Day 36 - 26th Sept 2025

| Basics of Junit Framework |
| --- |
| Features of Junit Framework, Unit Test Case in Junit, Test Methods, Assertions, Annotations, Hands on: Writing Junit Tests using eclipse |
| Test Suite |
| Exception Testing, Test Timeouts, Parameterized Tests, Hands on: Parameterized Tests, Introduction to,Hamcrest Library, Using assertThat, Testing using mock objects, Hands on: assertThat, Mocking |
| What Is Automation Testing?, Why Automation Testing?, Types of Testing: Manual vs. Automated, Automation Testing Tools and Frameworks, Test Automation Life Cycle, Introduction to Automation Frameworks (e.g., Data-Driven, Keyword-Driven, Page Object Model), Identifying Test Scenarios for Automation |

JDK - 1.5 and above

IDE -

set java environment

- JAVA\_HOME = c:\pro......\java\jdk 1.8.....

--- set path for bin folder

c:\program........\java\jdk 1.8\bin

in cmd check if

java --version

download JUnit archive

junit5.x. jar

set junit environment

JUNIT\_HOME = c:\progra....\JUNIT

set class path also

give the location of the jar file

%CLASSPATH%, %JUNIT\_HOME%\junit5.x.jar;;

test Junit setup

Pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>

<artifactId>JunitDemos</artifactId>

<version>1.0-SNAPSHOT</version>

<name>Archetype - JunitDemos</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>5.10.0</version> <!-- Use the latest stable version -->

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.10.0</version> <!-- Use the latest stable version -->

<scope>test</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/org.hamcrest/hamcrest-all -->

<dependency>

<groupId>org.hamcrest</groupId>

<artifactId>hamcrest-all</artifactId>

<version>1.3</version>

<scope>test</scope>

</dependency>

<!-- Maven -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-junit-jupiter</artifactId>

<version>5.12.0</version> <!-- Use the latest stable version -->

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.12.0</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

Task 01:

package org.example;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*assertEquals*;

public class DemoTest001TestJunit {

@Test

public void Testcase1() {

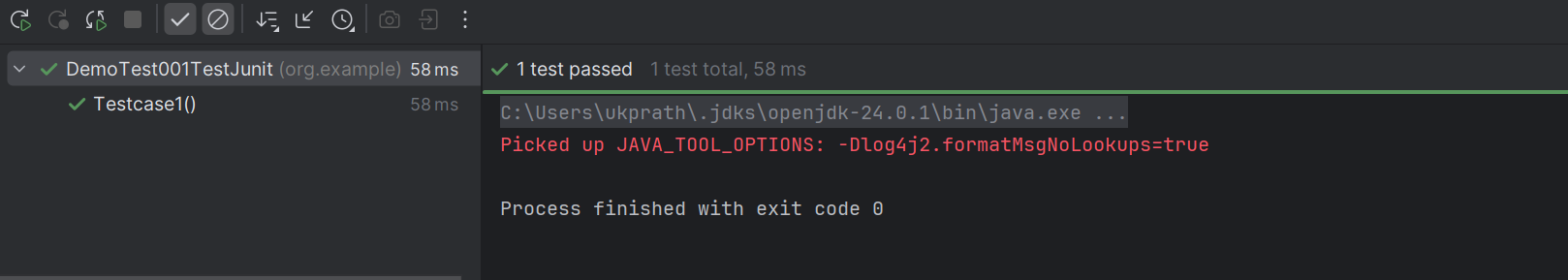
String str = "Pratheesh";

*assertEquals*("Pratheesh", str);

}

}

Output:



Changing name lowercase

package org.example;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*assertEquals*;

public class DemoTest001TestJunit {

@Test

public void Testcase1() {

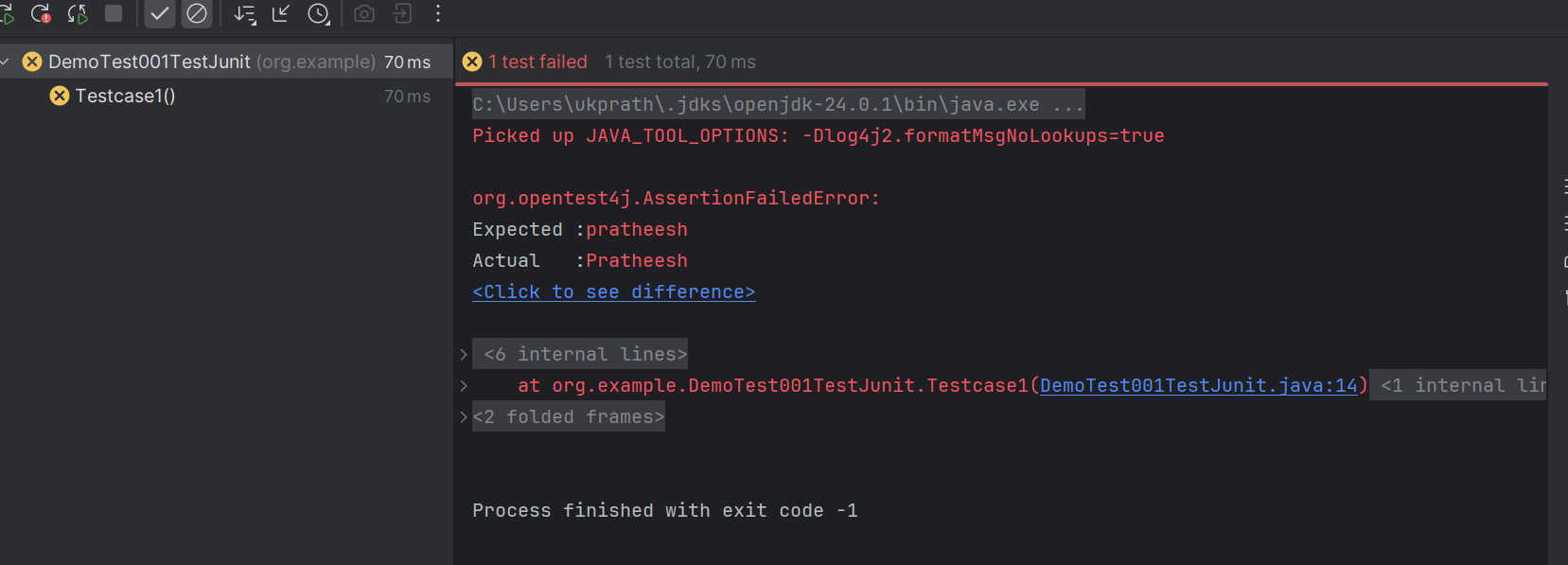
String str = "Pratheesh";

*assertEquals*("pratheesh", str);

}

}

Output



Task 2

package org.example;

import org.junit.jupiter.api.Test;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class DemoTest002TestJunitAssertions {

private static final Logger *logger* = LoggerFactory.*getLogger*(DemoTest002TestJunitAssertions.class);

@Test

void testcase01() {

*logger*.info("start testing");

String res = testcase02();

*logger*.info("testing is done "+res);

}

private String testcase02() {

*logger*.info(" we are in testcase02");

try{

Thread.*sleep*(1000);

}catch(InterruptedException ex) {

Thread.*currentThread*().interrupt();

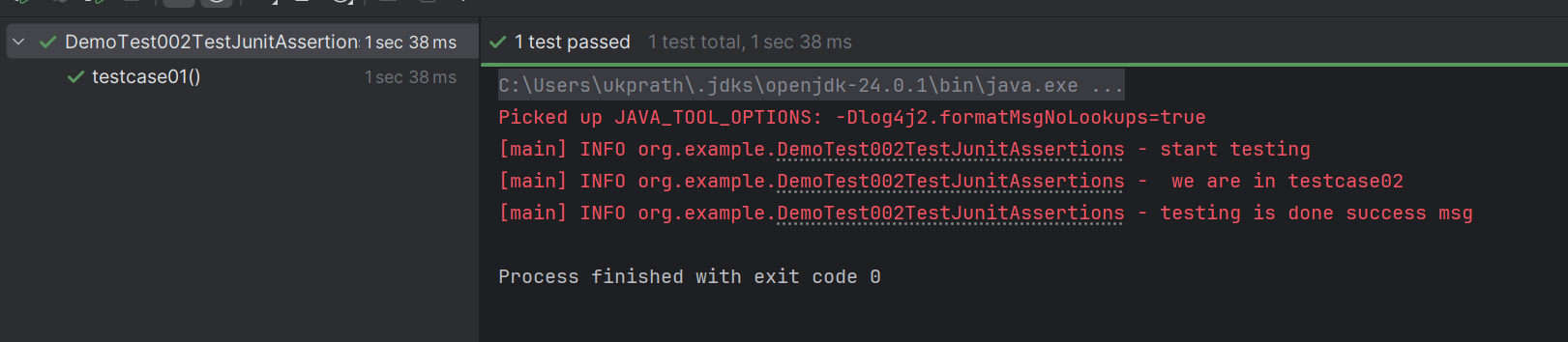
*logger*.error("execution got interruped", ex);

}

return "success msg";

}

}



Task 03 runner

package org.example;

import com.sun.net.httpserver.Authenticator;

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

//DemoTest001TestJunit.class

public class DemoTest002TestJunitRunner {

public static void main(String[] args) {

System.*out*.println("running a test runner code");

Result res = JUnitCore.*runClasses*(DemoTest001TestJunit.class);

for (Failure fail: res.getFailures()) {

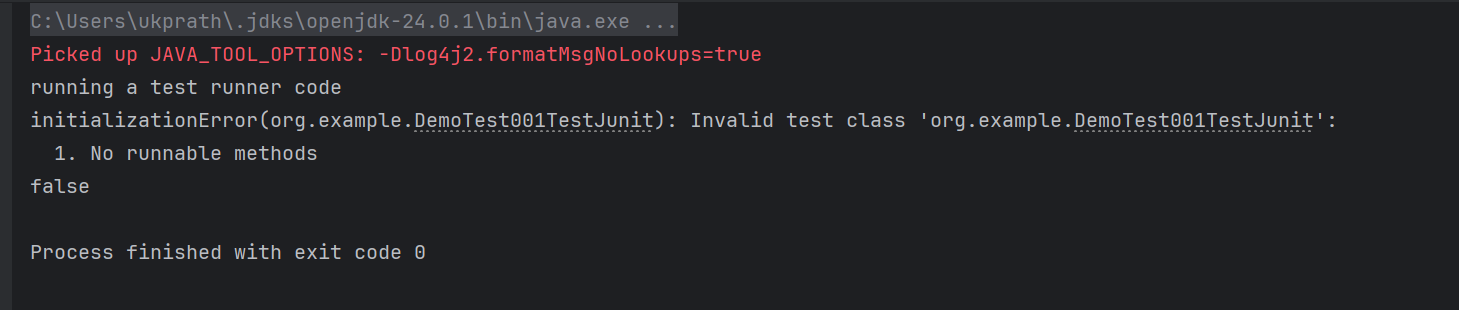
System.*out*.println(fail.toString());

}

System.*out*.println(res.wasSuccessful());

}

}



Task 03:

Launcher code

package org.example;

import org.junit.platform.launcher.Launcher;

import org.junit.platform.launcher.LauncherDiscoveryRequest;

import org.junit.platform.launcher.core.LauncherDiscoveryRequestBuilder;

import org.junit.platform.launcher.core.LauncherFactory;

import org.junit.platform.launcher.listeners.SummaryGeneratingListener;

import org.junit.platform.launcher.listeners.TestExecutionSummary;

import static org.junit.platform.engine.discovery.DiscoverySelectors.*selectClass*;

import static org.junit.platform.engine.discovery.DiscoverySelectors.*selectClass*;

public class DemoTest003TestJunitLauncher {

public static void main(String[] args) {

// Create a listener to collect test results

SummaryGeneratingListener listener = new SummaryGeneratingListener();

// Build a discovery request for the specific test class

LauncherDiscoveryRequest request = LauncherDiscoveryRequestBuilder.*request*()

.selectors(*selectClass*(DemoTest002TestJunitAssertions.class))

.build();

// Create a launcher and register the listener

Launcher launcher = LauncherFactory.*create*();

launcher.registerTestExecutionListeners(listener);

// Execute the tests

launcher.execute(request);

// Print summary

TestExecutionSummary summary = listener.getSummary();

System.*out*.println("Total tests found: " + summary.getTestsFoundCount());

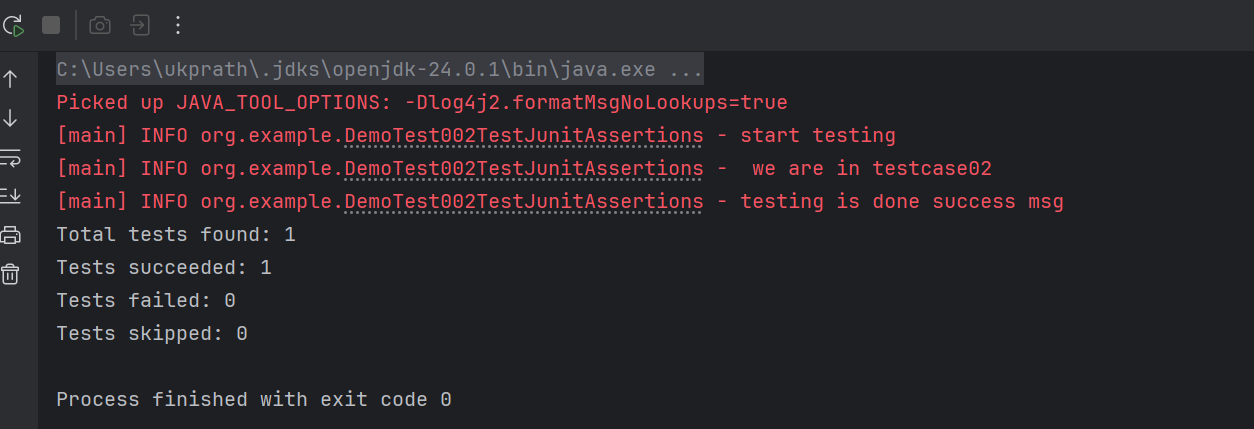
System.*out*.println("Tests succeeded: " + summary.getTestsSucceededCount());

System.*out*.println("Tests failed: " + summary.getTestsFailedCount());

System.*out*.println("Tests skipped: " + summary.getTestsSkippedCount());

}

}



Task 04

Assumptions in junit 5

package org.example;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.Assumptions;

import org.junit.jupiter.api.Test;

/\*Assumptions in JUnit 5 - checks the test cases conditionally based on preconditions.

if Assumption is failed then it is marked as skipped rather than failed.

Means Assumptions are used when we want to skip a test.

assumeTrue() - used to skip the test when a specified condition is not true for assumeTrue or not false for assumeFalse.

assumingThat() - to execute a block of code based on a Boolean Assumptions, If the Assumptions is false the block is skipped.\*/

public class DemoTest004TestJunit5Assumptions {

@Test

void Testcase1() {

boolean condition = "true".equalsIgnoreCase(System.*getProperty*("runTest"));

Assumptions.*assumeTrue*(condition, "as the condition is not met skip test case");

int result = testcase2();

Assertions.*assertEquals*(10, result, "value need to be 10");

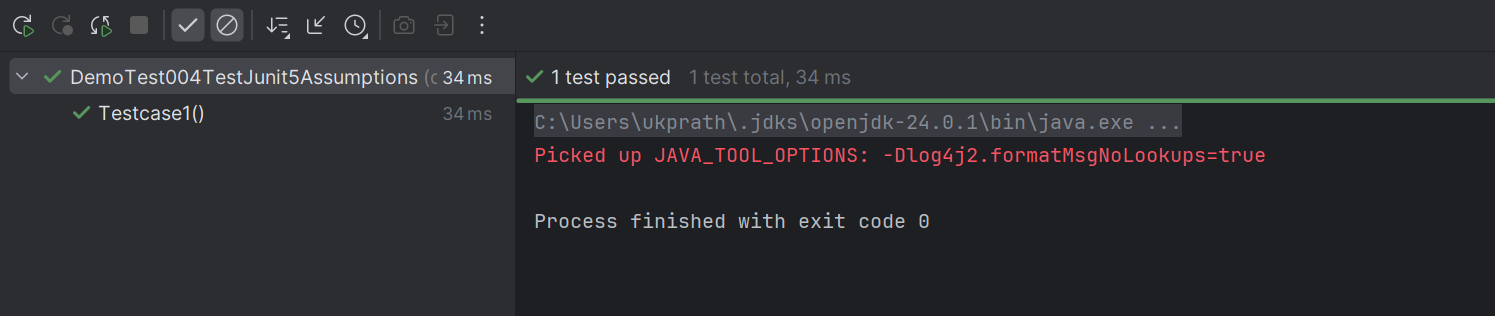
}

private int testcase2() {

return 10;

}

}



Task 05:

Parameterized Test demo

package org.example;

/\*

Parameterized Test - used to test a Test case with different parameters.

use @ParameterizedTest annotations.

\*/

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.*assertEquals*;

public class DemoTest005ParameterizedTestJunit5 {

@ParameterizedTest

@ValueSource(ints = {100, 25, 30, 70, 40})

void testSquare(int value) {

int result = square(value);

*assertEquals*(value \* value, result, " if wrong");

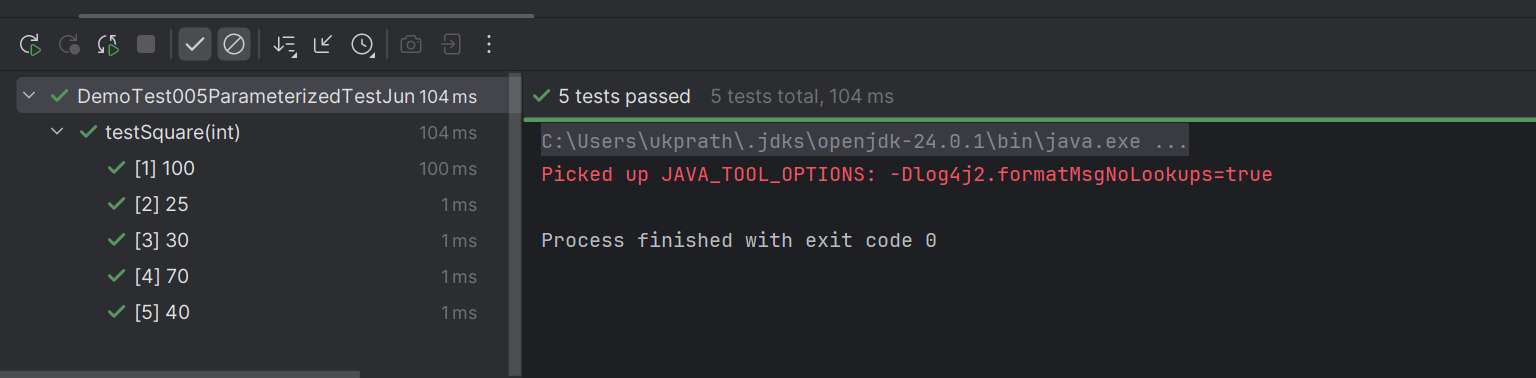
}

private int square(int number) {

return number \* number;

}

}



Task 06:

Dynamic Tests:

package org.example;

/\*Dynamic Tests in JUnit 5 - can be generated at runtime.

using @TestFactory annotation.

\*/

import org.junit.jupiter.api.DynamicTest;

import org.junit.jupiter.api.TestFactory;

import java.util.Collection;

import java.util.stream.Stream;

import static java.util.stream.Collectors.*toList*;

import static org.junit.jupiter.api.Assertions.*assertTrue*;

import static org.junit.jupiter.api.DynamicTest.*dynamicTest*;

public class DemoTest006DynamicTestJunit5 {

@TestFactory

Collection<DynamicTest> DynamicTestCase() {

return TestCases().map(val ->

*dynamicTest*("Dynamic Test: " + val, () -> *assertTrue*(val % 2 == 0))

).toList();

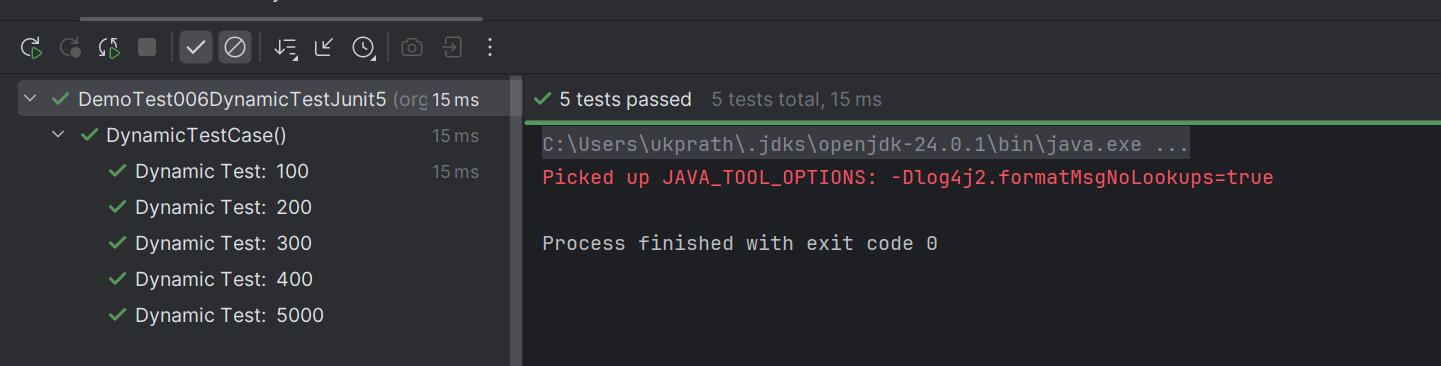
}

private Stream<Integer> TestCases() {

return Stream.*of*(100, 200, 300, 400, 5000);

}

}



Task 07:

Tags fast , slow and then integration filter

package org.example;

/\* Tag and Filter - tag test cases using @tag annotation.

Tags are labels to categorize the test cases.

Filter - used to filter and run the test cases by using the Tags.

\*/

import org.junit.jupiter.api.Tag;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*assertTrue*;

public class DemoTest007Tag\_FilterJunit5 {

@Test

@Tag("slow")

void slowTest() {

*assertTrue*(true, "slow test method");

}

@Test

@Tag("fast")

void fastTest() {

*assertTrue*(true, "fast test method");

}

@Test

@Tag("fast")

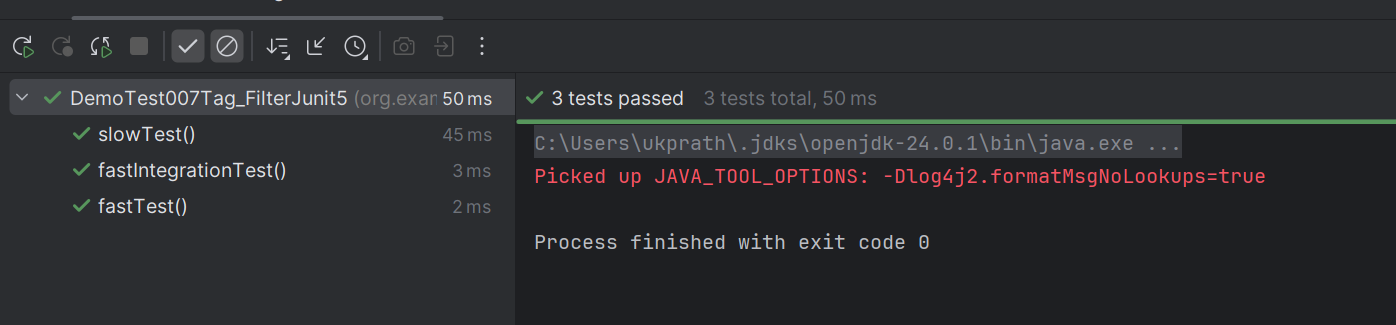
@Tag("integration")

void fastIntegrationTest() {

*assertTrue*(true, "fast integration test methid");

}

}



Task 08:

@AfterEach ,@AfterAll, @BeforeAll, @BeforeEach Annotations use

package org.example;

/\*

@AfterEach ,@AfterAll, @BeforeAll, @BeforeEach Annotations use

\*/

import org.junit.jupiter.api.AfterAll;

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*assertEquals*;

public class DemoTest008AfterBeforeJunit5 {

private int val1;

private int val2;

@Test

void test2() {

System.*out*.println("sample test method");

}

@BeforeAll

static void setupBeforeAll() {

System.*out*.println("run it b4 all tests");

}

@AfterAll

static void cleanupAfterAll() {

System.*out*.println("run it after all tests");

}

@BeforeEach

void setupBeforeEach() {

System.*out*.println("run it b4 each test");

val1 = 10;

val2 = 5;

}

@AfterEach

void cleanupAfterEach() {

System.*out*.println("run it after each test");

}

@Test

void test() {

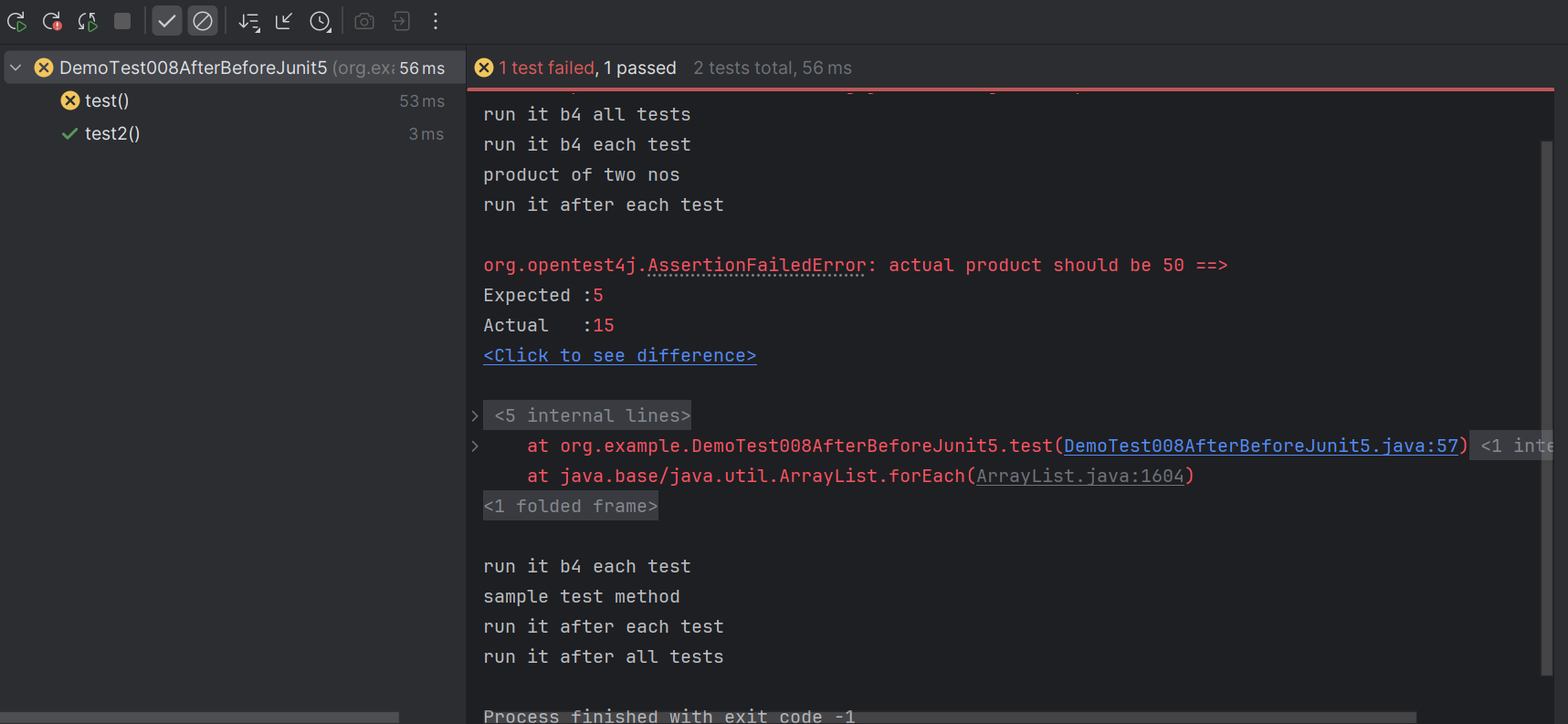
System.*out*.println("product of two nos");

int result = val1 + val2;

*assertEquals*(5, result, "actual product should be 50");

}

}



Task 09:

Testsuite annotation

<dependency>

<groupId>org.junit.platform</groupId>

<artifactId>junit-platform-suite</artifactId>

<version>1.10.0</version> <!-- Use the latest stable version -->

<scope>test</scope>

</dependency>

package org.example;

import org.junit.platform.suite.api.SelectClasses;

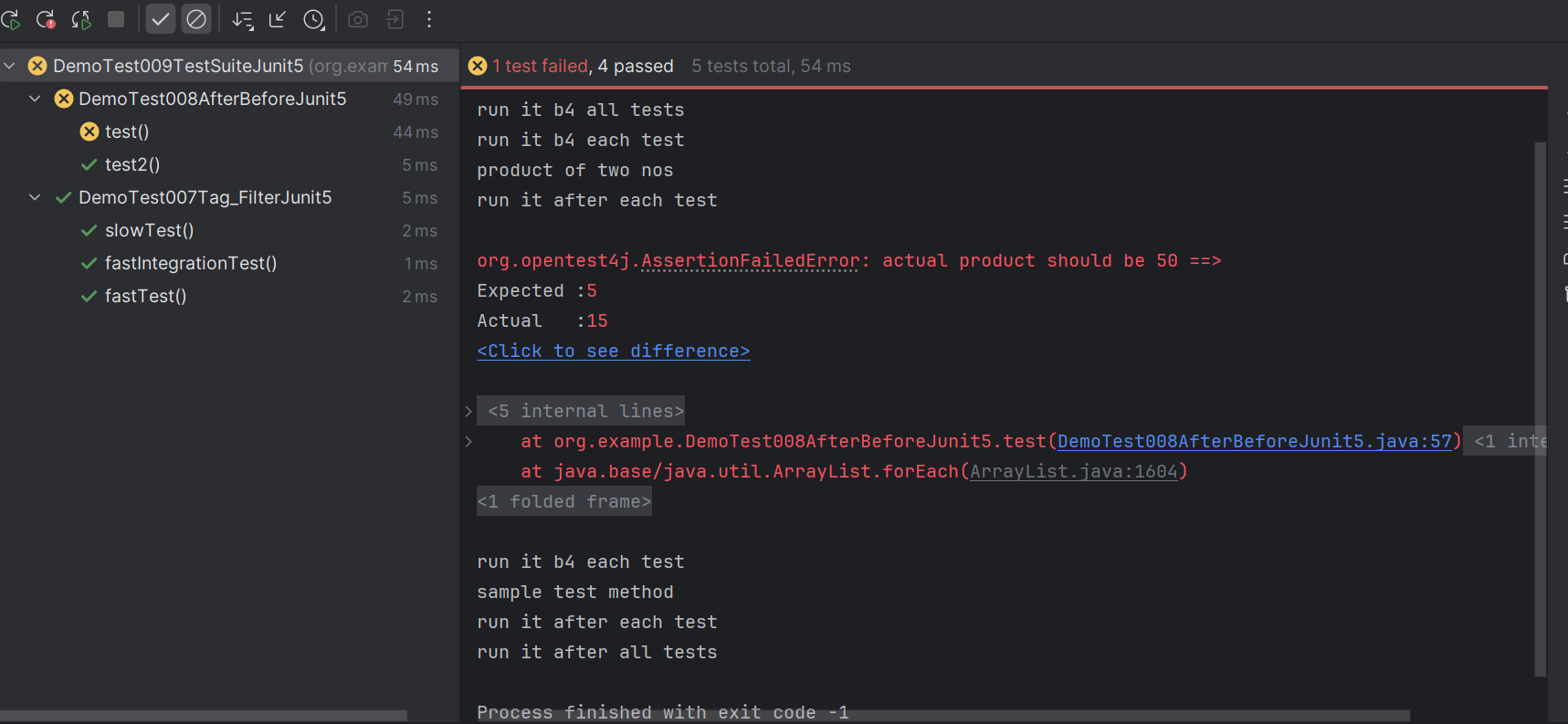
import org.junit.platform.suite.api.Suite;

@Suite

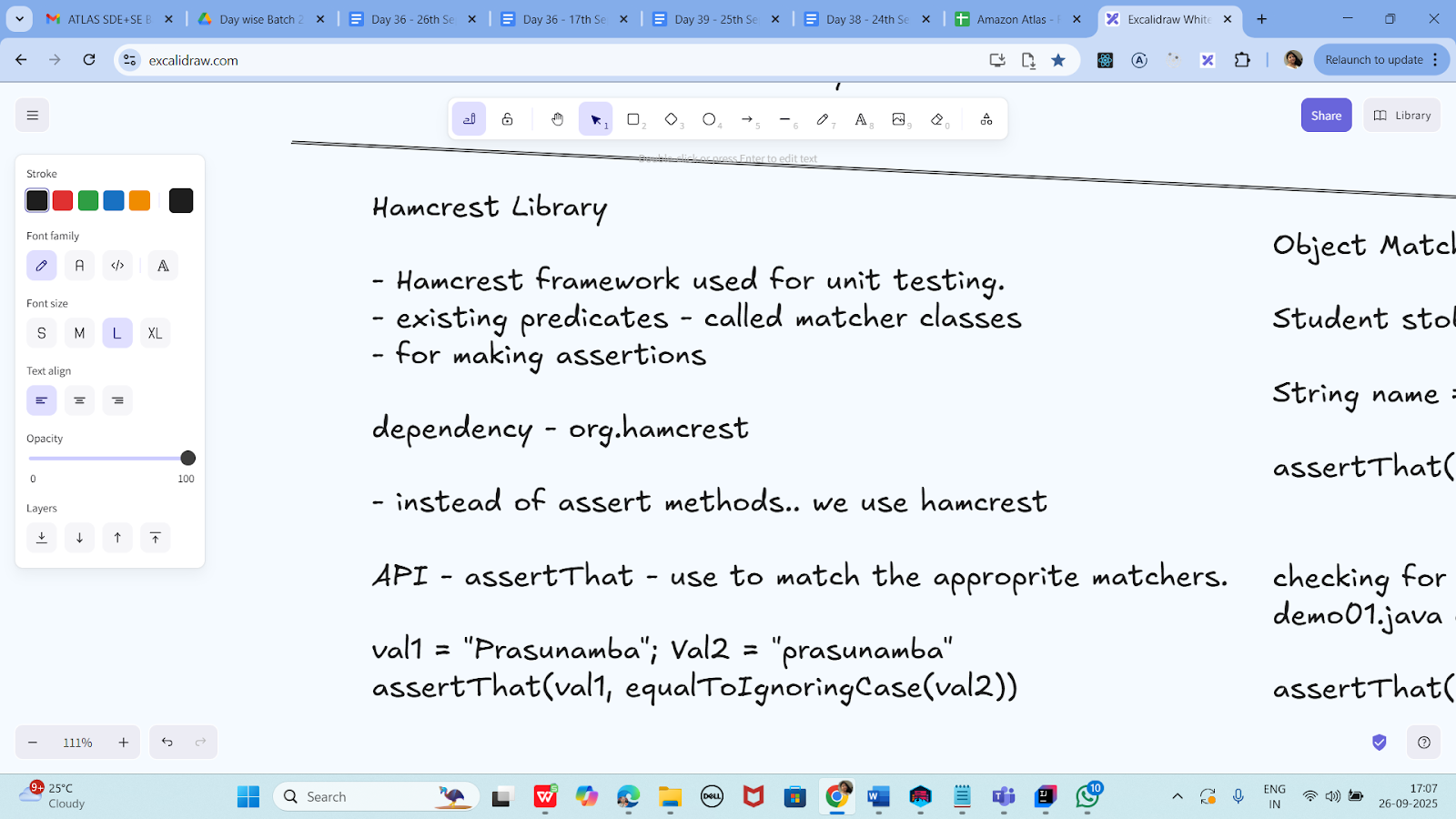
@SelectClasses({DemoTest008AfterBeforeJunit5.class, DemoTest007Tag\_FilterJunit5.class})

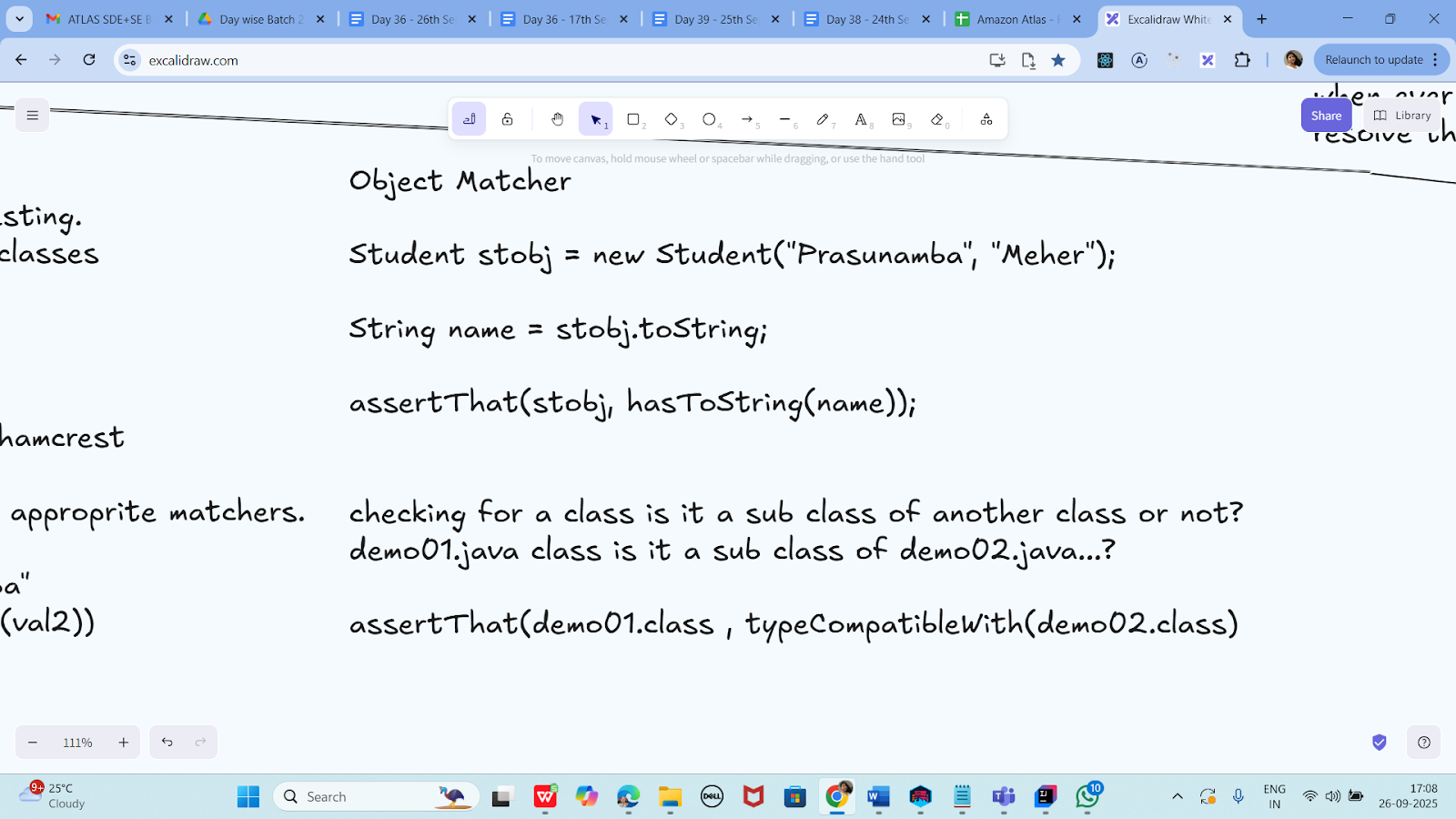
public class DemoTest009TestSuiteJunit5 {

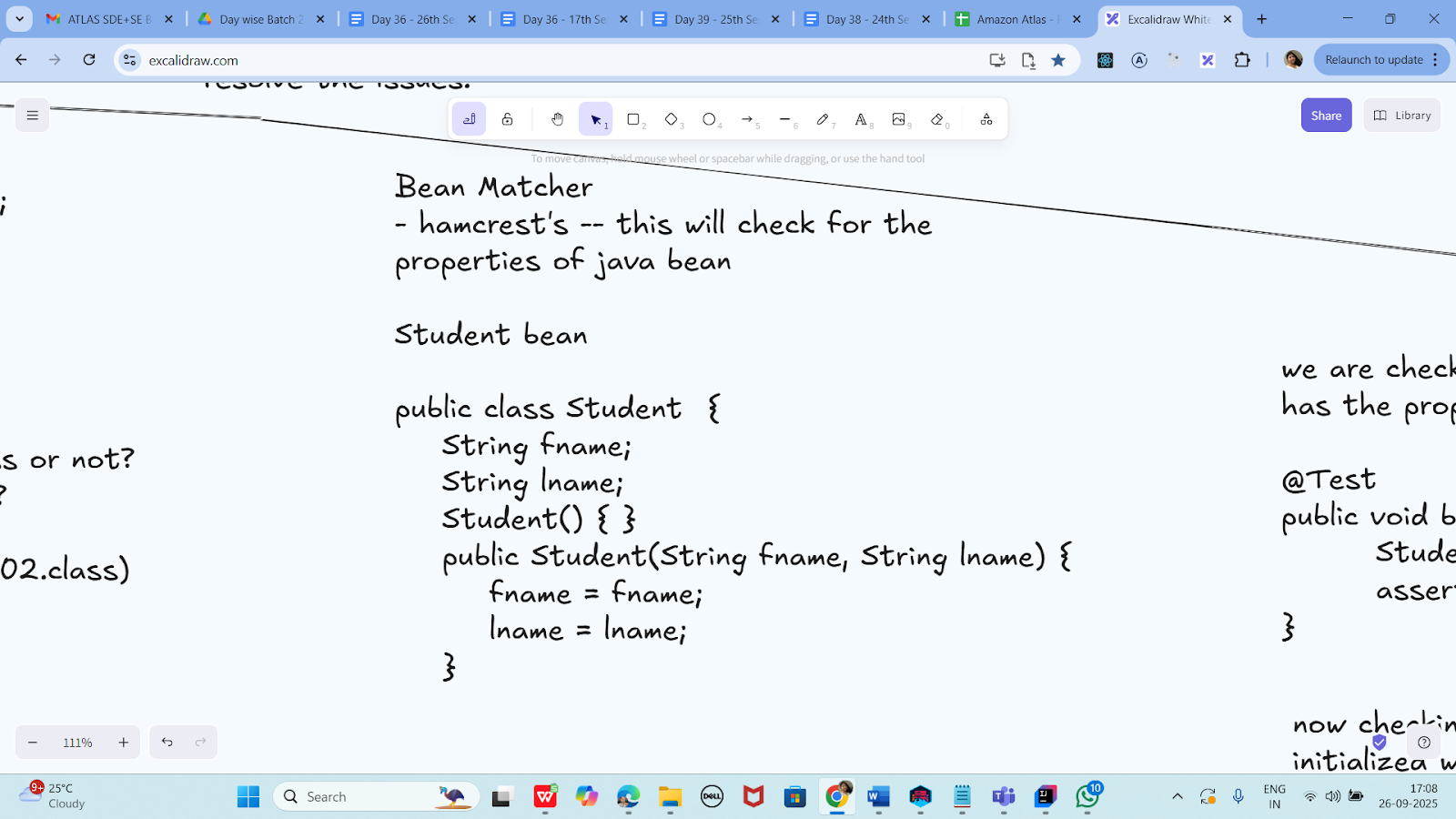
}

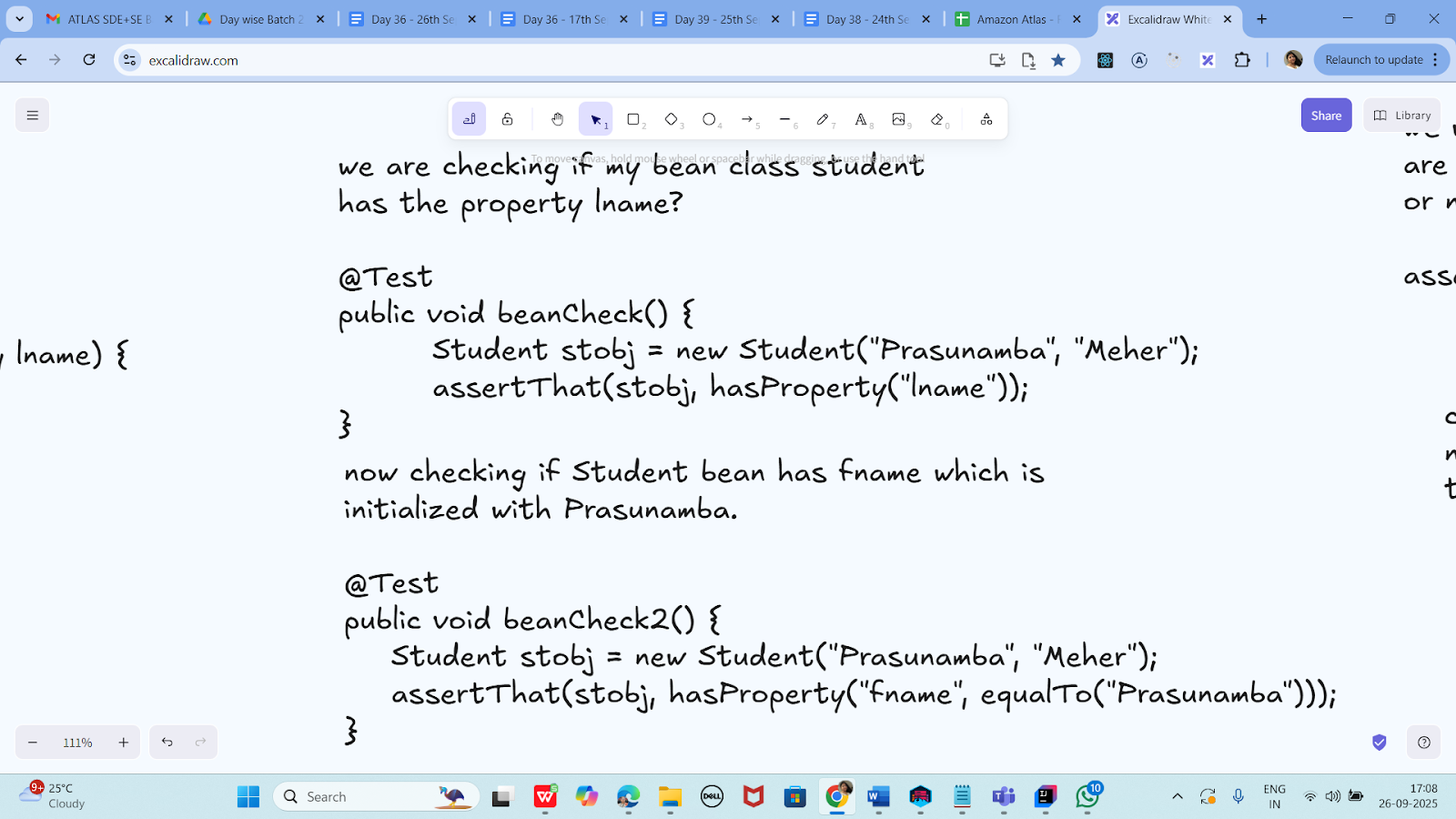


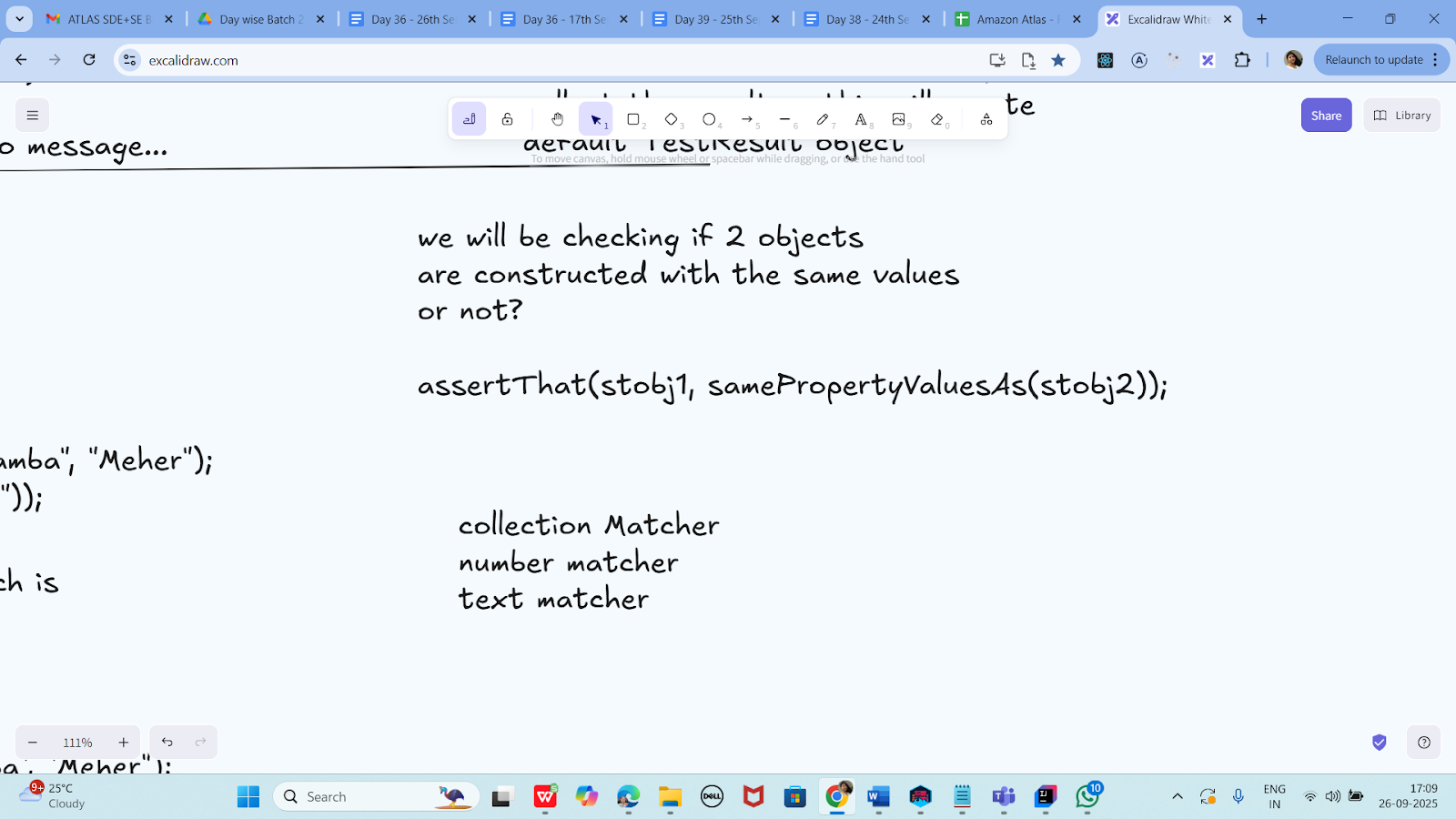
Hamcrest,











pom,

Task 10:

Hamcrest:

<!-- https://mvnrepository.com/artifact/org.hamcrest/hamcrest-all -->

<dependency>

<groupId>org.hamcrest</groupId>

<artifactId>hamcrest-all</artifactId>

<version>1.3</version>

<scope>test

package org.example;

import org.junit.jupiter.api.Test;

import static org.hamcrest.MatcherAssert.*assertThat*;

import static org.hamcrest.Matchers.\*;

public class DemoTest01Hamcrest {

@Test

public void hamcrestMethod() {

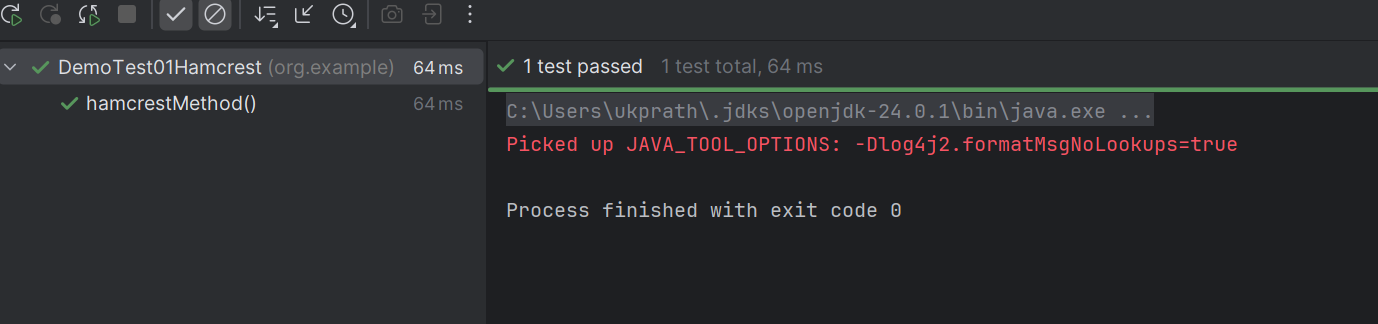
String str1 = "Apple";

String str2 = "Apple";

*assertThat*(str1 ,*equalToIgnoringCase*(str2));

}

}



Task 11:

Customer bean:

package org.example;

public class Customer {

private String Fname;

private String Lname;

public Customer(String Fname, String Lname) {

this.Fname = Fname;

this.Lname = Lname;

}

public String getFname() {

return Fname;

}

public String getLname() {

return Lname;

}

}

Based on the customer bean

package org.example;

import org.junit.jupiter.api.Test;

import static org.hamcrest.MatcherAssert.*assertThat*;//Object Matcher:

import static org.hamcrest.Matchers.*hasToString*;

// we are given a bean ... string correctness

public class DemoTest02Hamcrest {

@Test

public void method1() {

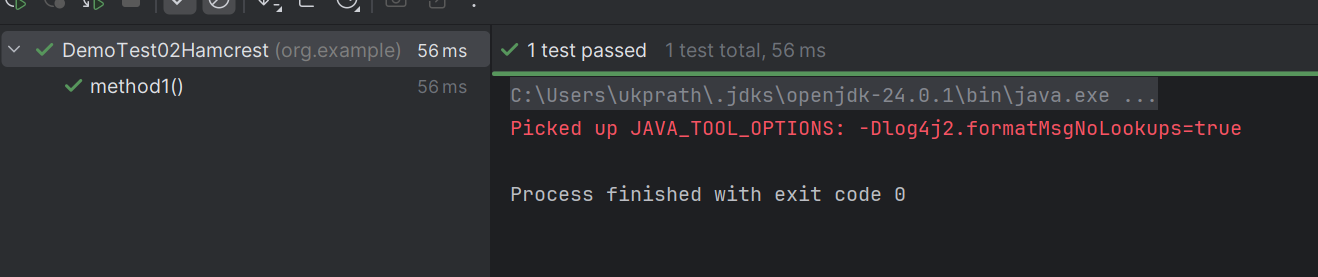
Customer customer = new Customer("John", "Abraham");

String str = customer.toString();

*assertThat*(customer, *hasToString*(str));

}

}



Task 12:

package org.example;

import org.junit.jupiter.api.Test;

import static org.hamcrest.MatcherAssert.*assertThat*;

//import static org.hamcrest.Matchers.hasProperty;

import static org.hamcrest.Matchers.\*;

//now check if bean has the property

public class DemoTest03Hamcrest {

// @Test

// public void checkMethod() {

// Customer customer = new Customer("John", "Abraham");

// assertThat(customer, hasProperty("Fname"));

// }

@Test

void method01() {

Customer customer = new Customer("john", "Abraham");

*assertThat*(customer, *hasProperty*("Lname"));

}

}

Task 13:

package org.example;

import org.junit.jupiter.api.Test;//collection Matcher

import java.util.ArrayList;

import java.util.List;

import static org.hamcrest.MatcherAssert.*assertThat*;

import static org.hamcrest.Matchers.*empty*;

public class DemoTest04Hamcrest {

@Test

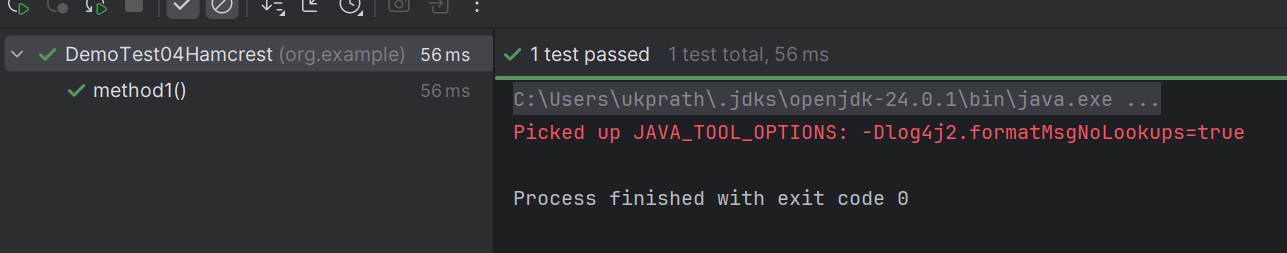
public void method1() {

List<String> custList = new ArrayList<>();

*assertThat*(custList, *empty*());

}

}



Task 14:

package org.example;

import org.junit.jupiter.api.Test;//collection Matcher

import java.util.Arrays;

import java.util.List;

import static org.hamcrest.MatcherAssert.*assertThat*;

import static org.hamcrest.Matchers.*hasSize*;

public class DemoTest05Hamcrest {

@Test

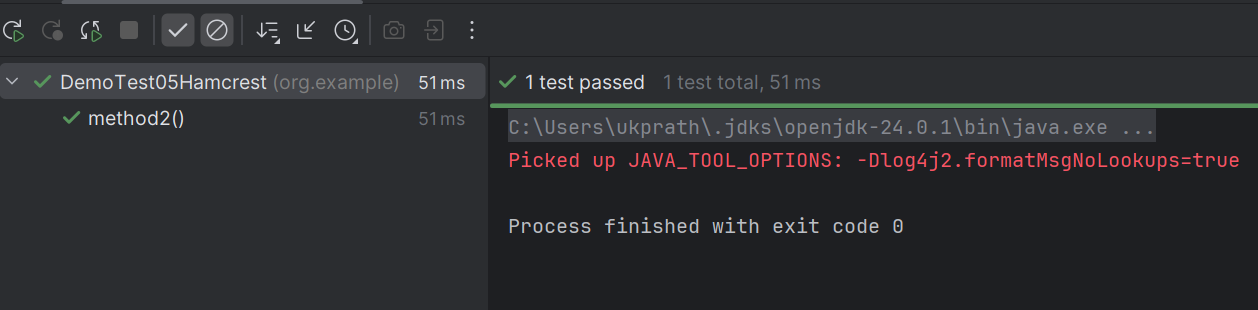
public void method2() {

List<String> custList = Arrays.*asList*("john", "Mary", "Sheik", "Singh");

*assertThat*(custList, *hasSize*(4));

}

}



Home tasks:

TASK 01: do it as home Task …. After the session

Work on Mobile app test cases…

src

main

java/com/example/MobileStore

controller

MobileController.java

dto

MobileDTO.java

entity

Mobile.java

exception

ExceptionHandlerController.java

ResouirceNotFoundException.java

repo

MobileRepository

service

impl

MobileServiceImpl.java

Mobileservices.java

test

MobileStoreTests.java

MobileApplication.java

The code links are provided in the info box..

Task 02:

Which of the following JUnit annotations is responsible for executing code only once before all test cases in a test class?

1. @BeforeEach executes before each test method and is run repeatedly for each test case.

2. @BeforeClass is used for static methods that are executed only once before any test methods in the class.

3. @Before is used for non-static methods and runs before every test case, not once.

4. @BeforeSuite belongs to TestNG and not valid in JUnit.

Task 03:

What makes parameterized tests in JUnit useful in test automation?

1. They allow for random test data generation during runtime without controlling input values explicitly.

2. They enable the creation of GUI-based test scenarios where user input is simulated for each test.

3. They help execute the same test logic repeatedly with different sets of input data, improving test coverage without redundant code.

4. They allow test cases to be grouped based on the execution time, helping to manage performance bottlenecks.

Task 04:

What is the primary role of an automation framework in the software testing life cycle?

1. It replaces the need for testers entirely by using AI-based code generators.

2. It helps design and run a collection of test cases with consistent coding standards, reusability, and maintainability.

3. It is used only to log the output of manual testing activities and store reports.

4. It ensures all testing is carried out on production systems with real-time user traffic.

Task 05:

What does the @Test(expected = Exception.class) annotation do in a JUnit test method?

1. It suppresses all exceptions thrown during the test execution regardless of type.

2. It validates that the test method throws the specific exception type; if not, the test fails.

3. It configures the test to ignore exceptions for logging purposes.

4. It is used to pass exceptions from test methods to external listeners for monitoring.

Task 06:

How does a data-driven automation framework typically manage test data?

1. It uses database tables to execute tests and validate outputs using stored procedures.

2. It externalizes test inputs and expected outputs, often in files like CSV, Excel, or XML.

3. It retrieves test data from real-time user sessions and uses them dynamically.

4. It stores test data as constants within test methods to reduce configuration.

Task 07:

You are testing a backend API method using JUnit that fetches user details. The actual response contains several optional fields which are not always present. What is a good approach to validate such variable structures?

1. Validate only the mandatory fields and skip the optional ones in the test.

2. Use assertTrue(response.length() > 0) to confirm the object exists.

3. Write a dynamic Hamcrest matcher that checks presence conditionally based on input scenario.

4. Compare the entire object string with a pre-saved expected value.

Task 08:

You are debugging a failing test where an object property is not matching the expected value. What’s a good use of JUnit's logging or assertion enhancement features to aid diagnosis?

1. Disable the test temporarily and test manually.

2. Add System.out.println() to print expected values only.

3. Use custom assertion messages with assertEquals(expected, actual, "Expected value mismatch") for clarity in failure logs.

4. Use assertions inside the application code itself.

Task 09:

Which best explains the concept of "Assertion" in unit testing?

1. Assertions are helper classes that display the execution order of test methods.

2. Assertions allow writing alternate code paths in tests to simulate branching logic.

3. Assertions are conditions embedded in test cases to confirm the correctness of expected versus actual behavior. A

4. Assertions track how many lines of code have been executed during a test.

Task 10:

Why is the assertThrows() method used in JUnit 5?

1. It validates whether a test method is ignored during execution.

2. It verifies whether a method throws a specified exception and allows capturing it for further validation.

3. It ensures all tests are executed sequentially.

4. It detects memory leaks by logging thrown objects.

Task 11:

In automation testing, what does the term “reusability” typically refer to?

1. The ability to execute manual tests on multiple platforms simultaneously.

2. The design of test scripts and functions in a way that they can be used across multiple test cases and modules with minimal changes.

3. The option to replay previous test logs for historical validation.

4. The integration of source control within test execution environments.

Task 12:

How does the data-driven automation framework differ from a traditional hardcoded test script?

1. Data-driven frameworks rely on UI events instead of actual inputs for validation.

2. It isolates test logic from test data, allowing multiple scenarios to run using external datasets without altering the code.

3. It merges expected and actual results into the same source file for traceability.

4. It executes tests only during system downtime, ensuring resource optimization.

Task 13:

In JUnit, what is the correct use case for @AfterAll annotation?

1. To execute a method after every test method in the test class completes.

2. To mark a method that runs only after all test methods in the current class are executed, and is typically used for cleanup.

3. To configure exception validation logic after each test execution.

4. To run the annotated method only if all tests in the suite fail.

Task 14:

Which annotation in JUnit is used to mark methods that should be run after each test case?

1. @AfterAll is used to run a method once after all test cases are executed.

2. @Cleanup ensures memory cleanup post test execution.

3. @AfterEach ensures a specific method is executed after every individual test case, often for cleanup purposes.

4. @PostTest is used in legacy JUnit versions for teardown activities.

Task 15:

Why is it important to use assertions rather than print statements in JUnit testing?

1. Assertions are required by the Java compiler for test execution to succeed.

2. Assertions do not require test case structuring, making them more convenient than print statements.

3. Assertions validate expected outcomes and allow automated failure detection, unlike print statements which require manual review.

4. Print statements are ignored by test runners while assertions are always enforced.

If done from task 2 to task 15 - raise your hands..

Task 16:

What does a keyword-driven automation framework focus on?

1. It maps business keywords to automated scripts, enabling non-technical users to write test scenarios using readable terms.

2. It uses keywords defined in source code comments to determine the execution flow.

3. It enables automated scripts to detect programming keywords such as if, else, and try.

4. It executes test cases based on chronological keywords extracted from system logs.

Task 17:

How does the Hamcrest library enhance the use of assertions in JUnit?

1. It transforms test results into color-coded HTML reports for better visualization.

2. It enables combining multiple assertions into a single, monolithic assertion block.

3. It allows writing test cases that adapt to dynamic UI elements in Selenium.

4. It provides a set of matcher objects to write readable and expressive conditions for asserting complex logical states.

Task 18:

What is the difference between @BeforeEach and @BeforeAll annotations in JUnit?

1. Both are functionally identical but differ in naming conventions for readability.

2. @BeforeAll applies only to test classes annotated with @ParameterizedTest; @BeforeEach does not.

3. @BeforeEach is run once before all test methods; @BeforeAll is run before each test.

4. @BeforeEach runs before every test method; @BeforeAll runs once before any test methods in the class.

Task 19:

How does the Hamcrest library enhance the use of assertions in JUnit?

1. It transforms test results into color-coded HTML reports for better visualization.

2. It enables combining multiple assertions into a single, monolithic assertion block.

3. It allows writing test cases that adapt to dynamic UI elements in Selenium.

4. It provides a set of matcher objects to write readable and expressive conditions for asserting complex logical states.

Task 20:

What is the main responsibility of the test automation life cycle phase called "Test Design"?

1. It involves identifying test conditions, selecting input data, and defining expected outcomes.

2. It focuses on selecting test tools and setting up environments for script execution.

3. It involves coding the test cases in the programming language supported by the framework.

4. It tracks test coverage and collects metrics post test execution.

Task 21:

Which characteristic of JUnit makes it a widely adopted tool for unit testing in Java applications?

1. JUnit integrates real-time cloud-based performance testing capabilities.

2. JUnit requires no assertions and passes tests automatically.

3. JUnit is annotation-based, supports automation and integration with build tools like Maven and Gradle.

4. JUnit enables graphical test development through drag-and-drop components.

Task 22:

Why are parameterized tests preferred in regression testing scenarios?

1. They make the test output easier to interpret by customizing the exception trace.

2. They help re-use a single test logic with multiple datasets, improving test coverage without duplicating code.

3. They execute only one test per data set to reduce overhead and improve performance.

4. They allow conditional test skipping based on configuration files.

Task 23:

What is the role of an automation tool such as Selenium in test frameworks?

1. It records audio-based voice commands for accessibility testing.

2. It provides hardware resource simulation for embedded systems testing.

3. It interacts with web elements like buttons, input fields, and validates UI behaviors across browsers.

4. It compiles and packages Java projects and deploys them to application servers.

Task 24:

What benefit does using test timeouts in JUnit offer during CI/CD integration?

1. It allows the test to self-heal and retry upon failure.

2. It controls logging verbosity for lengthy tests.

3. It schedules test execution based on server load to optimize performance.

4. It ensures each test completes within a defined time window, avoiding long-running or stuck tests during automation pipelines.

Task 25:

Why is exception testing essential in unit tests for enterprise applications?

1. It prevents the test class from being garbage collected.

2. It helps identify all unused code branches during test execution.

3. It allows skipping business rules validation in enterprise logic.

4. It enables checking whether code throws expected exceptions for invalid inputs or failed operations, improving robustness.

Task 26:

What is the significance of the @DisplayName annotation in JUnit 5?

1. It defines environment-specific parameters for parameterized tests.

2. It marks the test case as deprecated.

3. It sets a custom name for the test method which will appear in test reports and logs, improving readability.

4. It is used to group tests based on their priority.

Task 27:

Why are test automation frameworks essential in enterprise-level software testing?

1. They simplify manual report generation by developers.

2. They reduce testing time and increase accuracy by providing structure, reusability, and consistency to test scripts.

3. They allow bypassing security validations in staging environments.

4. They store user credentials securely for end-to-end validation.

Task 28:

How does the use of @ParameterizedTest benefit testing in JUnit 5?

1. It allows conditional execution of tests based on runtime memory.

2. It enables a single test method to run multiple times with different input values, reducing redundancy.

3. It forces tests to run in parallel threads by default.

4. It integrates external APIs into test methods using parameter injection.

Task 29:

What distinguishes JUnit as a framework suitable for Test-Driven Development (TDD)?

1. It provides graphical debugging and tracing capabilities that aid in test automation.

2. It enforces a mandatory rule of test creation before code implementation.

3. It allows fast feedback by executing tests frequently during development with minimal configuration.

4. It automatically converts manual test steps into code.

Task 30:

What happens when you use assertEquals() with mismatching expected and actual values in JUnit?

1. The test continues execution without any logging.

2. The test logs a warning but proceeds to the next test method.

3. The test fails immediately and throws an AssertionError highlighting the mismatched values.

4. The mismatch is silently ignored unless a debugger is attached.

If done till here plz raise your hands..

Task 31:

How does a keyword-driven automation framework enhance collaboration between technical and non-technical stakeholders?

1. It supports version control through internal commit logs.

2. It uses descriptive action words mapped to low-level functions, enabling non-programmers to create and manage test scripts.

3. It restricts test design only to business analysts, reducing developer dependency.

4. It auto-generates test cases from requirement documents.

Task 32:

Which of the following is a limitation of not using test automation in Agile development?

1. Manual testing allows more frequent feedback cycles.

2. Manual testing handles integration testing more efficiently.

3. Lack of automation increases repetitive effort and delays feedback on code changes.

4. Test automation creates unnecessary complexity in short release cycles.

Task 33:

A large enterprise uses multiple test automation tools. Some testers prefer TestNG, others prefer JUnit. The team wants to standardize reporting across all test runs. What is the best approach?

1. Use separate HTML report generators per framework.

2. Export results to a central database manually.

3. Integrate a reporting tool like Allure or ReportPortal that supports multiple test frameworks.

4. Replace all frameworks with custom logging.

Task 34:

In a parameterized test, a developer observes that values are not being passed into the test method. What could be the probable issue?

1. Test data is declared inside the method instead of an annotation source.

2. The data provider method is not returning a stream or array of the correct type.

3. JUnit doesn’t support parameterized tests in version 5.

4. Test method is not static.

Task 35:

A tester is using assertThat() to verify object equality in a domain class. Despite identical fields, the test fails. What might be the reason?

1. assertThat() doesn't support object comparisons.

2. The object comparison logic is flawed and needs explicit use of matchers like samePropertyValuesAs.

3. The objects being compared are null.

4. Hamcrest cannot be used on custom types.

Task 36:

A test for a payment module fails when run in parallel with other tests due to shared data being overwritten. What best practice should be followed in automation frameworks?

1. Ensure test data is randomized or isolated per execution context to avoid conflicts.

2. Schedule payment tests to run after all others.

3. Use Thread.sleep() to wait for other tests to complete.

4. Disable concurrent test execution in the build pipeline.

Task 37:

A test suite using the Page Object Model has become difficult to maintain due to duplication across different flows. What should the team consider to improve code reuse?

1. Introduce conditional logic in each test to reuse steps across flows.

2. Merge all page classes into one for simpler structure.

3. Create reusable utility methods in base classes or helper classes to abstract common page interactions.

4. Use external test case management tools to handle reuse.

Task 38:

During a regression run, certain JUnit test methods are failing only on specific environments. Testers suspect an environmental issue. What feature can be used to conditionally execute tests based on runtime environment?

1. Use @BeforeEach to check the environment and log it.

2. Add conditional if statements inside the test method.

3. Use @EnabledIfEnvironmentVariable or @EnabledOnOs annotations in JUnit 5 to control test execution.

4. Wrap tests in shell scripts that filter tests based on OS.

Task 39:

What is the purpose of organizing test methods in a test suite in JUnit?

1. It prevents execution of unrelated test methods.

2. It allows running multiple test classes together in a batch, improving execution efficiency and logical grouping.

3. It ensures each method runs with a different JVM instance.

4. It disables dependency injection during test execution.

Task 40:

What is the advantage of using assertTrue() and assertFalse() in JUnit tests?

1. They enable capturing screenshots on test failure.

2. They validate Boolean expressions and ensure conditional logic behaves as intended during test execution.

3. They skip test execution based on runtime parameters.

4. They automatically log warnings in case of unexpected test behavior.

Task 41:

A QA engineer is developing a JUnit test suite for a complex financial application where hundreds of individual unit test classes need to be executed regularly. The current setup executes each class separately, causing delays and inconsistency. What feature should be used to address this?

1. Group all test classes into a single package to automatically execute them.

2. Annotate each test method with @TestGroup for grouped execution.

3. Create a test suite using @Suite, grouping test classes for batch execution via JUnit's runner.

4. Use a loop to invoke test classes manually from the main method.

Task 42:

You are testing a method that calculates tax based on income and residency. The logic throws IllegalArgumentException if income is negative. You want to ensure this behavior in your test. What is the most appropriate way to test this in JUnit 5?

1. Surround the method in try-catch and fail the test if no exception occurs.

2. Use assertThrows(IllegalArgumentException.class, () -> methodCall()) to verify the exception.

3. Use @ExpectedException on the test method to mark it.

4. Annotate the test with @ExceptionTest(IllegalArgumentException.class).

Task 43:

While writing tests for a currency conversion module, a developer wants to validate the output using complex matching rules such as verifying range, type, and conditions. The current assertions are too rigid. Which approach should be adopted?

1. Use Hamcrest matchers to express flexible, human-readable conditions like greaterThan, instanceOf, etc.

2. Replace assertions with log checks that confirm the value is printed correctly.

3. Switch to print-based validations where ranges are visually verified.

4. Use assertEquals with rounding logic to handle conditions.

Task 44:

An automation tester wants to validate the login functionality across multiple sets of user credentials. The current approach duplicates the same test logic multiple times. What is the best JUnit feature to avoid redundancy?

1 Run the same test method in a loop with inputs hardcoded inside it.

2. Implement @TestFactory to generate dynamic test cases manually.

3. Use @ParameterizedTest with data sources like @CsvSource or @MethodSource to feed multiple test inputs.

4. Create multiple test methods each calling the login with different inputs.

Task 45:

During CI pipeline execution, some tests intermittently fail due to asynchronous background jobs not completing. The team is using JUnit with Selenium WebDriver. What is the best solution to make the tests more reliable?

1. Use Thread.sleep() for a safe time buffer.

2. Implement WebDriver waits combined with assertion polling to wait for specific conditions before asserting.

3. Move all assertions to the end of the test method to avoid early failures.

4. Execute the tests in isolation from the pipeline to avoid interference.

Task 46:

A developer working on a REST API wants to validate JSON responses based on key-value pairs, ensuring the exact response content. Traditional assertions are becoming too verbose. What library or approach can simplify this within JUnit tests?

1. Use regular expressions inside assertTrue() to match JSON strings.

2. Parse the JSON manually using string splitting and compare fields.

3. Integrate Hamcrest matchers or use JSONAssert for structured JSON comparison.

4. Convert responses to POJOs and compare using equals().

Task 47:

A financial services firm uses a keyword-driven automation framework to test a legacy ERP system. Manual testers often struggle to define scenarios without technical help. What change would improve efficiency without rewriting the framework?

1. Add developer-centric logs to each keyword for internal debugging.

2. Convert the keyword system to a data-driven one to simplify execution.

3. Enhance keyword definitions with descriptions and tooltips, and document them in a user-friendly format.

4. Integrate version control into the test keyword definition files.

Task 48:

A project has thousands of automated test cases built using JUnit, but execution time is increasing rapidly. What JUnit feature can be used to manage which tests run based on environment or context?

1. Split test classes into smaller ones to reduce runtime.

2. Use @Tag annotations to categorize tests and run selected groups using filters.

3. Use Maven to limit test class size through configuration.

4. Implement System.exit() in long tests to terminate early.

Task 49:

An e-commerce company has automated UI flows with Selenium and JUnit. Developers often skip running all tests before commits due to long test cycles. What is a practical strategy to maintain coverage without compromising speed?

1. Run all test cases manually every alternate day.

2. Convert the entire framework to manual testing temporarily.

3. Categorize tests using JUnit @Tag and prioritize critical ones to run on commit, others in nightly builds.

4. Reduce test count by removing low-priority test cases.

Task 50:

While writing tests for a RESTful web service that returns variable responses based on user location, developers face difficulty verifying dynamic content. How can Hamcrest help in this context?

1. Hamcrest allows strict matching of fixed outputs for reliable test behavior.

2. Hamcrest provides matchers like containsString, startsWith, and matchesPattern for validating dynamic strings flexibly.

3. Hamcrest lets you simulate geographical parameters for test cases.

4. Hamcrest disables assertions for dynamic content to avoid false positives.

Task 51:

In a stock trading platform, a JUnit test verifies exception scenarios for trade execution failures. The test is passing even when no exception is thrown. Which might be the issue with the test implementation?

1. The exception was caught in the application and silently logged.

2. The method under test is annotated with @Disabled, preventing it from executing.

3. The test environment isn't throwing exceptions due to disabled validations.

4. The test is using try-catch without rethrowing or asserting failure within the catch block.

Task 52:

A QA team is using JUnit 5 in a CI/CD pipeline. They wish to isolate long-running integration tests from fast-running unit tests. What feature should they adopt?

1. Write all integration tests in a separate class and exclude that file manually.

2. Introduce conditional test execution using if statements in test methods.

3. Replace JUnit with TestNG which has better performance filtering.

4. @Tag("integration") for long-running tests and run unit tests only with specific tags.

Task 53:

A developer builds an automated test using a data-driven framework. However, the same dataset is accidentally used across test cases, causing logical errors. What should be introduced to prevent such mistakes?

1. Introduce variable scoping rules in the test runner class.

2. Isolate datasets per test using parameter providers like @CsvFileSource or @MethodSource.

3. Use assertions to validate that each test case starts with different data.

4. Hardcode separate values directly in each test method.

Task 54:

Your team uses Selenium and JUnit for automation. Due to frequent UI changes, element locators often break, resulting in flaky tests. What is the best strategy to mitigate this?

1. Use advanced CSS selectors and hardcoded waits.

2. Introduce a page factory design and maintain locators in one central location per page.

3. all locators into environment variables.

4. Delay test execution until UI is finalized.

Info box:

Excalidraw link updated at 11.02

<https://excalidraw.com/#json=lDPvjgOIPrFFQ3SIcSF2l,aa2bHnuOtcPqYfgXfkWfSw>

<https://drive.google.com/drive/folders/1LwhNov1s1-vHzF9GPAObLSnP9kAvipmw?usp=sharing>

For mobile app.. Code to write the test cases…

write Junit & Mockito testcases to verify mobileController and mobileServiceImplclasses.