1. Introduction to Unit Testing

1.1 What is Unit Testing?

Unit testing is the process of verifying the correctness of the smallest testable parts of an application (usually methods or classes) in isolation.

A **unit**:

- A method
- A class
- A small logical component

The goal is to:

- Ensure correctness
- Prevent regressions
- Enable safe refactoring
- Improve code quality

1.2 Characteristics of Good Unit Tests

A good unit test is:

- **Isolated** (independent of other tests)
- **Deterministic** (always produces same result)
- **Fast**
- **Readable**
- **Self-validating** (automatic pass/fail)

1.3 Testing Terminology

Term	Meaning
SUT	System Under Test
Test Case	A single executable test
Test Suite	Collection of test cases
Assertion	Verification of expected result
Mock	Simulated dependency
Stub	Simplified implementation
Fixture	Test setup state

---

2. Creating a Maven Project for JUnit

JUnit 5 integrates cleanly with Maven.

2.1 Maven Standard Directory Structure

```
project-name/
 ├── pom.xml
 └── src/
     ├── main/java/       → Production code
     └── test/java/       → Test code
```

Tests must always reside under:

`src/test/java`

2.2 Required Dependencies (JUnit 5)

Add the following to `pom.xml`:

```
<dependencies>
    <dependency>
```

```
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.11.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Surefire Plugin (required to run tests)

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.5.2</version>
    </plugin>
  </plugins>
</build>
```

Run tests using:

```
mvn test
```

---

3. JUnit 5 Architecture

JUnit 5 consists of three main modules:

1. **JUnit Platform**
    - Launches testing frameworks
    - Executes test engines
2. **JUnit Jupiter**
    - New programming model
    - Contains annotations like `@Test`, `@BeforeEach`
3. **JUnit Vintage**

- Supports JUnit 3 and 4 tests

In modern projects, we primarily use **JUnit Jupiter**.

---

4. Writing the First Test

4.1 Example Production Class

```
package com.example;

public class Calculator {

    public int add(int a, int b) {
        return a + b;
    }

    public int divide(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Division by zero not
allowed");
        }
        return a / b;
    }
}
```

4.2 Writing a Test Class

Test file:

```
src/test/java/com/example/CalculatorTest.java

package com.example;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {
```

```
    @Test
    void shouldAddTwoNumbersCorrectly() {
        Calculator calculator = new Calculator();

        int result = calculator.add(2, 3);

        assertEquals(5, result);
    }
}
```

Explanation

- `@Test` marks the method as a test case.
- `assertEquals(expected, actual)` verifies correctness.
- If assertion fails → test fails.

---

5. Test Lifecycle in JUnit 5

JUnit provides lifecycle annotations:

Annotation	Purpose
`@BeforeEach`	Runs before every test
`@AfterEach`	Runs after every test
`@BeforeAll`	Runs once before all tests
`@AfterAll`	Runs once after all tests

5.1 Lifecycle Example

```
import org.junit.jupiter.api.*;
```

```
class LifecycleExampleTest {

    @BeforeAll
    static void beforeAllTests() {
        System.out.println("Executed once before all tests");
    }

    @BeforeEach
    void beforeEachTest() {
        System.out.println("Executed before each test");
    }

    @Test
    void test1() {}

    @Test
    void test2() {}

    @AfterEach
    void afterEachTest() {
        System.out.println("Executed after each test");
    }

    @AfterAll
    static void afterAllTests() {
        System.out.println("Executed once after all tests");
    }
}
```

Note:

- `@BeforeAll` and `@AfterAll` must be static unless using `@TestInstance(PER_CLASS)`.

---

6. Assertions in JUnit

Assertions validate expected outcomes.

6.1 Common Assertions

```
assertEquals(expected, actual);
assertNotEquals(expected, actual);
assertTrue(condition);
assertFalse(condition);
assertNull(object);
assertNotNull(object);
assertThrows(Exception.class, executable);
```

6.2 Testing Exceptions

```
@Test
void shouldThrowExceptionWhenDividingByZero() {
    Calculator calculator = new Calculator();

    Exception exception = assertThrows(
        IllegalArgumentException.class,
        () -> calculator.divide(10, 0)
    );

    assertEquals("Division by zero not allowed",
exception.getMessage());
}
```

Explanation:

- `assertThrows()` verifies type and allows inspection of message.

---

7. Grouped Assertions

Used to validate multiple conditions together.

```
@Test
void groupedAssertionsExample() {
    Calculator calculator = new Calculator();
    int result = calculator.add(10, 5);
```

```
    assertAll("result checks",
        () -> assertTrue(result > 0),
        () -> assertEquals(15, result),
        () -> assertNotNull(result)
    );
}
```

---

8. Parameterized Tests

Parameterized tests allow running the same test with multiple inputs.

8.1 Using @ValueSource

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

@ParameterizedTest
@ValueSource(ints = {1, 2, 3})
void shouldReturnPositiveForPositiveNumbers(int value) {
    Calculator calculator = new Calculator();
    int result = calculator.add(value, 5);
    assertTrue(result > 0);
}
```

8.2 Using @CsvSource

```
@ParameterizedTest
@CsvSource({
    "2,3,5",
    "5,5,10",
    "10,15,25"
})
void shouldAddNumbersCorrectly(int a, int b, int expected) {
    Calculator calculator = new Calculator();
    assertEquals(expected, calculator.add(a, b));
}
```

---

9. Nested Tests (Behavior Organization)

Used to structure tests logically.

```
import org.junit.jupiter.api.Nested;

class CalculatorNestedTest {

    Calculator calculator = new Calculator();

    @Nested
    class AddTests {
        @Test
        void shouldAddPositiveNumbers() {
            assertEquals(5, calculator.add(2,3));
        }
    }

    @Nested
    class DivideTests {
        @Test
        void shouldThrowExceptionWhenZero() {
            assertThrows(
                IllegalArgumentException.class,
                () -> calculator.divide(5,0)
            );
        }
    }
}
```

---

10. Repeated Tests

```
import org.junit.jupiter.api.RepeatedTest;

@RepeatedTest(3)
void repeatedTestExample() {
    Calculator calculator = new Calculator();
```

```
    assertEquals(4, calculator.add(2,2));
}
```

---

11. Tags (Test Categorization)

```
import org.junit.jupiter.api.Tag;

@Test
@Tag("slow")
void slowTestExample() {}
```

Run specific group using Maven configuration.

---

12. Mocking with Mockito (Testing Dependencies)

12.1 Why Mocking?

Unit tests must isolate the class under test.

Example:

- A service depends on database
- Instead of real DB → use mock

12.2 Example Service

```
public interface UserRepository {
    String findUserNameById(int id);
}

public class UserService {
```

```
    private final UserRepository repository;

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

    public String getUserName(int id) {
        return repository.findUserNameById(id);
    }
}
```

12.3 Writing Test with Mockito

```
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;

class UserServiceTest {

    @Test
    void shouldReturnUserNameFromRepository() {
        UserRepository mockRepo = mock(UserRepository.class);

        when(mockRepo.findUserNameById(1))
            .thenReturn("Vishal");

        UserService service = new UserService(mockRepo);

        String result = service.getUserName(1);

        assertEquals("Vishal", result);
        verify(mockRepo).findUserNameById(1);
    }
}
```

Concepts:

- `mock()` creates dummy implementation

- `when().thenReturn()` defines behavior
- `verify()` checks interaction

---

13. Integration Testing with Maven Failsafe

Naming convention:

- Unit tests → `*Test`
- Integration tests → `*IT`

Run integration tests:

```
mvn verify
```

---

14. Best Practices in JUnit Testing

1. Follow AAA pattern:
    - Arrange
    - Act
    - Assert
2. Test behavior, not implementation.
3. Keep tests independent.
4. Avoid testing private methods directly.
5. Use meaningful test names.
6. Cover:

    - Happy path
    - Boundary cases
    - Negative cases
    - Exception scenarios
7. Maintain high coverage but focus on quality.

---

15. Summary

JUnit 5 provides:

- Modern annotation-driven testing
- Rich assertion library
- Parameterized tests
- Nested tests
- Lifecycle management
- Integration with build tools
- Support for mocking frameworks

It is the industry standard framework for unit testing in Java and forms the foundation for advanced frameworks such as Spring Boot Testing, Testcontainers, and Behavior-Driven Development tools.