Foundations of Software Testing

2.4 Introduction to Automated Testing with JUnit 5

Learning objectives

- What is JUnit 5 and why it matters
- Anatomy of a test class
- JUnit 5 annotations and lifecycle
- Writing your first unit test
- Using assertions and testing exceptions

Why Automated Testing

- Manual testing is time-consuming and error-prone
- Automated tests verify code automatically
- Runs instantly after each change (fast feedback)
- Forms the foundation for continuous integration (CI)
- Allows safe refactoring and confident releases

What Is JUnit 5?

- A testing framework for Java
- Industry standard for unit and integration tests
- Provides annotations to define test methods
- Runs tests automatically via IDE or Maven/Gradle
- Integrates with Spring Boot, Rest Assured, and CI tools

JUnit 5 Architecture (Simplified)

- JUnit Platform: runs tests and connects to IDE/build tools
- JUnit Jupiter: new programming model (modern API)
- JUnit Vintage: backward compatibility for JUnit 4 tests
- In this course, we use JUnit Jupiter (JUnit 5).

Common JUnit 5 Annotations

Annotation	Purpose
@Test	Marks a test method
@BeforeEach	Runs before each test (setup)
@AfterEach	Runs after each test (cleanup)
@BeforeAll	Runs once before all tests (static)
@AfterAll	Runs once after all tests
@DisplayName	Describes the test clearly

Assertions in JUnit 5

- Assertions check that code behaves as expected
- Common ones:
  - assertEquals(expected, actual)
  - assertTrue(condition)
  - assertFalse(condition)
  - assertNotNull(object)
  - assertThrows(Exception.class, () -> …)
- Failures show differences in expected vs. actual values

Running Tests

- Tests can be run directly in IntelliJ IDEA:
  - Green check ✅ = success
  - Red cross ❌ = failure
- Or via Maven
- Reports show how many tests passed/failed

Demo 1: JUnit 5 in Action

- Goal: Show how a simple JUnit 5 test looks and runs.
- 1. Use the demo with the calculator from section 2.1
- 2. Add JUnit dependency in pom.xml
- 4. Create CalculatorTest with one test method using @Test and assertEquals.
- 5. Run test in IntelliJ and show output.
- 6. Create another test for division by zero and run the test

Demo 2: Test Lifecycle

- Goal: Demonstrate setup and teardown using lifecycle annotations.
- 1. Add @BeforeEach method printing “Starting test”.
- 2. Add @AfterEach printing “Finished test”.
- 3. Add a second test method (e.g., add or subtract).
- 4. Run tests to show how setup/cleanup run around each test.

Lab: Extending the Calculator Tests

- 1. Use the CalculatorTest class in the same demo project, or create a new one.
- 2. Add methods in Calculator:
    - add(int a, int b)
    - subtract(int a, int b)
    - multiply(int a, int b)
    - Keep the divide(double a, double b) method.
- 3. In CalculatorTest:
    - Write one test per operation.
    - Add @BeforeEach to initialize the calculator.
    - Use assertions (assertEquals, assertThrows, etc.).
- Run all tests to verify success.

Lab: Optional Extra Challenge

- Add validation logic to divide():
  - if (b == 0) {
  -     throw new IllegalArgumentException("Cannot divide by zero");
  - }
- Then update your test to verify the new exception type.
- Run all tests again to confirm they still pass.

Key Takeaways

- JUnit 5 automates what we previously checked manually
- Each test isolates logic and verifies expected behavior
- Assertions make tests self-verifying
- Exceptions can be tested safely
- Lifecycle hooks (@BeforeEach, @AfterEach) ensure clean setup
- This foundation prepares us for Spring Boot & API testing