**ID: hrajranj**

**Day 13 – 26th June 2025**

**Task 1**

import java.util.LinkedList;

public class Task001\_DS\_LinkedList {

    public static void main(String[] args) {

        LinkedList<String> fruits = new LinkedList<>();

        fruits.add("Apple");

        fruits.add("Banana");

        fruits.addFirst("Orange");

        fruits.addLast("Grapes");

        System.out.println("First Element: " + fruits.getFirst());

        System.out.println("Last Element: " + fruits.getLast());

        fruits.removeFirst();

        fruits.removeLast();

        for (String fruit : fruits) {

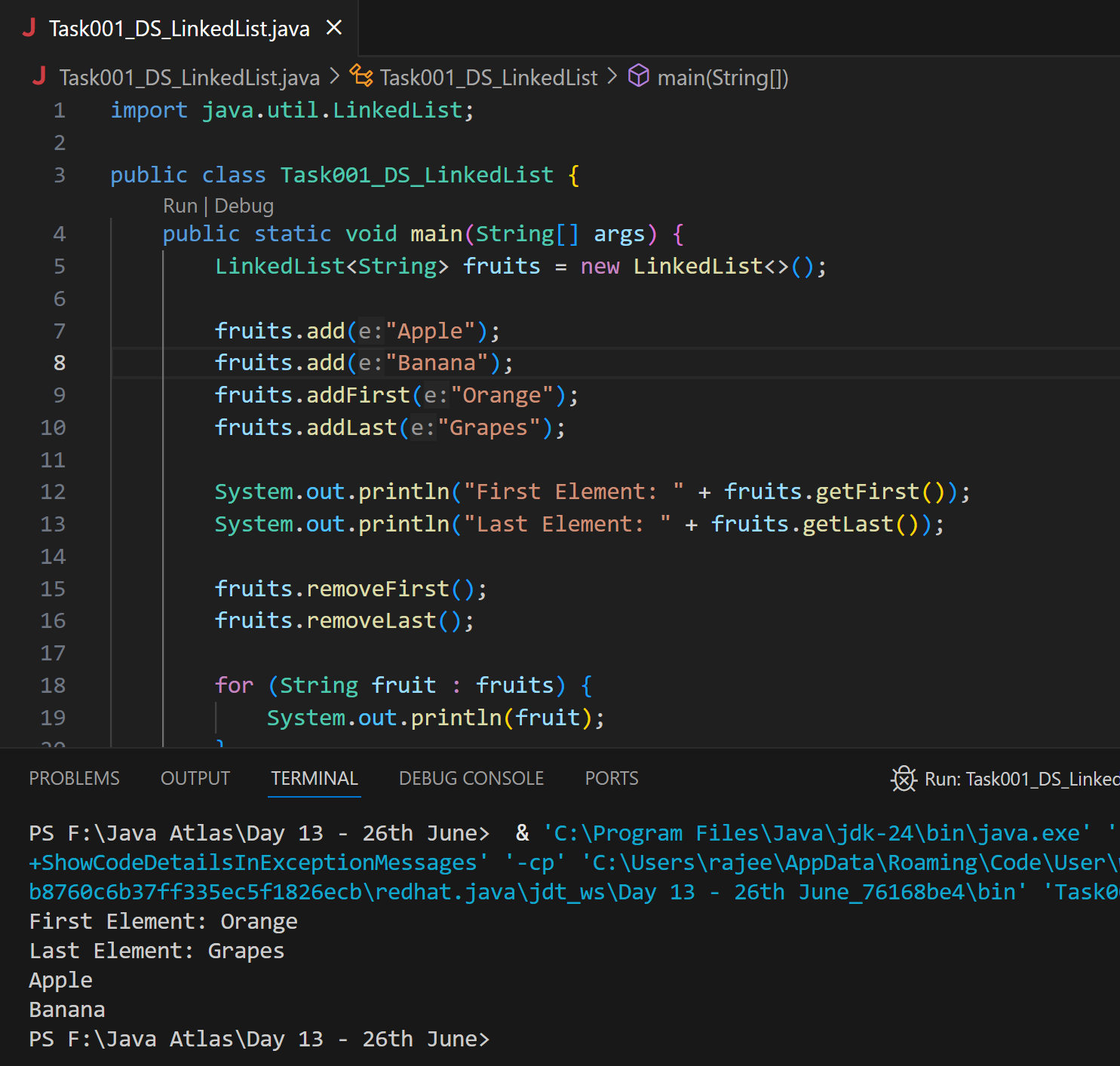
            System.out.println(fruit);

        }

    }

}

**Output**



**Task 2**

**Try to create a node and add a value to it..**

Create a node

Try to add element in the list

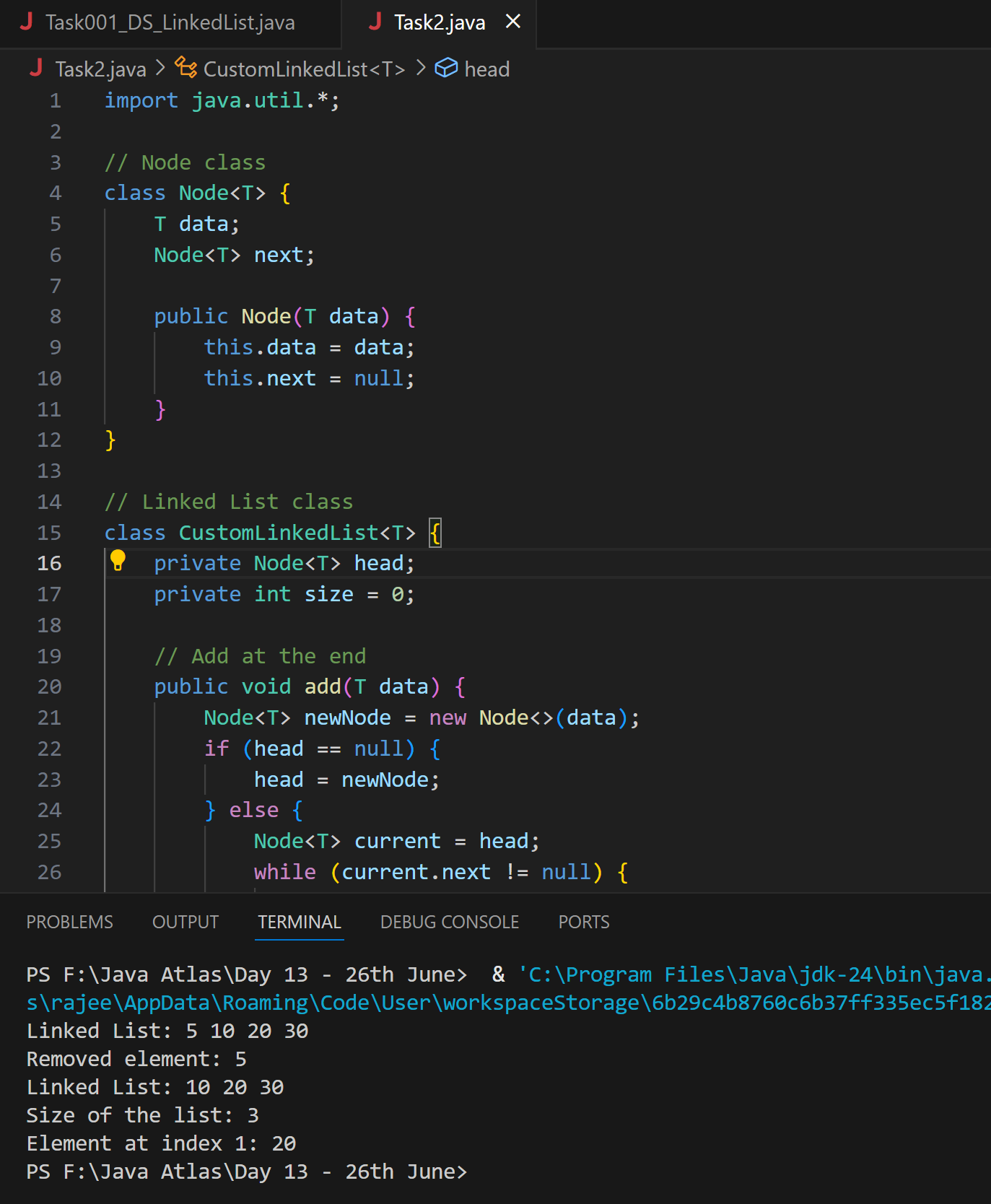
Add the element

Revolve the node

Display all the elements of the node

Find size of the node

**Ans:**



**Code**

import java.util.\*;

// Node class

class Node<T> {

    T data;

    Node<T> next;

    public Node(T data) {

        this.data = data;

        this.next = null;

    }

}

// Linked List class

class CustomLinkedList<T> {

    private Node<T> head;

    private int size = 0;

    // Add at the end

    public void add(T data) {

        Node<T> newNode = new Node<>(data);

        if (head == null) {

            head = newNode;

        } else {

            Node<T> current = head;

            while (current.next != null) {

                current = current.next;

            }

            current.next = newNode;

        }

        size++;

    }

    // Add at the beginning

    public void addFirst(T data) {

        Node<T> newNode = new Node<>(data);

        newNode.next = head;

        head = newNode;

        size++;

    }

    // Remove from the beginning

    public T removeFirst() {

        if (head == null) {

            throw new NoSuchElementException("List is empty");

        }

        T removedData = head.data;

        head = head.next;

        size--;

        return removedData;

    }

    // Get element at a specific index

    public T get(int index) {

        checkBounds(index);

        Node<T> current = head;

        for (int i = 0; i < index; i++) {

            current = current.next;

        }

        return current.data;

    }

    // Display all elements

    public void display() {

        Node<T> current = head;

        System.out.print("Linked List: ");

        while (current != null) {

            System.out.print(current.data + " ");

            current = current.next;

        }

        System.out.println();

    }

    // Size of the list

    public int size() {

        return size;

    }

    // Bounds check for get()

    private void checkBounds(int index) {

        if (index < 0 || index >= size) {

            throw new IndexOutOfBoundsException("Index out of range");

        }

    }

}

// Driver class with main method

public class Task2 {

    public static void main(String[] args) {

        CustomLinkedList<Integer> list = new CustomLinkedList<>();

        // Add elements

        list.add(10);

        list.add(20);

        list.add(30);

        list.addFirst(5); // Adding at beginning

        // Display list

        list.display();

        // Remove first element

        int removed = list.removeFirst();

        System.out.println("Removed element: " + removed);

        // Display again

        list.display();

        // Print size

        System.out.println("Size of the list: " + list.size());

        // Get element at index 1

        System.out.println("Element at index 1: " + list.get(1));

    }

}

**Task 3**

**List down the methods of linked lists.**

**Ans:**

Here are some common methods of LinkedList class in Java:

1. **add(E e)**: Adds the specified element to the end of the list.
2. **add(int index, E element)**: Inserts the specified element at the specified position in the list.
3. **addFirst(E e)**: Inserts the specified element at the beginning of the list.
4. **addLast(E e)**: Appends the specified element to the end of the list.
5. **clear()**: Removes all elements from the list.
6. **contains(Object o)**: Returns true if the list contains the specified element.
7. **get(int index)**: Returns the element at the specified position in the list.
8. **getFirst()**: Returns the first element in the list.
9. **getLast()**: Returns the last element in the list.
10. **indexOf(Object o)**: Returns the index of the first occurrence of the specified element in the list.
11. **lastIndexOf(Object o)**: Returns the index of the last occurrence of the specified element in the list.
12. **isEmpty()**: Returns true if the list contains no elements.
13. **remove(int index)**: Removes the element at the specified position in the list.
14. **remove(Object o)**: Removes the first occurrence of the specified element from the list.
15. **removeFirst()**: Removes and returns the first element from the list.
16. **removeLast()**: Removes and returns the last element from the list.
17. **size()**: Returns the number of elements in the list.
18. **set(int index, E element)**: Replaces the element at the specified position in the list with the specified element.

**Task 4:**

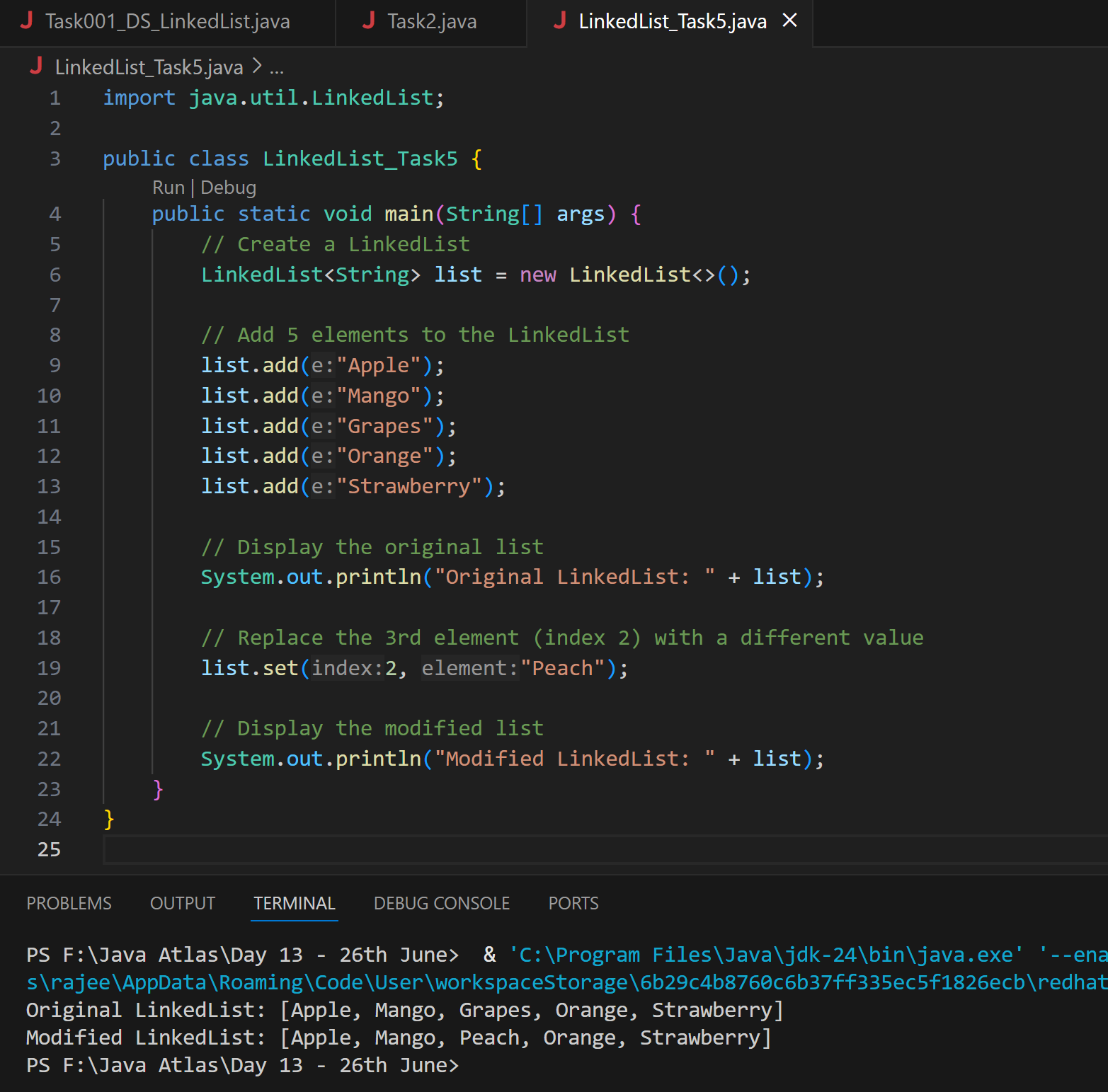
**What are the operations of data structures**

**Ans:**

1. Insertion – Add a new element to the data structure.
2. Deletion – Remove an existing element from the structure.
3. Traversal – Visit and process each element one by one.
4. Searching – Find whether an element exists or locate its position.
5. Sorting – Arrange elements in a specific order (ascending/descending).
6. Access (Retrieval) – Get the value of an element at a given index or position.
7. Update (Modification) – Change the value of an existing element.
8. Merging – Combine two or more data structures into one.
9. Splitting – Divide a data structure into two or more parts.
10. Resizing – Expand or shrink the capacity of a dynamic structure.
11. Push – Add an element to the top of a stack.
12. Pop – Remove the top element from a stack.
13. Enqueue – Add an element to the rear of a queue.
14. Dequeue – Remove an element from the front of a queue.
15. InsertLeft – Insert an element at the left end of a deque.
16. InsertRight – Insert an element at the right end of a deque.
17. Peek – View the top/front element without removing it.
18. isEmpty – Check if the data structure has no elements.
19. isFull – Check if the structure has reached its capacity (for fixed-size).
20. Size – Return the number of elements currently stored.

**Task 5:**

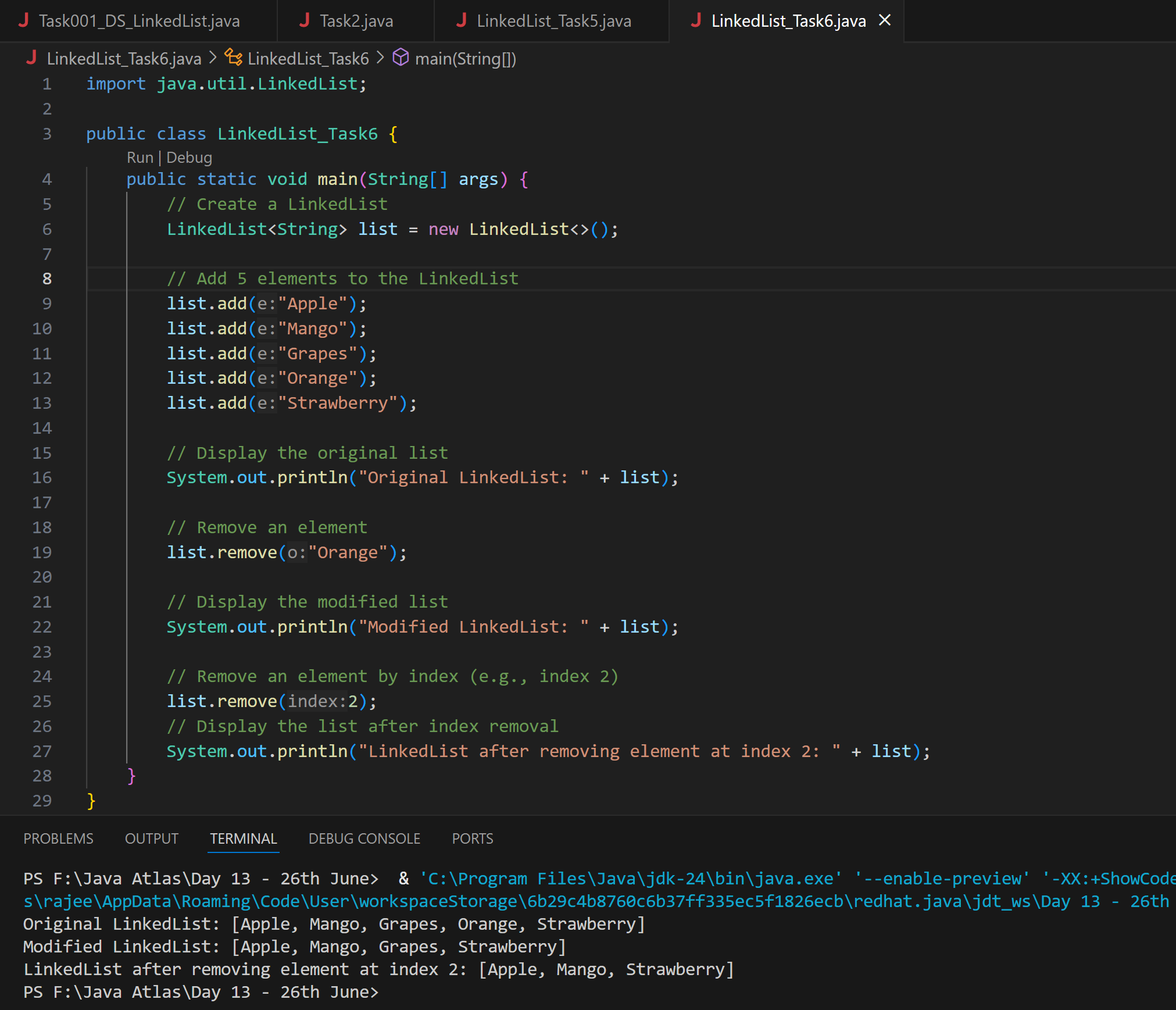
**Wap to create linked list add 5 elements to it and replace 3 rd element with different value..**



**Task 6:**

**Wap to create a linked list to add 5 elements and remove any element and display**

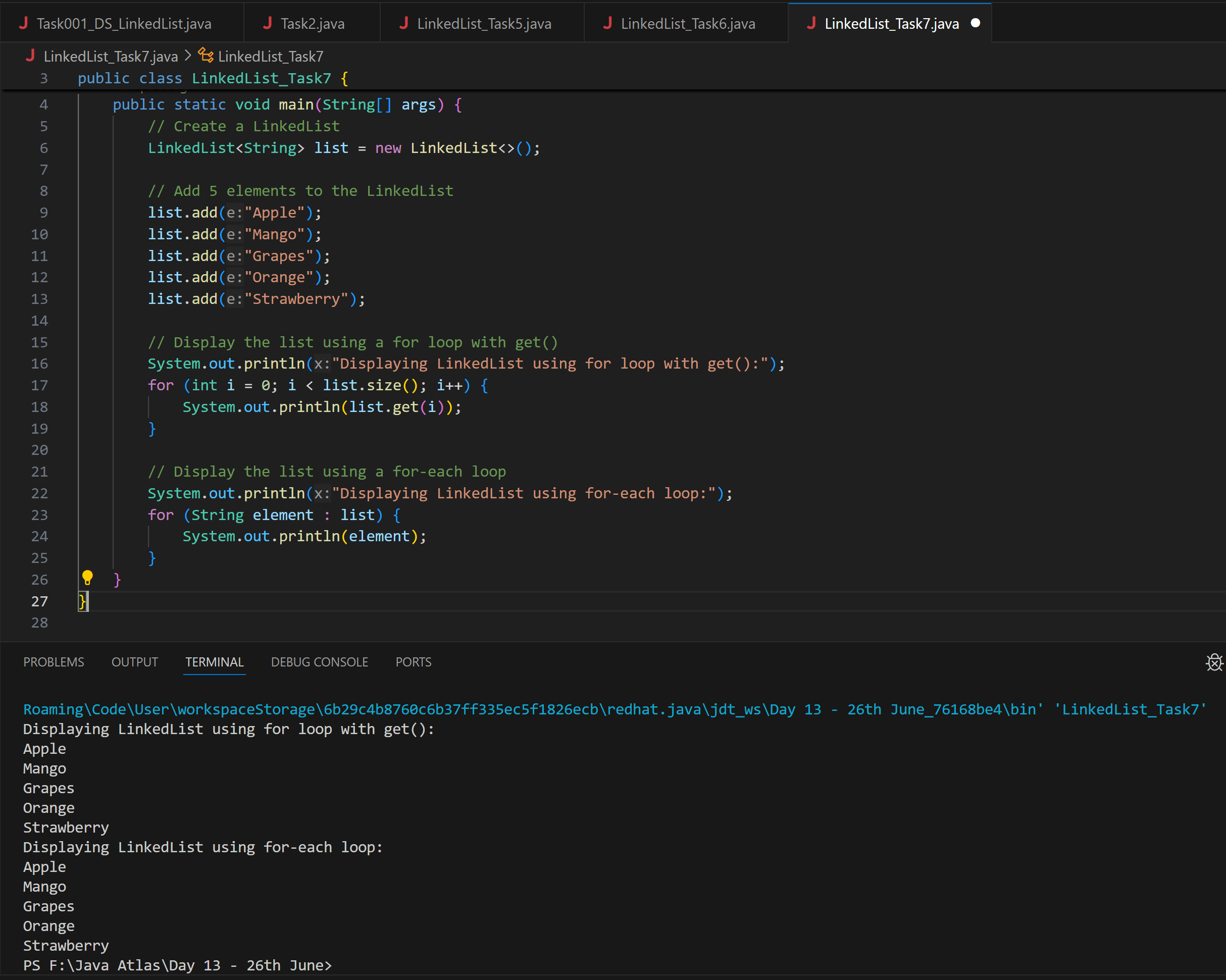
**Ans:**



**Task 7:**

**Wap to create a linked list to add 5 elements and display the list using for (use get() ) and for each loops**

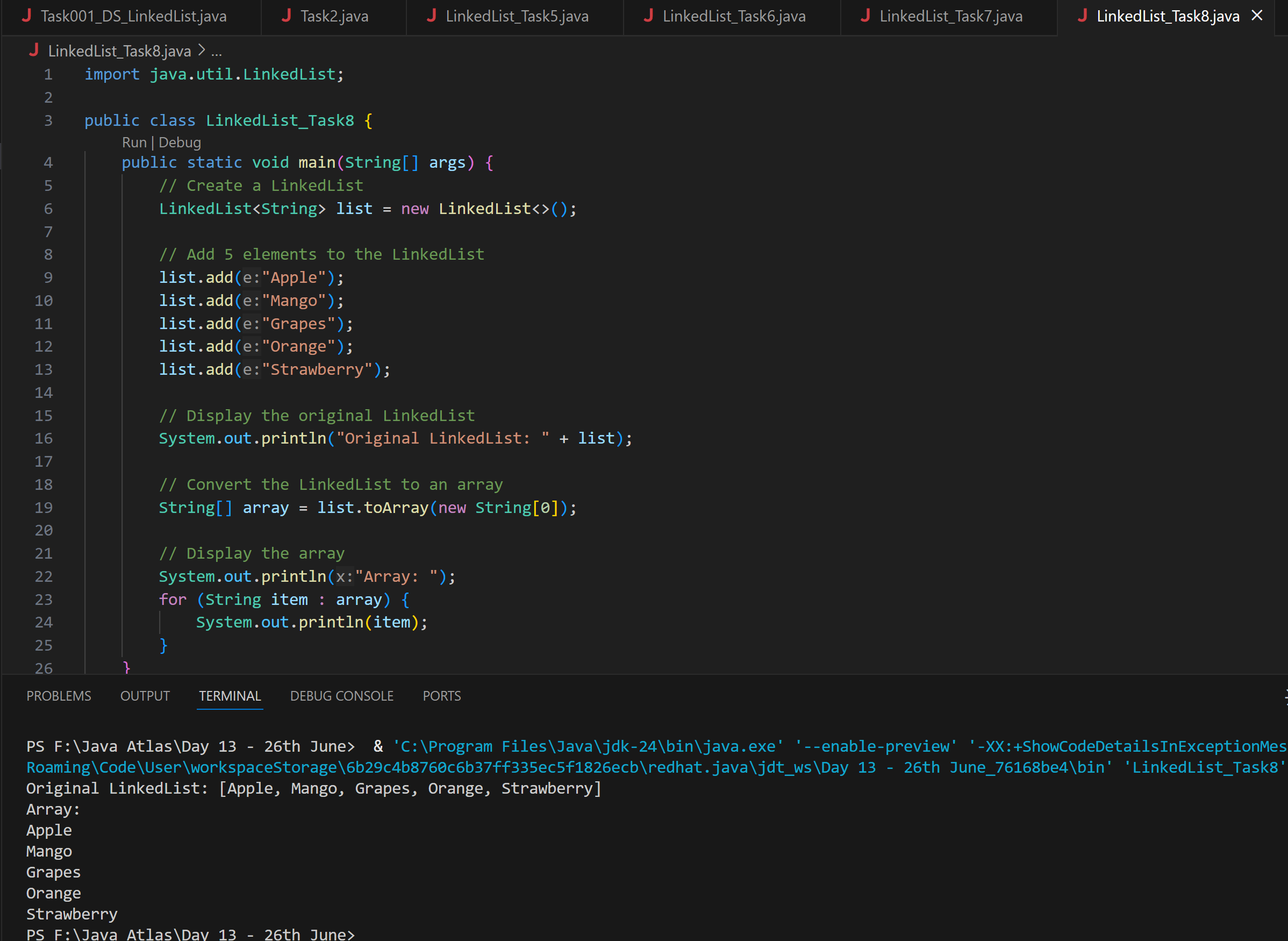
**Ans:**

****

**Task 8:**

**Create  a linked list and few items and convert it into an array**

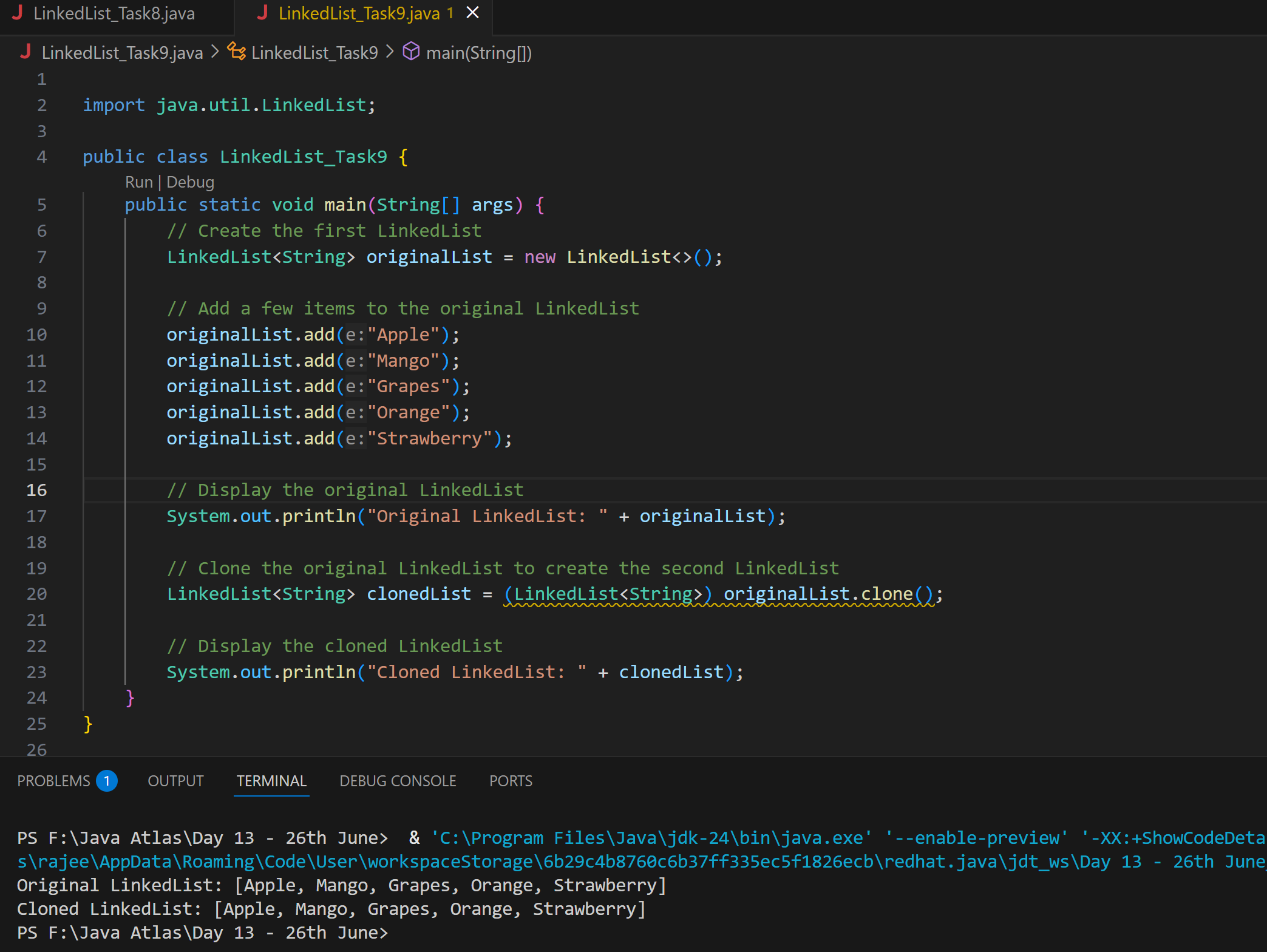
**Ans:**



**Task 9:**

**Create a linked list add few items and clone the 1st linked list with the 2nd linked list**

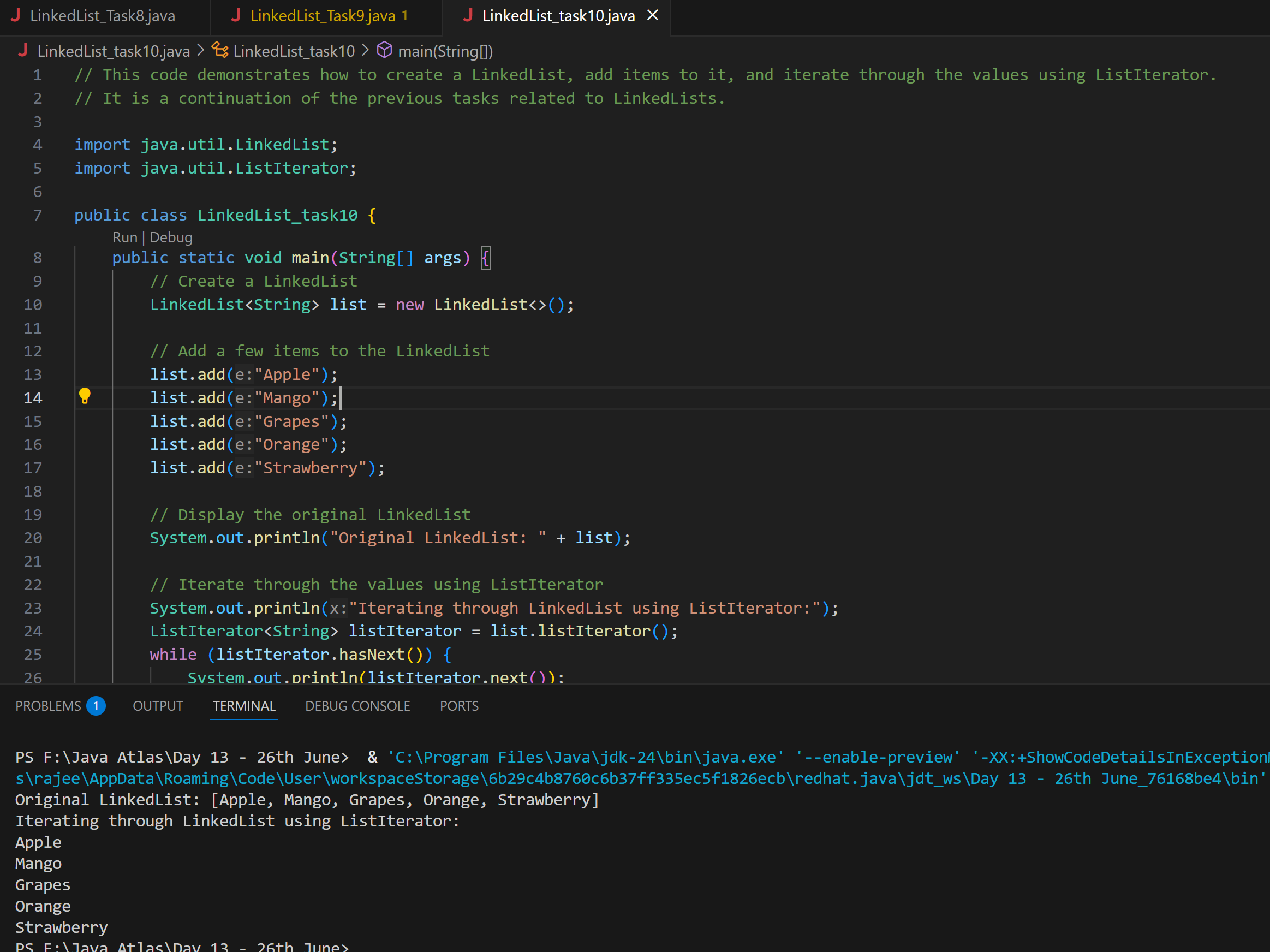
**Ans:**



**Task 10:**

**Create  linked list and iterate the values using ListIterator class in util package**

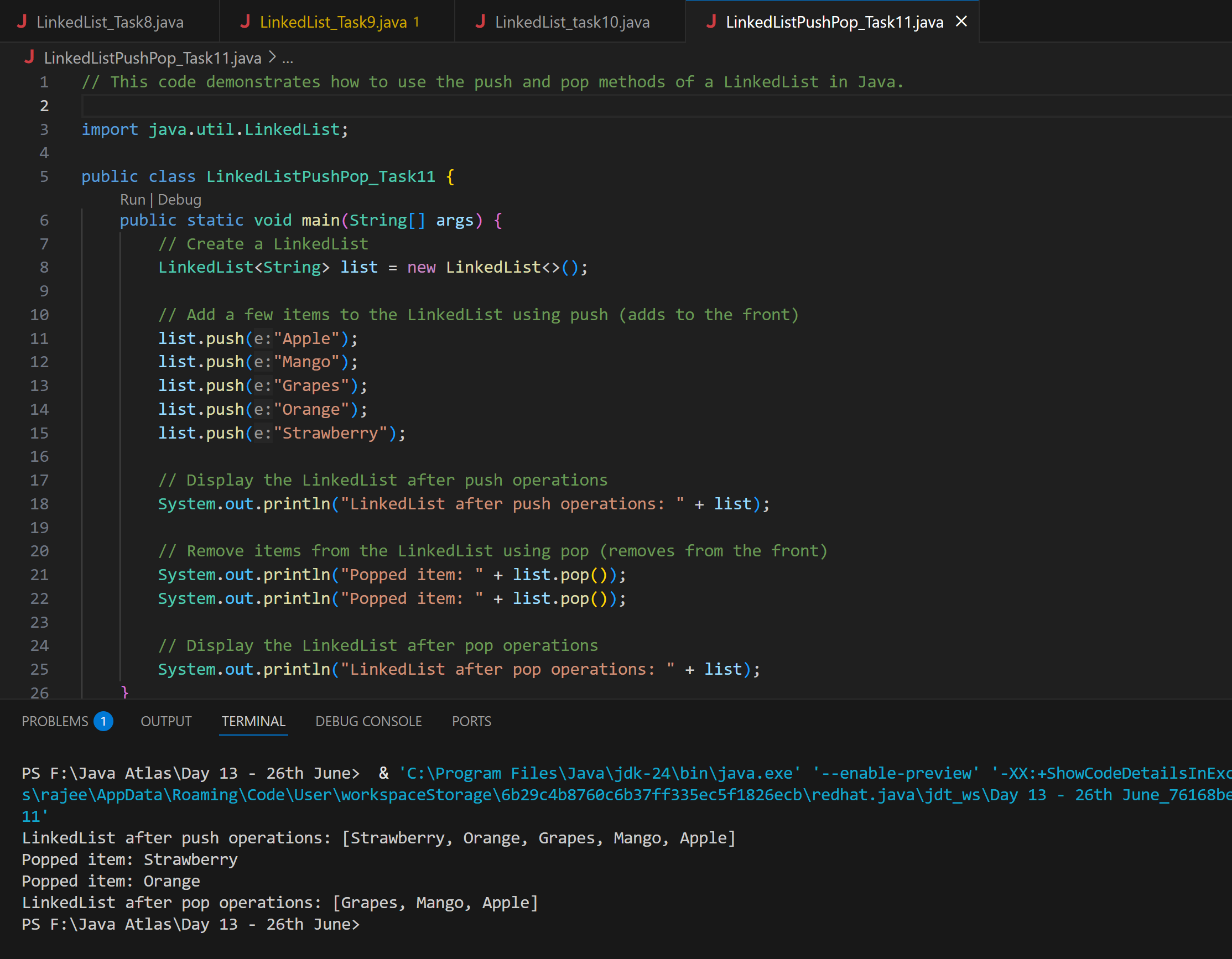
Ans:



**Task 11:**

**Create a linked list and use push and pop methods.**

**Ans:**



**Task 12:**

| **Sr. No.** | **Key** | **Iterator** | **Split iterator** |
| --- | --- | --- | --- |
| 1 | Basic | It can be used to traverse the element of the collection | It can be used with Stream also. |
| 2 | Bulk Operation | It can be used to traverse the element one by one only | It can be used to traverse the elements in bulk. |
| 3 | Sequential /Parallel | It can traverse the element in sequential manner only | It can traverse the element in sequential  as well as parallel manner. |
| 4. | External /Internal  Iterator | Iterator uses External Iteration to iterate Collections | Spliterator uses Internal Iteration |

**Difference between Iterator and splitIterator**

**Task 13:**

**Below is the code for Split iterator… run it to see the output..**

**Can you it to sout()... and see .**

**import java.util.\*;**

**public class Task0013\_DS\_Linkedlist\_SplitIterator {**

**public static void main(String[] args) {**

**LinkedList<String> lobj = new LinkedList<>();**

**lobj.add("Prasunamba");**

**lobj.add("Meher");**

**lobj.add(".MK");**

**Spliterator<String> sitobj = lobj.spliterator();**

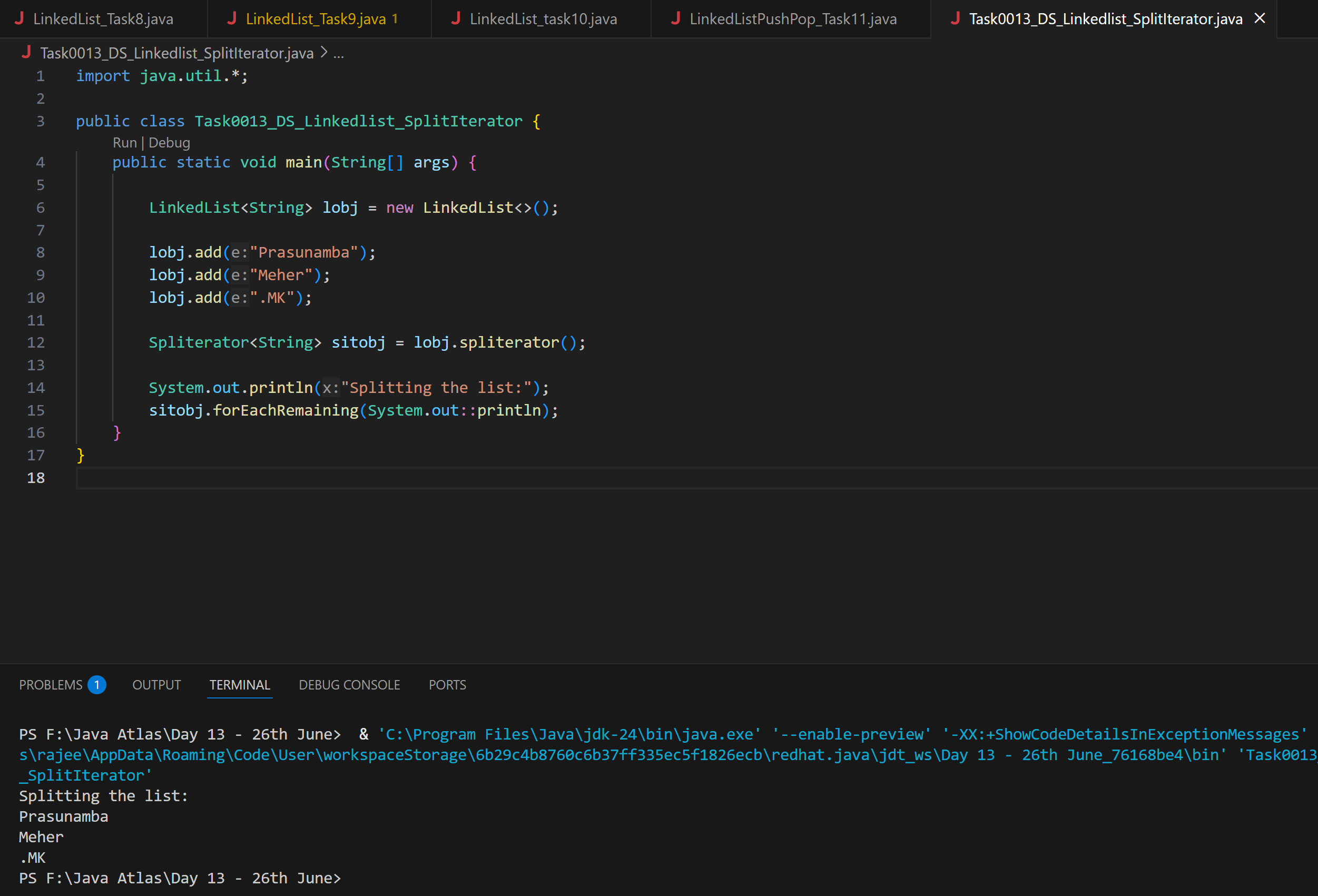
**System.out.println("Splitting the list:");**

**sitobj.forEachRemaining(System.out::println);**

**}**

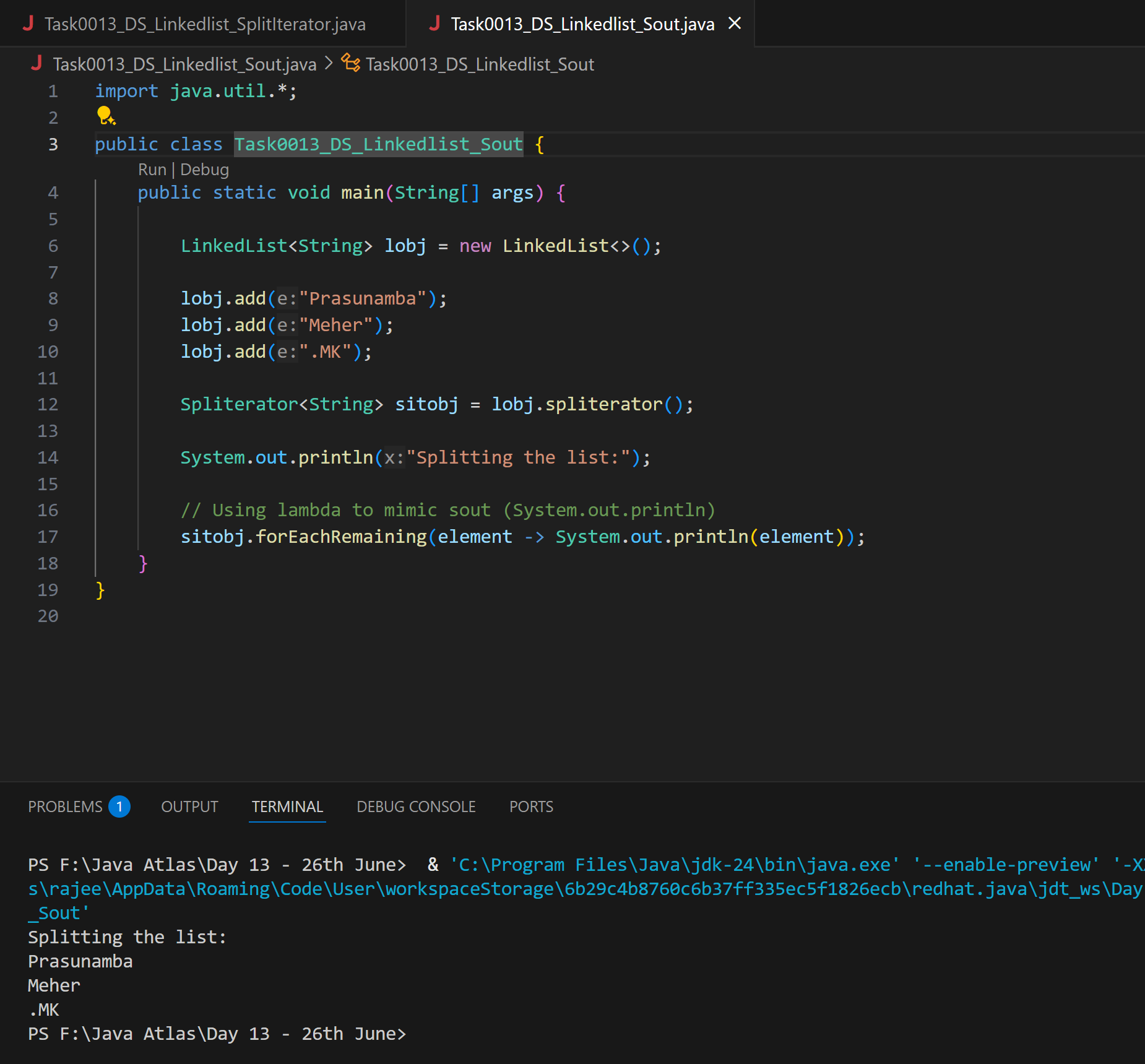
**}**

**Output**



**Code for sout**

**Ans:**



**Task 14:**

**Create alinkedlist and display items into 2 lists using split  iterator**

**Hint:**

**Spliterator<String> itobj2 = itobj1.trySplit();**

**while( itobj1.tryAdvance( (n) -> { System.out.println(n); } ) );**

**Ans:**

import java.util.LinkedList;

import java.util.Spliterator;

public class Task14\_SplitLinkedListS {

    public static void main(String[] args) {

        // Step 1: Create a LinkedList and add items

        LinkedList<String> items = new LinkedList<>();

        items.add("Apple");

        items.add("Banana");

        items.add("Cherry");

        items.add("Date");

        items.add("Elderberry");

        items.add("Fig");

        // Step 2: Create a Spliterator from the LinkedList

        Spliterator<String> it1 = items.spliterator();

        // Step 3: Split the iterator

        Spliterator<String> it2 = it1.trySplit();

        // Step 4: Traverse first half using it2

        System.out.println("Items from Spliterator 1:");

        it1.forEachRemaining((item) -> System.out.println("  " + item));

        // Step 5: Traverse second half using it1

        System.out.println("Items from Spliterator 2:");

        if (it2 != null) {

            it2.forEachRemaining((item) -> System.out.println("  " + item));

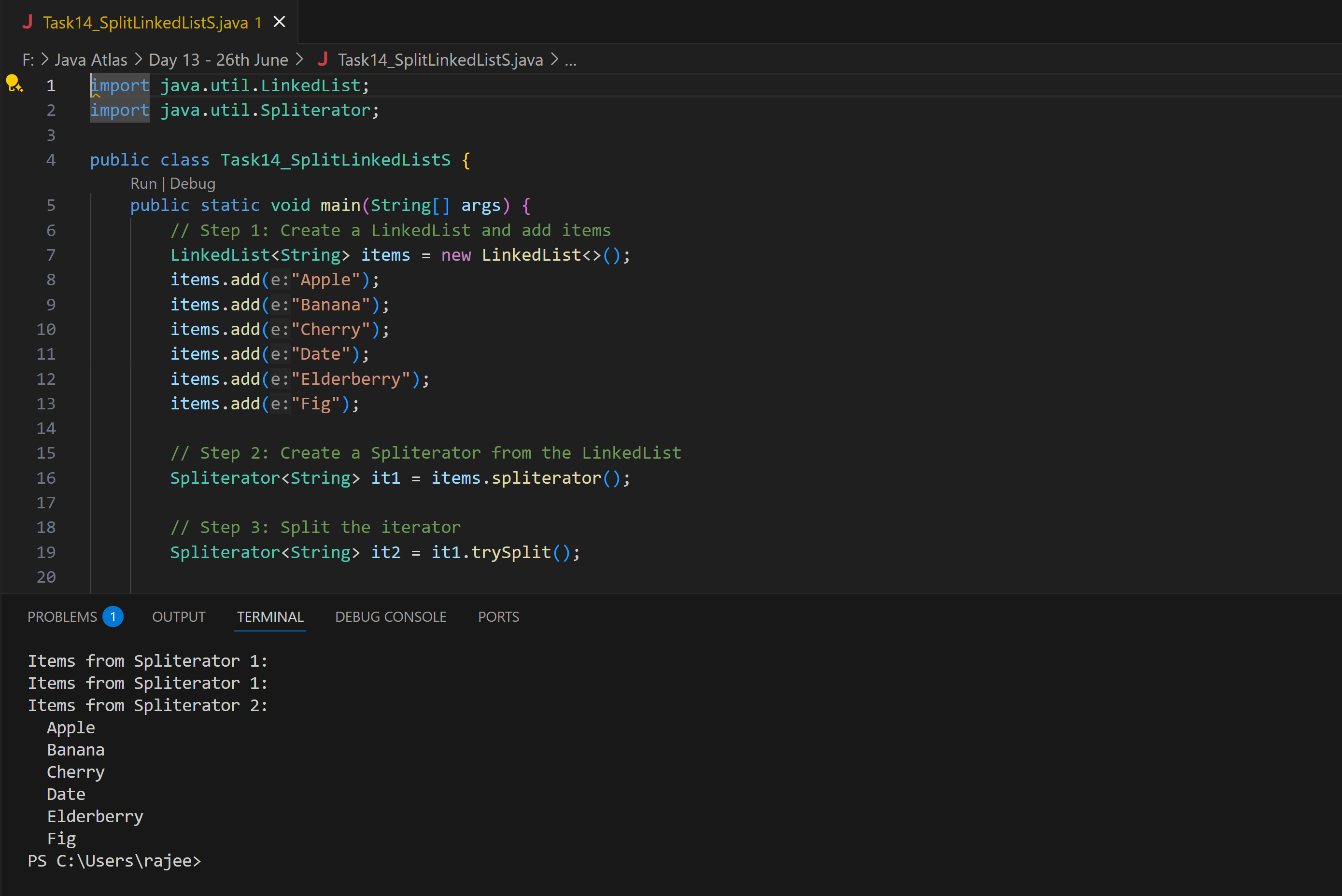
        } else {

            System.out.println("Spliterator couldn't split (too small).");

        }

    }

}

****

**Task 15:**

**What do you understand by a pointer?**

**Ans:**

A pointer is a variable in programming that stores the memory address of another variable. Instead of holding a data value directly, a pointer points to the location where the data is stored. This is particularly useful for dynamic memory allocation, arrays, and data structures like linked lists and trees.

**Task 16:**

**Difference between \* and & in pointers?**

**\* (Dereference Operator):** Used to access the value at the memory address stored in a pointer.

Example: If p is a pointer to an integer, \*p gives the value of the integer that p points to.

**& (Address-of Operator):** Used to get the memory address of a variable.

Example: If x is an integer, &x gives the memory address of x.

**Task 17:**

**Wap in c or c++ to implement the use of pointers**

**Ans:**

#include <iostream>

using namespace std;

int main() {

    int x = 10;       // Declare an integer variable

    int \*p = &x;      // Declare a pointer and assign it the address of x

    cout << "Value of x: " << x << endl;          // Output the value of x

    cout << "Address of x: " << &x << endl;       // Output the address of x

    cout << "Value of p (address of x): " << p << endl; // Output the value of p (address of x)

    cout << "Value at address p: " << \*p << endl; // Output the value at the address stored in p

    \*p = 20;           // Change the value at the address stored in p

    cout << "New value of x: " << x << endl;      // Output the new value of x

    return 0;

}

**Task 18:**

**Wap to create  a doubly linked list**

**Ans:**

**Code:**

public class MyDoublyLinkedList {

    // Node structure with data, prev, and next pointers

    private static class Node {

        String data;

        Node prev, next;

        Node(String d) { data = d; }

    }

    private Node head, tail;

    public void addFirst(String data) {

        Node n = new Node(data);

        if (head == null) {

            head = tail = n;

        } else {

            n.next = head;

            head.prev = n;

            head = n;

        }

    }

    public void addLast(String data) {

        Node n = new Node(data);

        if (tail == null) {

            head = tail = n;

        } else {

            tail.next = n;

            n.prev = tail;

            tail = n;

        }

    }

    public String removeFirst() {

        if (head == null) return null;

        String val = head.data;

        head = head.next;

        if (head != null) head.prev = null;

        else tail = null;

        return val;

    }

    public String removeLast() {

        if (tail == null) return null;

        String val = tail.data;

        tail = tail.prev;

        if (tail != null) tail.next = null;

        else head = null;

        return val;

    }

    public void traverseForward() {

        for (Node cur = head; cur != null; cur = cur.next) {

            System.out.print(cur.data + " ");

        }

        System.out.println();

    }

    public void traverseBackward() {

        for (Node cur = tail; cur != null; cur = cur.prev) {

            System.out.print(cur.data + " ");

        }

        System.out.println();

    }

    public static void main(String[] args) {

        MyDoublyLinkedList dll = new MyDoublyLinkedList();

        dll.addFirst("Apple");

        dll.addLast("Mango");

        dll.addFirst("Grapes");

        dll.addLast("Orange");

        System.out.print("Forward: ");

        dll.traverseForward();      // Grapes Apple Mango Orange

        System.out.print("Backward: ");

        dll.traverseBackward();     // Orange Mango Apple Grapes

        dll.removeFirst();          // removes Grapes

        dll.removeLast();           // removes Orange

        System.out.print("After removals, Forward: ");

        dll.traverseForward();      // Apple Mango

    }

}

