**day13\_107856406\_dsdipt\_sudipto\_26june2025**

**Employee Code:** 107856406

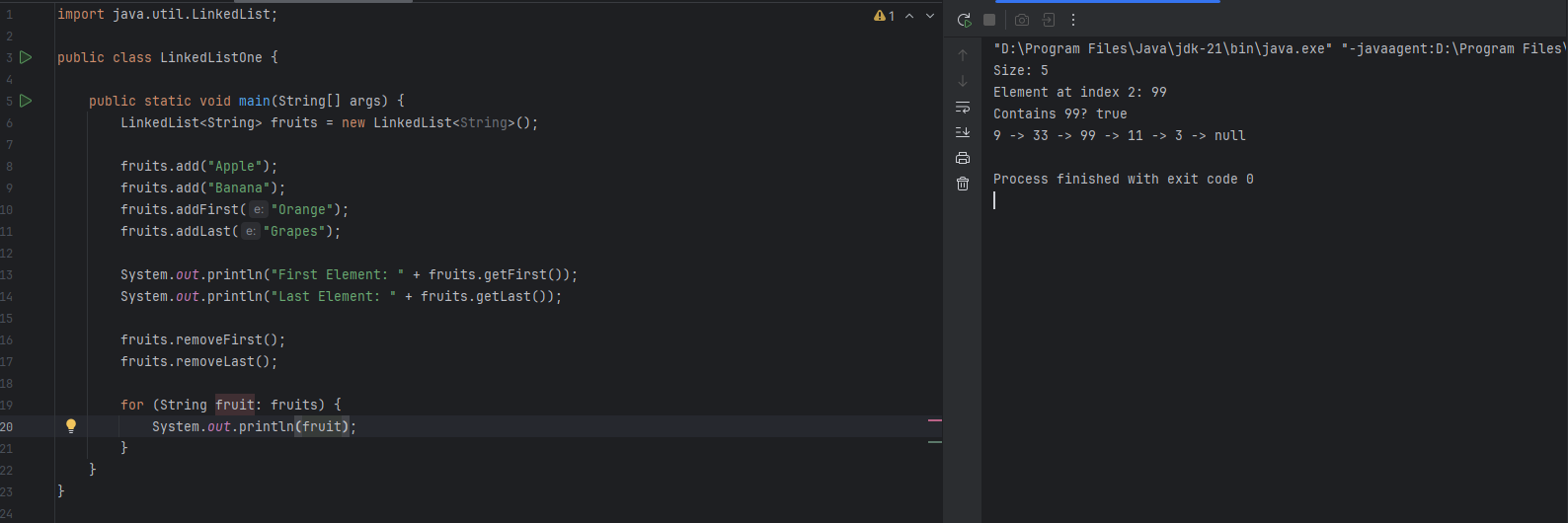
**Login ID:** dsdipt

**Email :** dsdipt@amazon.com

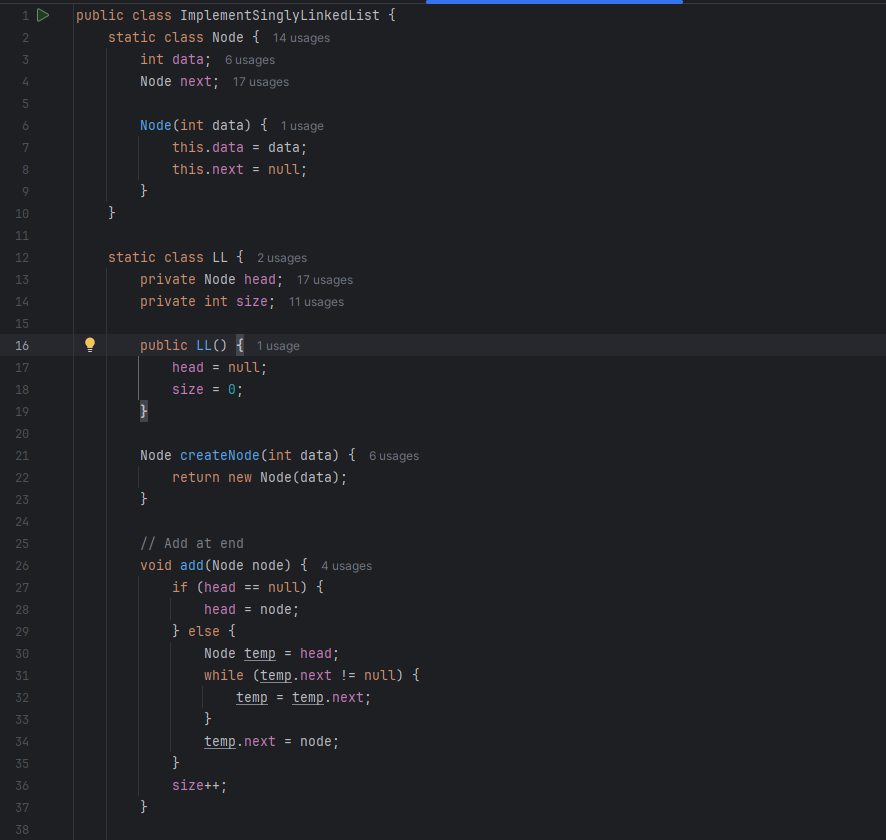
**Name:** Sudipto Das

**Date:** 26 June 2025 (Day 13)

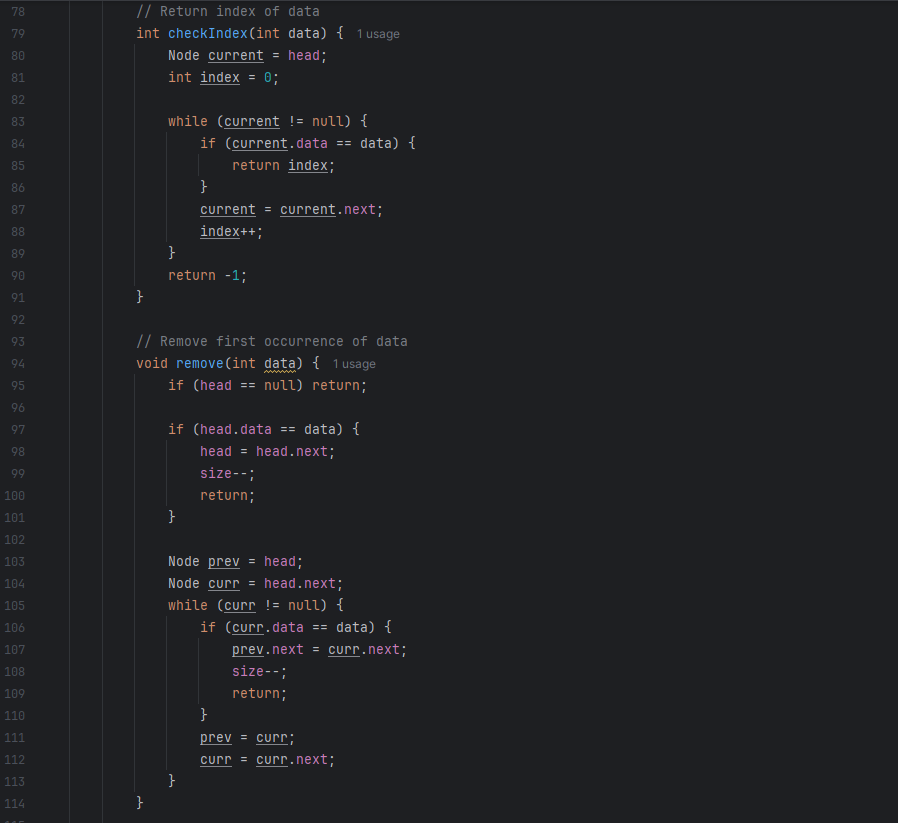
### ***Task 1: Demonstrate basic operations on a LinkedList***



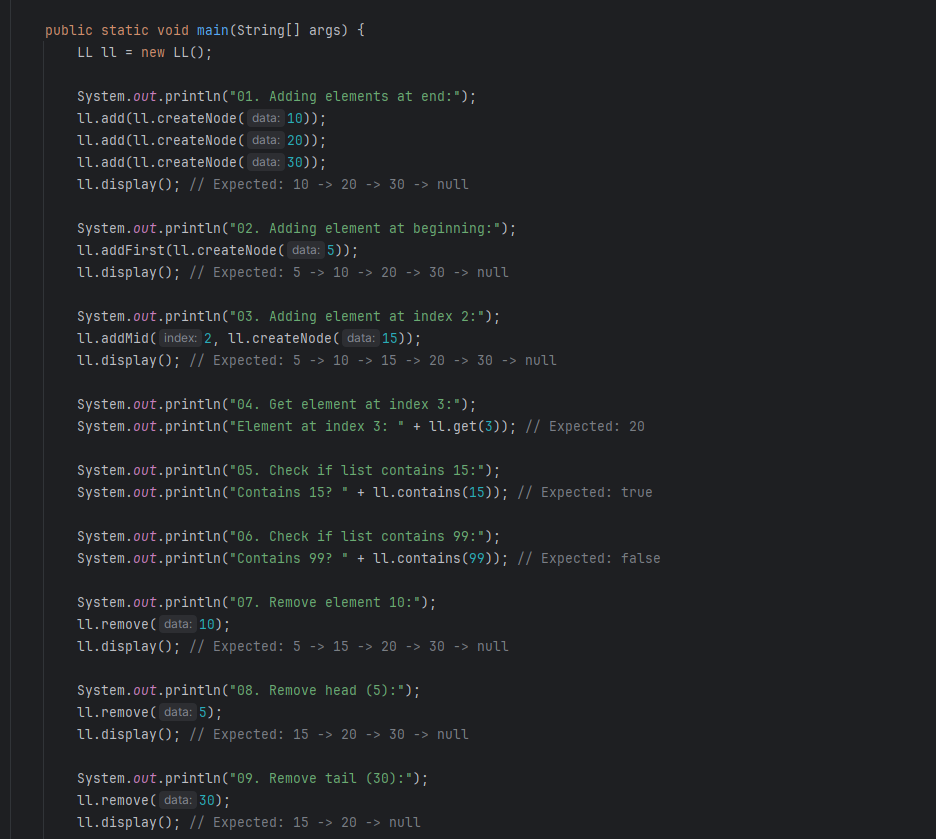
### ***Task 2: Create a custom Node and implement Singly LinkedList methods.***

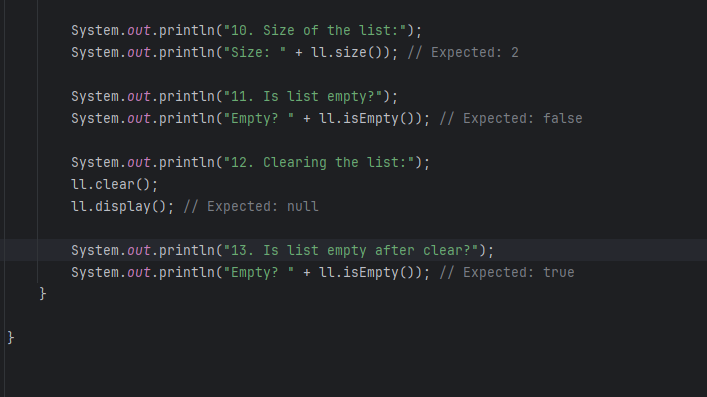


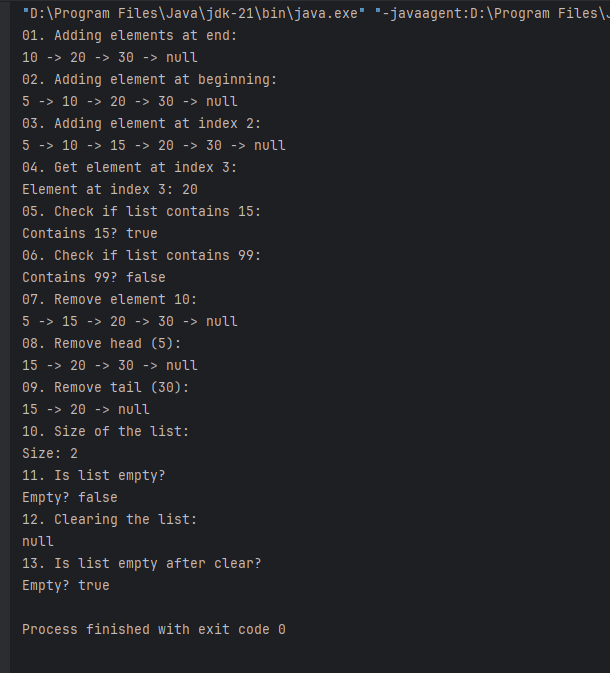












### ***Task 3: Methods of LinkedList in Java***

**🔹 Adding Elements**

* add(E e) – Add element at the end
* add(int index, E element) – Add at specific index
* addFirst(E e) – Add at the beginning
* addLast(E e) – Add at the end

**🔹 Removing Elements**

* remove() – Remove first element
* remove(int index) – Remove element at given index
* remove(Object o) – Remove first occurrence of element
* removeFirst() – Remove first element
* removeLast() – Remove last element
* clear() – Remove all elements

**🔹 Accessing Elements**

* get(int index) – Get element at index
* getFirst() – Get first element
* getLast() – Get last element
* peek() – View first element without removing
* peekFirst() – Same as peek()
* peekLast() – View last element without removing

**🔹 Updating Elements**

* set(int index, E element) – Replace element at index

**🔹 Searching**

* contains(Object o) – Check if element exists
* indexOf(Object o) – Get index of first occurrence
* lastIndexOf(Object o) – Get index of last occurrence

**🔹 Size & Check**

* size() – Get number of elements
* isEmpty() – Check if list is empty

**🔹 Stack Operations**

* push(E e) – Push element to front (like stack)
* pop() – Pop element from front

**🔹 Queue Operations**

* offer(E e) – Add to end (like queue)
* poll() – Remove from front
* offerFirst(E e) – Add to front
* offerLast(E e) – Add to end
* pollFirst() – Remove from front
* pollLast() – Remove from end

### ***Task 4: What are the operations of data structures***

**🔹 1. Insertion**

* Add a new element into the data structure.
* Example: add() in LinkedList, push() in Stack.

**🔹 2. Deletion**

* Remove an element from the data structure.
* Example: remove() in List, pop() in Stack, dequeue() in Queue.

**🔹 3. Traversal**

* Access each element of the data structure **one by one**.
* Example: Looping through an array or list.

**🔹 4. Searching**

* Find the **location** or **presence** of an element.
* Example: contains() or binary search.

**🔹 5. Sorting**

* Arrange the elements in **ascending or descending order**.
* Example: Bubble sort, Merge sort, Quick sort, etc.

**🔹 6. Updating**

* Modify the value of an existing element.
* Example: set(index, newValue) in a list.

**🔹 7. Accessing**

* Directly retrieve a specific element (often by index or key).
* Example: get(index) in an array or list, or map.get(key) in a HashMap.

**🔹 8. Merging**

* Combine two or more data structures into one.
* Example: Merging two sorted arrays.

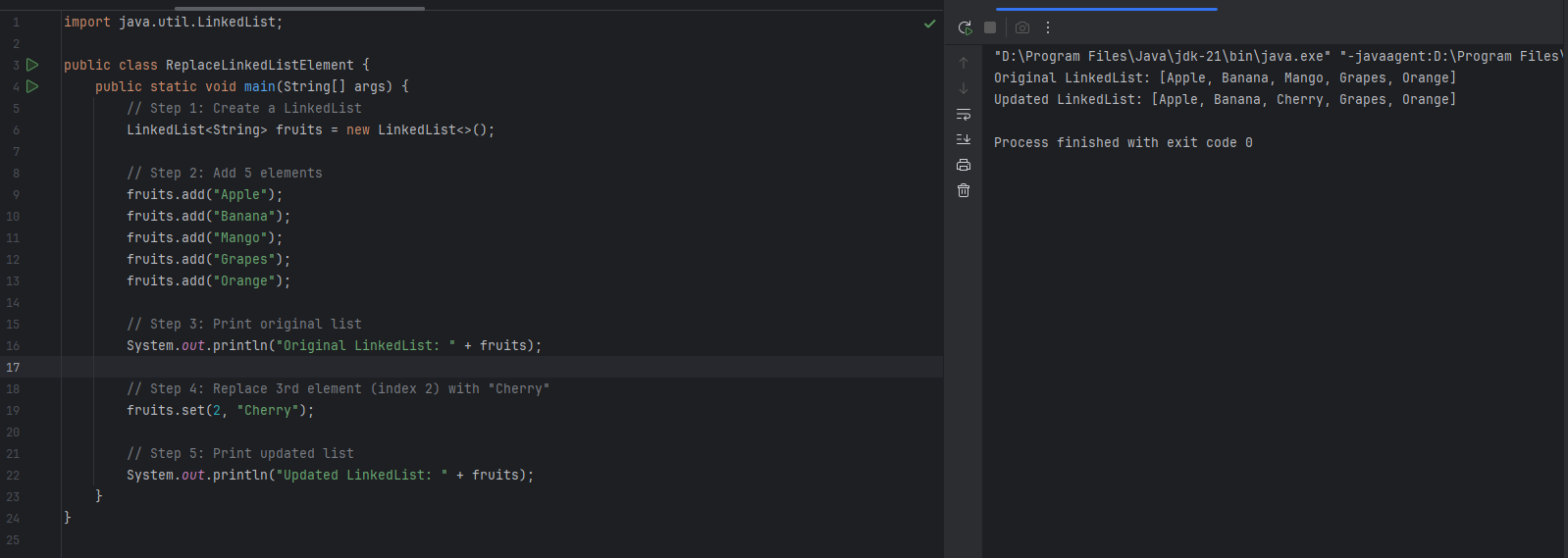
**🔹 9. Splitting**

* Break one data structure into two or more.
* Example: Splitting a list into sublists.

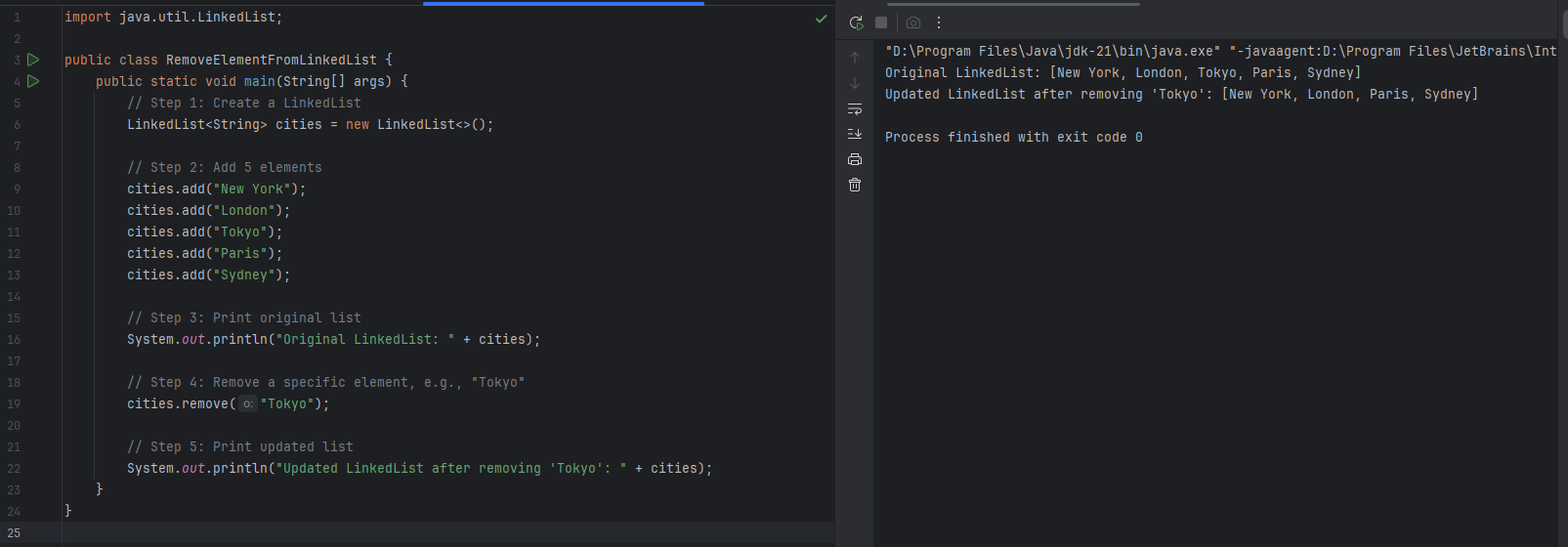
**🔹 10. Filtering (Optional)**

* Return only elements that meet a specific condition.
* Example: Find all even numbers in a list.

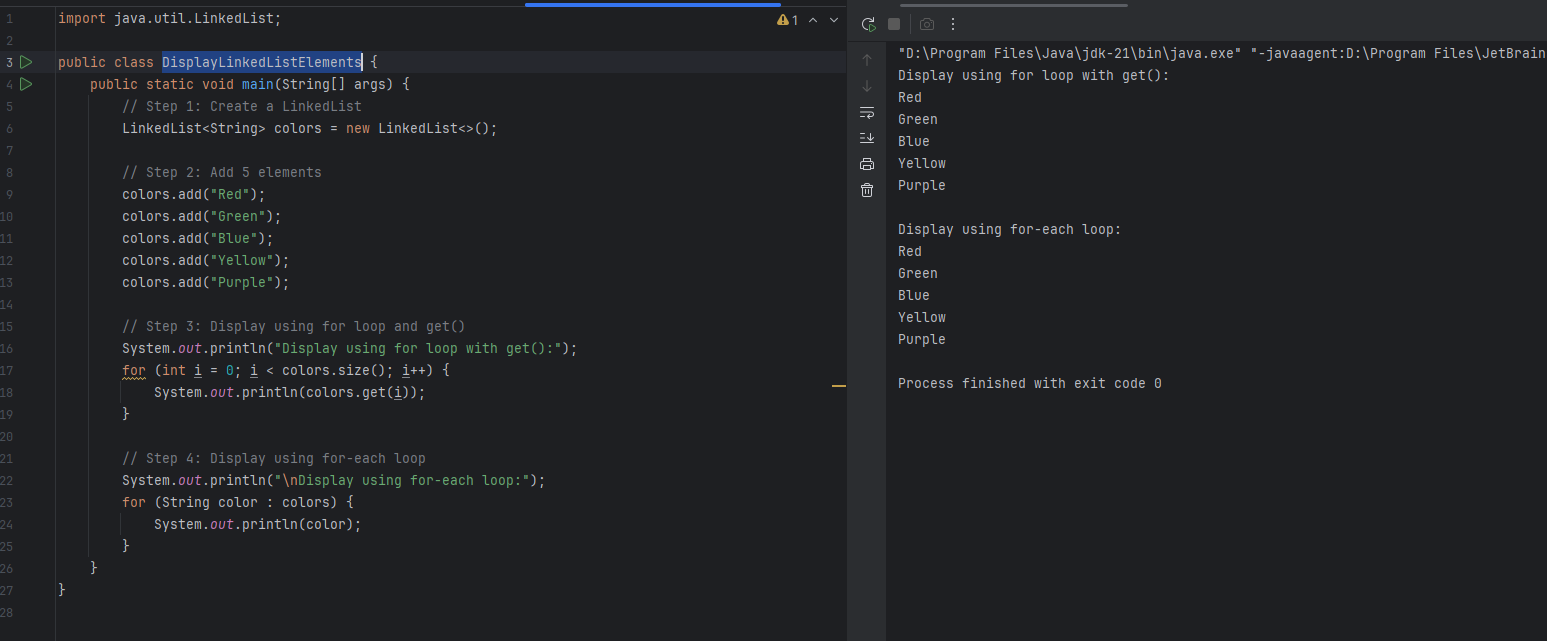
### ***Task 5: Update an Element at a Specific Index in a LinkedList in Java***



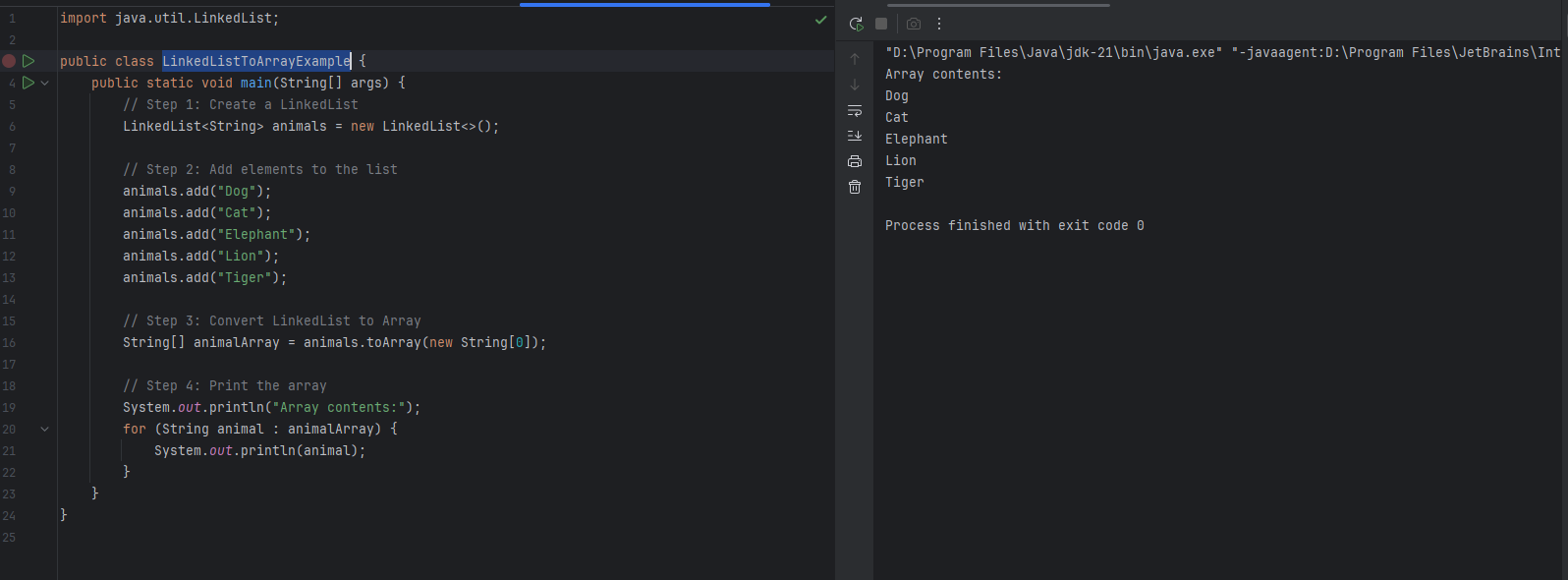
### ***Task 6: Removing an Element from a LinkedList in Java***



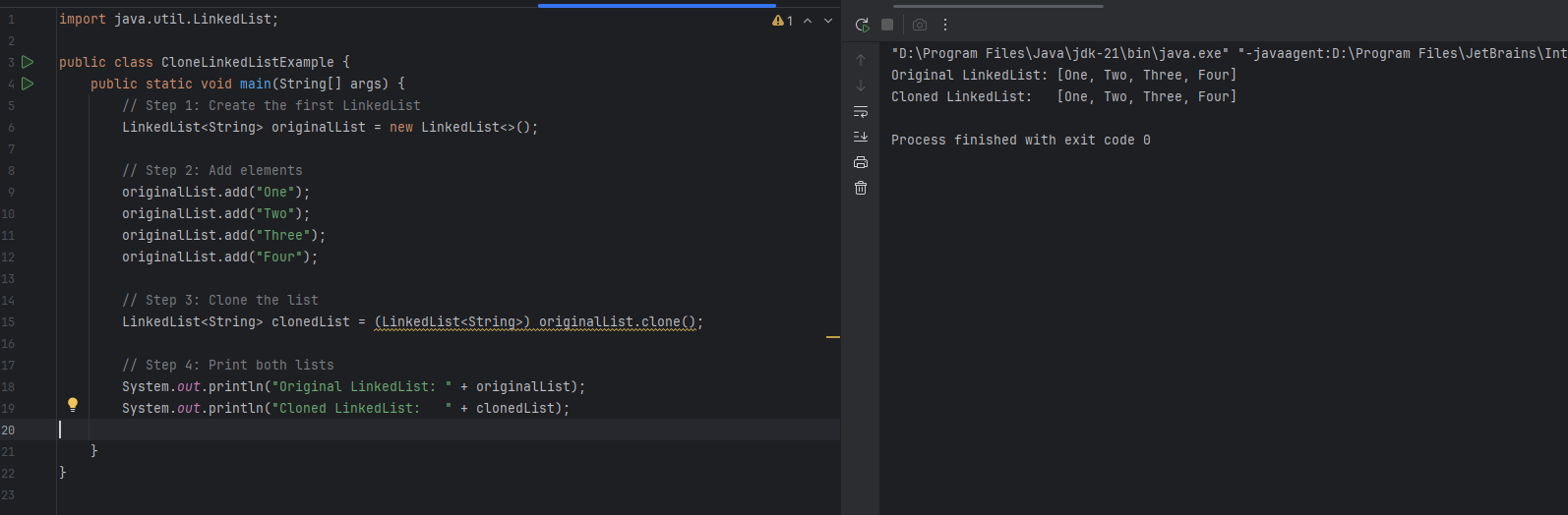
### ***Task 7: Displaying Elements of a LinkedList Using For & For Each Loop in Java***



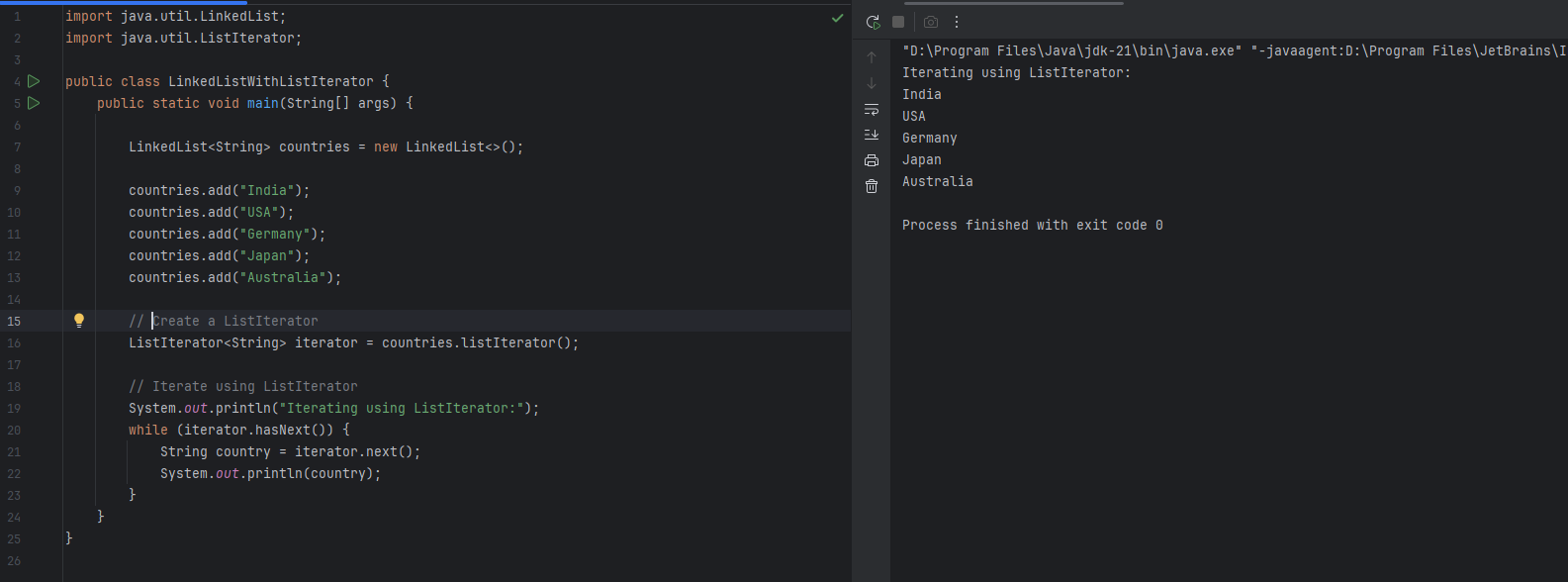
### ***Task 8: Convert a LinkedList to an Array in Java***



### ***Task 9: Cloning a LinkedList in Java***



### ***Task 10: Using ListIterator to Traverse a LinkedList in Java***



### ***Task 11: Stack Using LinkedList with push() and pop() in Java***

### ***Task 12: Stack Using LinkedList with push() and pop() in Java***

**Purpose**

* **Iterator**: For **sequential** traversal of collections.
* **Spliterator**: For **parallel** and sequential traversal, with the ability to **split** the data.

**Traversal Direction**

* **Iterator**: Only **forward** traversal.
* **Spliterator**: Only **forward**, but can **split** data for parallel processing.

**Parallel Processing**

* **Iterator**: **Not suitable** for parallel processing.
* **Spliterator**: **Designed for parallelism**, works well with parallelStream().

**Key Methods**

* **Iterator**:
  + hasNext()
  + next()
  + remove()
* **Spliterator**:
  + trySplit()
  + tryAdvance()
  + forEachRemaining()

**Element Removal**

* **Iterator**: Supports remove().
* **Spliterator**: Does **not** support element removal.

**Use With Streams**

* **Iterator**: Not compatible with Stream API.
* **Spliterator**: Backbone of Java Stream API (used internally).

**Performance Use Case**

* **Iterator**: Good for **simple, small** tasks.
* **Spliterator**: Best for **large collections** and **parallel execution**.

### ***Task 13: Traverse a LinkedList using Spliterator***

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);

}

}

### ***Task 14: Create a linked list and display items into 2 lists using split iterator***

import java.util.LinkedList;

import java.util.Spliterator;

public class Task0014\_DS\_Linkedlist\_SplitItr2Lists {

public static void main(String[] args) {

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

llobj.add("Prasunamba");

llobj.add("Meher");

llobj.add(".MK");

llobj.add("MP");

Spliterator<String> itobj1 = llobj.spliterator();

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

System.out.println("spliterator 1");

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

System.out.println("spliterator 2");

while( itobj2.tryAdvance( (n) -> { System.out.println(n); } ) );

}

}

***Task 15: What do you understand by a pointer?***

**Example (C Language):**

int num = 10;

int\* ptr = &num;

printf("%d", \*ptr); // Output: 10

* num is a regular integer.
* ptr is a pointer to num.
* &num gets the address of num.
* \*ptr is called dereferencing, which accesses the value at that address.

### ***Task 16: Difference Between \* and & in Pointers***

**1. & (Address-of Operator)**

* Used to get the **memory address** of a variable.
* Pronounced as **"address of"**.
* Returns the address in memory where the variable is stored.

int a = 5;

int\* p = &a;

* &a gets the **address** of variable a.
* That address is assigned to pointer p.

**2. \* (Dereference Operator)**

* Used to **access the value** stored at a memory address.
* Pronounced as **"value at address"**.
* When used in a declaration, it defines a **pointer** variable.
* When used with a pointer, it **dereferences** it to access the actual value.

int a = 5;

int\* p = &a;

printf("%d", \*p); // Output: 5

* \*p gives the **value at the address** stored in p, which is 5.

***Task 17: Simple C Program to Demonstrate the Use of Pointers***

#include <stdio.h>

int main() {

int num = 10; // Step 1: Declare an integer

int\* ptr = &num; // Step 2: Create a pointer to that integer

// Step 3: Display values

printf("Value of num : %d\n", num); // Prints 10

printf("Address of num : %p\n", &num); // Prints address

printf("Value of ptr : %p\n", ptr); // Should match &num

printf("Value at \*ptr : %d\n", \*ptr); // Dereference pointer, prints 10

// Step 4: Change value using pointer

\*ptr = 25;

printf("\nValue of num after changing through pointer: %d\n", num); // Prints 25

return 0;

}

***Task 18: Create a custom Node and implement Doubly LinkedList methods.***

