Day 13 -

Employee ID: 201933938

Login ID: iamasif

Name: Shaik Asif

Linked Lists:

&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

Linked list in c++

#include <bits/stdc++.h>

using namespace std;

// Define a Node class

class Node{

public:

int data; // Data part of the node

Node\* next; // Pointer to the next node

// Constructor for convenience

Node(int value) : data(value), next(nullptr) {}

};

// Class for singly linked list

class Linkedlist{

private:

Node\* head; // Pointer to the head of the list

public:

// Constructor to initialize an empty list

Linkedlist(){

head = nullptr;

}

// Function to insert a node at the end

void insertAtEnd(int value){

Node\* newNode = new Node(value);

if(head == nullptr){

head = newNode; // If list is empty, make newNode the head

}

else{

Node\* temp = head;

while (temp->next != nullptr){

temp = temp->next; // Traverse to the last node

}

temp->next = newNode; // Link the last node to newNode

}

}

// Function to delete a Node by Value

void deleteByValue(int value){

if(head == nullptr){

return;

}

if(head->data == value){

Node\* temp = head;

head = head->next; // Move head to the next node

delete temp; // Free memory of the deleted node

return;

}

Node\* temp = head;

while(temp->next && temp->next->data != value){

temp = temp->next; // Traverse to find the node to delete

}

if(temp->next){

Node\* nodeToDelete = temp->next;

temp->next = temp->next->next; // Unlink the node

delete nodeToDelete; //Free Memory

}

}

// Function to display the list

void display(){

Node\* temp = head;

while(temp != nullptr){

cout << temp->data << "->";

temp = temp->next;

}

cout << "NULL" <<endl;

}

// Destructor to free all allocated memory

~LinkedList() {

Node\* temp;

while (head) {

temp = head;

head = head->next;

delete temp;

}

}

};

int main() {

LinkedList list;

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

cout << "Linked List: ";

list.display();

list.deleteByValue(20);

cout << "After Deleting 20: ";

list.display();

return 0;

}

======================================================================================

<https://pythontutor.com/render.html#mode=display>

By Ahmed.. You can visualise the data structures while executing the code line by line..

**Task 001:**

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);

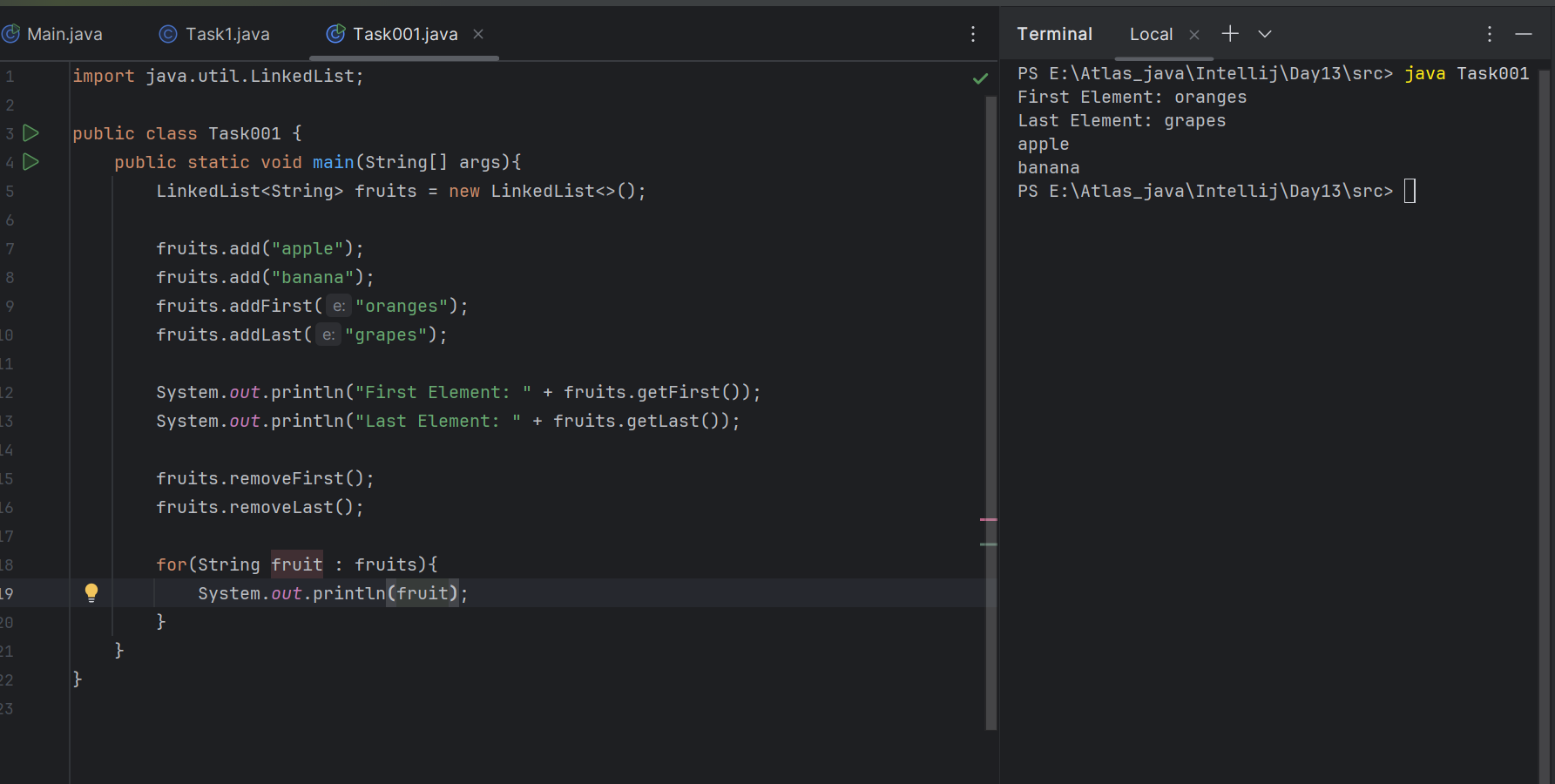
}

}

}

11.07 to 11.12

==========================================================================================

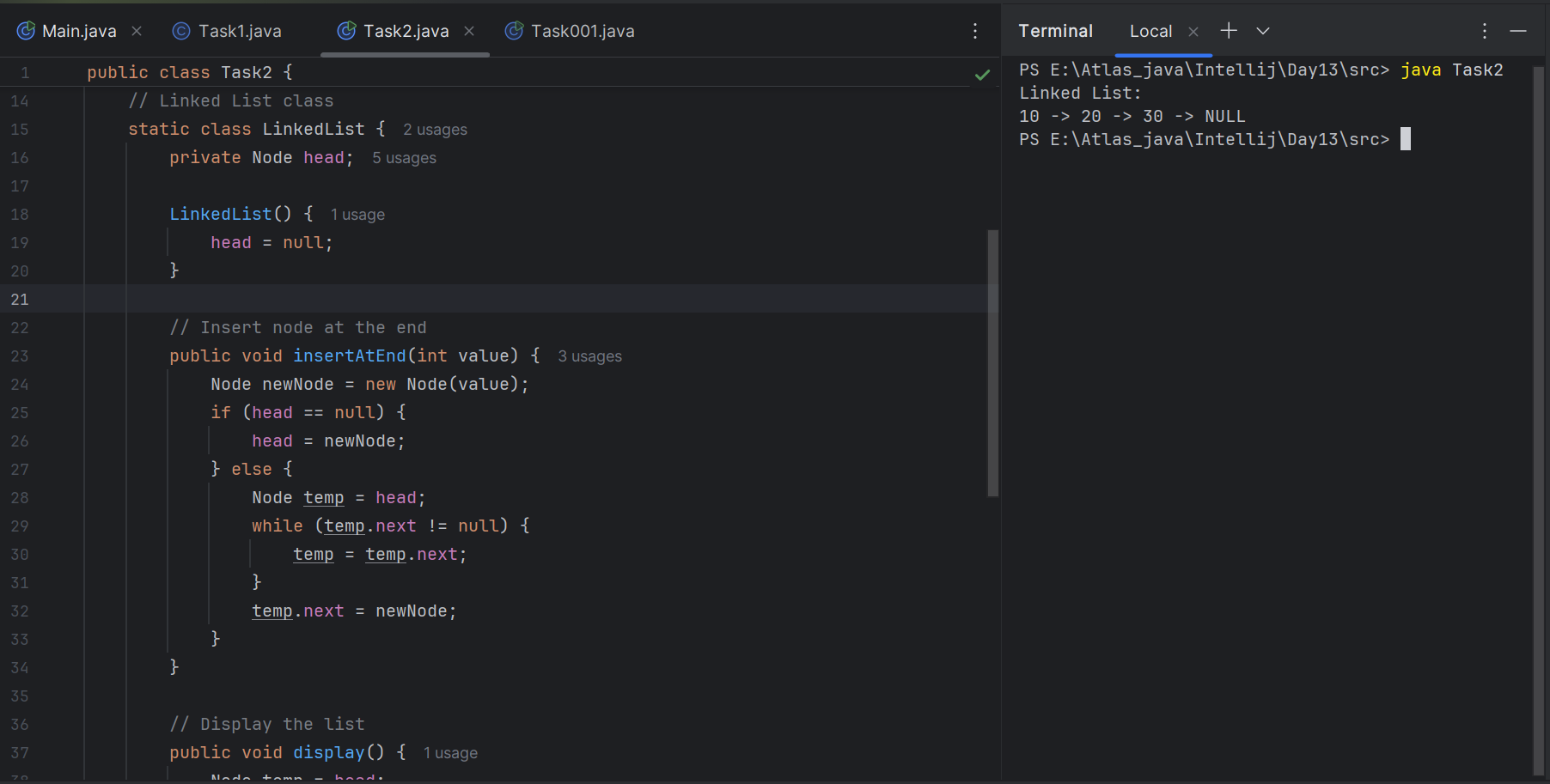


**T**

**Task 002:**

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

11.35 to 11.40



/ create a node

class Node<T> {

T data;

Node<T> next;

public Node(T data) {

this.data = data;

this.next = null;

}

}

// try to add element in the list

public 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 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;

}

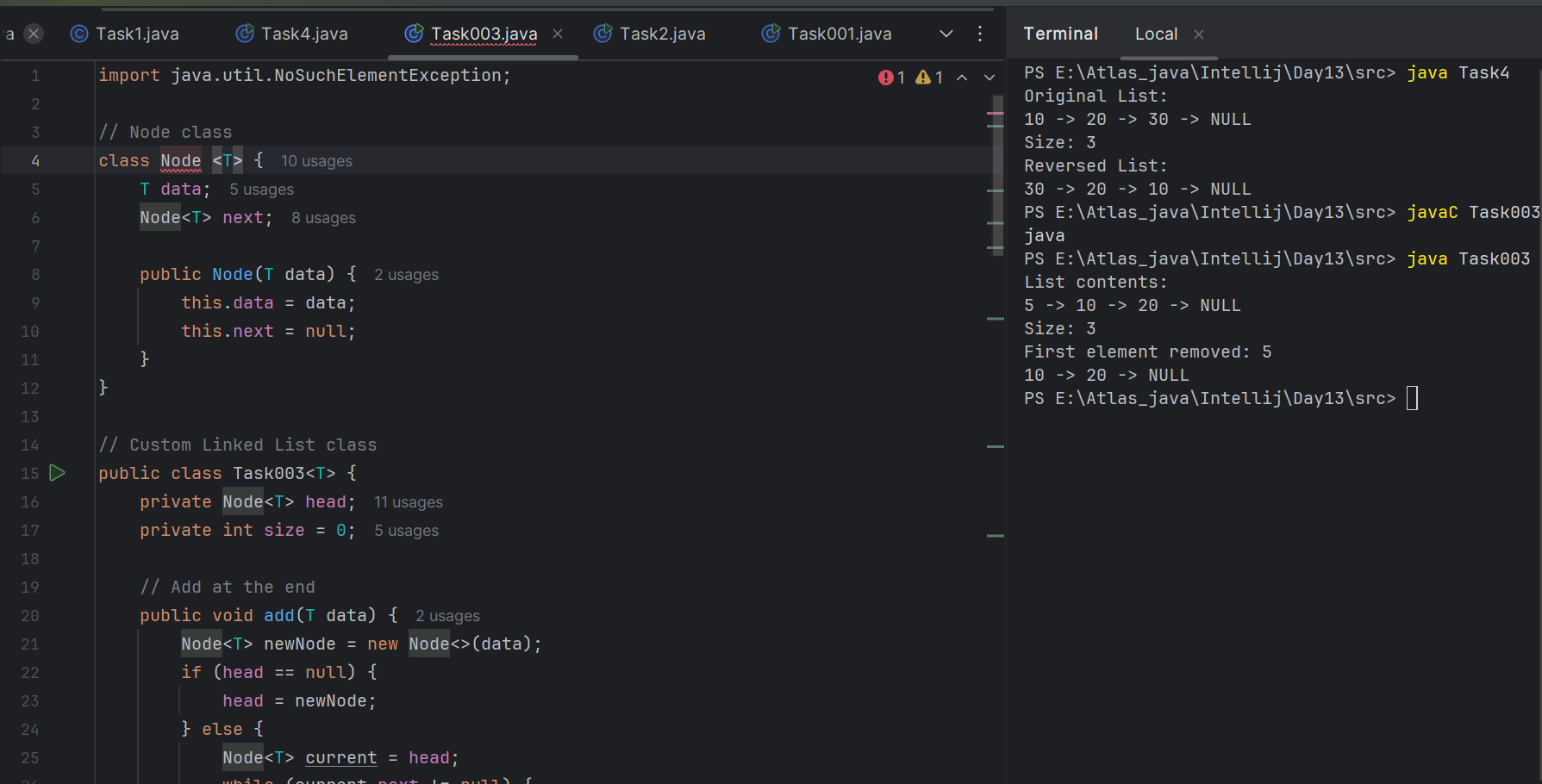
// Size of the list

public int size() {

return size;

}

}



**Task4:**

Create a node

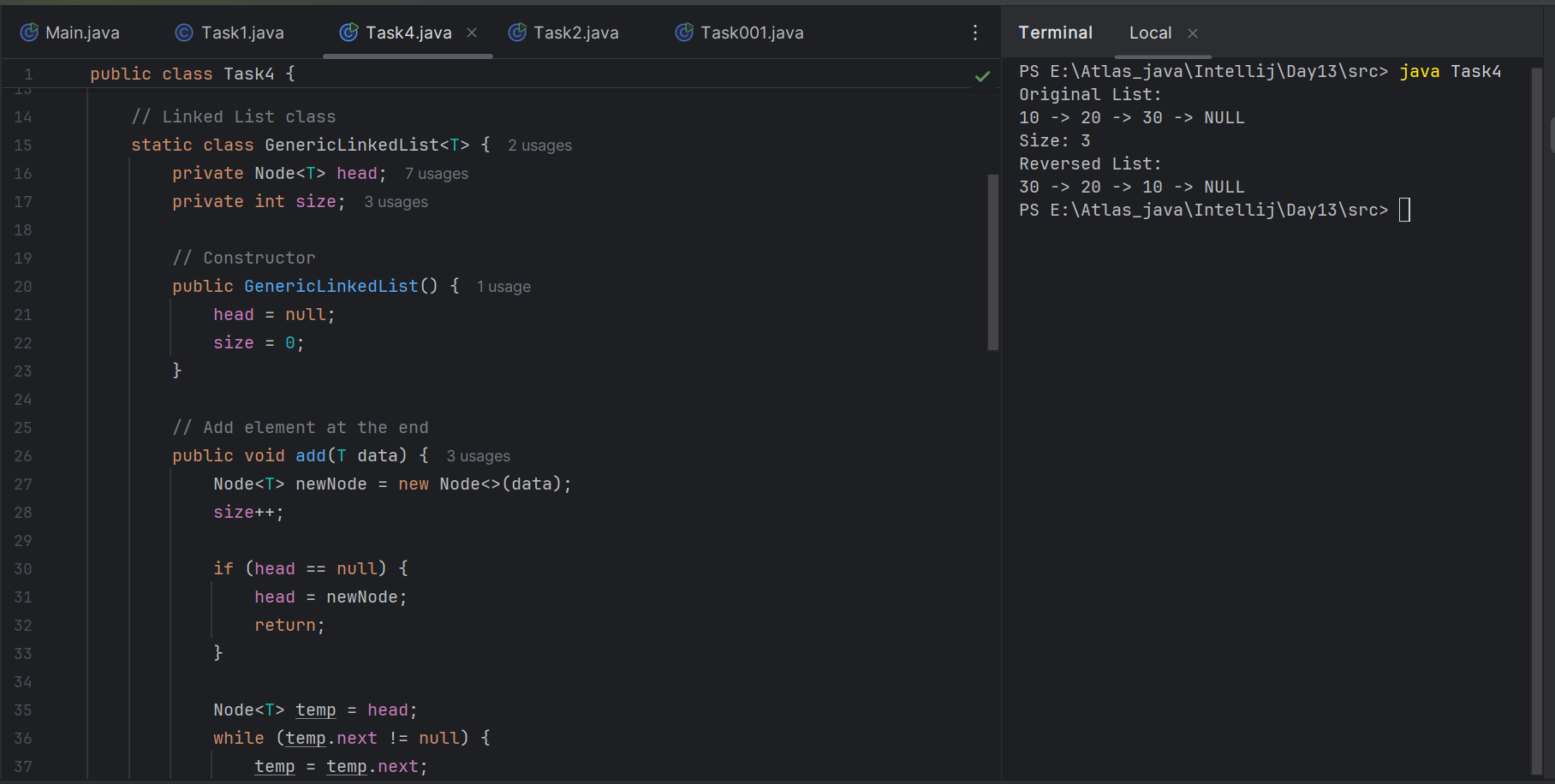
Try to add element in the list

Add the element

Remove the node

Display all the elements of the node

Find size of the node



**Task0004:**

addfirst

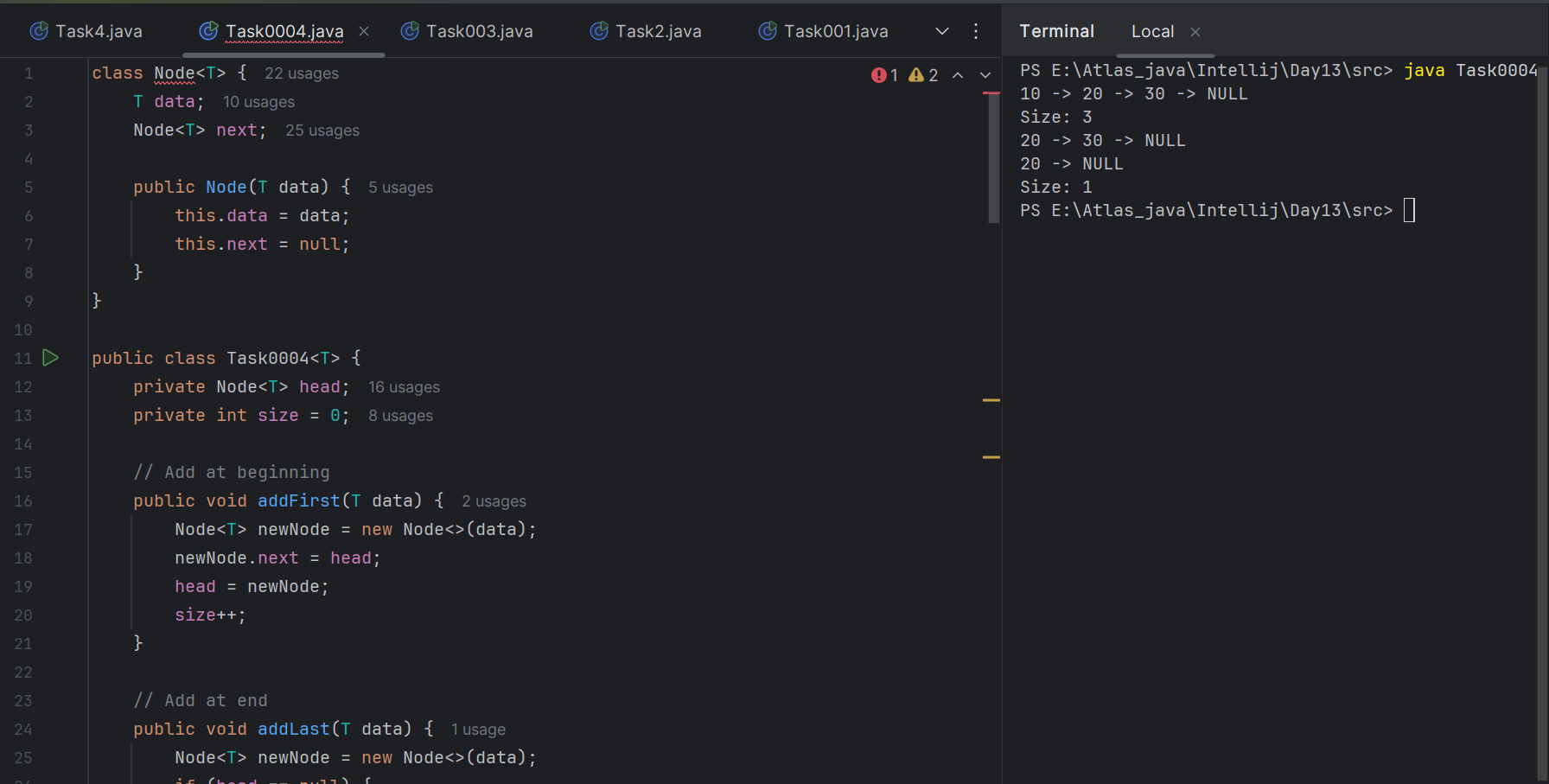
add last

removefirst

remove last

traverse

adding in between



**Task 5:**

List down the methods of linked lists..

| **Method Name** | **Purpose** |
| --- | --- |
| addFirst(T data) | Add a node at the beginning of the list |
| addLast(T data) | Add a node at the end of the list |
| addAt(int index, T data) | Insert a node at a specific index |
| removeFirst() | Remove the first node from the list |
| removeLast() | Remove the last node from the list |
| removeAt(int index) | Remove a node at a specific index |
| get(int index) | Get the data at a specific index |
| set(int index, T data) | Update the data at a specific index |
| contains(T data) | Check if a value exists in the list |
| indexOf(T data) | Return the index of the first occurrence of the value |

14.40 to 14.45

=====================================================================

### **Traversal:**

| **Method Name** | **Purpose** |
| --- | --- |
| traverse() | Print or visit all elements in order |
| reverseTraversal() | Print all elements in reverse order *(usually recursive)* |

### **Size and Emptiness:**

| **Method Name** | **Purpose** |
| --- | --- |
| size() | Return the number of elements in the list |
| isEmpty() | Check whether the list is empty |

### **Utility / Maintenance:**

| **Method Name** | **Purpose** |
| --- | --- |
| clear() | Remove all elements from the list |
| toString() | Convert the list to a string format (for printing) |
| clone() | Create a copy of the current list |

| **Doubly Linked List Only:** |  |
| --- | --- |

| prev pointer in node | Points to the previous node |
| --- | --- |

| addBefore() | Insert before a given node |
| --- | --- |

| addAfter() | Insert after a given node |
| --- | --- |

| reverse() | Reverses the entire list |
| --- | --- |

**Task 5**

**:**

What are the operations of data structures.. I liner

14.46 to 14.50

Traversing

Insertion

Deletion

Searching

Sorting

### **1. Traversing**

* **Definition**: Visiting each element of the data structure exactly once in order.
* **Purpose**: Often used to display elements, apply operations, or perform computations.
* **Example**: Printing all elements in an array or linked list.

### **2. Insertion**

* **Definition**: Adding a new element to the data structure.
* **Places to Insert**:  
  + At the beginning
  + At the end
  + At a specific position/index
* **Example**: list.add(2, value);

### **3. Deletion**

* **Definition**: Removing an element from the data structure.
* **Places to Delete**:  
  + First element
  + Last element
  + Specific index or value
* **Example**: list.removeFirst();, list.removeAt(3);

### **4. Searching**

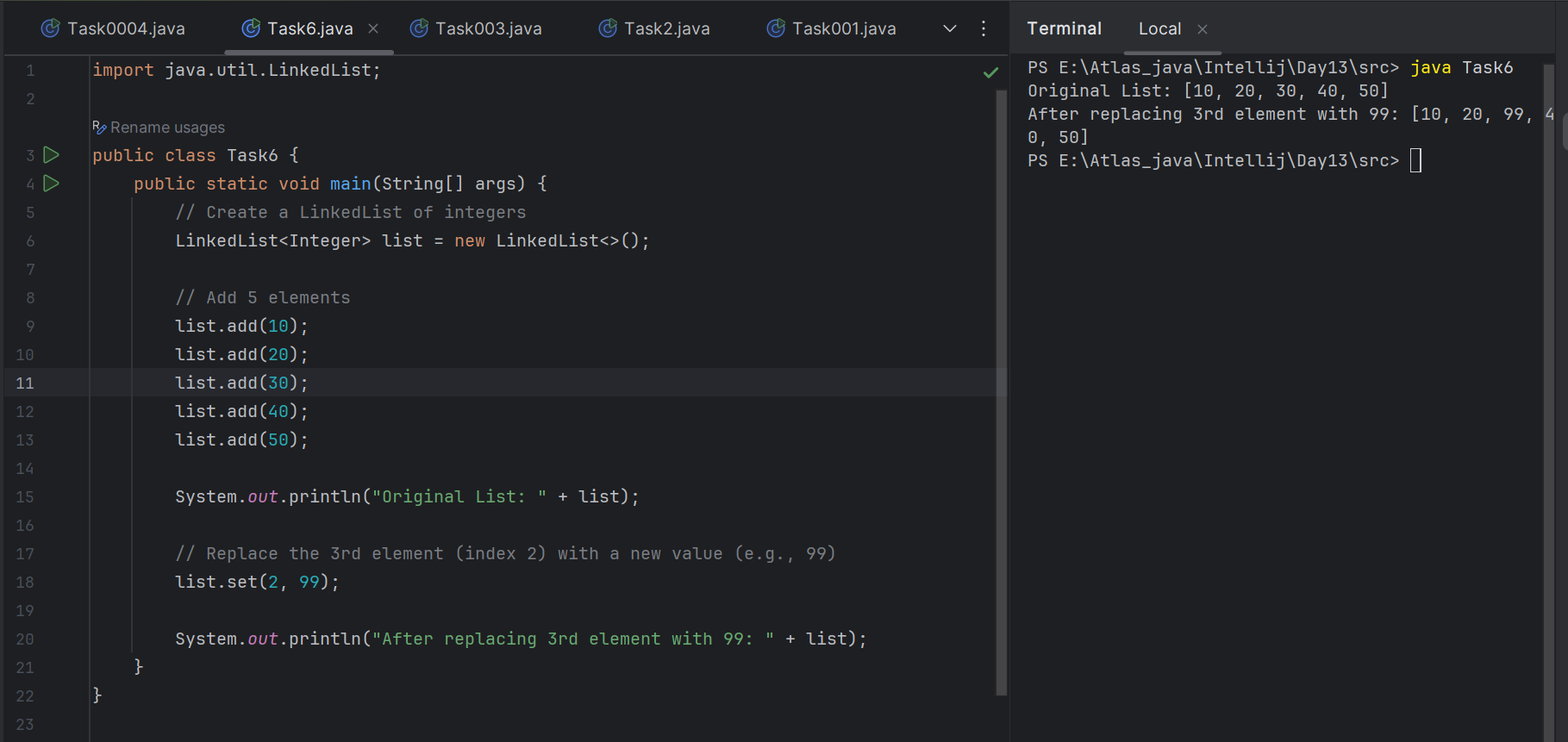
* **Definition**: Finding whether an element exists in the data structure (and possibly its position).
* **Techniques**:  
  + Linear Search (for unsorted data)
  + Binary Search (for sorted data)
* **Example**: list.contains(value);, indexOf(value);

### **5. Sorting**

* **Definition**: Rearranging elements in a specific order (ascending or descending).
* **Common Algorithms**:  
  + Bubble Sort
  + Selection Sort
  + Merge Sort
  + Quick Sort
* **Example**: Sorting a list of numbers in increasing order.

**Task 6:**

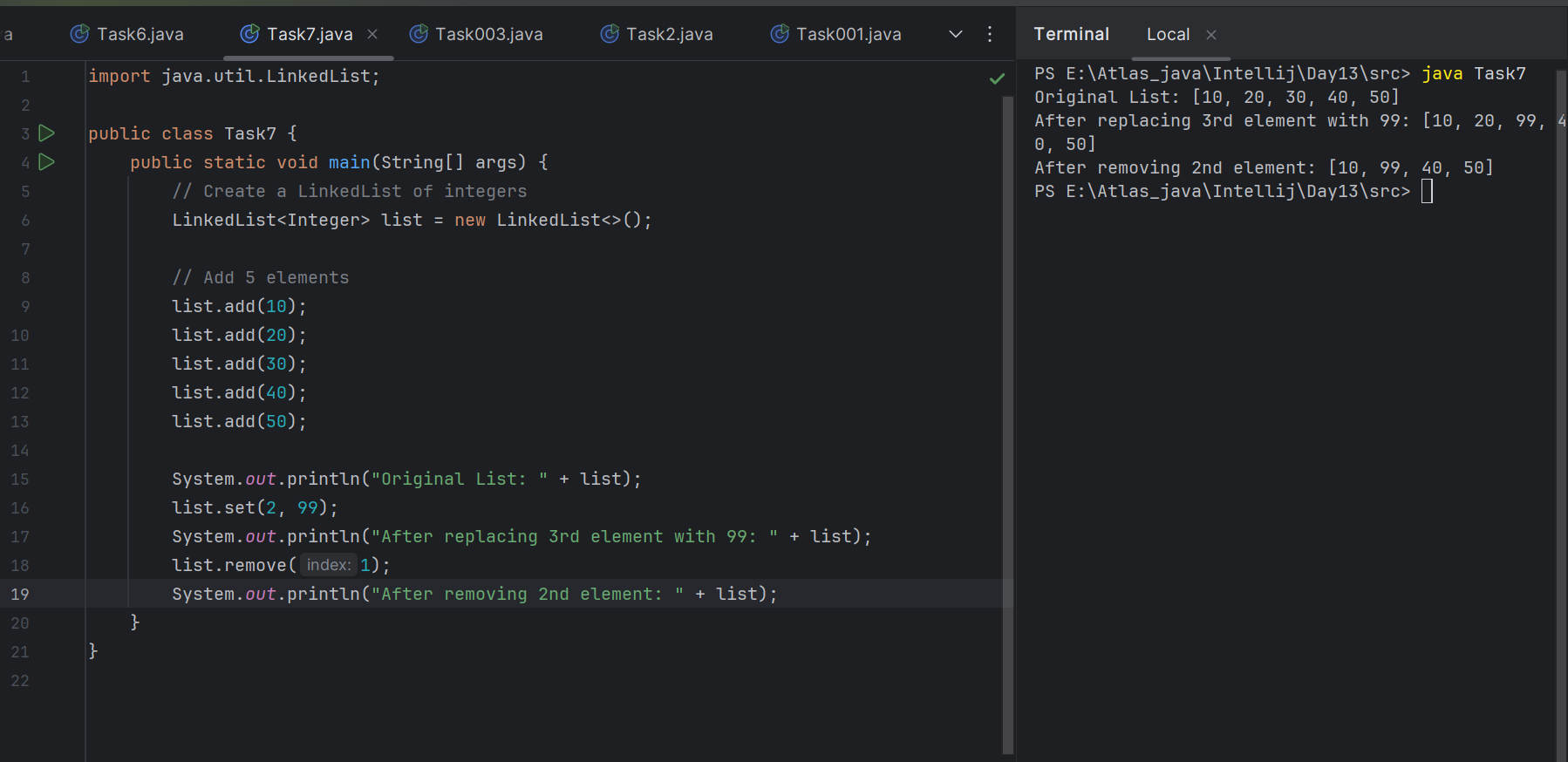
Wap to create linked list add 5 elements to it and replace 3 rd element with different value..



**Task 6:**

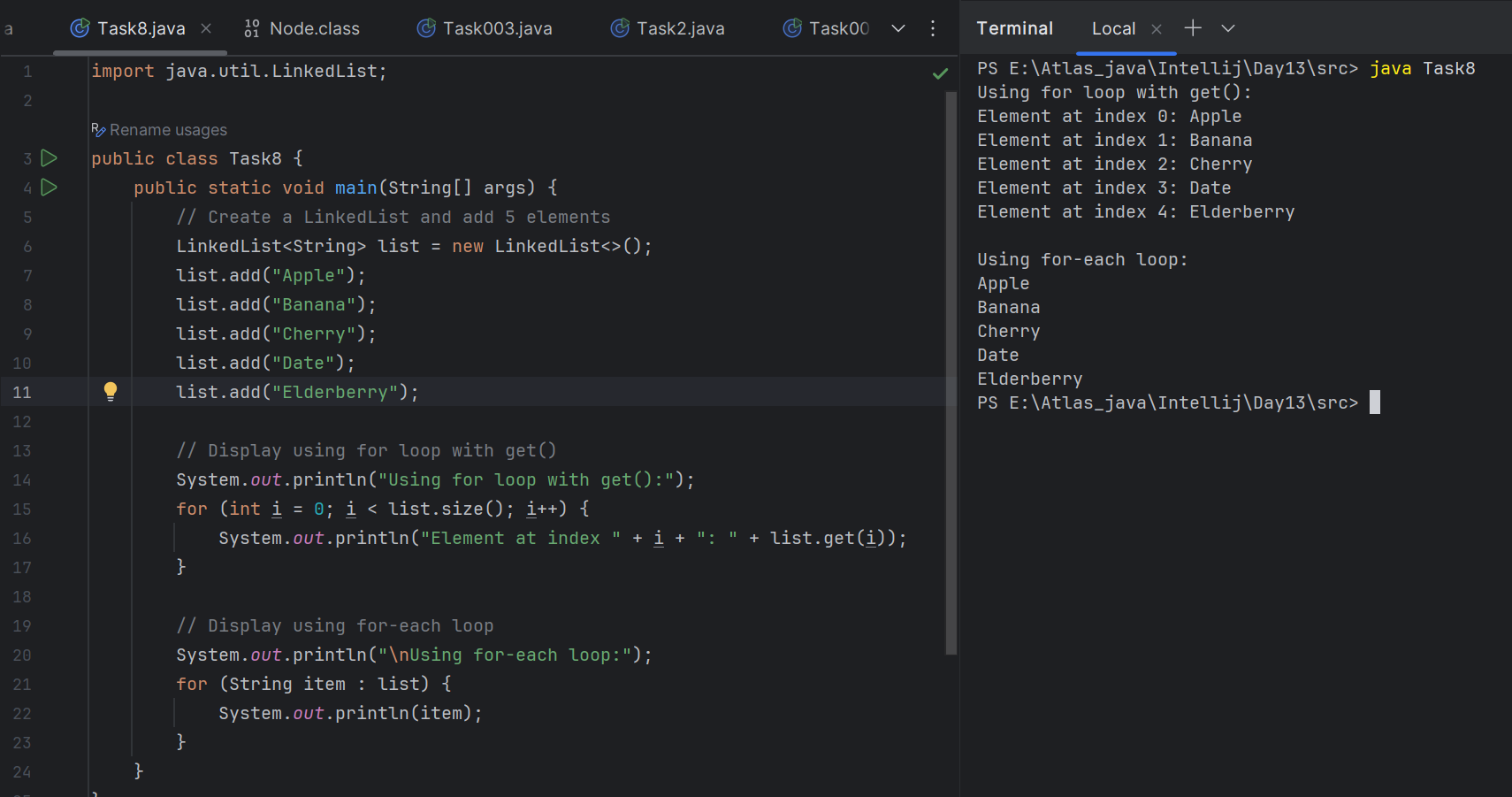
Wap to create a linked list to add 5 elements and remove any element and display

14.56 to 15.00



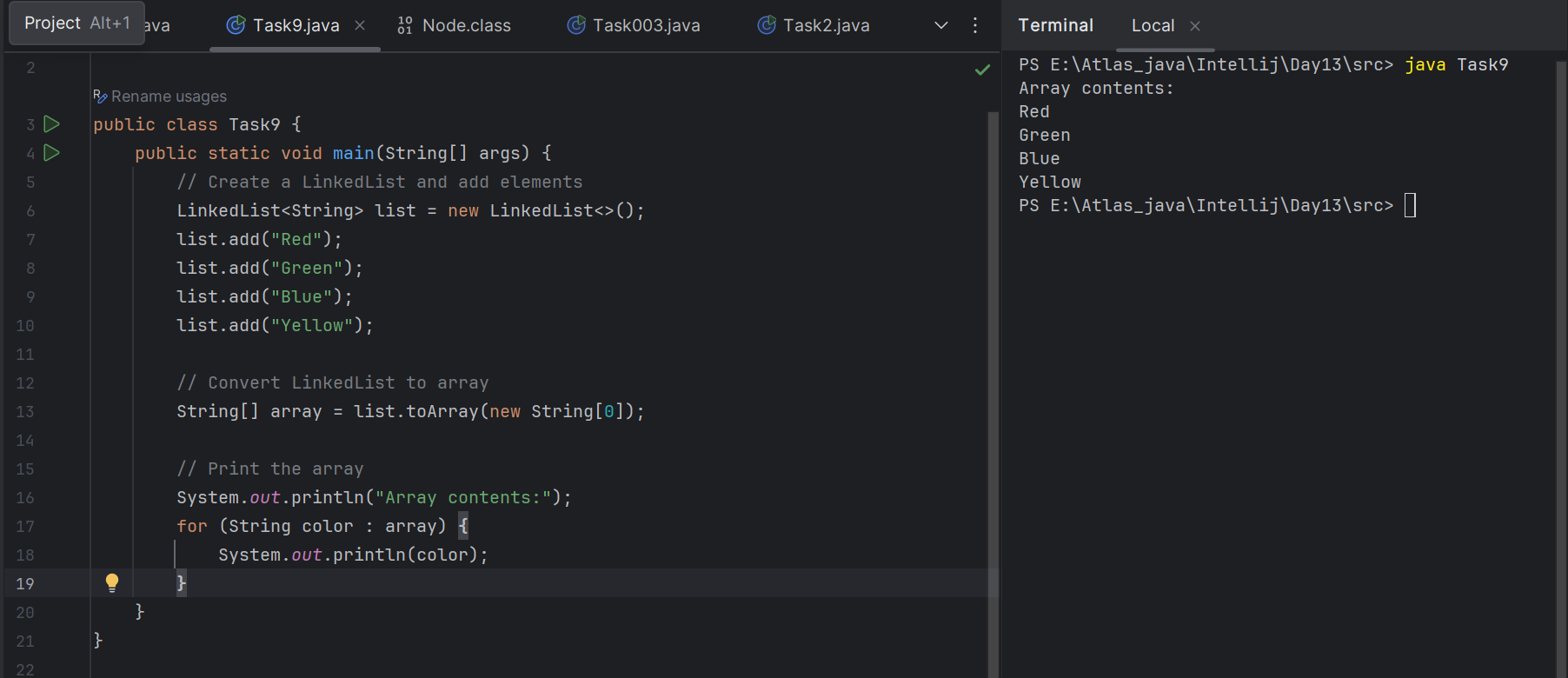
**Task 7:**

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



**Task 8:**

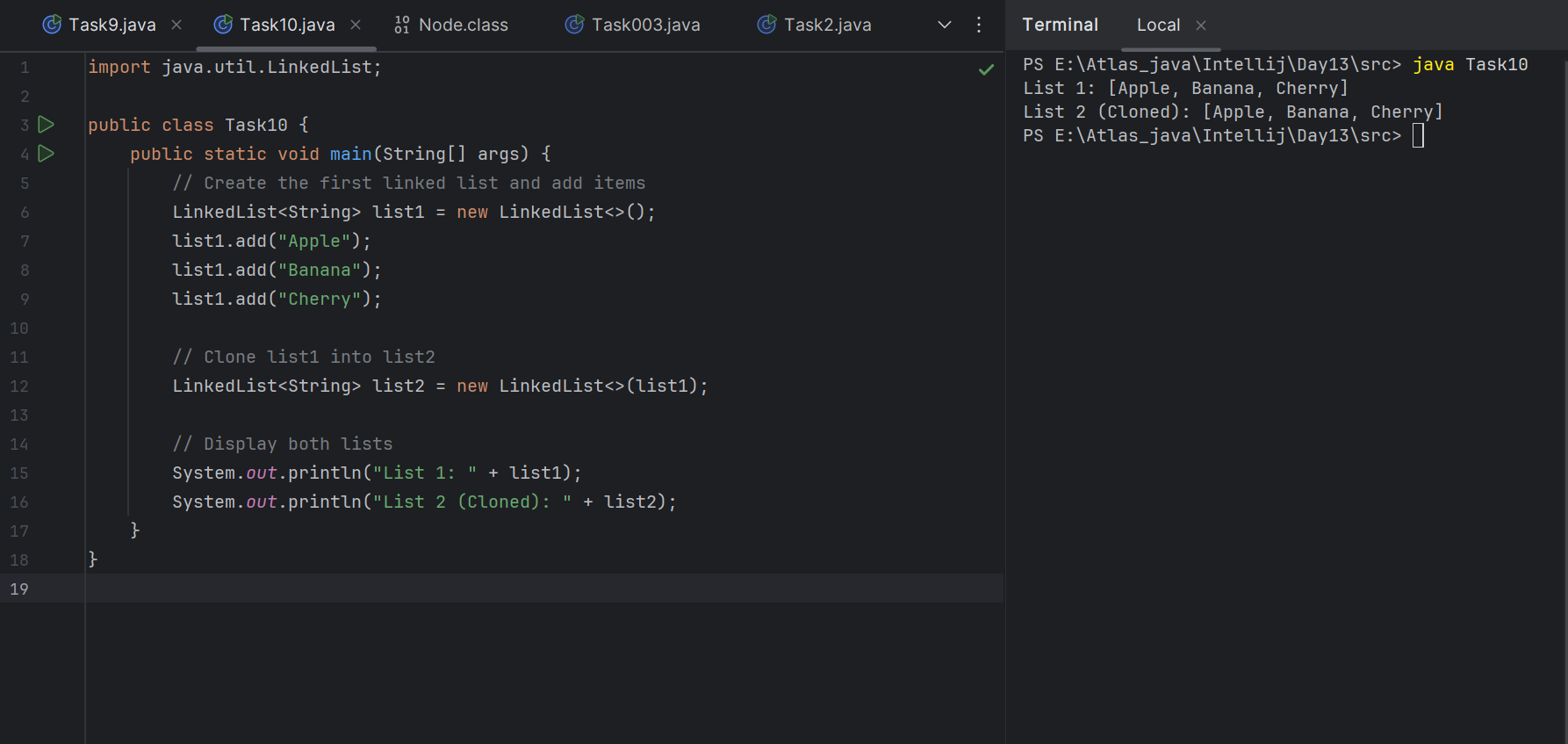
Createa a linked list and few items and convert it into an array



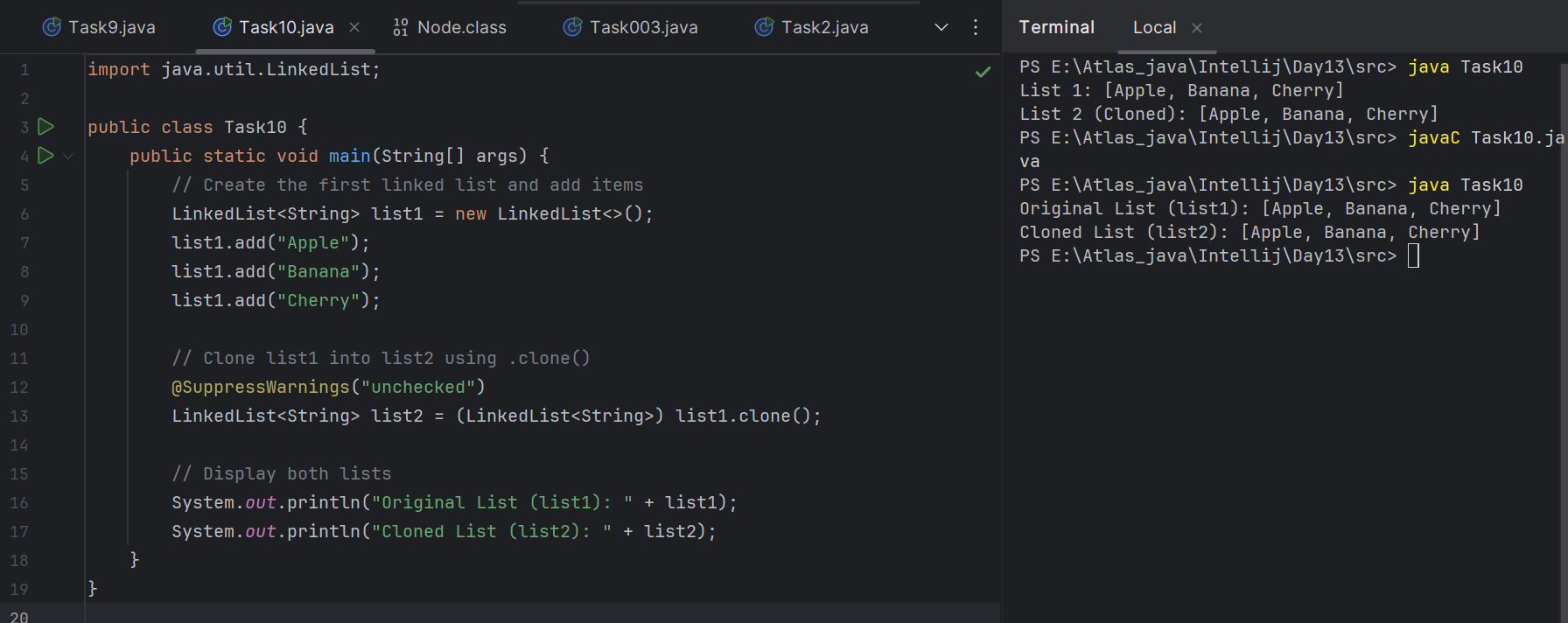
154.10 to 15.15

**Task 9:**

Createa alinked list add few items and clone the 1st linked list with the 2nd linked list



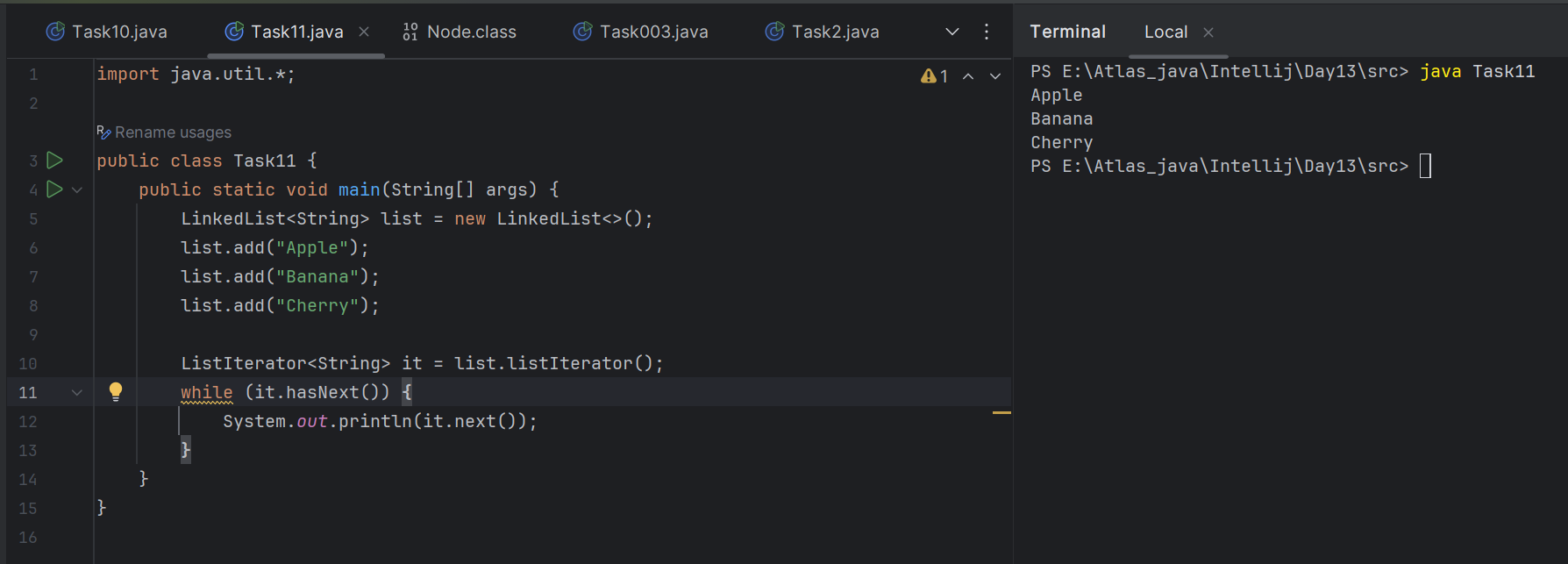
By using **.clone:**

****

15.16 to 15.20

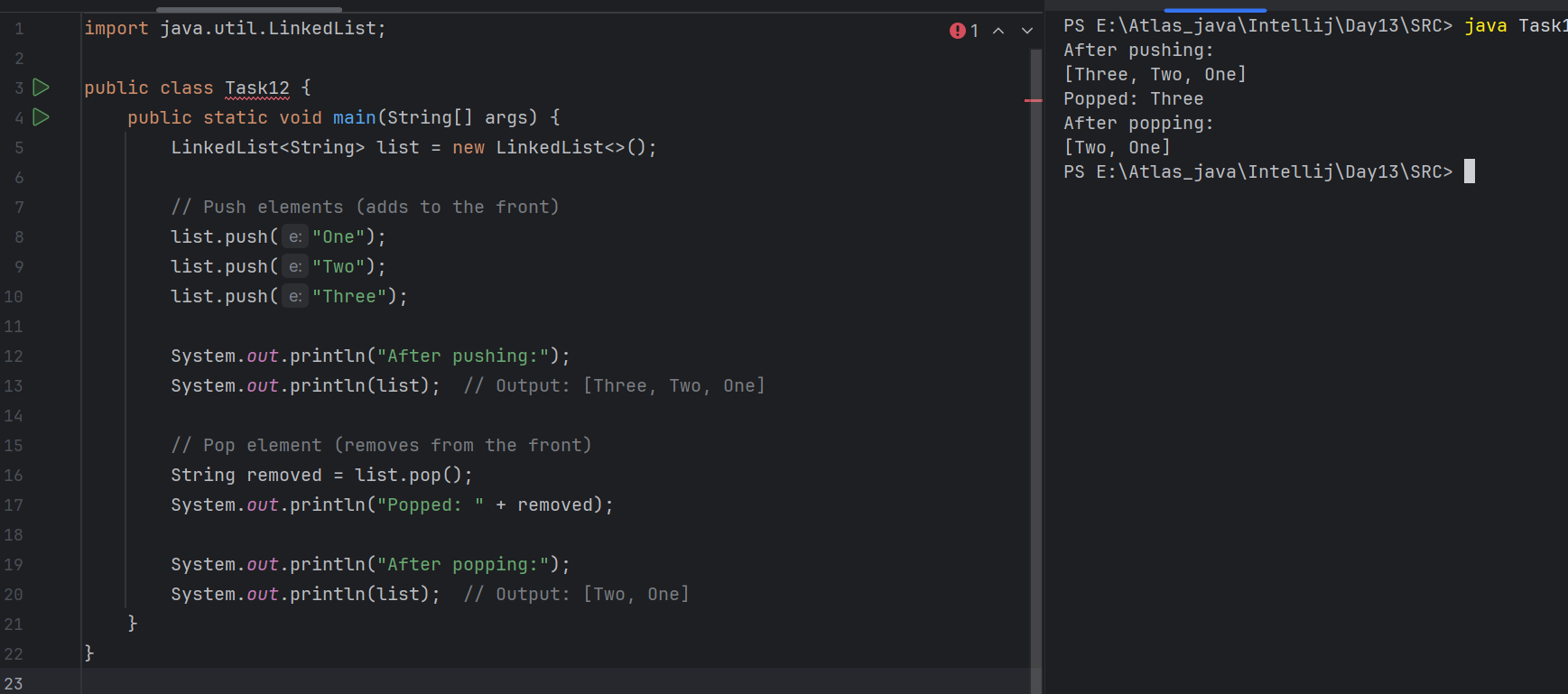
**Task 10:**

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



**Task 11:**

Create a linked list and use push and pop methods.



15.50 to 15.55

**Task 12:**

Difference between Iterator and splitIterator

**Iterator**: Used to traverse elements of a collection sequentially in a single thread.

**Spliterator**: Used to traverse and partition elements for parallel processing in streams.

**1. Iterator:**

| **Feature** | **Description** |
| --- | --- |
| **Introduced In** | Java 1.2 |
| **Traverses** | Sequentially (one by one) |
| **Direction** | Forward only |
| **Parallelism Support** | Not supported |
| **Method Used** | hasNext(), next(), remove() |
| **Use Case** | Simple iteration of collections |
| **Fail-fast** | Yes (throws ConcurrentModificationException if structure modified) |

**2. Spliterator:**

| **Feature** | **Description** |
| --- | --- |
| **Introduced In** | Java 8 |
| **Traverses** | Sequentially or in parallel |
| **Direction** | Forward only |
| **Parallelism Support** | Supported using trySplit() |
| **Method Used** | tryAdvance(), forEachRemaining(), trySplit() |
| **Use Case** | Efficient stream-based and parallel iteration |
| **Fail-fast** | Yes (similar to Iterator) |

Task 13:

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");

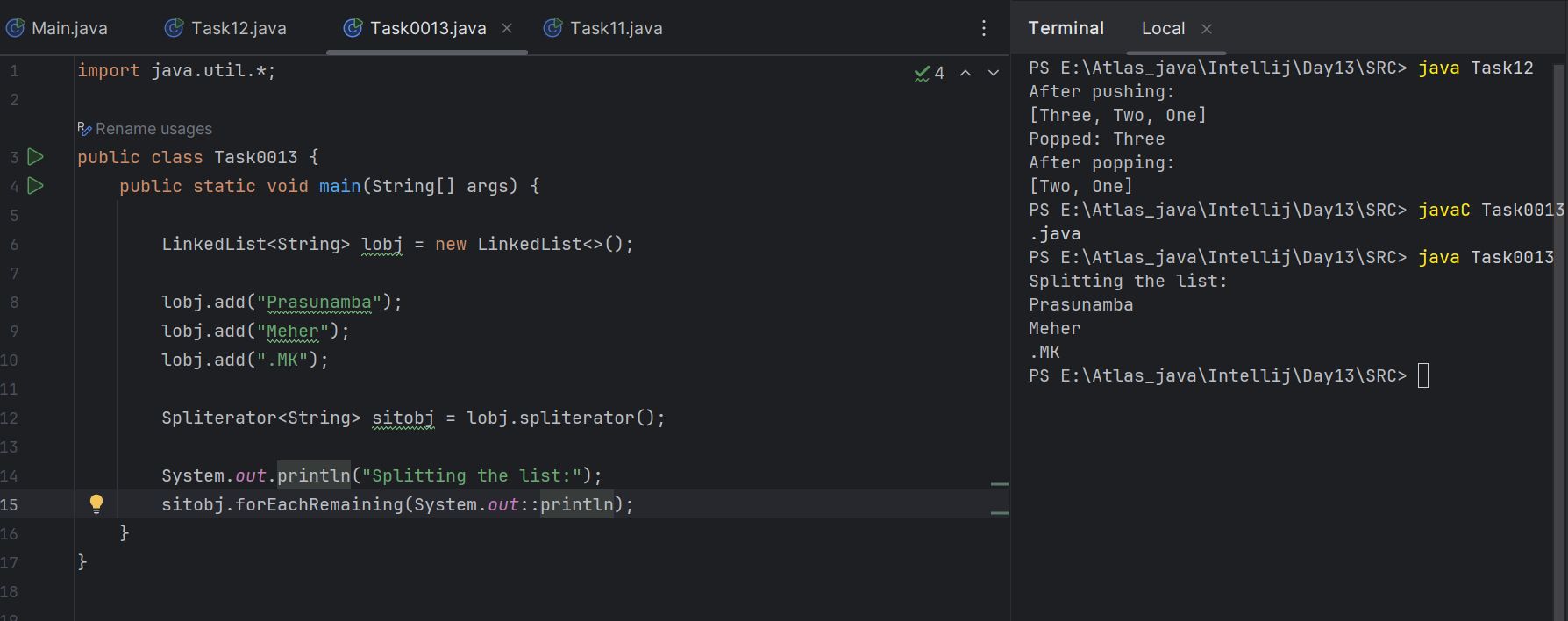
Spliterator<String> sitobj = lobj.spliterator();

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

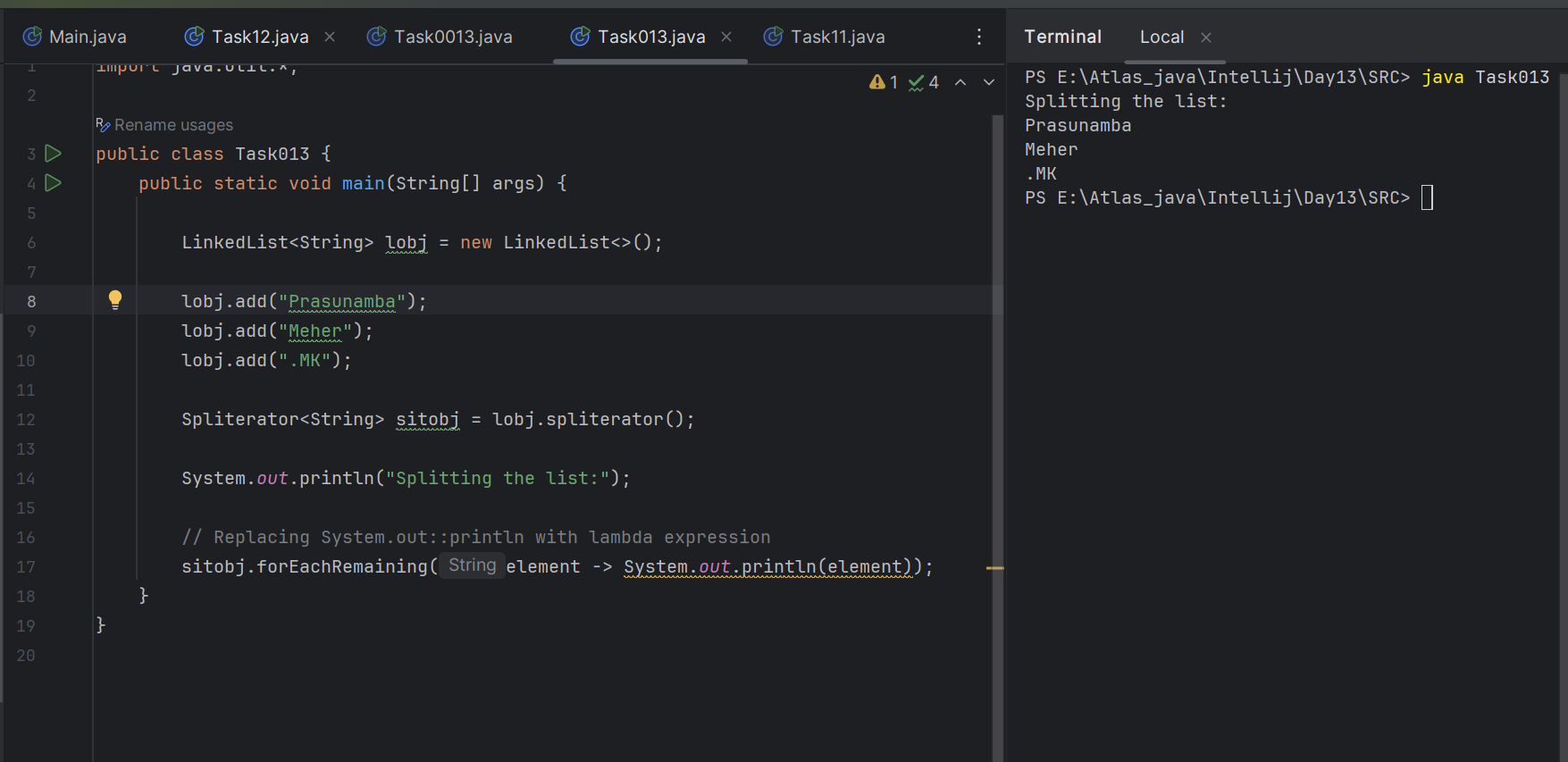
sitobj.forEachRemaining(System.out::println);

}

}

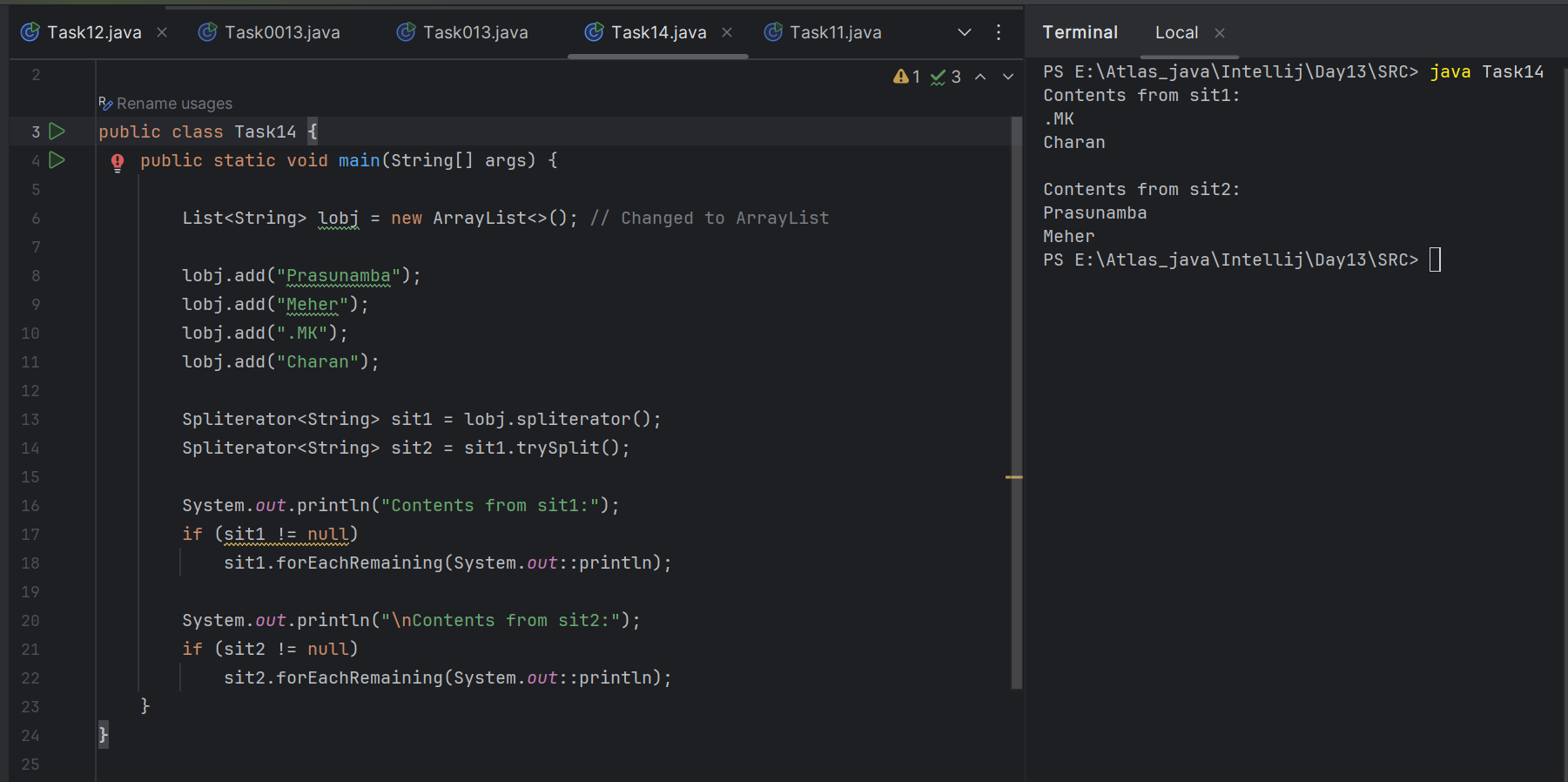


Now Can you change it to sout()... and see ..



**Task 14:**

Createa a linkedlist and display items into 2 lists using split iterator



**Task 15:**

What do you understand by a pointer?

A **pointer** is a variable that **stores the memory address** of another variable.

A pointer doesn’t hold data directly, but it tells you **where the data is stored in memory**.

Ex:

int x = 10;

int\* p = &x;

**Task 16:**

Difference between \* and & in pointers?

### & (Address-of Operator)

* Meaning: Gets the address of a variable.
* Used to create a pointer.

Exl:

int x = 10;

int\* p = &x;

### \* (Dereference Operator)

* Meaning: Accesses the value at the address stored by a pointer.
* Used to get or change the value at the pointer’s address.

Exl:

int x = 10;

int\* p = &x;

cout << \*p;

**Task 17:**

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

#include <iostream>

using namespace std;

int main() {

int number = 42;

int\* ptr = &number; // Pointer to the variable 'number'

// Display value using variable and pointer

cout << "Value of number: " << number << endl;

cout << "Address of number: " << &number << endl;

cout << "Value stored in pointer: " << ptr << endl;

cout << "Value pointed to by pointer: " << \*ptr << endl;

// Modify value using pointer

\*ptr = 100;

cout << "New value of number after modification through pointer: " << number << endl;

return 0;

}

Op:

Value of number: 42

Address of number: 0x61ff08 // (This is just an example address; it will vary)

Value stored in pointer: 0x61ff08 // Same address as above

Value pointed to by pointer: 42

New value of number after modification through pointer: 100

**Task 18:**

Wap to create a doubly linked list

class Node {

int data;

Node prev;

Node next;

Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

public class Task18 {

Node head;

// Insert at end

public void insertEnd(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

newNode.prev = temp;

}

// Display forward

public void displayForward() {

Node temp = head;

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

while (temp != null) {

System.out.print(temp.data + " <-> ");

temp = temp.next;

}

System.out.println("null");

}

// Display backward

public void displayBackward() {

Node temp = head;

if (temp == null) return;

// Go to last node

while (temp.next != null) {

temp = temp.next;

}

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

while (temp != null) {

System.out.print(temp.data + " <-> ");

temp = temp.prev;

}

System.out.println("null");

}

// Main method

public static void main(String[] args) {

Task18 dll = new Task18();

dll.insertEnd(10);

dll.insertEnd(20);

dll.insertEnd(30);

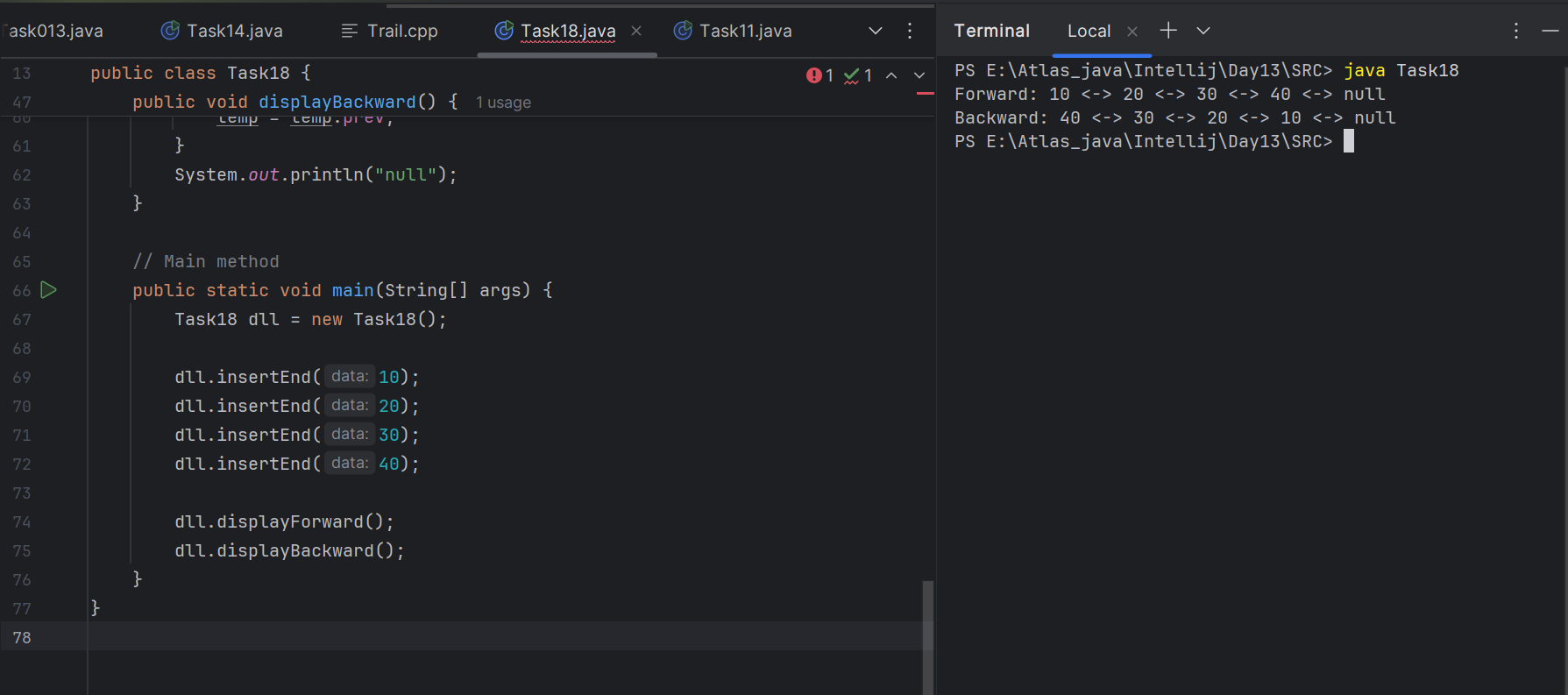
dll.insertEnd(40);

dll.displayForward();

dll.displayBackward();

}

}



10 min 18.00 to

===============================================================================================

**Home Task:**

**What the advantages and disadvantages of the Linked list:**

## **Advantages of Linked List:**

1. **Dynamic Size** Can grow and shrink during runtime, unlike arrays which have fixed size.
2. **Efficient Insertions/Deletions** Adding or removing elements (especially at the beginning or middle) is fast—no shifting like in arrays.
3. **Memory Efficient (for large insertions/deletions)** Doesn’t waste space for unused indexes, unlike arrays with reserved but unused slots.
4. **No Memory Waste Due to Resizing** No need to resize or copy the list when it grows (unlike dynamic arrays like ArrayList).
5. **Good for Implementing Other Data Structures** Useful for stacks, queues, graphs, etc.

## **Disadvantages of Linked List:**

1. **Extra Memory Usage** Each node stores not just data, but also a pointer (next), using more memory.
2. **Slow Access (No Random Access)** Can't directly access the middle; need to traverse from head every time.
3. **Not Cache-Friendly** Since nodes are scattered in memory, it's slower compared to arrays stored in continuous memory.
4. **More Complex to Implement** Needs careful pointer management; can lead to bugs like memory leaks or null pointer errors.
5. **Reverse Traversal Not Possible (in singly linked list)** You can't move backward unless you use a **doubly linked list**.

**Applications of linkedlist:**

### **1**. Dynamic Memory Allocation

* Linked lists allow memory to be allocated at runtime, making them ideal for implementing data structures that grow or shrink.
* Used in custom memory management systems (like heaps or free lists).

### 2. Stacks and Queues

* Stacks (LIFO) can be implemented using singly linked lists by inserting/removing at the head.
* Queues (FIFO) use linked lists with head and tail pointers for efficient enqueue/dequeue.

### 3. Graphs and Trees

* Linked lists are used to represent:  
  + Adjacency lists in graphs
  + Tree structures (e.g., binary trees, heaps)

### 4. Polynomial Arithmetic

* Polynomials can be represented using linked lists where each node holds a coefficient and exponent.

### 5. Implementing Hash Tables

* Linked lists handle collisions in hash tables via chaining, where each bucket contains a linked list of entries.

### 6. Music/Video Playlist Management

* Doubly linked lists are ideal for navigating forward and backward through songs/videos.

### 7. Undo/Redo Functionality

* Applications like word processors use doubly linked lists to track state changes for undo/redo operations.

### 8. Navigation Systems

* Linked lists are useful in implementing pathfinding systems, where each node points to the next location.

9. Browser History

* A doubly linked list allows back and forward navigation between visited web pages.

### 10. Job Scheduling / Task Queues

* OS-level task scheduling often uses linked lists to manage job/task queues efficiently.

========================================================================================================================================================

**Info Box**

========================================================================================================================================================

========================================================================================================================================================