Class-31: Unit Testing

by Pial Kanti Samadder

What is Unit Testing?

Unit testing is the practice of validating **small, isolated units of code**—typically individual methods or functions—to ensure they behave as expected.

Key Characteristics

- Tests a *single unit* of code
- No external dependencies (DB, network, file system)
- Fast, repeatable, and deterministic
- Written by developers during implementation

Goal

Ensure each building block of your application works correctly *in isolation*.

Why Unit Testing is Important

Builds Confidence in Code

- Ensures core business logic works reliably
- Prevents unexpected issues during changes or refactoring

Improves Code Quality

- Encourages modular and clean design
- Identifies logical errors early

Reduces Development Cost

- Catching bugs early is cheaper than fixing them later
- Reduces time spent debugging in QA or production

Supports Continuous Delivery

- Tests run automatically in CI/CD pipelines
- Helps maintain stability during fast releases

Key Benefit

Unit tests act as a **safety net**, allowing you to develop and refactor with confidence.

Types of Testing

1. Unit Testing

Focus:
Tests the **smallest pieces of code** (methods/classes) in complete isolation.

Characteristics:

- No external dependencies
- Fast and reliable
- Written by developers
- Validates individual business logic

Purpose:
Ensures each component works correctly by itself.

2. Integration Testing

Focus:
Tests how **multiple components interact** with each other.

Characteristics:

- May involve in-memory DB, real repositories, or external services
- Slightly slower than unit tests
- Ensures system layers work together as expected

Purpose:
Verifies modules integrate correctly (e.g., Service ↔️ Repository ↔️ Database).

3. End-to-End (E2E) Testing

Focus:
Tests the **entire application flow** from start to finish.

Characteristics:

- Simulates real user actions
- Involves actual UI, backend, and database
- Slowest type of test
- Covered usually by QA automation

Purpose:
Ensures the whole system works exactly as a user would expect.

Introduction to JUnit 5

What is JUnit 5?

JUnit 5 is a modern Java testing framework used to write **unit tests**, helping developers validate their code with simple, readable tests.

Why Developers Use JUnit 5

- Easy and expressive annotations
- Clean and readable test methods
- Strong support in IDEs (IntelliJ, Eclipse, VS Code)
- Works naturally with Maven/Gradle
- Supports parameterized tests
- Faster and more flexible than older versions

What You Can Do With JUnit 5

- Write basic unit tests (`@Test`)
- Run setup code before each test (`@BeforeEach`)
- Verify expected outputs using assertions
- Test exception behavior (`assertThrows`)
- Group tests and improve readability

JUnit 5 Core Annotations

1. **@Test**
   Marks a method as a test case.
   - Runs the method during test execution
   - Should not return anything
   - Should not accept parameters (unless parameterized)
2. **@BeforeEach**
   - Runs before each test method.
   - Use it to prepare common test data or initialize objects.
   - Example: creating a service instance before each test.
3. **@AfterEach**
   - Runs after each test method.
   - Use it to clean up resources if needed (closing streams, clearing data).
4. **@BeforeAll**
   Runs once before all tests in the class.
   - Must be static
   - Ideal for expensive setup (DB connection mock, loading config)
5. **@AfterAll**
   Runs once after all tests in the class.
   - Must be static
   - Used for releasing shared resources
6. **@DisplayName**
   - Provides a readable, custom name for a test.
   - Makes test results more descriptive.
7. **@Disabled**
   - Temporarily disables a test or test class.
   - Useful when a feature is not ready or unstable.

Assertions in JUnit 5

What Are Assertions?

Assertions are methods used in tests to **verify that the actual output of your code matches the expected result**. If an assertion fails, the test fails.

Assertions help you confirm that:

- Inputs produce the correct outputs
- Methods behave as expected
- Errors/exceptions occur when they should

Common Assertion Methods

1. **assertEquals(expected, actual)**
   - Verifies that two values are equal.
   - Useful for validating return values, calculations, etc.
2. **assertNotNull(object)**
   - Checks that an object is not null.
   - Common when testing repository or service results.
3. **assertTrue(condition) / assertFalse(condition)**
   - Verifies boolean conditions (validation flags, checks, status).
4. **assertThrows(Exception.class, () -> {...})**
   - Confirms that a method throws the expected exception.
   - Great for validating error handling and invalid inputs.
5. **assertAll(...)**
   - Runs multiple assertions together.
   - Useful when testing multiple fields of a returned object.
6. **assertTimeout(Duration, executable)**
   - Checks if a piece of code completes within a time limit.
   - Useful for performance-sensitive operations.

Basic JUnit 5 Test Example

Example: Testing a Calculator Service

```
class CalculatorTest {

    static Calculator calculator;

    // Runs once before all tests
    @BeforeAll
    static void init() {
        calculator = new Calculator();
        System.out.println("Calculator initialized");
    }

    // Runs before each test
    @BeforeEach
    void setup() {
        System.out.println("Starting a test...");
    }

    @Test
    @DisplayName("Addition: 2 + 3 = 5")
    void testAddition() {
        int result = calculator.add(2, 3);
        assertEquals(5, result, "2 + 3 should equal 5");
    }

    @Test
    @DisplayName("Subtraction: 5 - 3 = 2")
    void testSubtraction() {
        int result = calculator.subtract(5, 3);
        assertEquals(2, result, "5 - 3 should equal 2");
    }
}
```

Explanation

- `@BeforeAll` → initializes **shared resources** once per class
- `@BeforeEach` → runs **before every test**, useful for setup
- `@Test` → marks a test method
- `@DisplayName` → gives a **readable name** for the test
- `assertEquals` → checks if the **actual result matches expected**

Writing Tests with Given–When–Then

What is Given–When–Then?

A **structured pattern** for writing unit tests that improves readability and clarity.

- **Given:** the initial context or setup
- **When:** the action or method under test
- **Then:** the expected outcome or assertion

Benefits

- Makes tests **self-explanatory**
- Improves **maintainability**
- Helps developers and reviewers **understand the test purpose quickly**

Example: Calculator Test Using GWT

```
@Test
@DisplayName("Addition: 2 + 3 = 5")
void testAddition_GWT() {
    // Given
    int a = 2;
    int b = 3;

    // When
    int result = calculator.add(a, b);

    // Then
    assertEquals(5, result, "2 + 3 should equal 5");
}
```

Introduction to Mockito

What is Mockito?

Mockito is a **Java mocking framework** used to **simulate dependencies** in unit tests. It allows you to test a class in isolation by **mocking external services or repositories**.

Why Use Mockito?

- Isolate the class under test → no reliance on DB, APIs, or other services
- Test only **business logic**
- Verify **method calls and interactions**
- Improves **test speed** and reliability

Key Features

- `@Mock` → creates mock objects
- `@InjectMocks` → injects mocks into the class under test
- `when(...).thenReturn(...)` → stubbing method responses
- `verify(...)` → check method calls

Basic Mockito Example

Scenario: Testing a `UserService` that depends on a `UserRepository`

```
@ExtendWith(MockitoExtension.class)
class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    @DisplayName("Find User by ID")
    void testFindUserById() {
        // Given
        User mockUser = new User(1L, "John");
        when(userRepository.findById(1L)).thenReturn(Optional.of(mockUser));

        // When
        User result = userService.findUserById(1L);

        // Then
        assertEquals("John", result.getName());
        verify(userRepository, times(1)).findById(1L);
    }
}
```

Explanation

- **@Mock** → creates a fake `UserRepository`
- **@InjectMocks** → injects the mock into `UserService`
- **when(...).thenReturn(...)** → simulates repository response
- **assertEquals** → verifies expected output
- **verify(...)** → ensures method is called exactly once

Test Naming Best Practices

Why Naming Matters

- Clear test names **communicate intent**
- Improves **readability and maintainability**
- Makes it easy for others to **understand what is being tested**
- Helps **debug failing tests quickly**

Recommended Pattern: Given–When–Then

- **Given:** Initial state or setup
- **When:** Action under test
- **Then:** Expected outcome

Format:

```
methodName_whenCondition_expectedOutcome
```

Examples

- `calculatePrice_whenInvalidInput_throwsException()`
- `getUserProfile_whenUserExists_returnsProfile()`
- `createOrder_whenStockUnavailable_throwsException()`
- `deleteUser_whenUserNotFound_returnsFalse()`

Tips for Naming

- Be **descriptive, not short**
- Avoid ambiguous names like `test1()` or `checkSomething()`
- Include **expected behavior** in the name
- Use **consistent naming convention** across the project

Key Takeaway

Good test names make your **tests self-explanatory**, **professional**, and **easier to maintain**.