Day 14 , EMP\_ID : 1129116146

Task1

public class Task1{

static class Node {

int data;

Node next;

Node(int data) { this.data = data; }

}

private Node head; // first node

private Node tail; // last node

// add: append data at the end

public void add(int data) {

Node newNode = new Node(data);

if (head == null) {

head = tail = newNode;

} else {

tail.next = newNode;

tail = newNode;

}

}

// traverse: print all nodes

public void traverse() {

Node curr = head;

while (curr != null) {

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

curr = curr.next;

}

System.out.println();

}

public static void main(String[] args) {

SimpleLinkedList list = new SimpleLinkedList();

list.add(5);

list.add(10);

list.add(15);

list.traverse(); // ➞ 5 10 15

}

}

Task 2

## **What is Traversing a Linked List?**

**Traversing** means **visiting each element** of the linked list **one by one**, from the **start (head)** to the **end**.

Task 3

public class Task3 {

static class Node {

int data;

Node next;

Node(int data) { this.data = data; }

}

private Node head; // first node

private Node tail; // last node

public void add(int data) {

Node newNode = new Node(data);

if (head == null) {

head = tail = newNode;

tail.next = head; // points to itself, circularly

} else {

tail.next = newNode; // old tail → newNode

tail = newNode;

tail.next = head; // new tail → head (circular)

}

}

// traverse: print all nodes once, starting from head

public void traverse() {

if (head == null) {

System.*out*.println("(empty)");

return;

}

Node curr = head;

do {

System.*out*.print(curr.data + " ");

curr = curr.next;

} while (curr != head);

System.*out*.println();

}

public static void main(String[] args) {

Task3 list = new Task3();

list.add(5);

list.add(10);

list.add(15);

list.traverse(); // ➞ 5 10 15

}

}

Task4 and 5 in 1

package StackQuestions;

import java.util.ArrayList;

import java.util.NoSuchElementException;

import java.util.Stack;

class Ds\_Stack\_Push {

public static void main(String[] args) {

Stack<String> names = new Stack<>();

names.push("Prasunamba");

names.push("Meher");

names.push(".MK");

System.*out*.println("Stack before pop: " + names);

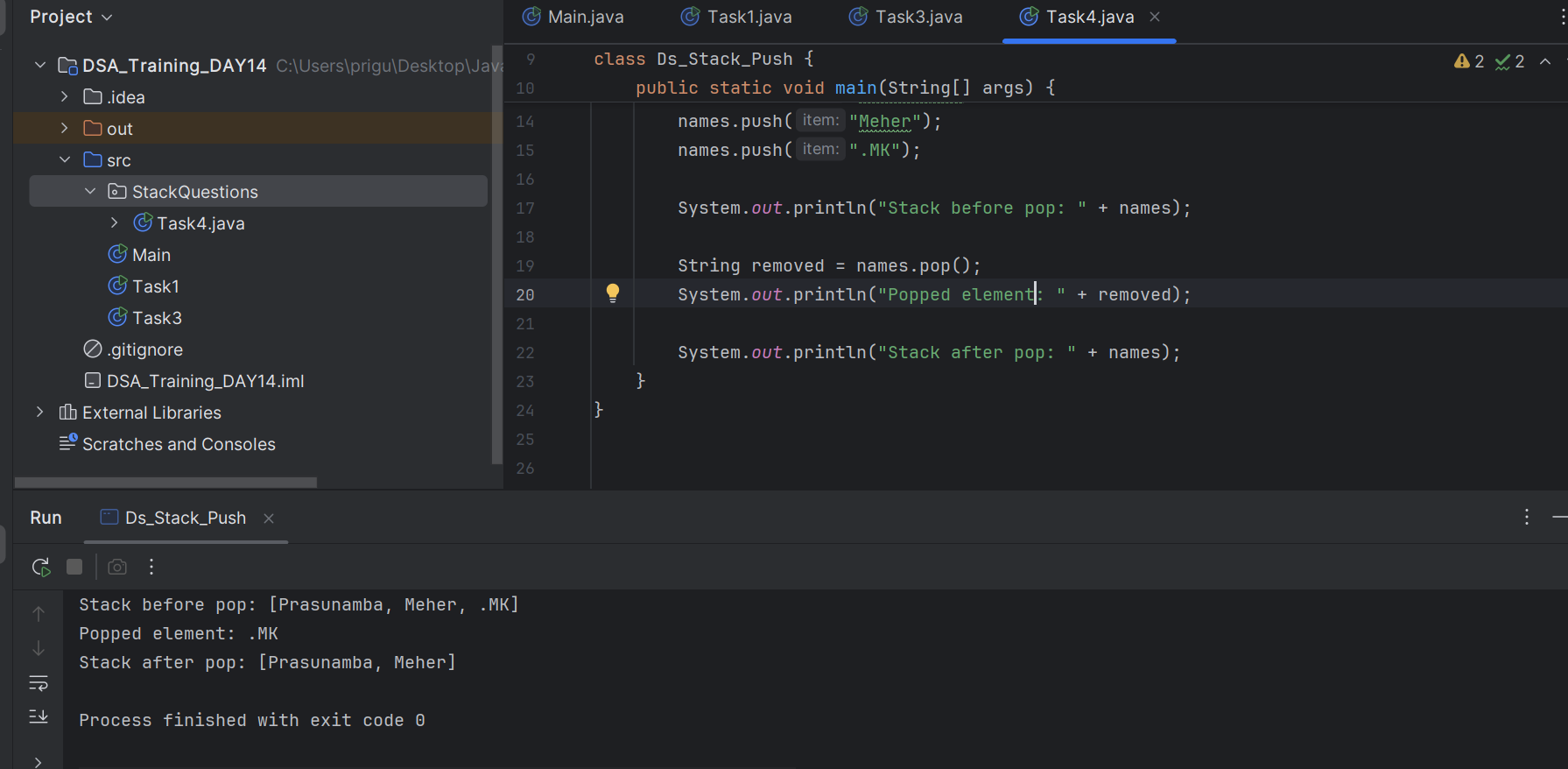
String removed = names.pop();

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

System.*out*.println("Stack after pop: " + names);

}

}



Task 6

package StackQuestions;

import java.util.Stack;

public class Task5 {

public static void main(String[] args) {

Stack<String> names = new Stack<>();

names.push("Prasunamba");

names.push("Meher");

names.push(".MK");

System.*out*.println("Current stack: " + names);

// Pop the top element and display it

String popped = names.pop();

System.*out*.println("Popped element: " + popped);

System.*out*.println("Stack now remaining names: " + names);

// Search for a specific value

String target = "Prasunamba";

int position = names.search(target);

if (position != -1) {

System.*out*.println("'" + target + "' is at position "

+ position + " from the top.");

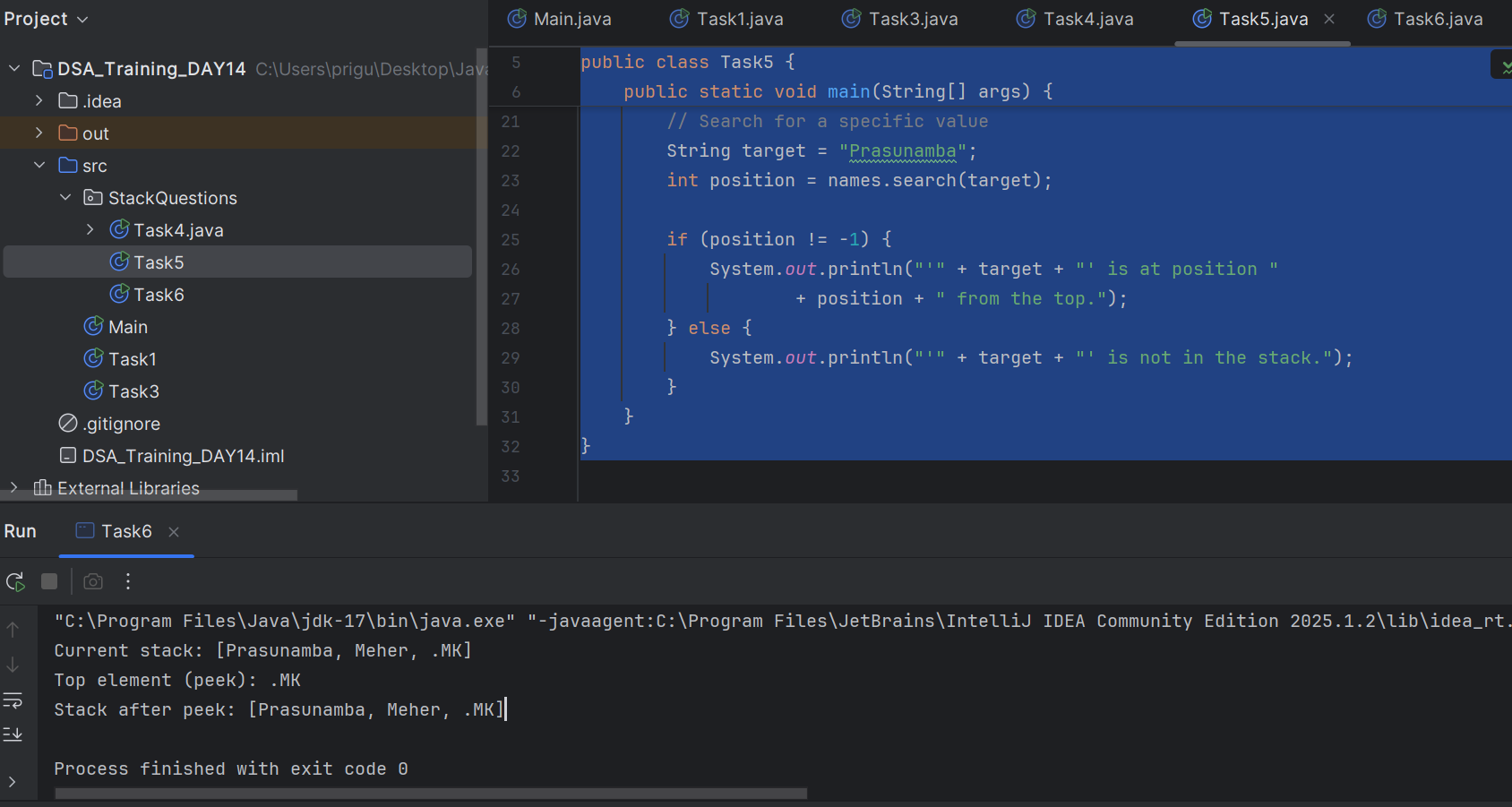
} else {

System.*out*.println("'" + target + "' is not in the stack.");

}

}

}



Task 7

package StackQuestions;

//peek

import java.util.Stack;

public class Task6 {

public static void main(String[] args) {

Stack<String> names = new Stack<>();

names.push("Prasunamba");

names.push("Meher");

names.push(".MK");

System.*out*.println("Current stack: " + names);

// Use peek() to look at the top without removing

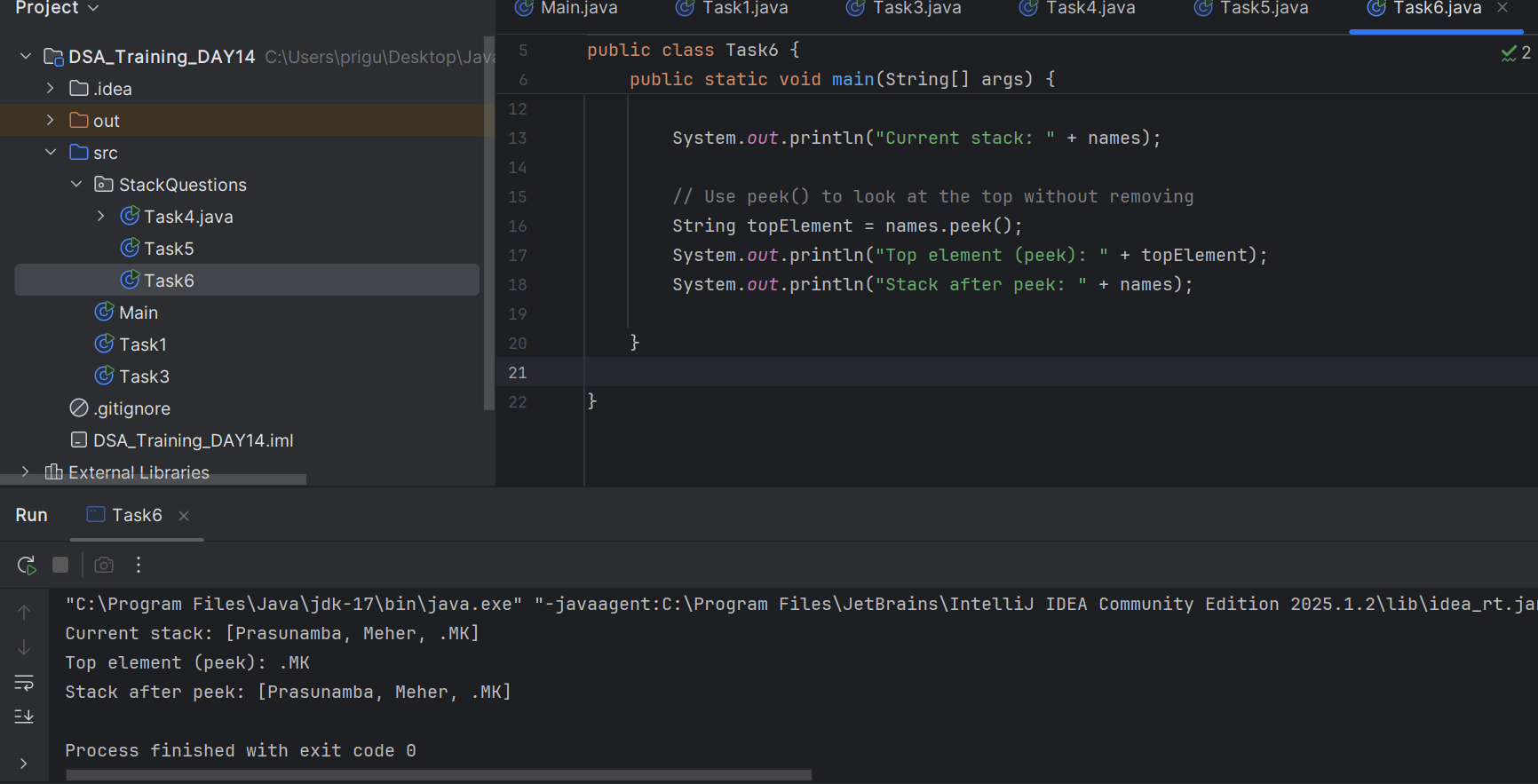
String topElement = names.peek();

System.*out*.println("Top element (peek): " + topElement);

System.*out*.println("Stack after peek: " + names);

}

}



Task 7 n 8

package StackQuestions;

import java.util.Stack;

public class Task7n8 {

public static void main(String[] args) {

Stack<String> names = new Stack<>();

names.push("Prasunamba");

names.push("Meher");

// Check emptiness

System.*out*.println("Is stack empty? " + names.isEmpty()); // false

// Pop both elements

names.pop();

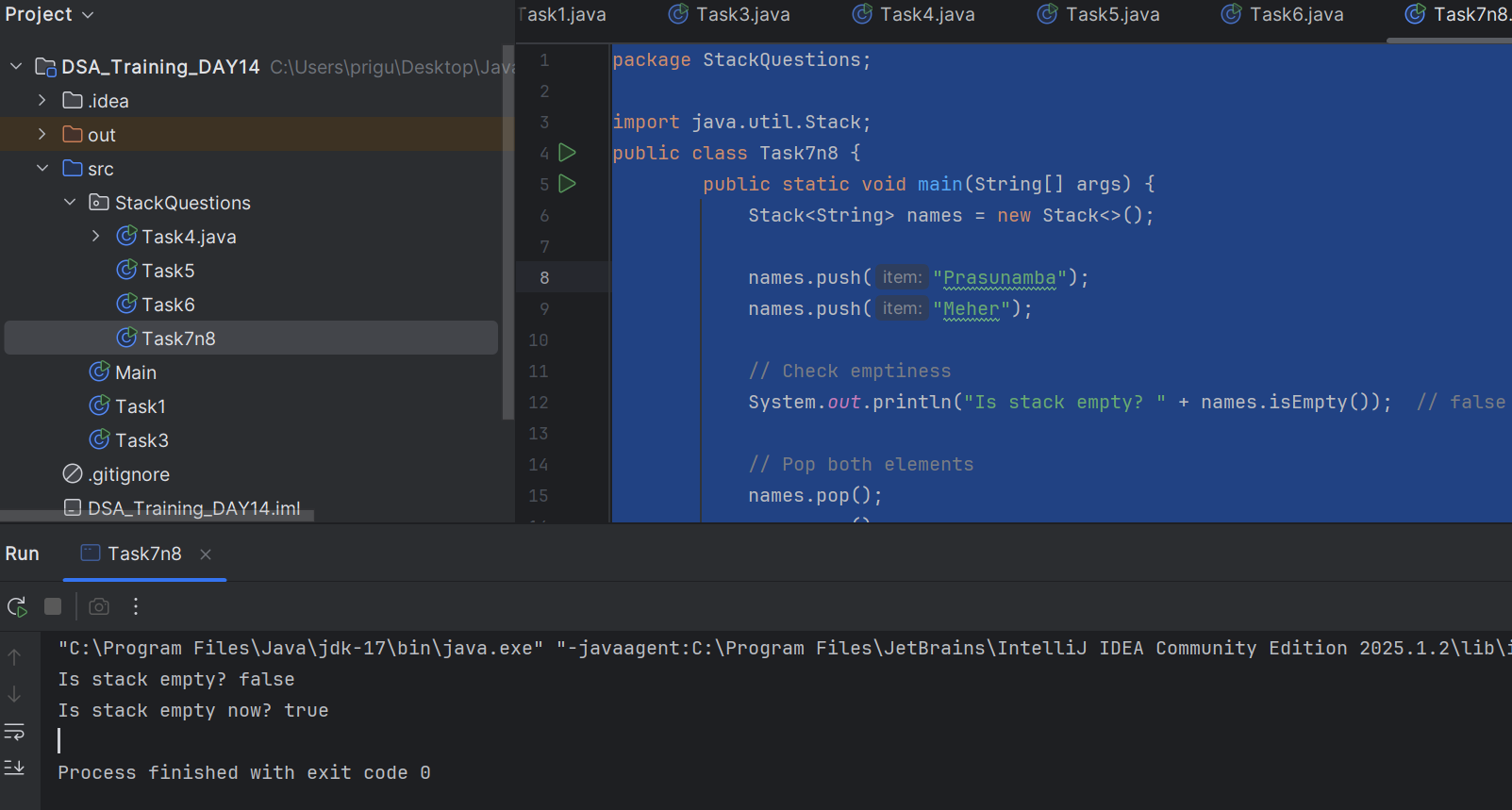
names.pop();

// After popping everything

System.*out*.println("Is stack empty now? " + names.isEmpty()); // true

}

}



Task 9

**push, pop, peek, empty, search** are the core stack-specific methods.

import java.util.Stack;

public class Task9 {

public static void main(String[] args) {

Stack<String> s = new Stack<>();

System.*out*.println("\nEmpty? " + s.empty()); // true

s.push("A"); // ["A"]

s.push("B"); // ["A", "B"]

System.*out*.println("Top is: " + s.peek()); // prints "B"

System.*out*.println("Position of A from top: " + s.search("A")); // prints 2

System.*out*.println("Popped: " + s.pop()); // removes and prints "B"

System.*out*.println("Empty now? " + s.empty()); // false

s.pop(); // removes "A"

System.*out*.println("Empty now? " + s.empty()); // true

}

}

Task 10

What are the common operations in Queues

1. **enqueue(item)**
   * Adds an element at the rear of the queue
   * Time Complexity: O(1)
2. **dequeue()**
   * Removes and returns the element from the front
   * Time Complexity: O(1)
3. **peek() / front()**
   * Returns the front element without removing it
   * Time Complexity: O(1)
4. **isEmpty()**
   * Checks if the queue has no elements
   * Time Complexity: O(1)
5. **isFull()** (for fixed-size queues only)  
   * Checks if the queue is full
   * Time Complexity: O(1)
6. **size()**
   * Returns the number of elements in the queue
   * Time Complexity: O(1) for some implementations, O(n) for others

Task 11

package QueueQuestions;

public class Task11 {

private int[] arr;

private int front, size, capacity;

public Task11 (int capacity) {

this.capacity = capacity;

arr = new int[capacity];

front = 0;

size = 0;

}

public boolean isFull() {

return size == capacity;

}

public boolean isEmpty() {

return size == 0;

}

public void enqueue(int item) {

if (isFull()) {

System.*out*.println("Queue is full");

return;

}

int rear = (front + size) % capacity;

arr[rear] = item;

size++;

System.*out*.println("Inserted " + item);

}

public int dequeue() {

if (isEmpty()) {

System.*out*.println("Queue is empty");

return -1;

}

int item = arr[front];

front = (front + 1) % capacity;

size--;

System.*out*.println("Removed " + item);

return item;

}

public int peek() {

if (isEmpty()) {

System.*out*.println("Queue is empty");

return -1;

}

return arr[front];

}

public void display() {

if (isEmpty()) {

System.*out*.println("(empty)");

return;

}

System.*out*.print("Queue: ");

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

System.*out*.print(arr[(front + i) % capacity] + " ");

}

System.*out*.println();

}

public static void main(String[] args) {

Task11 q = new Task11(5);

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

q.display(); // Queue: 10 20 30

System.*out*.println("Peek: " + q.peek()); // Peek: 10

q.dequeue(); // Removes 10

q.display(); // Queue: 20 30

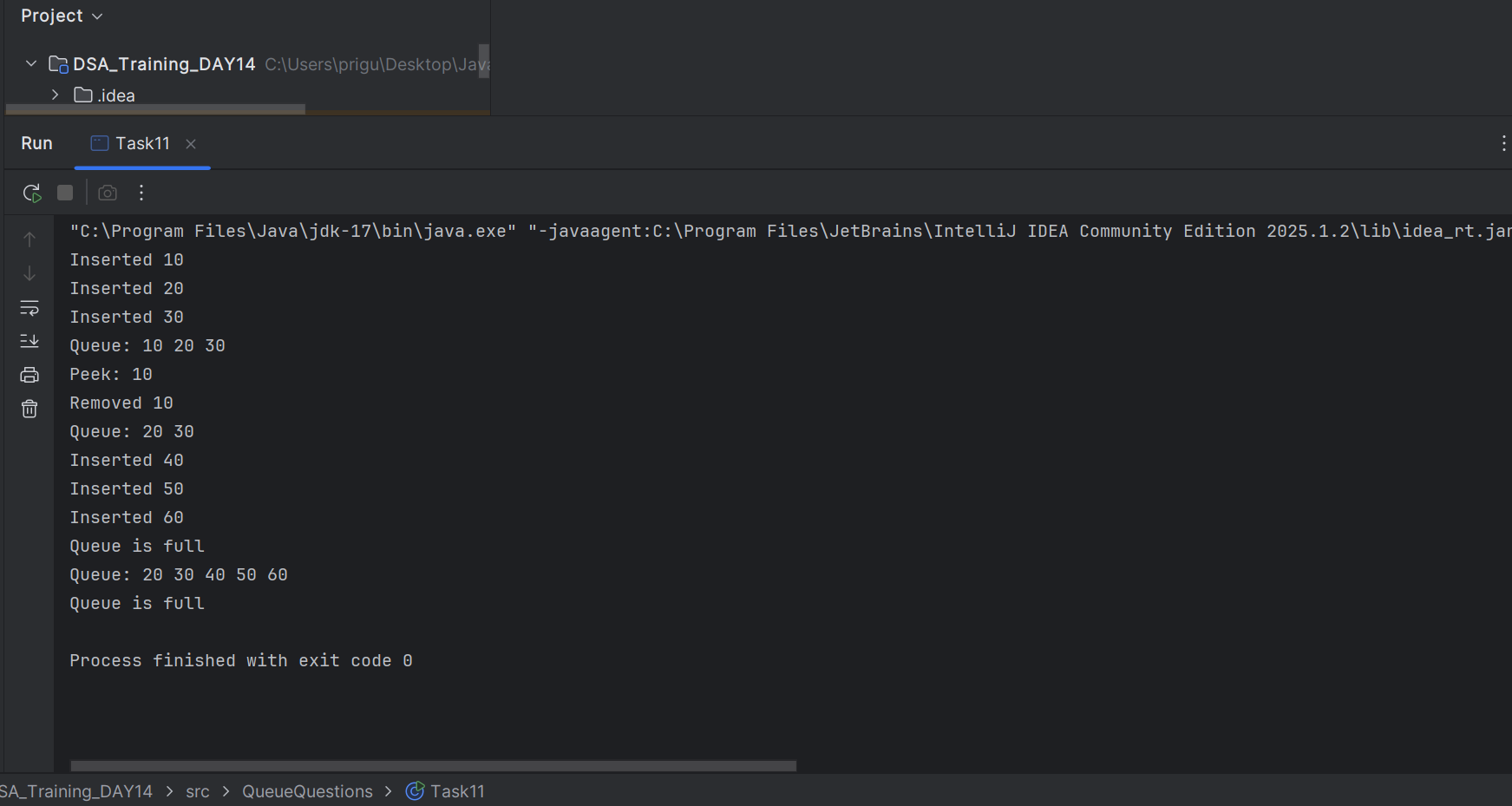
for(int i = 40; i <= 70; i+=10) q.enqueue(i);

q.display(); // Queue: 20 30 40 50 60

q.enqueue(80); // Queue is full

}

}



**HOME TASKS**

package RecursiveQuestions;

public class Task13 {

// Recursive method to find nth Fibonacci number

public static int fib(int n) {

if (n < 2) {

return n; // base case: fib(0) = 0, fib(1) = 1

}

return *fib*(n - 1) + *fib*(n - 2); // recursive case

}

public static void main(String[] args) {

int n = 10; // how many Fibonacci numbers you want to print

System.*out*.println("Fibonacci series up to " + n + " terms:");

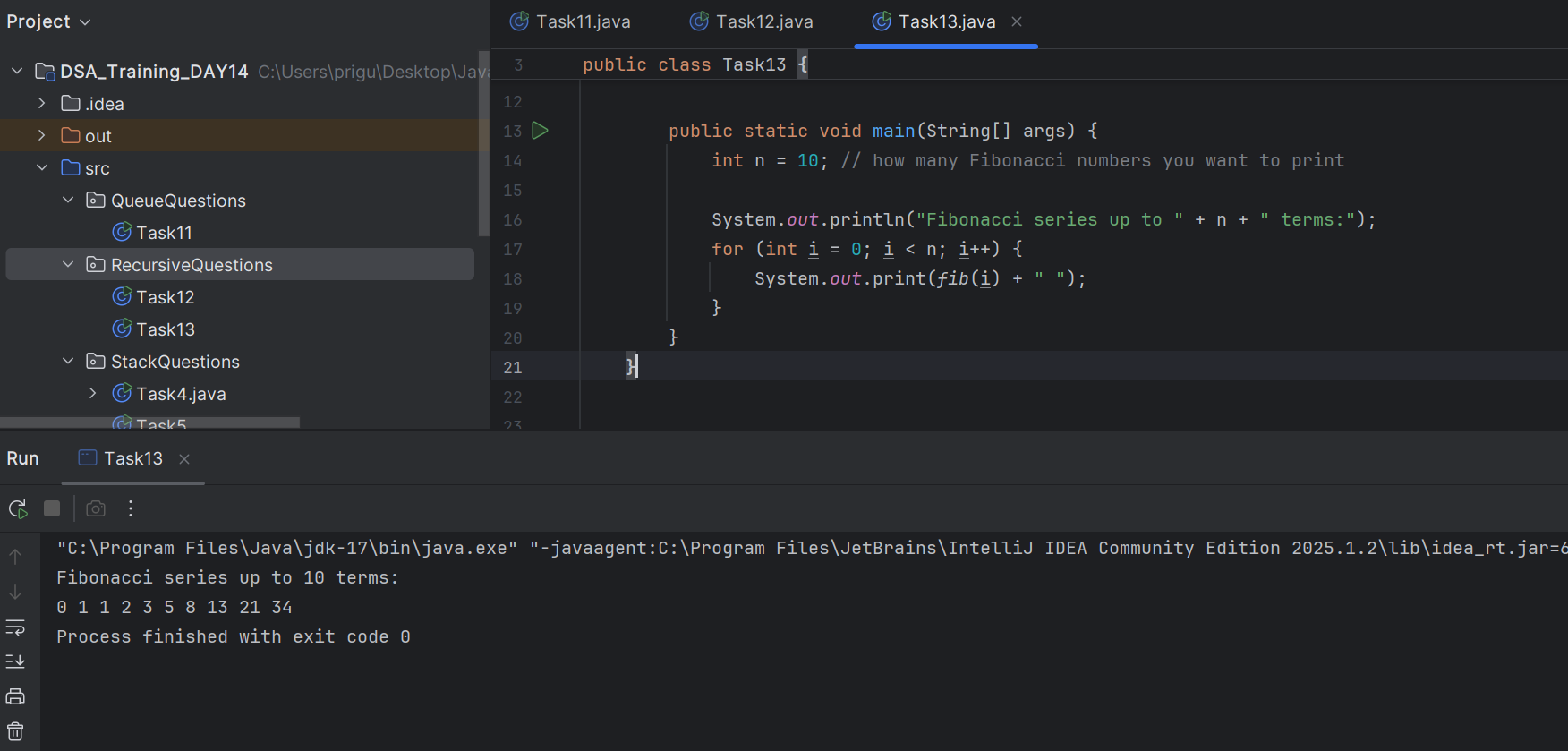
for (int i = 0; i < n; i++) {

System.*out*.print(*fib*(i) + " ");

}

}

}



Task14

What is the difference between recursion and iteration

Function calls **itself in recursion**

**Repeats a block of code using loops (for, while) in iteration.**

**Task 15**

**package RecursiveQuestions;**

**public class Task15 {**

**// Recursive method to reverse string**

**public static String reverse(String str) {**

**if (str.isEmpty()) {**

**return str; // base case: empty string**

**}**

**// Take first char and put it at the end of reversed rest**

**return *reverse*(str.substring(1)) + str.charAt(0);**

**}**

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

**String input = "hello";**

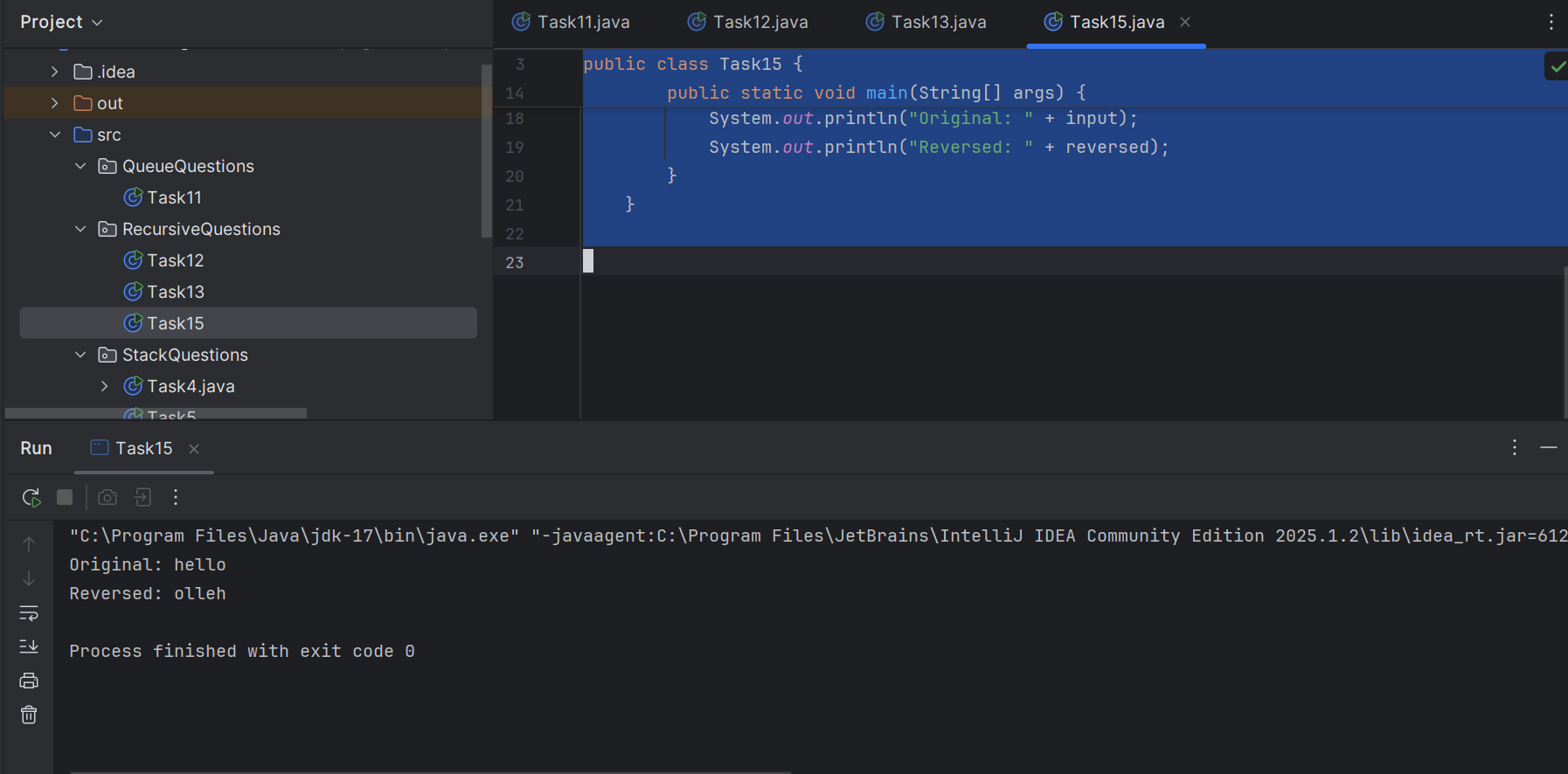
**String reversed = *reverse*(input);**

**System.*out*.println("Original: " + input);**

**System.*out*.println("Reversed: " + reversed);**

**}**

**}**

****

**HOMETasks**

**17.**

Write a recursive function to search for an element in an array

package RecursiveQuestions;

public class SearchingELement {

// Recursive function to search for an element in the array

public static int search(int[] arr, int index, int target) {

// Base case: if index goes out of bounds

if (index >= arr.length) {

return -1; // element not found

}

// If current element is the target

if (arr[index] == target) {

return index;

}

// Recursive call for the next element

return *search*(arr, index + 1, target);

}

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40, 50};

int target = 30;

int result = *search*(arr, 0, target);

if (result != -1) {

System.*out*.println("Element found at index: " + result);

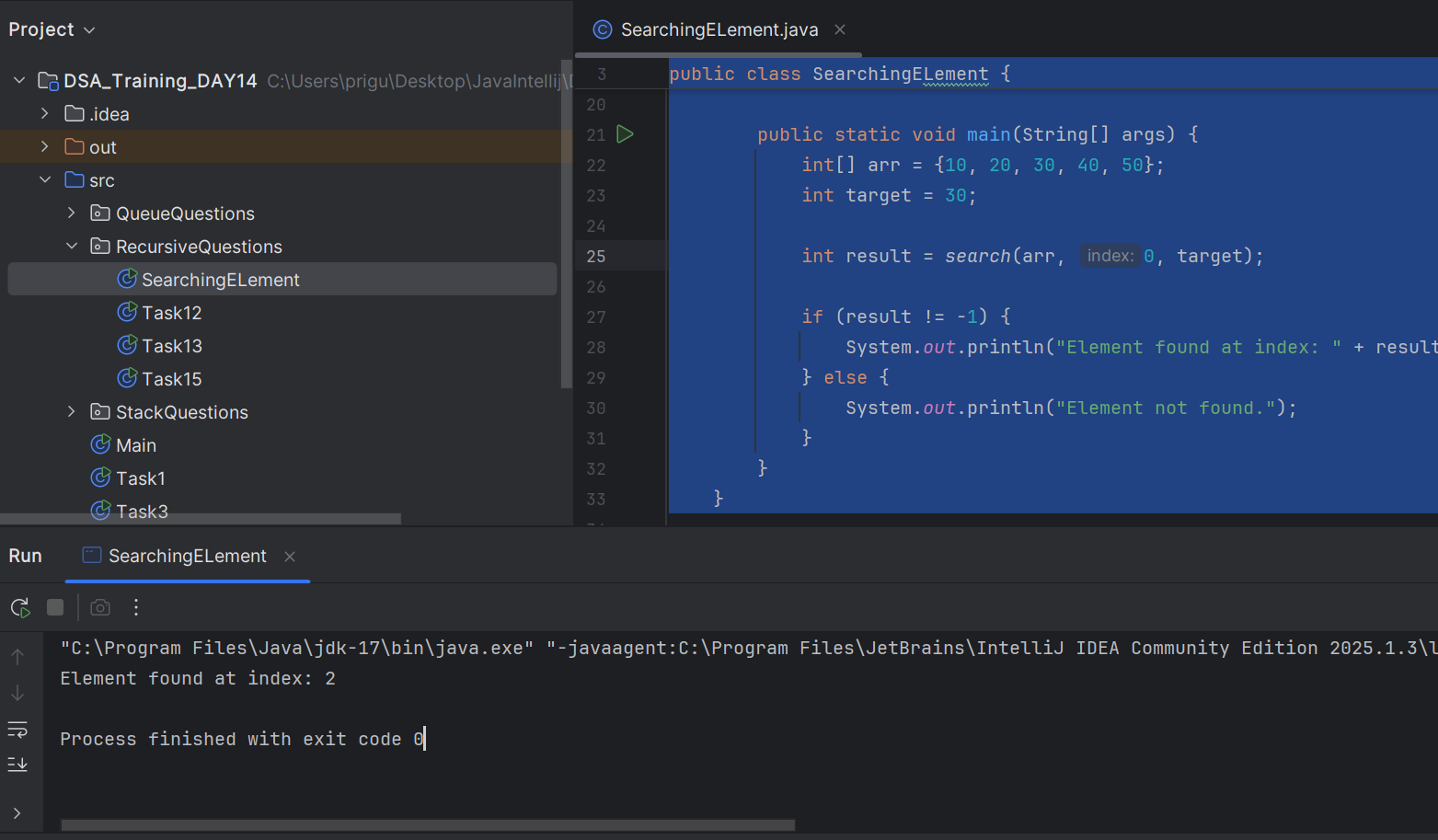
} else {

System.*out*.println("Element not found.");

}

}

}



**Task 18**

**package RecursiveQuestions;**

**public class sumOfNum {**

**// Recursive function to count the number of digits**

**public static int countDigits(int num) {**

**// Base case**

**if (num == 0) {**

**return 0;**

**}**

**// Recursive step**

**return 1 + *countDigits*(num / 10);**

**}**

**// Recursive function to calculate the sum of digits**

**public static int sumDigits(int num) {**

**// Base case**

**if (num == 0) {**

**return 0;**

**}**

**// Recursive step**

**return (num % 10) + *sumDigits*(num / 10);**

**}**

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

**int number = 12345;**

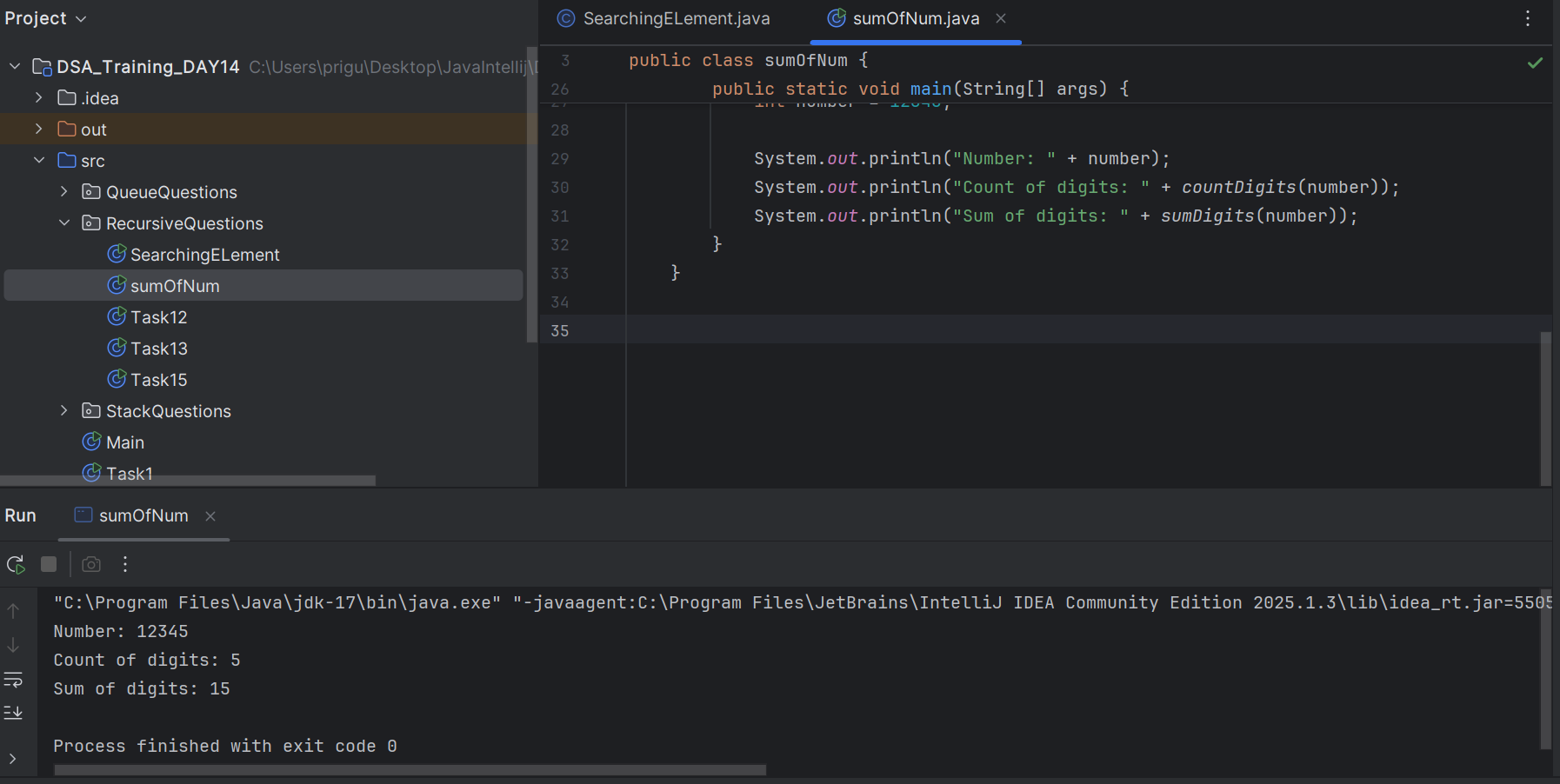
**System.*out*.println("Number: " + number);**

**System.*out*.println("Count of digits: " + *countDigits*(number));**

**System.*out*.println("Sum of digits: " + *sumDigits*(number));**

**}**

**}**

****

**Task 18**  Write a recursive function to reverse a null-terminated string

package RecursiveQuestions;

public class ReverseNull {

// Recursive function to reverse a null-terminated string

public static void reverse(char[] str, int index) {

// Base case: if current character is null terminator, stop

if (str[index] == '\0') {

return;

}

// Recursive call

*reverse*(str, index + 1);

// Print the character while backtracking

System.*out*.print(str[index]);

}

public static void main(String[] args) {

// Simulate a null-terminated string (extra '\0' at the end)

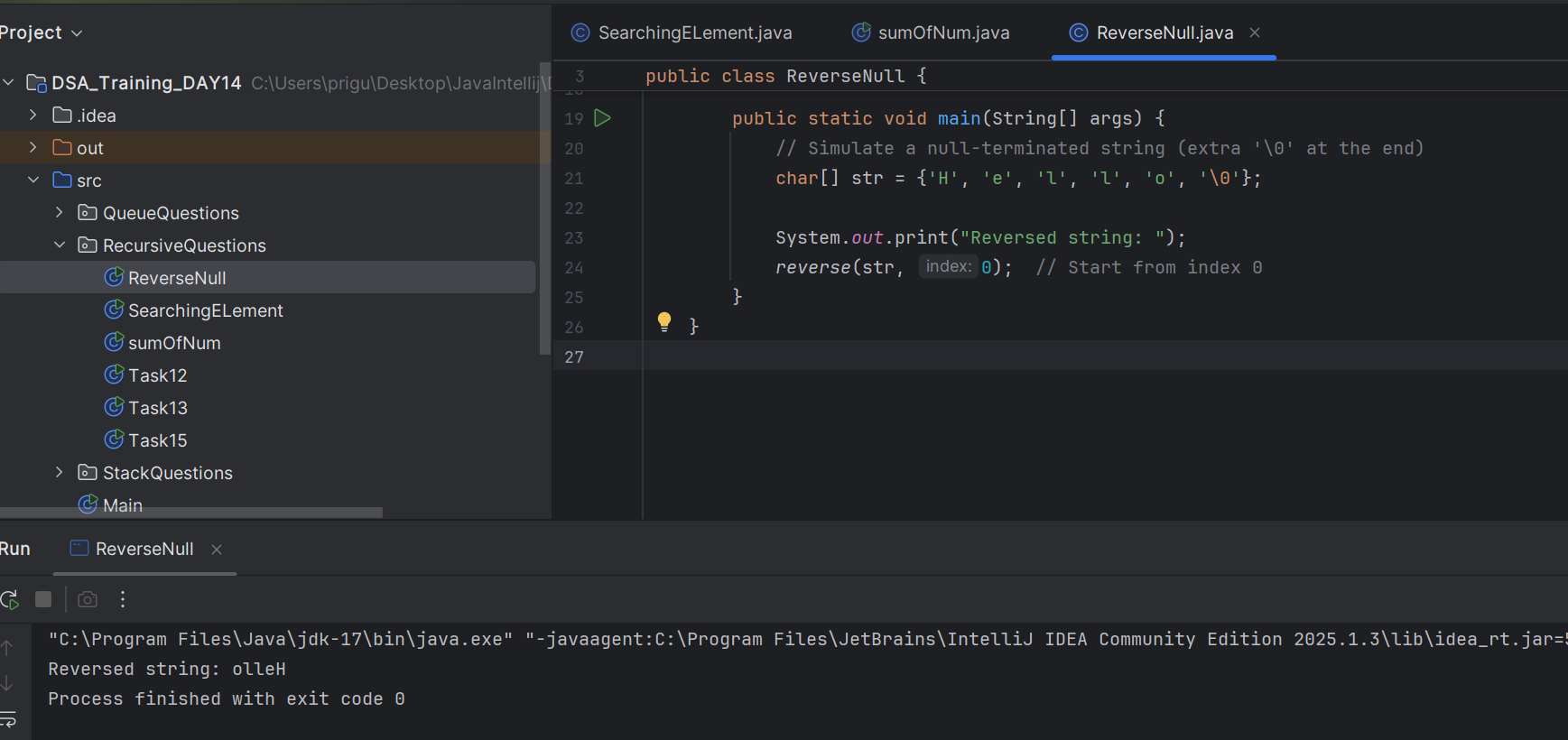
char[] str = {'H', 'e', 'l', 'l', 'o', '\0'};

System.*out*.print("Reversed string: ");

*reverse*(str, 0); // Start from index 0

}

}



Task 19

package RecursiveQuestions;

public class Conversion {

// Recursive function to print binary of a number

public static void decimalToBinary(int num) {

if (num == 0) {

return;

}

// Recursive call with quotient

*decimalToBinary*(num / 2);

// Print remainder (either 0 or 1)

System.*out*.print(num % 2);

}

public static void main(String[] args) {

int number = 13;

System.*out*.print("Binary of " + number + " is: ");

// Special case if number is 0

if (number == 0) {

System.*out*.print(0);

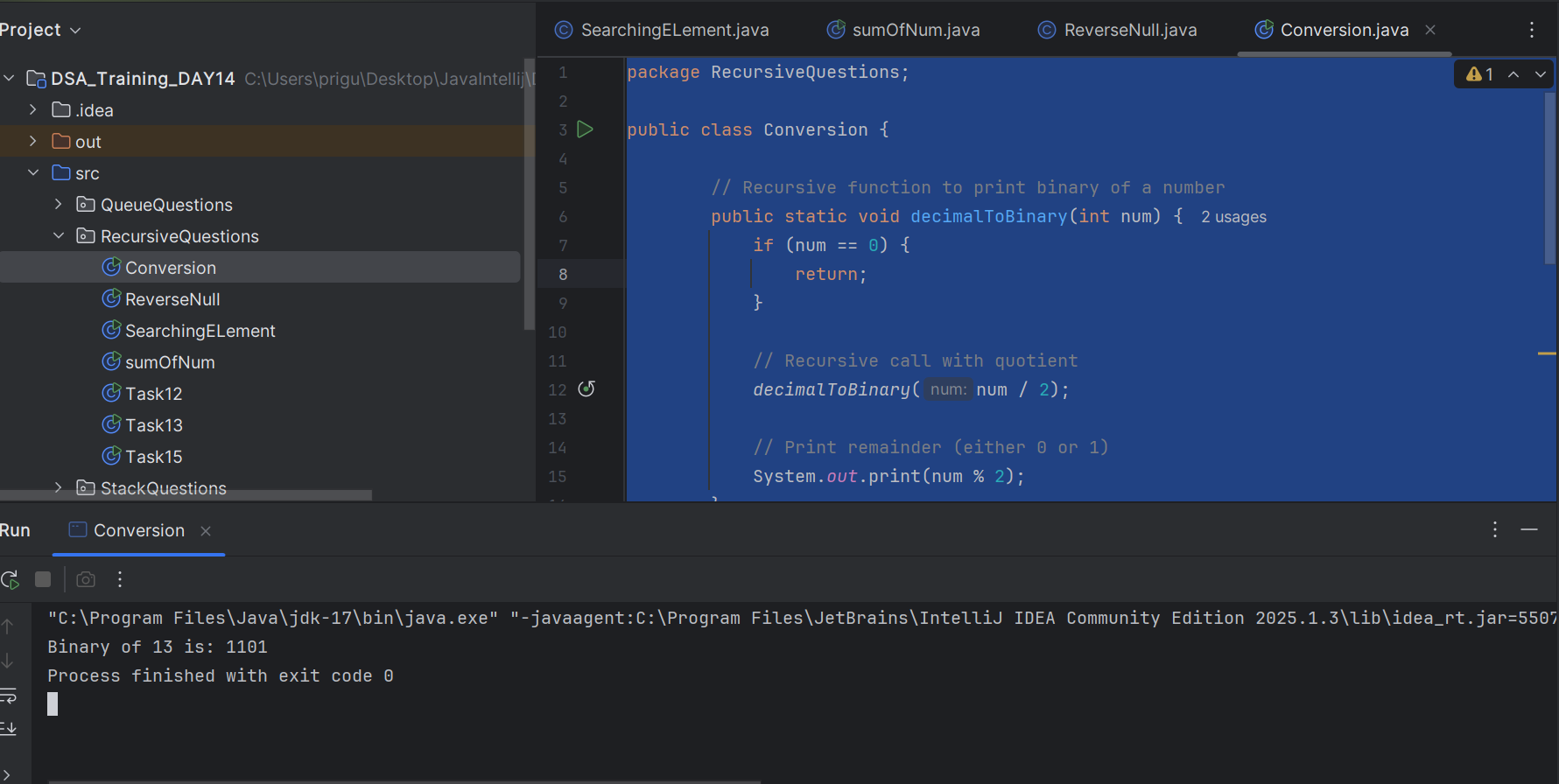
} else {

*decimalToBinary*(number);

}

}

}



Task 20

package QueueQuestions;

public class Palidrome {

// Recursive function to check for palindrome

public static boolean isPalindrome(String str, int start, int end) {

// Base case: if pointers cross, it's a palindrome

if (start >= end) {

return true;

}

// If characters don't match, it's not a palindrome

if (str.charAt(start) != str.charAt(end)) {

return false;

}

// Recursive check for the remaining string

return *isPalindrome*(str, start + 1, end - 1);

}

public static void main(String[] args) {

String str = "madam";

if (*isPalindrome*(str, 0, str.length() - 1)) {

System.*out*.println(str + " is a palindrome.");

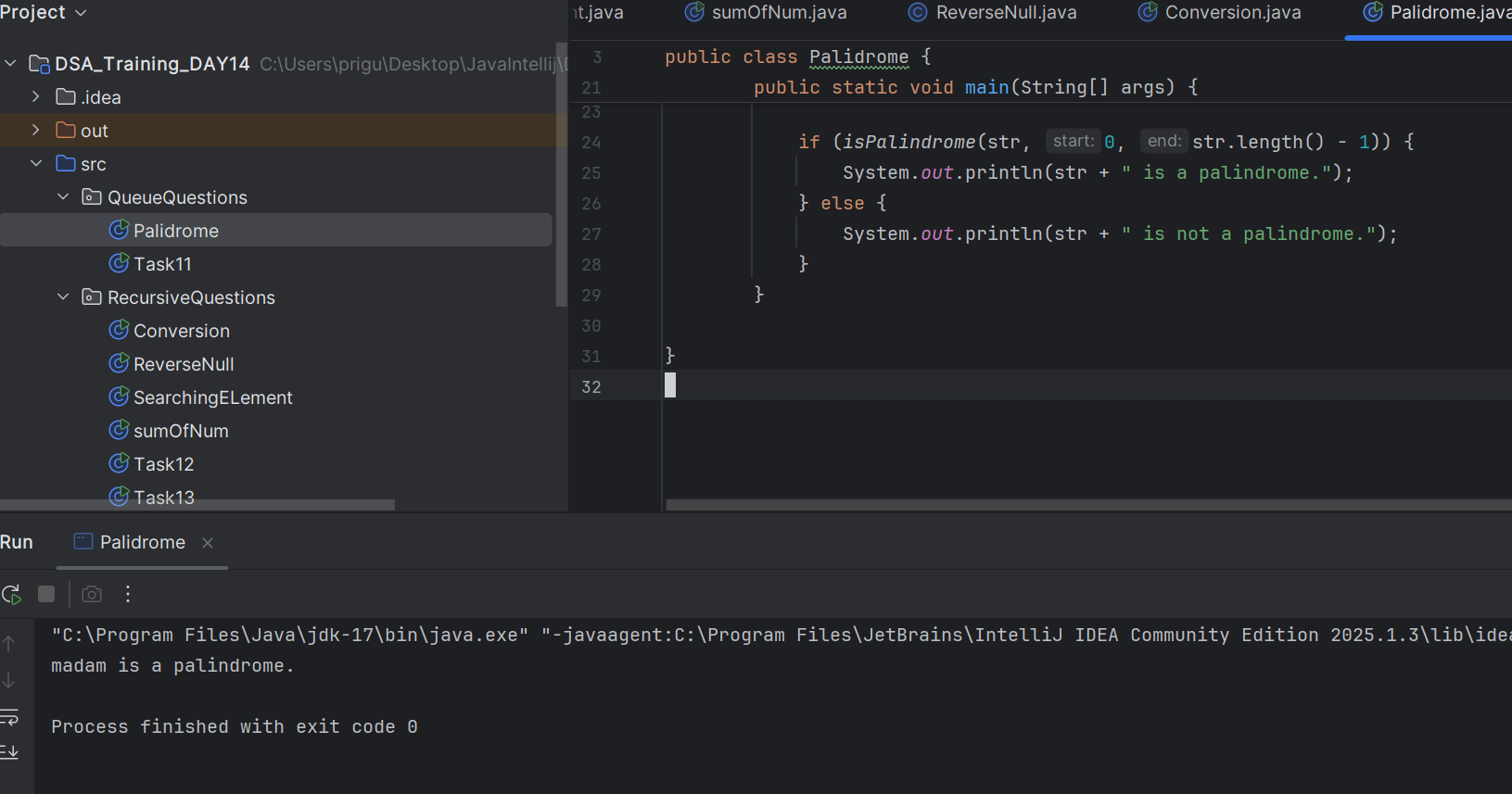
} else {

System.*out*.println(str + " is not a palindrome.");

}

}

}



Task 21

public class Array {

// Recursive function to copy elements from src[] to dest[]

public static void copyArray(int[] src, int[] dest, int index) {

// Base case: when index reaches array length

if (index >= src.length) {

return;

}

// Copy current element

dest[index] = src[index];

// Recursive call for next index

*copyArray*(src, dest, index + 1);

}

public static void main(String[] args) {

int[] source = {10, 20, 30, 40, 50};

int[] destination = new int[source.length];

// Call recursive function starting at index 0

*copyArray*(source, destination, 0);

// Print copied array

System.*out*.print("Copied array: ");

for (int val : destination) {

System.*out*.print(val + " ");

}

}

}

