**day36\_107856406\_dsdipt\_sudipto\_17september2025**

**Employee Code:** 107856406

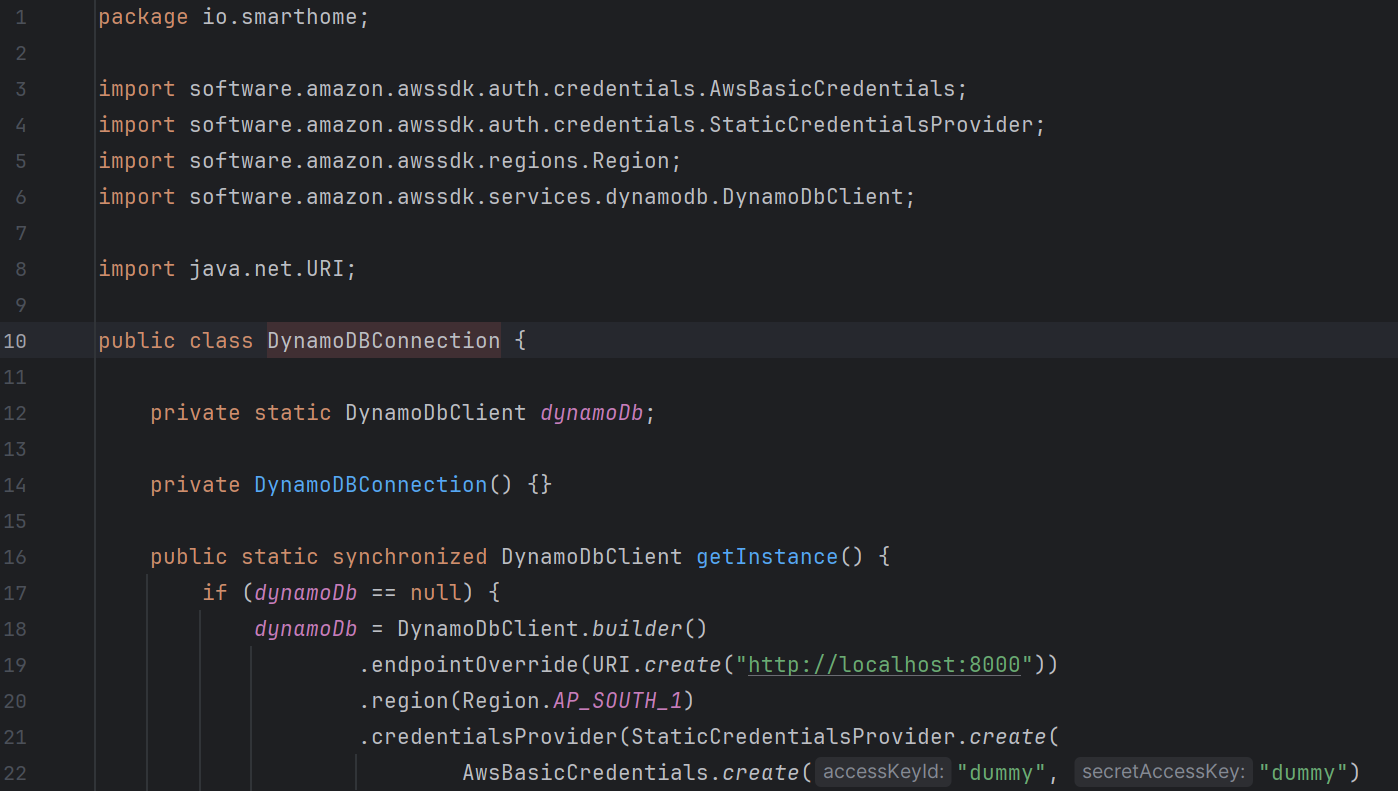
**Login ID:** dsdipt

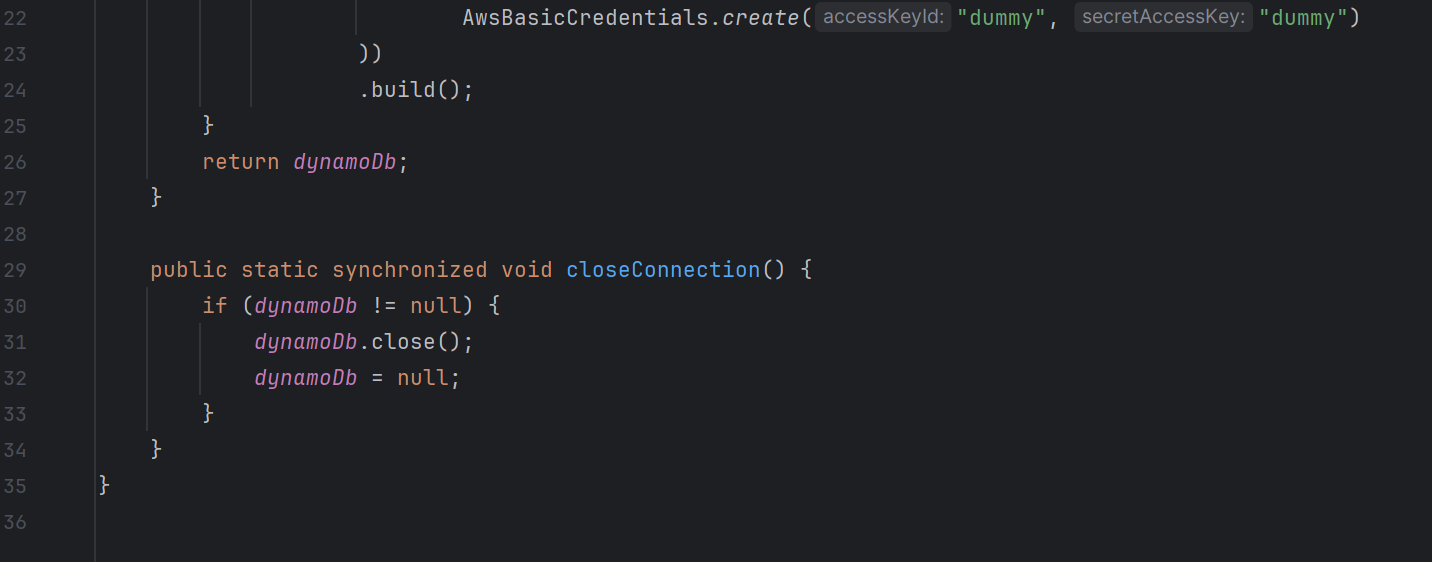
**Email:** dsdipt@amazon.com

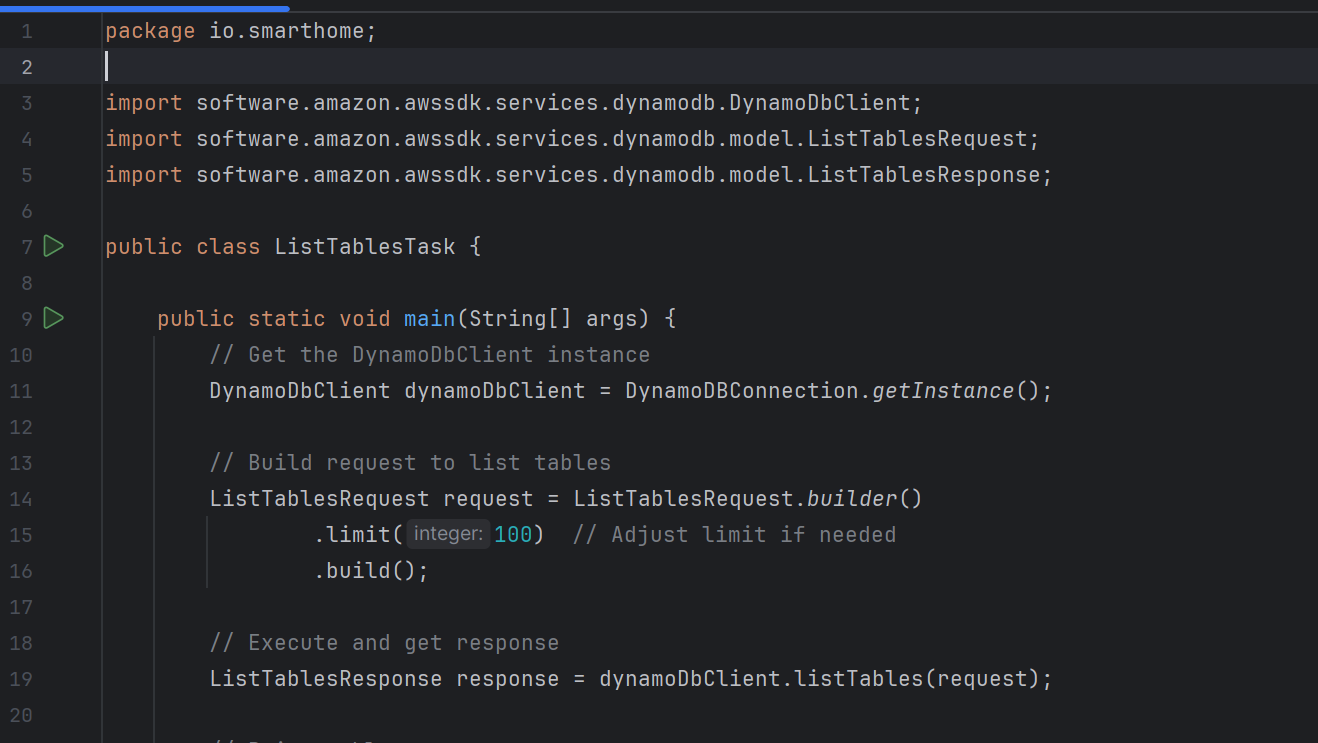
**Name:** Sudipto Das

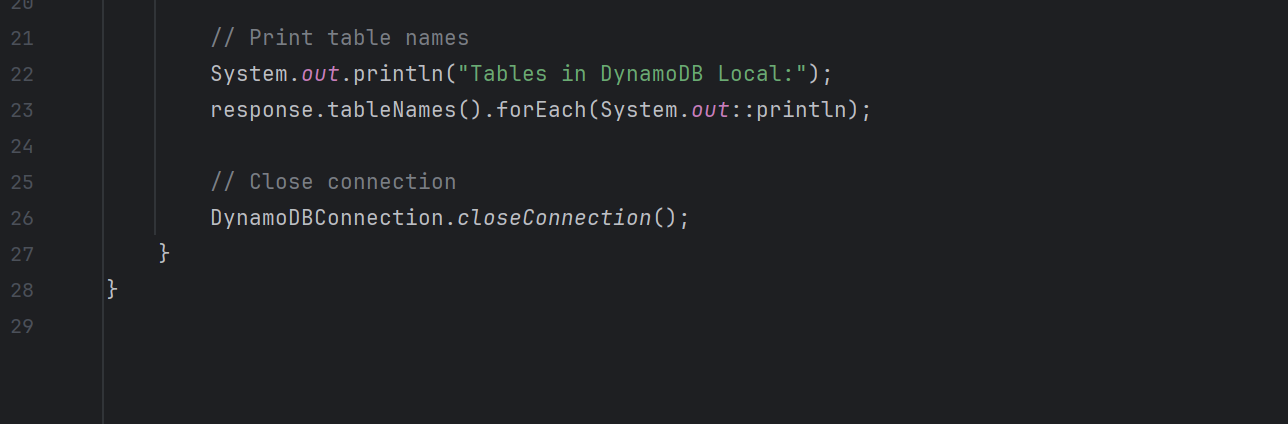
**Date:** 17 September 2025 (Day 36)

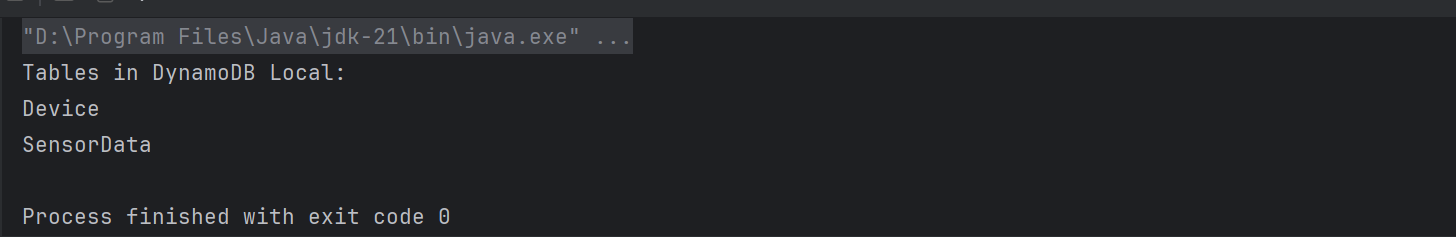
### ***Task 01: Display list of tables from the server***











### ***Task 02: DAX***

package org.example;

import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedAsyncClient;

import software.amazon.awssdk.regions.Region;

import software.amazon.awssdk.services.dynamodb.model.AttributeValue;

import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;

import software.amazon.dax.ClusterDaxAsyncClient;

import software.amazon.dax.Configuration;

import java.util.HashMap;

import java.util.Map;

import java.util.concurrent.CompletableFuture;

public class DemoDAXClient {

void daxClient() {

System.out.println("Demo for DAX Client");

String daxClusterEndpoint = "daxs://daxcluster01.ee3lf0.dax-clusters.ap-south-1.amazonaws.com";

Region region = Region.AP\_SOUTH\_1;

// Build DAX client

ClusterDaxAsyncClient daxClient = ClusterDaxAsyncClient.builder()

.overrideConfiguration(Configuration.builder()

.url(daxClusterEndpoint)

.region(region)

.build())

.build();

// Enhanced async client (optional, for high-level API)

DynamoDbEnhancedAsyncClient enhancedClient = DynamoDbEnhancedAsyncClient.builder()

.dynamoDbClient(daxClient)

.build();

try {

String tableName = "Prasunamba'sTable";

String partitionKeyName = "Prasunamba'sPKname";

String partitionKeyValue = "Prasunamba'sPKvalue";

// Key to get

Map<String, AttributeValue> keyToGet = new HashMap<>();

keyToGet.put(partitionKeyName, AttributeValue.builder().s(partitionKeyValue).build());

GetItemRequest getItemRequest = GetItemRequest.builder()

.tableName(tableName)

.key(keyToGet)

.build();

System.out.println("Getting item from DAX...");

// Async getItem call

CompletableFuture<Map<String, AttributeValue>> futureItem = daxClient.getItem(getItemRequest)

.thenApply(response -> response.item());

futureItem.thenAccept(item -> {

if (item == null || item.isEmpty()) {

System.out.println("No item found");

} else {

System.out.println("Got an item: " + item);

}

}).join(); // Wait for completion

} catch (Exception e) {

System.err.println("DAX Error: " + e.getMessage());

e.printStackTrace();

} finally {

// Clean up DAX client

daxClient.close();

}

}

public static void main(String[] args) {

DemoDAXClient demo = new DemoDAXClient();

demo.daxClient();

}

}

### ***Task 03: What is JUNIT?***

JUnit is a **Java testing framework** used for writing and running **unit tests**.

It helps verify that individual methods and components of your code work as expected, preventing bugs and ensuring code quality.

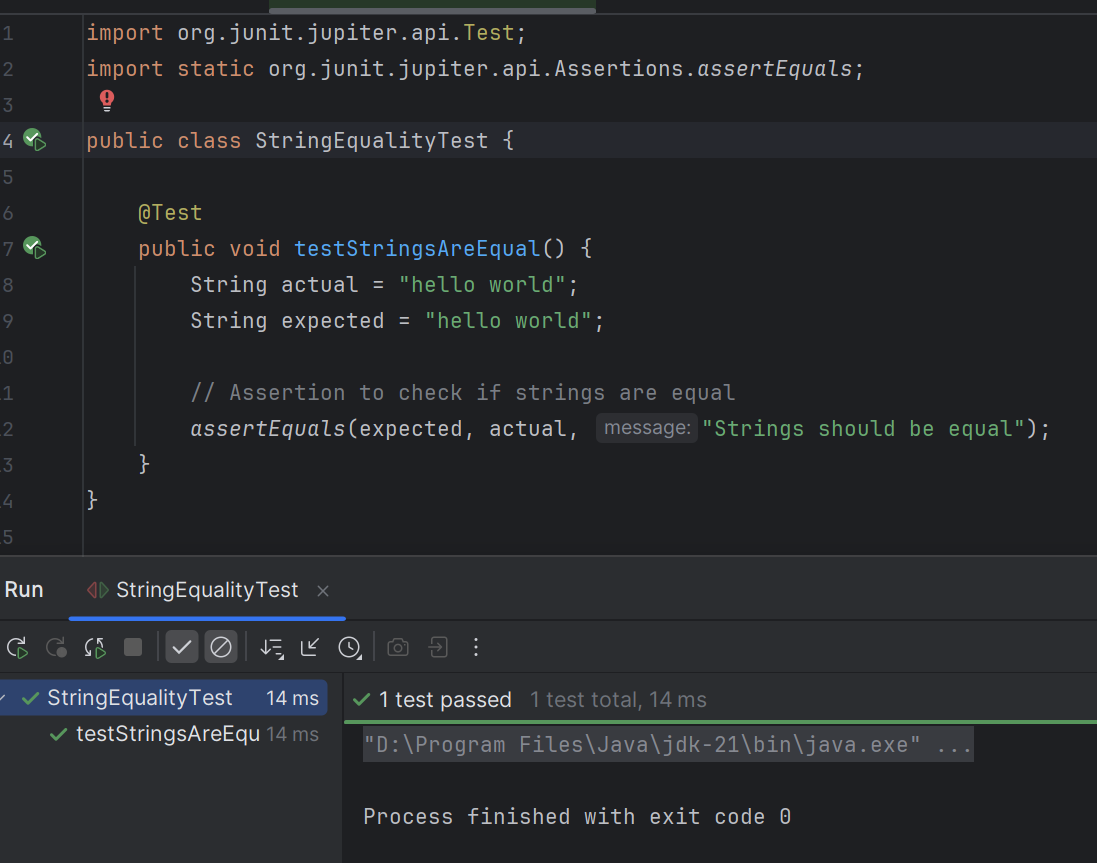
Tests are written using simple annotations like @Test and assertions such as assertEquals(expected, actual).

Commonly used in development and CI/CD pipelines for automated testing.

### ***Task 04: What is the latest version of Junit***

The latest stable version of JUnit 5 is **5.13.4**, released on **September 17, 2025**. This version includes various enhancements and bug fixes across JUnit Platform, Jupiter, and Vintage components [docs.junit.org](https://docs.junit.org/5.13.4/release-notes/?utm_source=chatgpt.com).

### ***Task 05: Write a test case to check if 2 strings or equal?***



### ***Task 06: Write the list of tags in Junit 5.***

* @Test  
   Marks a method as a test case.
* @BeforeEach  
   Runs before **each** test method (for setup).
* @AfterEach  
   Runs after **each** test method (for cleanup).
* @BeforeAll  
   Runs **once before all tests** in the class (static method).
* @AfterAll  
   Runs **once after all tests** in the class (static method).
* @Disabled  
   Disables a test (skip it temporarily).
* @DisplayName  
   Sets a custom name for the test method in reports.
* @Nested  
   Groups related tests into a nested class for organization.
* @Tag  
   Tags a test for grouping/filtering during execution.
* @ParameterizedTest  
   Runs the same test multiple times with different arguments.
* @ValueSource  
   Supplies simple values (like ints or strings) for parameterized tests.
* @CsvSource  
   Supplies comma-separated values for parameterized tests.
* @MethodSource  
   Supplies test arguments from a method (good for complex objects).
* @RepeatedTest  
   Repeats the same test a specified number of times (good for stability).
* @TestInstance  
   Configures the test instance lifecycle (PER\_METHOD or PER\_CLASS).

### ***Task 07: What are meta annotations & composed annotations?***

**Meta Annotations**

* Meta annotations are **annotations that are applied to other annotations**.
* They define **how your custom annotations behave** (e.g., where they can be applied, how long they are retained).
* Common meta annotations:  
  + @Retention → Defines annotation lifetime (source, class, runtime).
  + @Target → Defines where annotation can be applied (class, method, field, etc.).
  + @Inherited → Allows annotation to be inherited by subclasses.
  + @Documented → Includes annotation in Javadoc.

**Composed Annotations**

* Composed annotations are **custom annotations that combine multiple annotations together**.
* Useful for **reusing annotation patterns** and reducing boilerplate.

**Example Code**

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

import java.lang.annotation.ElementType;

import java.lang.annotation.Documented;

// Meta Annotations

@Retention(RetentionPolicy.RUNTIME) // Available at runtime

@Target(ElementType.METHOD) // Can be applied on methods

@Documented

@interface MyMetaAnnotation {

String value() default "Default Meta";

}

// Composed Annotation

@MyMetaAnnotation(value = "Composed Example") // Uses meta annotation

@interface MyComposedAnnotation {

String description() default "Composed Annotation";

}

// Usage

public class AnnotationDemo {

@MyComposedAnnotation(description = "This is a test method")

public void testMethod() {

System.out.println("Running test method with composed annotation");

}

}

**What’s Happening:**

* MyMetaAnnotation → A custom annotation using meta annotations (@Retention, @Target, @Documented).
* MyComposedAnnotation → Combines MyMetaAnnotation and adds its own property.
* testMethod() → Uses the composed annotation, inheriting behavior from MyMetaAnnotation.

**Meta Annotations** = annotations on annotations, controlling behavior.

**Composed Annotations** = custom annotations built by combining other annotations for reuse.

### ***Task 08: What Are Assertions in JUnit?***

* Assertions are statements used in test cases to **verify if the code behaves as expected**.
* They automatically check if the actual result matches the expected result.
* If an assertion fails → the test fails
* If all assertions pass → the test passes

**Common Assertion Methods**

* assertEquals(expected, actual) → Checks if two values are equal.
* assertNotEquals(expected, actual) → Checks if two values are NOT equal.
* assertTrue(condition) → Checks if a condition is true.
* assertFalse(condition) → Checks if a condition is false.
* assertNull(object) → Checks if an object is null.
* assertNotNull(object) → Checks if an object is NOT null.
* assertArrayEquals(expectedArray, actualArray) → Checks if two arrays are equal.
* assertThrows(Exception.class, executable) → Checks if an exception is thrown.

### ***Task 09: What Are Assumptions in JUnit?***

* Assumptions are like **pre-checks** in your tests.
* They help you **skip tests automatically** if certain conditions aren’t met, instead of failing them.

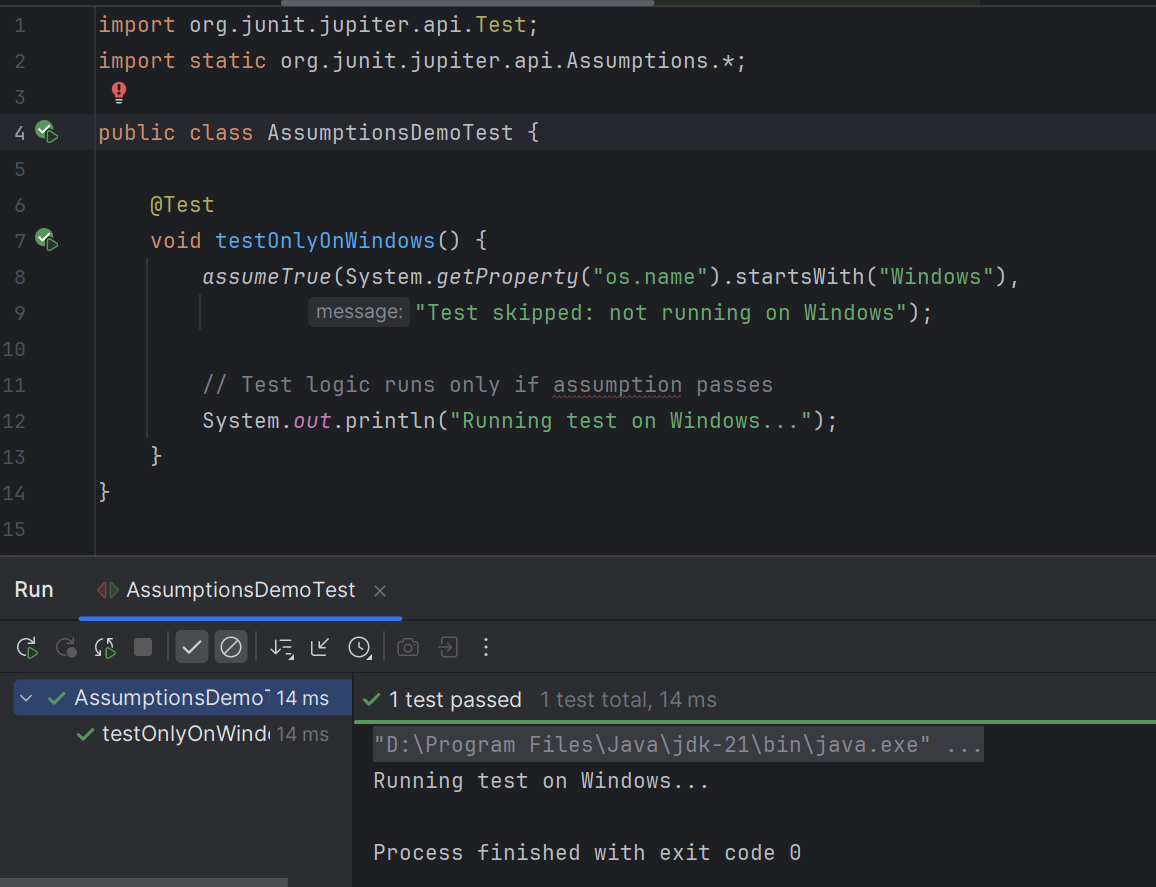
**Why Use Assumptions?**

* Useful when your test depends on environment factors (like OS, external services, configs).
* If an assumption fails → test is skipped, not marked as failed
* Keeps your test reports clean from false negatives 🚫

**Common Assumption Methods**

* assumeTrue(condition) → Runs the test only if the condition is true.
* assumeFalse(condition) → Runs the test only if the condition is false.
* assumingThat(condition, executable) → Runs code only if the condition is true.

**Example :**



**In this example:**

* If the OS is NOT Windows → test is skipped, not failed.

### ***Task 10: What Is Disabling Test Cases?***

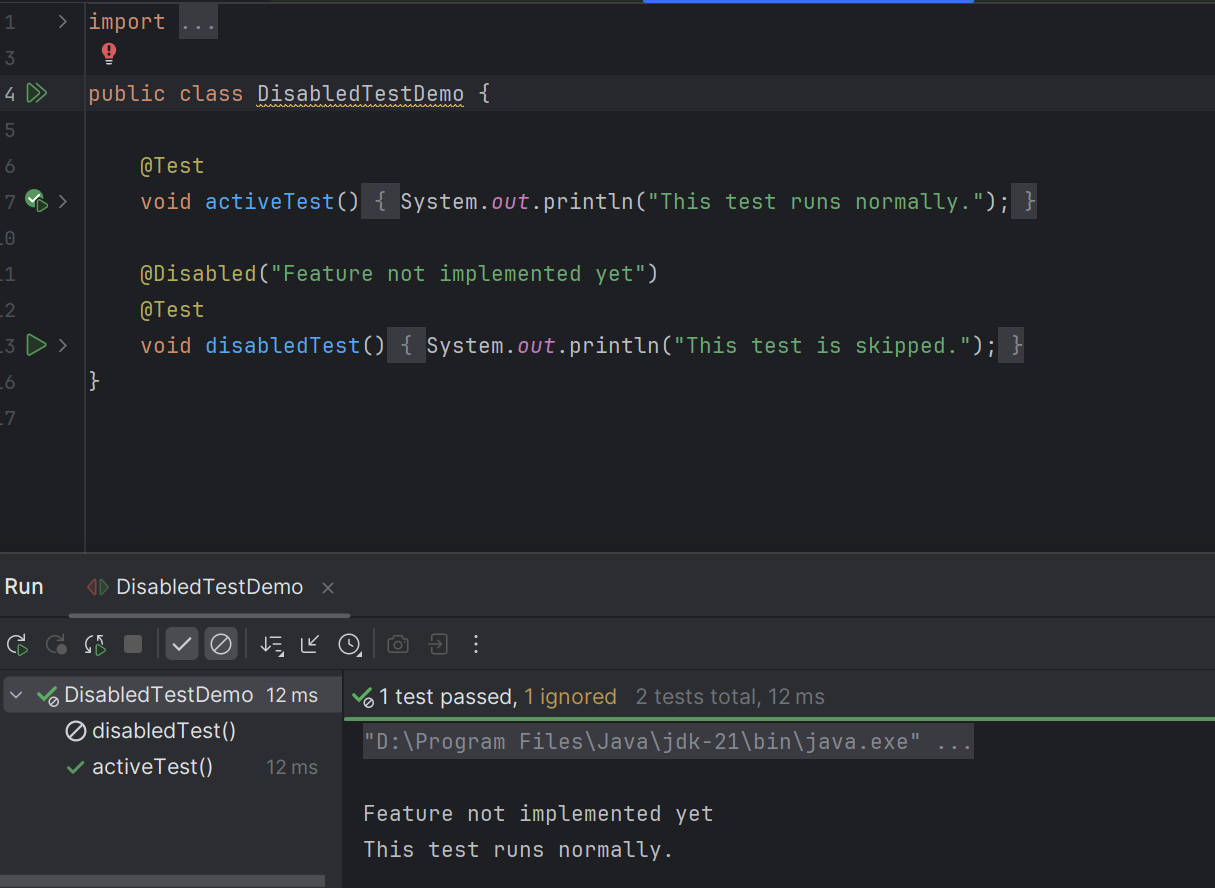
**What Is Disabling Test Cases?**

* Disabling a test case means **temporarily preventing it from running**.
* In JUnit, this is done using the **@Disabled annotation**.
* Disabled tests are **skipped**, not marked as failed

**Why/When To Use Them**

* **Under development:** Test is not fully implemented yet.
* **Known bugs:** Test fails due to a known issue that’s being fixed.
* **Environment-dependent:** Test should run only in certain environments.
* **Temporary skip:** To speed up test runs or avoid unstable tests.

**Example :**



**In this example:**

* activeTest() → Runs normally.
* disabledTest() → Skipped, with a message explaining why.

### ***Task 11: Conditional Test Executions in JUnit 5***

JUnit 5 allows tests to **run or skip based on certain conditions** using these annotations:

* @EnabledOnOs → Run test only on specified operating systems.
* @DisabledOnOs → Skip test on specified operating systems.
* @EnabledOnJre → Run test only on specified Java versions.
* @DisabledOnJre → Skip test on specified Java versions.
* @EnabledIf → Run test only if a custom condition (boolean expression) is true.
* @DisabledIf → Skip test if a custom condition (boolean expression) is true.
* @EnabledIfEnvironmentVariable → Run test only if a specific environment variable has a certain value.
* @DisabledIfEnvironmentVariable → Skip test if a specific environment variable has a certain value.

### ***Task 12: What Is Automated Testing?***

* Automated Testing is the process of **using software tools to execute tests** on your application automatically.
* Instead of manually checking if your code works, the tests **run automatically** and verify expected outcomes.
* It’s widely used in **unit testing, integration testing, and UI testing**.

**Key Points**

* Reduces **manual effort** and saves time.
* Helps catch **bugs early** in the development cycle.
* Ensures **consistent and repeatable** test results.
* Essential for **Continuous Integration (CI) and Continuous Delivery (CD)** pipelines.

### ***Task 13: Automated Testing Tools & Frameworks***

**For Java / Backend**

* **JUnit** → Unit testing framework for Java.
* **TestNG** → Advanced Java testing framework with annotations and parallel execution.
* **Mockito** → Framework for mocking objects in unit tests.
* **Arquillian** → Integration testing for Java EE applications.

**For Web / UI Testing**

* **Selenium** → Web browser automation for functional UI testing.
* **Cypress** → Modern JS framework for end-to-end testing.
* **Playwright** → Cross-browser web automation testing.
* **Puppeteer** → Headless browser testing for Chrome.

**For API / Backend Testing**

* **RestAssured** → Java DSL for testing REST APIs.
* **Postman** → API testing tool with automation support.
* **Karate** → API test automation framework with BDD syntax.

**For CI/CD and Automation**

* **Jenkins** → Automates running tests in pipelines.
* **GitHub Actions** → Run automated tests on commits or PRs.
* **CircleCI / TravisCI** → Cloud-based CI/CD with test automation.

### ***Task 14: Life Cycle of Test Automation***

**1. Test Automation Planning**

* Identify **test objectives** and scope for automation.
* Decide **which test cases to automate** (usually repetitive, critical, or high-risk).
* Choose **tools and frameworks** suitable for the application.

**2. Test Design / Scripting**

* Create **automation scripts** based on the test scenarios.
* Define **test data**, input values, and expected results.
* Use best practices for **modular and reusable code**.

**3. Test Environment Setup**

* Prepare **testing environment** (OS, browsers, databases, servers).
* Configure tools, frameworks, and dependencies.

**4. Test Execution**

* Run automated tests using selected tools or CI/CD pipelines.
* Capture **results, logs, and screenshots** (if UI testing).

**5. Test Reporting**

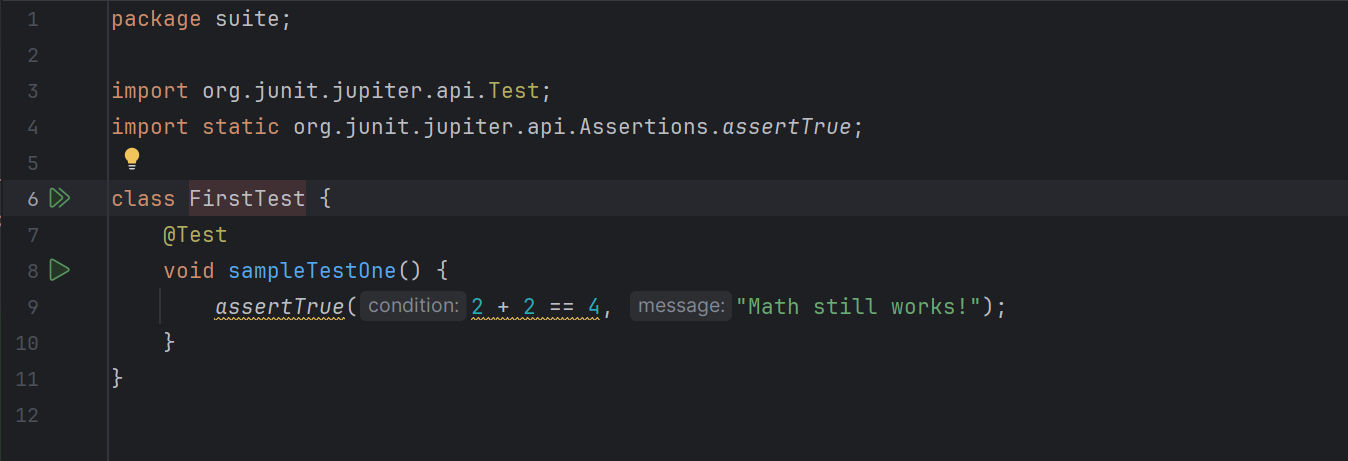
* Analyze test results for **pass/fail status**.
* Generate **reports** for stakeholders and developers.
* Identify **bugs or failures** and log them for fixing.

**6. Maintenance**

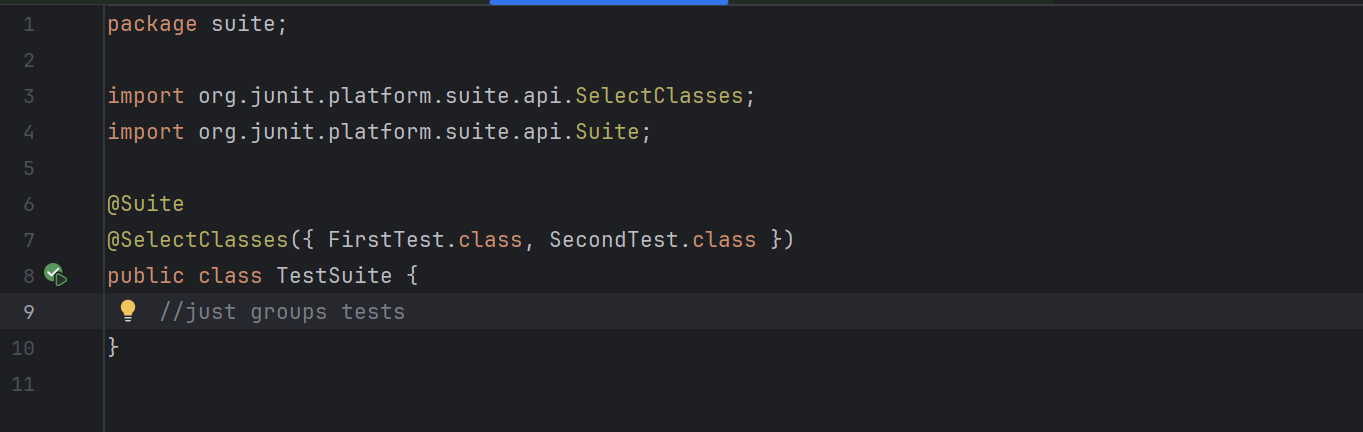
* Update test scripts as the application **evolves or new features are added**.
* Fix broken tests due to changes in UI, API, or logic.
* Refactor scripts to improve efficiency and reliability.

Test automation life cycle = **Plan → Design → Setup → Execute → Report → Maintain**Following this cycle ensures automated tests are **robust, reusable, and valuable** in the long run

### ***Task 15: Write a JUNIT 5 code to use @Suite tag***







### ***Task 16: Write JUNIT 5 code to use @fast and @slow tags.***

