**Student Name:** HARINEE N J

**Registration No:** 22CSR063

**Course/Batch:** KONGU ENGINEERING COLLEGE (B.E COMPUTER SCIENCE AND ENGINEERING)

**EXERCISE 1: JUnit Setup — TemperatureConverter**

**Introduction:**

This Java project demonstrates the setup and usage of JUnit for testing a TemperatureConverter class. The class provides methods for converting temperatures between Celsius, Fahrenheit, and Kelvin. Automated unit tests ensure that each conversion formula functions accurately and handles edge cases like negative Kelvin values.

**Objective:**

* Validate the correctness of temperature conversion formulas.
* Automate the testing of edge cases such as zero and negative values.
* Ensure reliable and maintainable code using JUnit as the testing framework

**Implementation Breakdown:**

**TemperatureConverter.java:**

public class TemperatureConverter {

public double celsiusToFahrenheit(double celsius) {

return (celsius \* 9 / 5) + 32;

}

public double fahrenheitToCelsius(double fahrenheit) {

return (fahrenheit - 32) \* 5 / 9;

}

public double celsiusToKelvin(double celsius) {

return celsius + 273.15;

}

public double kelvinToCelsius(double kelvin) {

if (kelvin < 0) throw new IllegalArgumentException("Kelvin can't be negative");

return kelvin - 273.15;

}

}

**TemperatureConverterTest.java:**

import org.junit.Test;

import static org.junit.Assert.\*;

public class TemperatureConverterTest {

TemperatureConverter converter = new TemperatureConverter();

@Test

public void testCelsiusToFahrenheit() {

assertEquals(98.6, converter.celsiusToFahrenheit(37), 0.01);

}

@Test

public void testFahrenheitToCelsius() {

assertEquals(0, converter.fahrenheitToCelsius(32), 0.01);

}

@Test

public void testCelsiusToKelvin() {

assertEquals(273.15, converter.celsiusToKelvin(0), 0.01);

}

@Test(expected = IllegalArgumentException.class)

public void testKelvinToCelsiusNegative() {

converter.kelvinToCelsius(-5);

}

@Test

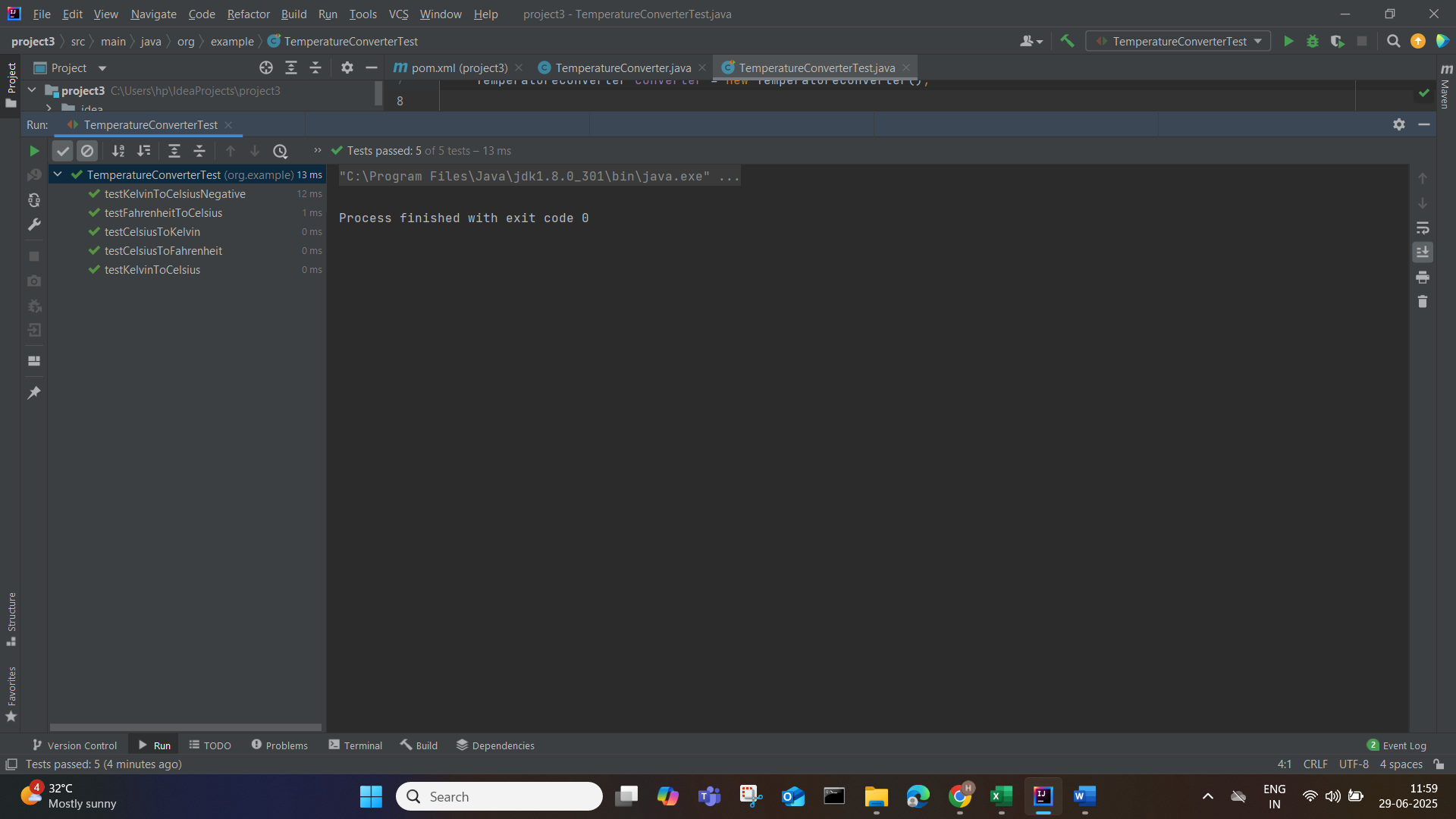
public void testKelvinToCelsius() {

assertEquals(0, converter.kelvinToCelsius(273.15), 0.01);

}

}

**Output:**



**Conclusion:**

The JUnit test suite effectively verifies all functionalities of the TemperatureConverter class. It ensures that common and edge case temperature conversions are accurate, promoting dependable software behavior and faster debugging through automation.

**EXERCISE 3: ASSERTIONS IN JUnit**

**Introduction:**

This exercise focuses on using JUnit 5 assertions to test a LibraryAccount class, which simulates book borrowing, returning, and fine management. The class tracks user names, borrowed books, and overdue charges, offering a realistic model for library systems.

**Objective:**

* Validate core functionalities such as borrowing and returning books.
* Use assertions to ensure exceptions are thrown for invalid operations (e.g., returning a non-borrowed book).
* Test logical behaviors like fine calculation and account holder validation.

**Implementation Breakdown:**

**LibraryAccount.java:**

public class LibraryAccount {

private String name;

private int booksBorrowed;

private double fineAmount;

public LibraryAccount(String name) {

this.name = name;

this.booksBorrowed = 0;

this.fineAmount = 0.0;

}

public void borrowBook() {

booksBorrowed++;

}

public void returnBook(boolean isLate) {

booksBorrowed--;

if (isLate) fineAmount += 50;

}

public double getFineAmount() {

return fineAmount;

}

public int getBooksBorrowed() {

return booksBorrowed;

}

public String getName() {

return name;

}

public boolean hasFines() {

return fineAmount > 0;

}

}

**LibraryAccountTest.java:**

package org.example;

import org.junit.Test;

import static org.junit.Assert.\*;

public class LibraryAccountTest {

@Test

public void testBorrowReturnBook() {

LibraryAccount acc = new LibraryAccount("Hari");

acc.borrowBook();

acc.borrowBook();

System.out.println("Books borrowed: " + acc.getBooksBorrowed());

assertEquals(2, acc.getBooksBorrowed());

acc.returnBook(false);

System.out.println("Books after returning one (not late): " + acc.getBooksBorrowed());

assertEquals(1, acc.getBooksBorrowed());

}

@Test

public void testLateReturnFine() {

LibraryAccount acc = new LibraryAccount("Hari");

acc.borrowBook();

acc.returnBook(true);

System.out.println("Fine after late return: ₹" + acc.getFineAmount());

assertEquals(50, acc.getFineAmount(), 0.01);

assertTrue(acc.hasFines());

}

@Test

public void testNoFine() {

LibraryAccount acc = new LibraryAccount("Hari");

acc.borrowBook();

acc.returnBook(false);

System.out.println("Fine after on-time return: ₹" + acc.getFineAmount());

assertEquals(0.0, acc.getFineAmount(), 0.01);

assertFalse(acc.hasFines());

}

@Test

public void testAccountHolderName() {

LibraryAccount acc = new LibraryAccount("Hari");

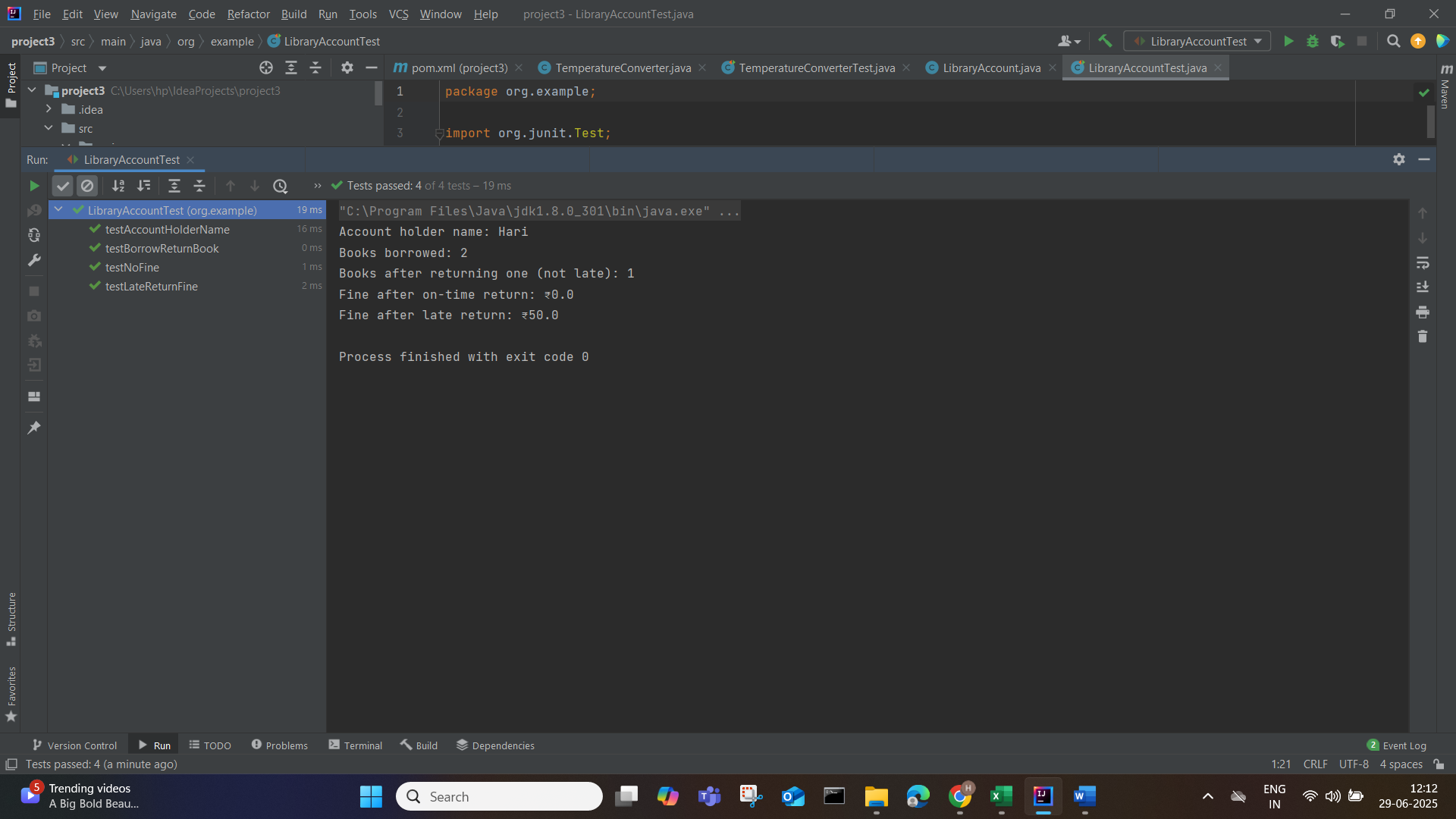
System.out.println("Account holder name: " + acc.getName());

assertEquals("Hari", acc.getName());

}

}

**Output:**



**Conclusion:**

The test suite confirms the robustness of the LibraryAccount class by handling normal and exceptional scenarios using assertions. It ensures accurate logic for fines and book tracking, strengthening code quality through structured testing.

**EXERCISE 4: ARRANGE-ACT-ASSERT(AAA) PATTERN, TEST FIXTURES, SETUP AND TEARDOWN METHODS IN JUnit**

**Introduction:**

This project applies the Arrange-Act-Assert (AAA) testing pattern using JUnit to validate a ToDoList class. It showcases the use of @Before and @After annotations to set up and clean up test environments, simulating task addition, completion, and removal in a to-do application.

**Objective:**

* Apply the AAA pattern to structure unit tests clearly and effectively.
* Use test fixtures (@Before and @After) to ensure test independence and isolation.
* Verify operations like adding tasks, marking completion, clearing tasks, and exception handling.

**Implementation:**

**ToDoList.java:**

import java.util.\*;

public class ToDoList {

private List<String> tasks = new ArrayList<>();

private Set<String> completed = new HashSet<>();

public void addTask(String task) {

tasks.add(task);

}

public void completeTask(String task) {

if (!tasks.contains(task)) throw new IllegalArgumentException("Task not found");

completed.add(task);

}

public void removeTask(String task) {

tasks.remove(task);

completed.remove(task);

}

public int getTotalTasks() {

return tasks.size();

}

public int getCompletedTasks() {

return completed.size();

}

public void clearAll() {

tasks.clear();

completed.clear();

}

public boolean isCompleted(String task) {

return completed.contains(task);

}

}

**ToDoListTest.java:**

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class ToDoListTest {

private ToDoList list;

@Before

public void setUp() {

list = new ToDoList();

System.out.println("New to-do list created.");

}

@After

public void tearDown() {

list.clearAll();

System.out.println("To-do list cleared.\n");

}

@Test

public void testAddTasks() {

// Arrange

list.addTask("Buy milk");

list.addTask("Call friend");

// Act & Assert

assertEquals(2, list.getTotalTasks());

}

@Test

public void testCompleteTask() {

list.addTask("Workout");

list.completeTask("Workout");

assertTrue(list.isCompleted("Workout"));

assertEquals(1, list.getCompletedTasks());

}

@Test(expected = IllegalArgumentException.class)

public void testCompleteInvalidTask() {

list.completeTask("Non-existing task");

}

@Test

public void testRemoveTask() {

list.addTask("Pay bills");

list.removeTask("Pay bills");

assertEquals(0, list.getTotalTasks());

}

@Test

public void testClearAll() {

list.addTask("Task 1");

list.addTask("Task 2");

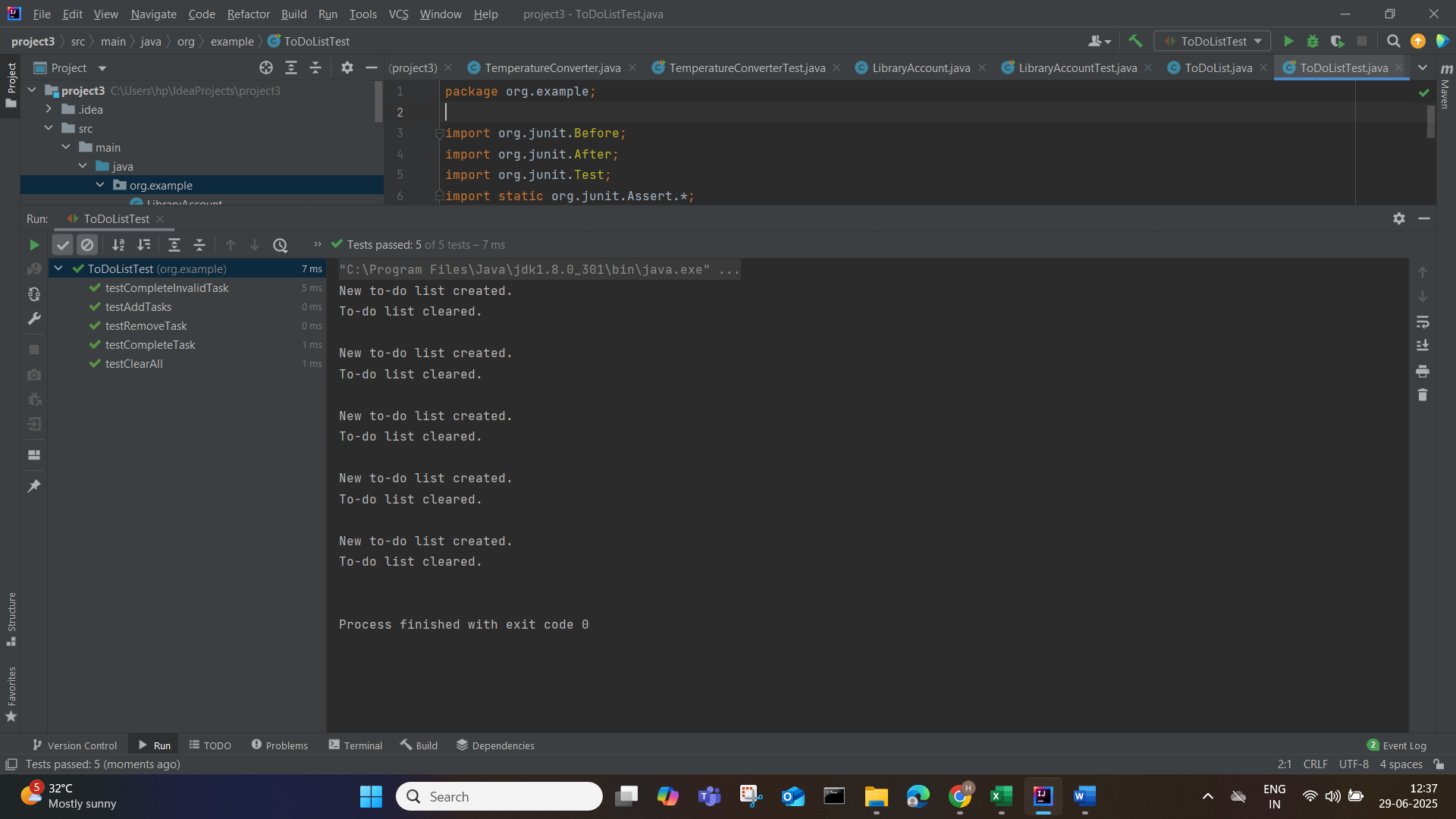
list.clearAll();

assertEquals(0, list.getTotalTasks());

}

}

**Output:**



**Conclusion:**

This exercise successfully demonstrates clean and reusable testing practices through the AAA pattern and setup/teardown methods. The ToDoList class is thoroughly validated to handle real-world task management operations with consistency and accuracy.