boolean isList1 = true;

        while (current != null) {
            if (isList1) {
                if (list1 == null) {
                    list1 = new ListNode(current.val);
                    current1 = list1;
                } else {
                    current1.next = new ListNode(current.val);
                    current1 = current1.next;
                }
            } else {
                if (list2 == null) {
                    list2 = new ListNode(current.val);
                    current2 = list2;
                } else {
                    current2.next = new ListNode(current.val);
                    current2 = current2.next;
                }
            }

            current = current.next;
            isList1 = !isList1;
        }

        if (current1 != null) {
            current1.next = null;
        }

        if (current2 != null) {
            current2.next = null;
        }

        ListNode[] result = new ListNode[2];
        result[0] = list1;
        result[1] = list2;
        return result;
    }

    public static void main(String[] args) {
        // Create a sample linked list
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(4);
        head.next.next.next.next = new ListNode(5);

        // Split the linked list into two lists
        solution solution = new solution();
        ListNode[] result = solution.splitLinkedList(head);

        // Print the elements of list1
        ListNode list1 = result[0];
        System.out.print("List 1: ");
        while (list1 != null) {
            System.out.print(list1.val + " ");
            list1 = list1.next;
```



```
        }
        System.out.println();

        // Print the elements of list2
        ListNode list2 = result[1];
        System.out.print("List 2: ");
        while (list2 != null) {
            System.out.print(list2.val + " ");
            list2 = list2.next;
        }
        System.out.println();
    }
}
```

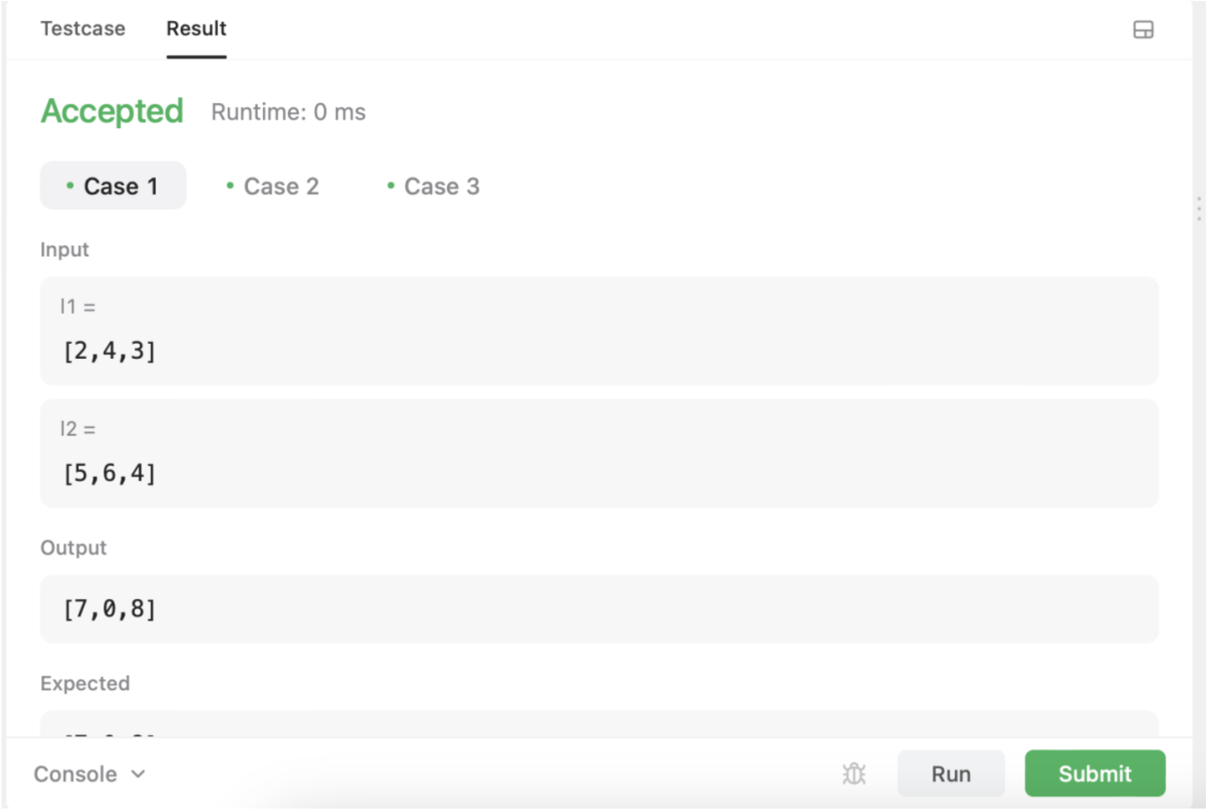


```
/Library/Java/JavaVirtualMachines/jdk-19.j
List 1: 1 3 5
List 2: 2 4

Process finished with exit code 0
```

9. Add 2 Big Numbers Using LinkedList (https://leetcode.com/problems/add-two-numbers/)

```
public class Solution {
  public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode dummyHead = new ListNode(0);
    ListNode p = l1;
    ListNode q = l2;
    ListNode current = dummyHead;
    int carry = 0;
    while (p != null || q != null) {
        int x = (p != null) ? p.val : 0;
        int y = (q != null) ? q.val : 0;
        int sum = carry + x + y;
        carry = sum / 10;
        current.next = new ListNode(sum % 10);
        current = current.next;

        if (p != null)
            p = p.next;
        if (q != null)
            q = q.next;
    }

    if (carry > 0) {
        current.next = new ListNode(carry);
    }

    return dummyHead.next;
}
```

```
}
```



Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

l1 =

```
[2,4,3]
```

l2 =

```
[5,6,4]
```

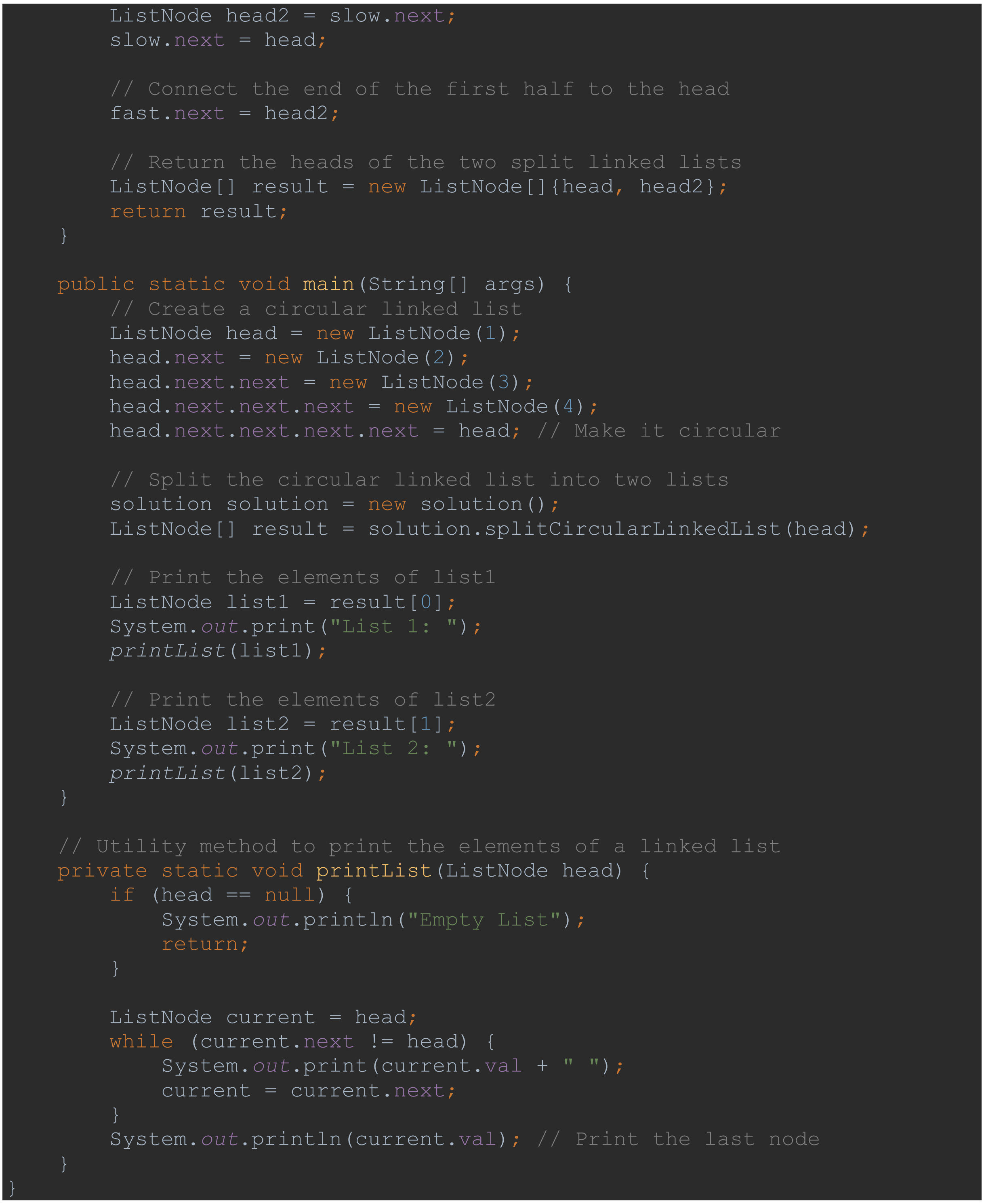
Output

```
[7,0,8]
```

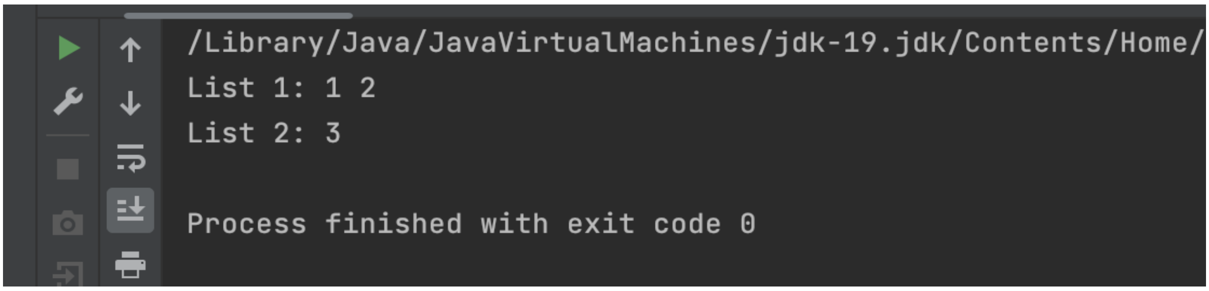
Expected

10. Split Circular LinkedList

```
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}

public class solution {
    public ListNode[] splitCircularLinkedList(ListNode head) {
        if (head == null || head.next == null) {
            return new ListNode[]{head, null};
        }

        ListNode slow = head;
        ListNode fast = head;

        // Find the midpoint of the circular linked list
        while (fast.next != head && fast.next.next != head) {
            slow = slow.next;
            fast = fast.next.next;
        }

        // Set the next pointer of the second half to the head
```



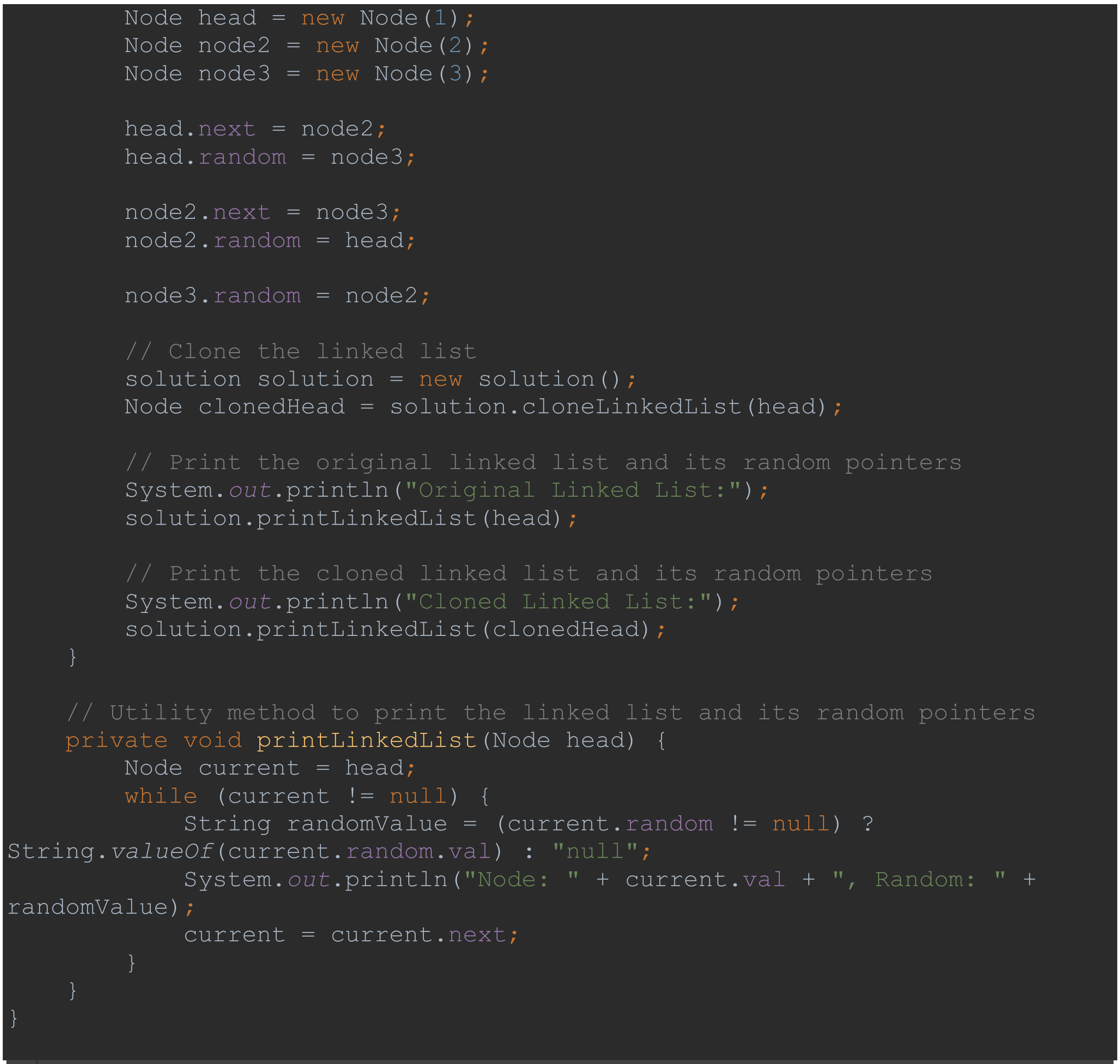
```java
        ListNode head2 = slow.next;
        slow.next = head;

        // Connect the end of the first half to the head
        fast.next = head2;

        // Return the heads of the two split linked lists
        ListNode[] result = new ListNode[]{head, head2};
        return result;
    }

    public static void main(String[] args) {
        // Create a circular linked list
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(4);
        head.next.next.next.next = head; // Make it circular

        // Split the circular linked list into two lists
        solution solution = new solution();
        ListNode[] result = solution.splitCircularLinkedList(head);

        // Print the elements of list1
        ListNode list1 = result[0];
        System.out.print("List 1: ");
        printList(list1);

        // Print the elements of list2
        ListNode list2 = result[1];
        System.out.print("List 2: ");
        printList(list2);
    }

    // Utility method to print the elements of a linked list
    private static void printList(ListNode head) {
        if (head == null) {
            System.out.println("Empty List");
            return;
        }

        ListNode current = head;
        while (current.next != head) {
            System.out.print(current.val + " ");
            current = current.next;
        }
        System.out.println(current.val); // Print the last node
    }
}
```



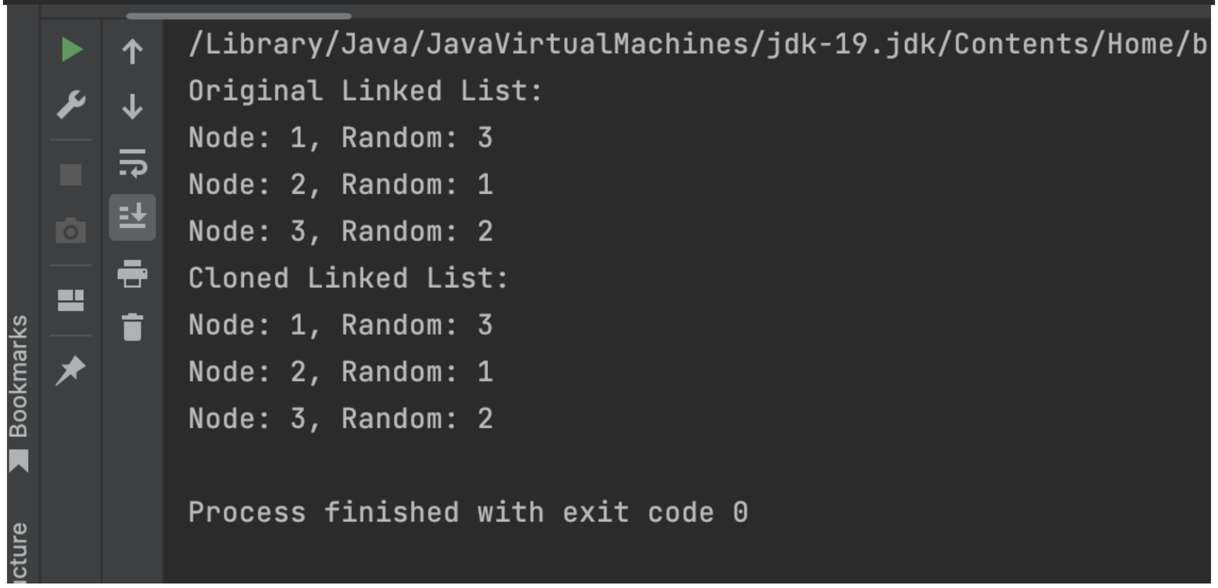
```
/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/
List 1: 1 2
List 2: 3

Process finished with exit code 0
```

11. Clone a LinkedList

```java
import java.util.HashMap;
import java.util.Map;

class Node {
    int val;
    Node next;
    Node random;

    Node(int val) {
        this.val = val;
        this.next = null;
        this.random = null;
    }
}

public class solution {
    public Node cloneLinkedList(Node head) {
        if (head == null) {
            return null;
        }

        Map<Node, Node> nodeMap = new HashMap<>();
        Node current = head;

        // Create new nodes and store the mapping
        while (current != null) {
            Node newNode = new Node(current.val);
            nodeMap.put(current, newNode);
            current = current.next;
        }

        // Set the next and random pointers of the new nodes
        current = head;
        while (current != null) {
            Node clonedNode = nodeMap.get(current);
            clonedNode.next = nodeMap.get(current.next);
            clonedNode.random = nodeMap.get(current.random);
            current = current.next;
        }

        return nodeMap.get(head);
    }

    public static void main(String[] args) {
        // Create a sample linked list with random pointers
```



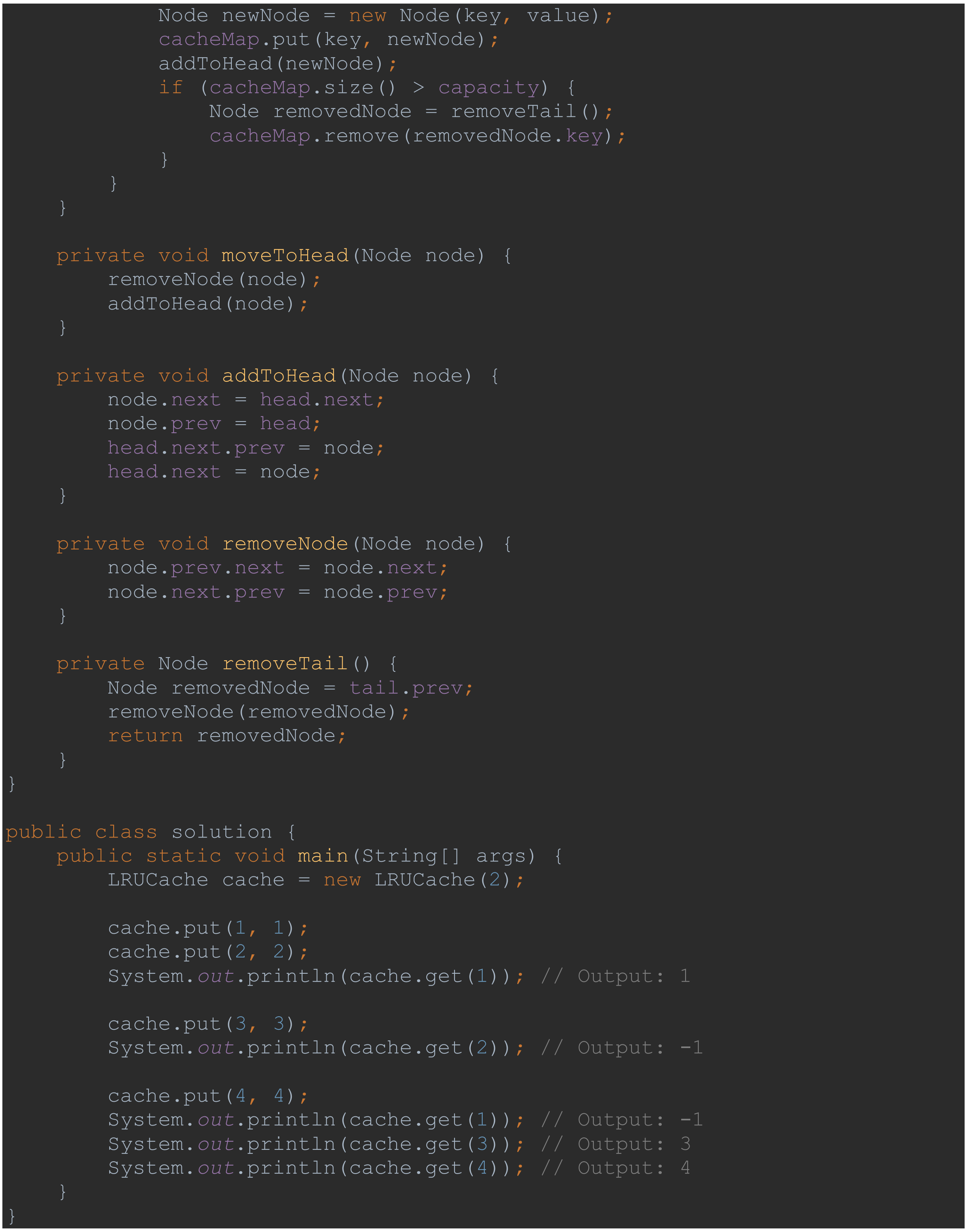
```
        Node head = new Node(1);
        Node node2 = new Node(2);
        Node node3 = new Node(3);

        head.next = node2;
        head.random = node3;

        node2.next = node3;
        node2.random = head;

        node3.random = node2;

        // Clone the linked list
        solution solution = new solution();
        Node clonedHead = solution.cloneLinkedList(head);

        // Print the original linked list and its random pointers
        System.out.println("Original Linked List:");
        solution.printLinkedList(head);

        // Print the cloned linked list and its random pointers
        System.out.println("Cloned Linked List:");
        solution.printLinkedList(clonedHead);
    }

    // Utility method to print the linked list and its random pointers
    private void printLinkedList(Node head) {
        Node current = head;
        while (current != null) {
            String randomValue = (current.random != null) ?
String.valueOf(current.random.val) : "null";
            System.out.println("Node: " + current.val + ", Random: " +
randomValue);
            current = current.next;
        }
    }
}
```



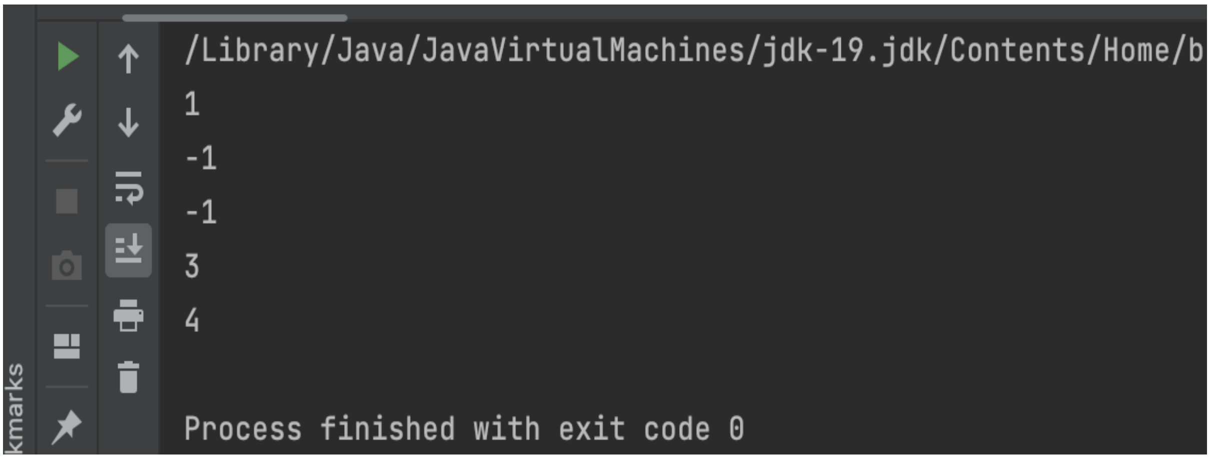
```
/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/b
Original Linked List:
Node: 1, Random: 3
Node: 2, Random: 1
Node: 3, Random: 2
Cloned Linked List:
Node: 1, Random: 3
Node: 2, Random: 1
Node: 3, Random: 2

Process finished with exit code 0
```

12.LRU Cache Implement Using LinkedList

```java
import java.util.HashMap;
import java.util.Map;

class LRUCache {
    class Node {
        int key;
        int value;
        Node prev;
        Node next;

        public Node(int key, int value) {
            this.key = key;
            this.value = value;
        }
    }

    private int capacity;
    private Map<Integer, Node> cacheMap;
    private Node head;
    private Node tail;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.cacheMap = new HashMap<>();
        this.head = new Node(-1, -1);
        this.tail = new Node(-1, -1);
        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        if (cacheMap.containsKey(key)) {
            Node node = cacheMap.get(key);
            moveToHead(node);
            return node.value;
        }
        return -1;
    }

    public void put(int key, int value) {
        if (cacheMap.containsKey(key)) {
            Node node = cacheMap.get(key);
            node.value = value;
            moveToHead(node);
        } else {
```

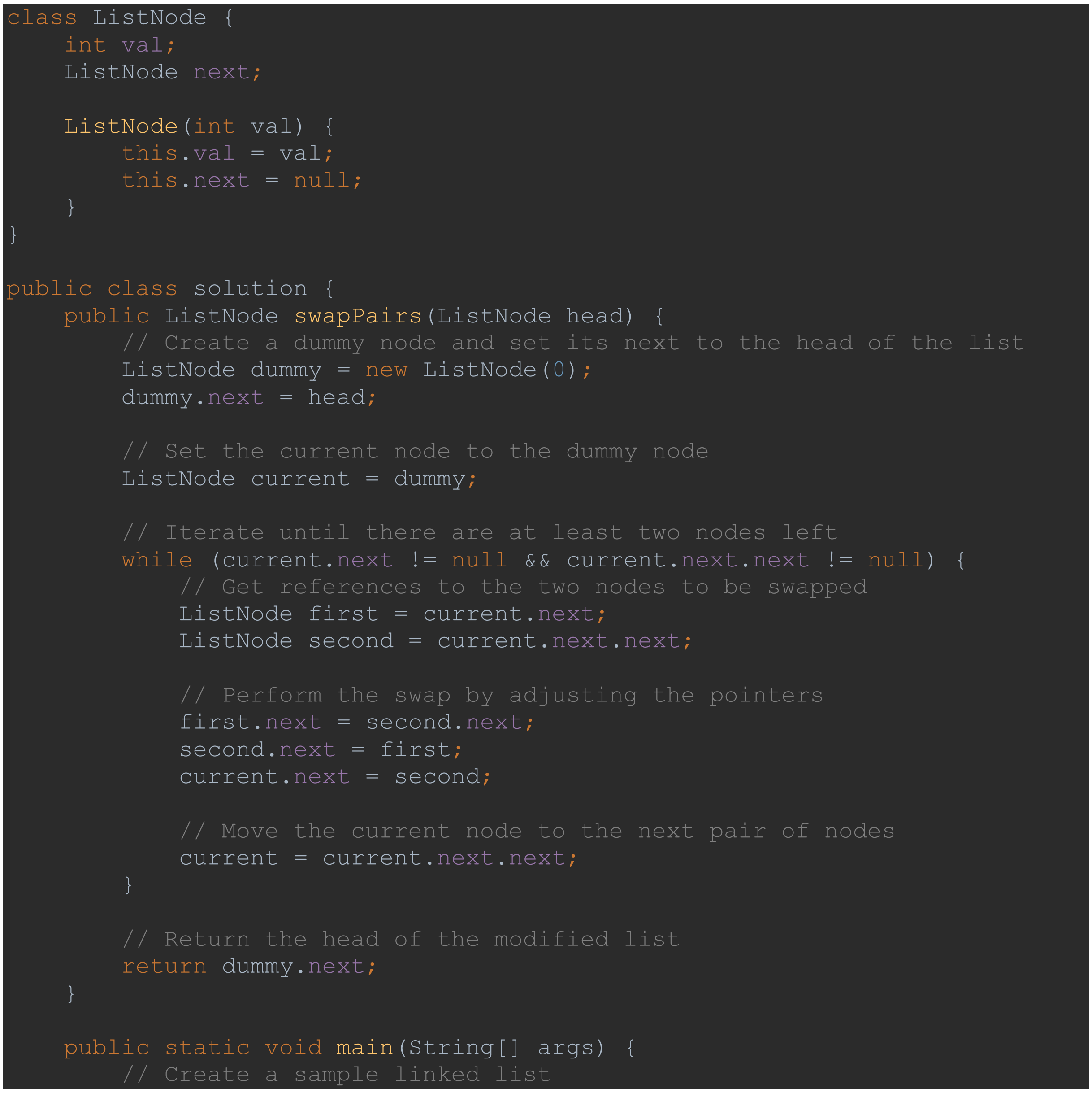


```
            Node newNode = new Node(key, value);
            cacheMap.put(key, newNode);
            addToHead(newNode);
            if (cacheMap.size() > capacity) {
                Node removedNode = removeTail();
                cacheMap.remove(removedNode.key);
            }
        }
    }

    private void moveToHead(Node node) {
        removeNode(node);
        addToHead(node);
    }

    private void addToHead(Node node) {
        node.next = head.next;
        node.prev = head;
        head.next.prev = node;
        head.next = node;
    }

    private void removeNode(Node node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }

    private Node removeTail() {
        Node removedNode = tail.prev;
        removeNode(removedNode);
        return removedNode;
    }
}

public class solution {
    public static void main(String[] args) {
        LRUCache cache = new LRUCache(2);

        cache.put(1, 1);
        cache.put(2, 2);
        System.out.println(cache.get(1)); // Output: 1

        cache.put(3, 3);
        System.out.println(cache.get(2)); // Output: -1

        cache.put(4, 4);
        System.out.println(cache.get(1)); // Output: -1
        System.out.println(cache.get(3)); // Output: 3
        System.out.println(cache.get(4)); // Output: 4
    }
}
```

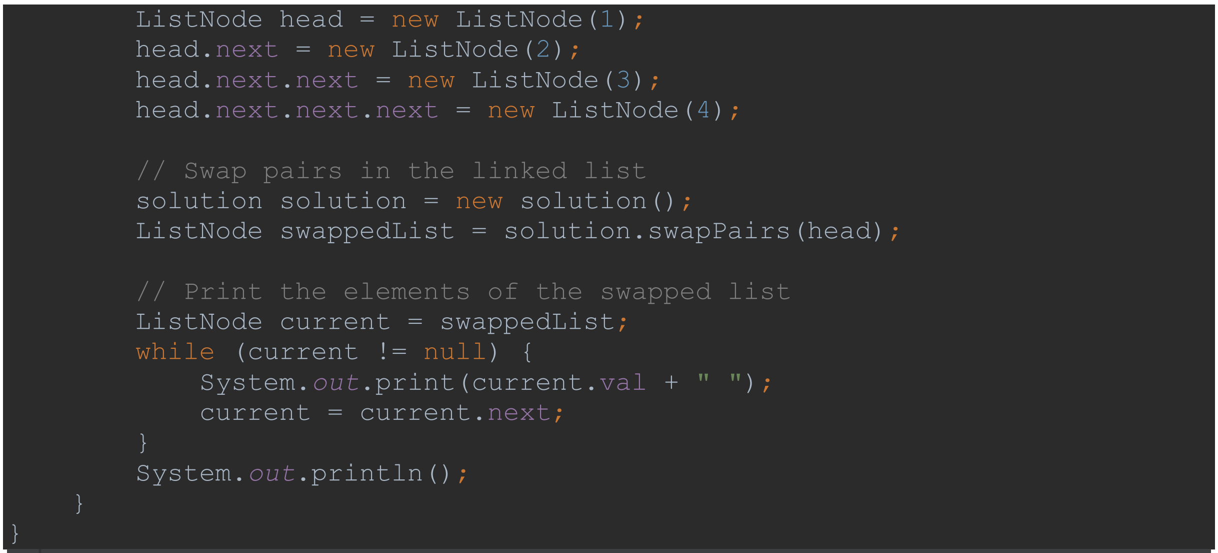


```
/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/b
1
-1
-1
3
4

Process finished with exit code 0
```

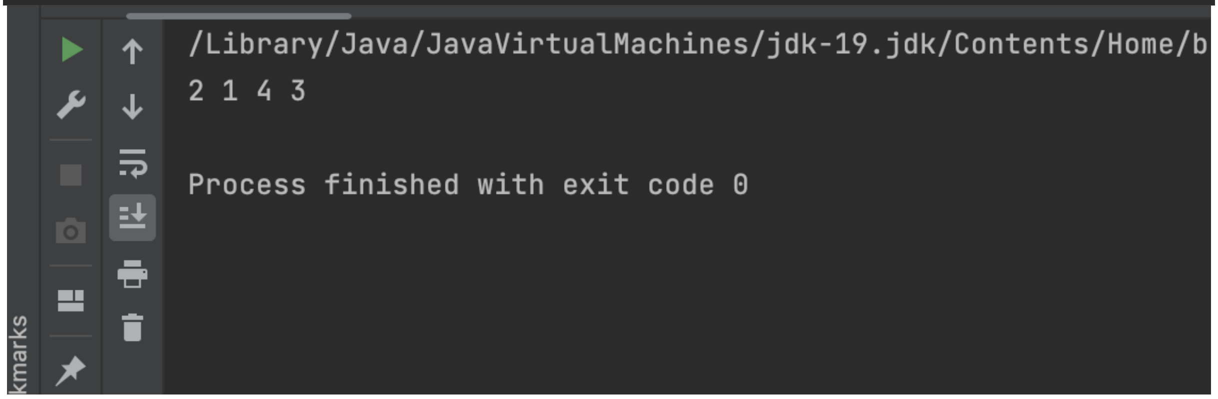
13.Pair Wise Swap in a LinkedList



```
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}

public class solution {
    public ListNode swapPairs(ListNode head) {
        // Create a dummy node and set its next to the head of the list
        ListNode dummy = new ListNode(0);
        dummy.next = head;

        // Set the current node to the dummy node
        ListNode current = dummy;

        // Iterate until there are at least two nodes left
        while (current.next != null && current.next.next != null) {
            // Get references to the two nodes to be swapped
            ListNode first = current.next;
            ListNode second = current.next.next;

            // Perform the swap by adjusting the pointers
            first.next = second.next;
            second.next = first;
            current.next = second;

            // Move the current node to the next pair of nodes
            current = current.next.next;
        }

        // Return the head of the modified list
        return dummy.next;
    }

    public static void main(String[] args) {
        // Create a sample linked list
```

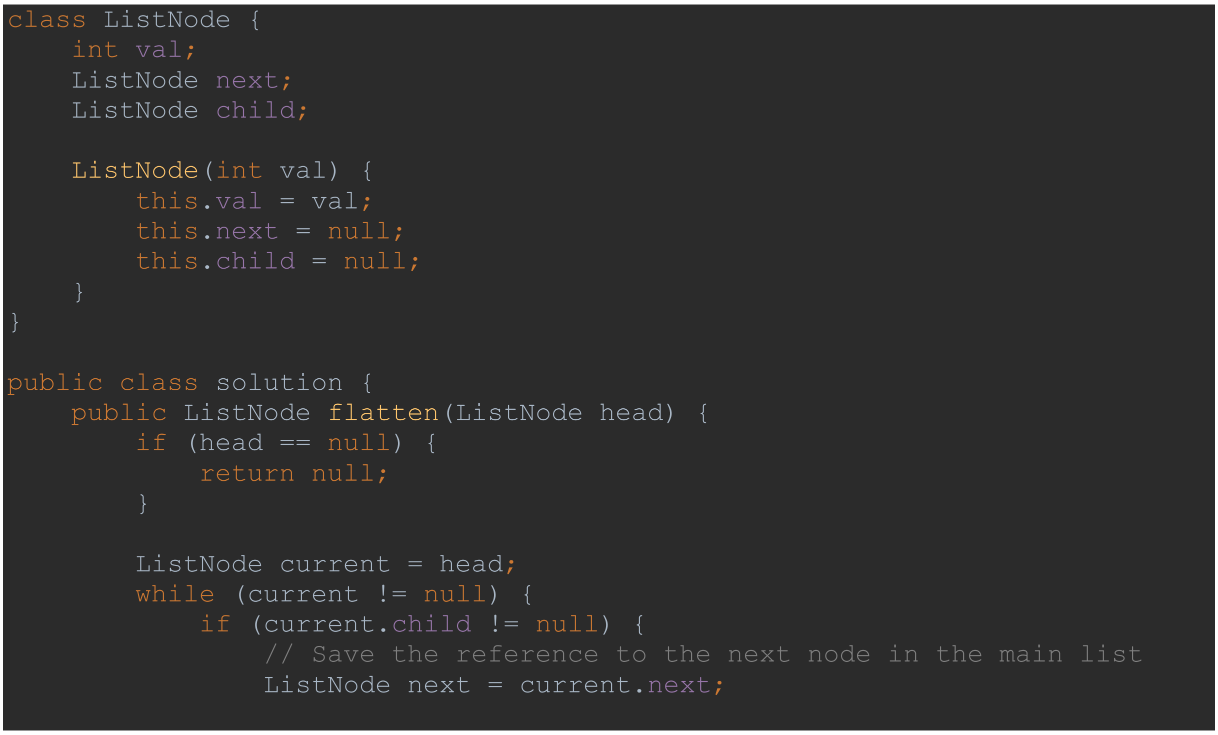


```
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(4);

        // Swap pairs in the linked list
        solution solution = new solution();
        ListNode swappedList = solution.swapPairs(head);

        // Print the elements of the swapped list
        ListNode current = swappedList;
        while (current != null) {
            System.out.print(current.val + " ");
            current = current.next;
        }
        System.out.println();
    }
}
```

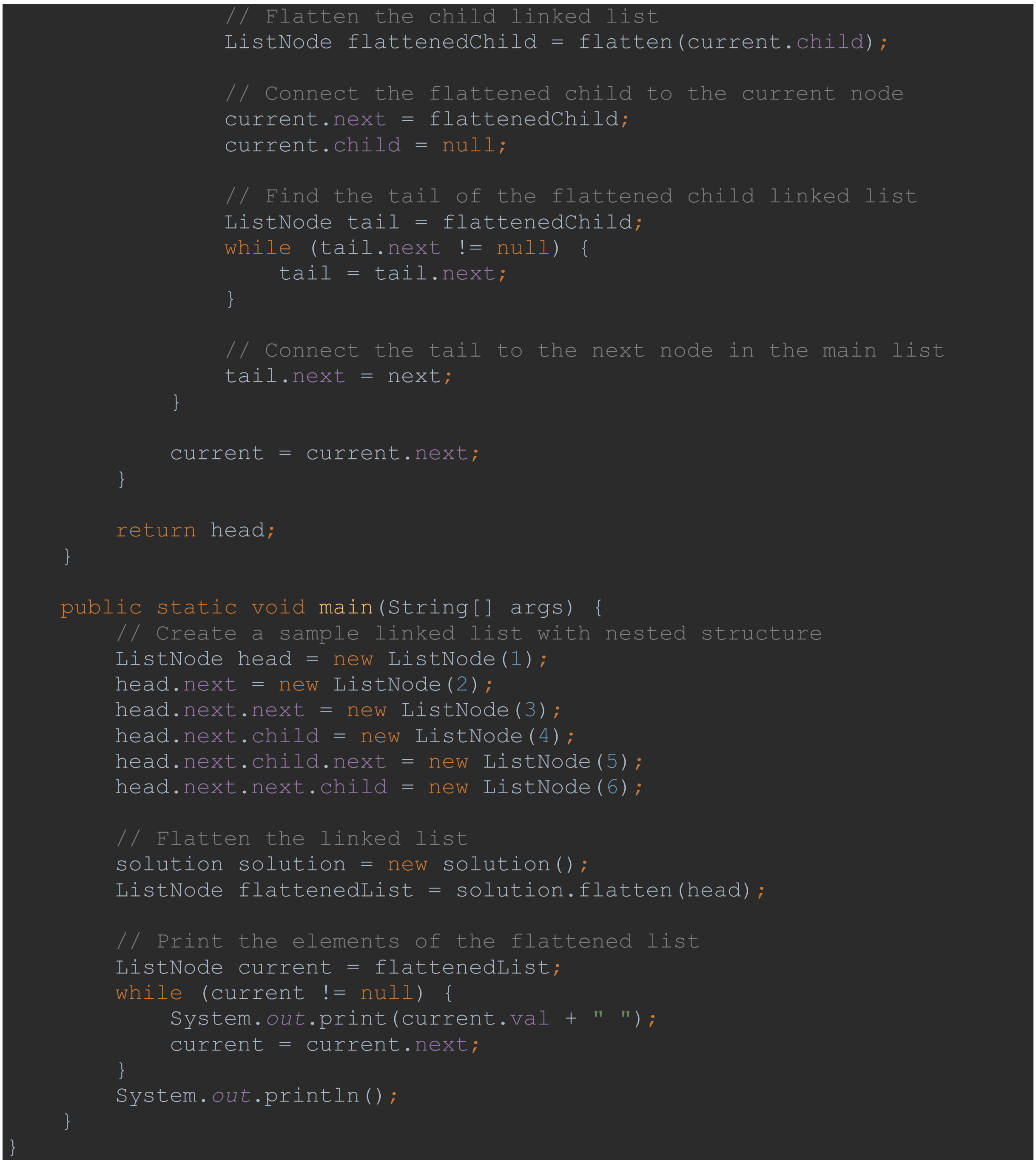


```
/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/b
2 1 4 3

Process finished with exit code 0
```

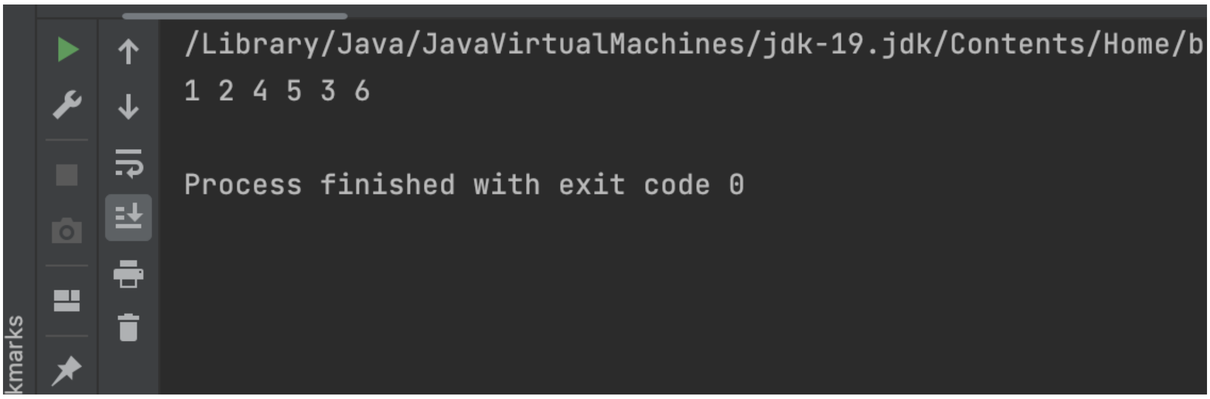
14. Flatting a LinkedList



```
class ListNode {
    int val;
    ListNode next;
    ListNode child;

    ListNode(int val) {
        this.val = val;
        this.next = null;
        this.child = null;
    }
}

public class solution {
    public ListNode flatten(ListNode head) {
        if (head == null) {
            return null;
        }

        ListNode current = head;
        while (current != null) {
            if (current.child != null) {
                // Save the reference to the next node in the main list
                ListNode next = current.next;
```



```java
                // Flatten the child linked list
                ListNode flattenedChild = flatten(current.child);

                // Connect the flattened child to the current node
                current.next = flattenedChild;
                current.child = null;

                // Find the tail of the flattened child linked list
                ListNode tail = flattenedChild;
                while (tail.next != null) {
                    tail = tail.next;
                }

                // Connect the tail to the next node in the main list
                tail.next = next;
            }

            current = current.next;
        }

        return head;
    }

    public static void main(String[] args) {
        // Create a sample linked list with nested structure
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(3);
        head.next.child = new ListNode(4);
        head.next.child.next = new ListNode(5);
        head.next.next.child = new ListNode(6);

        // Flatten the linked list
        solution solution = new solution();
        ListNode flattenedList = solution.flatten(head);

        // Print the elements of the flattened list
        ListNode current = flattenedList;
        while (current != null) {
            System.out.print(current.val + " ");
            current = current.next;
        }
        System.out.println();
    }
}
```



```
/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/b
1 2 4 5 3 6


Process finished with exit code 0
```