Unit Testing with Test-Driven Development (TDD) in Java

Title: Unit Testing with TDD in Java

Subtitle: Leveraging JUnit and Mockito for Effective Testing

Presenter Name: Rama Shanker

Date: 27/01/2025

Agenda

What is Unit Testing?

Introduction to Test-Driven Development (TDD)

Overview of JUnit

Introduction to Mockito

Lab: Writing Unit Tests with Mockito

Key Takeaways

What is Unit Testing?

Definition: Testing individual components or units of code to verify their correctness.

Purpose:

- Detect bugs early.
- Improve code quality.
- Facilitate refactoring.

Benefits:

- Saves time and cost in the long run.
- Ensures functionality meets requirements.

Characteristics:

- Isolated.
- Fast and repeatable.

Test-Driven Development (TDD)

Definition: A development methodology where tests are written before the code.

Process:

Write a failing test.

Write code to make the test pass.

Refactor code while ensuring tests still pass.

Benefits:

Improves design.

Encourages small, testable units.

Provides a safety net for refactoring.

JUnit Overview

What is JUnit?

A popular Java framework for unit testing.

Simplifies test creation and execution.

Key Features:

Assertions (e.g., assertEquals, assertTrue).

Annotations (e.g., @Test, @BeforeEach, @AfterEach).

Test suites for grouping tests.

Example

```
@Test
void testAddition() {
    int result = Calculator.add(2, 3);
    assertEquals(5, result);
}
```

```
@Test
void testSubstraction() {
    int result = Calculator.sub(5, 3);
    assertEquals(2, result);
}
```

Introduction to Mockito

What is Mockito?

A framework for creating mock objects.

Helps test classes in isolation by mocking dependencies.

Why Use Mockito?

Simplifies testing of complex systems.

Mimics behavior of real objects.

Key Features:

Mocking (mock(Class.class)).

Stubbing (when(...).thenReturn(...)).

Verifications (verify(...)).

Mockito Example

```
@Mock
UserRepository userRepository;


@Test
void testFindUserById() {
    when(userRepository.findById(1)).thenReturn(new User(1, "John"));

    User user = userService.findById(1);

    assertEquals("John", user.getName());
    verify(userRepository).findById(1);
}
```

Explanation:

@Mock: Creates a mock object for UserRepository.

when: Defines behavior for the mock.

verify: Ensures the mock was used as expected.

Lab: Writing Unit Tests with Mockito

Setup:

Add JUnit and Mockito dependencies to your project.

Example (Maven):

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>4.5.1</version>
    <scope>test</scope>
</dependency>
```

Exercise:

Write a unit test for a service class using Mockito.

Mock a repository and stub its methods.

Verify interactions with the mock.

Challenge:

Add a failing test and implement the code to make it pass.

UserService Implementation

```
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User getUserDetails(int userId) {
        return userRepository.findById(userId).orElseThrow(() -> new RuntimeException("User not found"));
    }
}
```

UserRepository Interface

```
public interface UserRepository {
    Optional<User> findById(int id);
}
```

User Entity

```
public class User {
    private int id;
    private String name;

    public User(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

Unit Test with Mockito

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {
    @Mock
    private UserRepository userRepository;
    @InjectMocks
    private UserService userService;
    @Test
    public void testGetUserDetails() {
        // Arrange
        User mockUser = new User(1, "Alice");
        when(userRepository.findById(1)).thenReturn(Optional.of(mockUser));
        // Act
        User result = userService.getUserDetails(1);
        // Assert
        assertNotNull(result);
        assertEquals("Alice", result.getName());
        verify(userRepository).findById(1);
    }
    @Test
    public void testGetUserDetails_UserNotFound() {
        // Arrange
        when(userRepository.findById(2)).thenReturn(Optional.empty());


        // Act & Assert
        Exception exception = assertThrows(RuntimeException.class, () -> userService.getUserDetails(2));
        assertEquals("User not found", exception.getMessage());
    }
}
```

Explanantion

@RunWith(MockitoJUnitRunner.class): Sets up the test with Mockito support.

@Mock: Mocks the UserRepository.

@InjectMocks: Injects the mock into the UserService being tested.

when(...).thenReturn(...): Defines mock behavior.

verify(...): Ensures the mock was called as expected.

Key Takeaways

Unit tests ensure code reliability and maintainability.

TDD promotes better design and confidence in code changes.

JUnit and Mockito are essential tools for Java developers.

Practice regularly to master testing techniques.