***Shiv Sunder Dash report***

1. Calculator and Unit Tests

I developed a Calculator class that includes methods for performing basic arithmetic operations: add, subtract, multiply, and divide. To validate the correctness of these operations, I wrote a test class named CalculatorTest. In this test class, each method was individually tested using assertions like assertEquals to check for expected results and assertNotEquals to ensure incorrect results are not falsely accepted. Additionally, I wrote a specific test to verify that dividing by zero throws an ArithmeticException, using the assertThrows assertion to handle the expected exception scenario.

2. Lifecycle Method Demonstration

To demonstrate the execution flow of JUnit 5 lifecycle methods, I created a class called LifecycleTest. This class included methods annotated with @BeforeAll, @BeforeEach, @Test, @AfterEach, and @AfterAll. Each method prints a simple message to the console, allowing me to observe the precise order in which these annotations are triggered during the test execution. This clearly showed how setup and teardown processes are managed around test cases in JUnit 5.

3. User Class and UserService Validation

I implemented a User class with basic fields such as name, email, and age. Then, I created a corresponding test class UserTest to validate these fields. I used various assertions including assertEquals, assertNotNull, and assertTrue to check for expected values and ensure proper initialization. All validations were grouped using assertAll to keep the test structured. Alongside this, I created a UserService class containing a method validateAge(int age) that throws an IllegalArgumentException if the provided age is below 18. This method was unit tested using assertThrows to verify that the exception is raised correctly and that the message matches the expected "Underage" string.

4. Product REST API and Integration Testing

For managing product data in an e-commerce context, I built a Spring Boot REST API with attributes such as id, name, description, price, stockQuantity, and category. The API includes endpoints for listing all products, retrieving a product by ID, adding a new product, updating an existing product, and deleting a product. These endpoints correspond to HTTP methods GET, POST, PUT, and DELETE. I then wrote integration tests using MockMvc along with annotations like @SpringBootTest and @AutoConfigureMockMvc to simulate and verify HTTP interactions. Each test checks for the expected response status and content.

5. Library Book API and Integration Testing

In this part, I designed and implemented a Spring Boot REST API for managing a library of books. Each book entry includes an id, title, author, and publishedYear. The application provides endpoints for basic CRUD operations: retrieving all books, getting a single book by ID, adding new books, updating book details, and deleting entries. To confirm that these endpoints function correctly, I wrote integration tests using MockMvc. Each test simulates client requests to the API and verifies the status codes and response content to ensure proper behavior of the application across all operations.

Mcq answers:

1 .C. @Test

2. B. @Before

3. B. @BeforeClass must be static.

4. C. To enable scanning for Spring Data JPA interfaces

5. A. To fail if no exception occurs.

6. C. @ParameterizedTest

7. B. Fail

8. B. Assertions.assertAll()

9. C. Asserts that an exception is thrown

10. A. The test fails

11. A. @ExtendWith

12. A. assertThrows(Exception.class, () -> ...)

13. C. @ParameterizedTest

14. C. To fail if the block takes too long

15. A. @TestFactory