TDD using JUnit 5 and Mockito

# ✅ Topic Explanation: TDD using JUnit 5 and Mockito

TDD (Test Driven Development) is a software development approach where you write tests before writing the actual code.  
You follow this cycle:  
1. Write a Test (Fail First)  
2. Write the Minimum Code to Pass the Test  
3. Refactor the Code  
4. Repeat

# ✅ About JUnit 5

JUnit 5 is the next generation of JUnit.  
It includes:  
- JUnit Platform: Test engine launcher  
- JUnit Jupiter: For writing tests  
- JUnit Vintage: Backward compatibility  
  
Common annotations in JUnit 5:  
- @Test  
- @BeforeEach  
- @AfterEach  
- @BeforeAll  
- @AfterAll

# ✅ About Mockito

Mockito is a Java framework for mocking dependencies in unit tests.  
It helps in testing classes in isolation by simulating behaviors of other classes.  
You use Mockito when you want to fake the behavior of external services, repositories, or APIs.

# ✅ Exercise 1: Setting Up JUnit 5 and Mockito

## 🎯 Scenario:

You need to set up JUnit 5 and Mockito in your Maven Java project and write your first basic unit test.

## ✅ Steps:

### Step 1: Create a Java Maven Project

- Open your IDE (Eclipse/IntelliJ)  
- Create a Maven project named: JUnitMockitoDemo

### Step 2: Add JUnit 5 and Mockito Dependencies to pom.xml

<dependencies>  
 <!-- JUnit 5 -->  
 <dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter</artifactId>  
 <version>5.10.0</version>  
 <scope>test</scope>  
 </dependency>  
  
 <!-- Mockito -->  
 <dependency>  
 <groupId>org.mockito</groupId>  
 <artifactId>mockito-core</artifactId>  
 <version>5.6.0</version>  
 <scope>test</scope>  
 </dependency>  
</dependencies>

### Step 3: Create a Simple Java Class to Test

// File: Calculator.java  
public class Calculator {  
 public int add(int a, int b) {  
 return a + b;  
 }  
  
 public int multiply(int a, int b) {  
 return a \* b;  
 }  
}

### Step 4: Create JUnit Test Class

// File: CalculatorTest.java  
import static org.junit.jupiter.api.Assertions.\*;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
class CalculatorTest {  
  
 private Calculator calculator;  
  
 @BeforeEach  
 void setUp() {  
 calculator = new Calculator();  
 }  
  
 @Test  
 void testAddition() {  
 assertEquals(5, calculator.add(2, 3));  
 }  
  
 @Test  
 void testMultiplication() {  
 assertEquals(6, calculator.multiply(2, 3));  
 }  
}

### ✅ Step 5: Run the Tests

- Right-click the test file → Run as → JUnit Test  
- Expected Output:  
✔ testAddition()  
✔ testMultiplication()  
(Both green ticks in JUnit view)

### ✅ (Optional) Step 6: Basic Mockito Usage Example

import static org.mockito.Mockito.\*;  
import static org.junit.jupiter.api.Assertions.\*;  
  
import org.junit.jupiter.api.Test;  
import org.mockito.Mockito;  
  
class SampleMockitoTest {  
  
 interface Service {  
 String fetchData();  
 }  
  
 @Test  
 void testWithMockito() {  
 Service serviceMock = mock(Service.class);  
 when(serviceMock.fetchData()).thenReturn("Mock Data");  
  
 String result = serviceMock.fetchData();  
  
 assertEquals("Mock Data", result);  
 }  
}

# ✅ Summary:

In this exercise, you learned:  
- How to set up JUnit 5  
- How to write a simple unit test  
- How to use Mockito for mocking

# TDD using JUnit5 and Mockito

## Exercise 3: Assertions in JUnit

In Test-Driven Development (TDD) using JUnit 5, assertions are used to verify the expected behavior of your code. This helps ensure that the logic you're testing behaves correctly. In Exercise 3, the goal is to write and understand different assertions that JUnit provides.

### Commonly Used JUnit Assertions

import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
  
public class AssertionsTest {  
 @Test  
 public void testAssertions() {  
 // Assert equals: Checks if two values are equal  
 assertEquals(5, 2 + 3);  
  
 // Assert true: Checks if the condition is true  
 assertTrue(5 > 3);  
  
 // Assert false: Checks if the condition is false  
 assertFalse(5 < 3);  
  
 // Assert null: Checks if the object is null  
 assertNull(null);  
  
 // Assert not null: Checks if the object is not null  
 assertNotNull(new Object());  
 }  
}

### Detailed Breakdown of Each Assertion

|  |  |  |
| --- | --- | --- |
| Assertion | Usage Example | What It Verifies |
| assertEquals() | assertEquals(5, 2 + 3) | Expected result (5) matches actual (2 + 3) |
| assertTrue() | assertTrue(5 > 3) | The condition is true |
| assertFalse() | assertFalse(5 < 3) | The condition is false |
| assertNull() | assertNull(null) | The value is null |
| assertNotNull() | assertNotNull(new Object()) | The value is not null |

### Why Assertions Matter in TDD

• They allow you to automatically verify behavior after each code change.

• Help identify bugs early in the development cycle.

• Ensure that refactoring does not break existing behavior.

### Extra Tip

JUnit 5 also includes additional assertions like:

• assertThrows() – Verifies that a method throws an exception.

• assertAll() – Groups multiple assertions.

• assertTimeout() – Fails if execution exceeds a time limit.

### Summary

In this exercise, you:

• Practiced using key JUnit 5 assertions.

• Learned how each assertion type helps in verifying different conditions.

• Gained insight into writing clean, reliable tests as part of TDD.

# TDD using JUnit5 and Mockito

## Exercise 4: Arrange-Act-Assert (AAA) Pattern, Test Fixtures, Setup and Teardown Methods in JUnit

This exercise focuses on two important aspects of writing clean and maintainable unit tests:

1. Using the Arrange-Act-Assert (AAA) pattern

2. Utilizing test fixtures with @BeforeEach and @AfterEach in JUnit 5

### 1. Arrange-Act-Assert (AAA) Pattern

The AAA pattern is a standard structure for writing unit tests. It improves readability and clarity.

AAA Structure:

|  |  |
| --- | --- |
| Phase | Description |
| Arrange | Set up the necessary test data or environment. |
| Act | Call the method or functionality being tested. |
| Assert | Verify that the expected results occurred. |

Example using AAA Pattern:

import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
  
public class CalculatorTest {  
  
 @Test  
 public void testAddition() {  
 // Arrange  
 Calculator calc = new Calculator();  
  
 // Act  
 int result = calc.add(2, 3);  
  
 // Assert  
 assertEquals(5, result);  
 }  
}

### 2. Test Fixtures: Setup and Teardown

JUnit 5 provides lifecycle methods to help reuse setup and cleanup logic.

Common Annotations:

|  |  |
| --- | --- |
| Annotation | Purpose |
| @BeforeEach | Runs before each test method (setup) |
| @AfterEach | Runs after each test method (teardown) |

Example with Setup and Teardown:

import org.junit.jupiter.api.\*;  
import static org.junit.jupiter.api.Assertions.\*;  
  
public class CalculatorLifecycleTest {  
  
 private Calculator calc;  
  
 @BeforeEach  
 public void setUp() {  
 calc = new Calculator();  
 System.out.println("Setup: Calculator instance created");  
 }  
  
 @AfterEach  
 public void tearDown() {  
 System.out.println("Teardown: Test completed");  
 }  
  
 @Test  
 public void testAddition() {  
 int result = calc.add(10, 20);  
 assertEquals(30, result);  
 }  
  
 @Test  
 public void testSubtraction() {  
 int result = calc.subtract(50, 20);  
 assertEquals(30, result);  
 }  
}

Calculator Class (for completeness):

public class Calculator {  
 public int add(int a, int b) {  
 return a + b;  
 }  
 public int subtract(int a, int b) {  
 return a - b;  
 }  
}

### Summary

|  |  |
| --- | --- |
| Concept | Description |
| AAA Pattern | Arrange (setup data), Act (execute), Assert (verify) |
| @BeforeEach | Used for setup before every test |
| @AfterEach | Used for teardown after every test |
| Benefit | Leads to clean, maintainable, and repeatable tests |

# TDD using JUnit 5 and Mockito 4

## Mockito Exercises - Exercise 1: Mocking and Stubbing

### What is Mockito?

Mockito is a popular mocking framework for Java. It allows you to simulate the behavior of external dependencies like APIs, databases, services, etc., without actually calling them. It helps in unit testing by allowing you to focus on the logic of the class under test, without depending on real implementations of dependencies.

### What is Mocking?

Mocking means creating fake objects (mocks) of classes/interfaces that your code depends on. You can define how these mocks should behave when their methods are called.

### What is Stubbing?

Stubbing means pre-defining return values for method calls on mock objects. This is useful when you want the mock to give a controlled, predictable output during the test.

### Scenario Explanation:

Goal: Test a service that depends on an external API, without actually calling the real external API.

### Steps Followed in the Example:

1. Create a Mock Object for the External API:  
ExternalApi mockApi = Mockito.mock(ExternalApi.class);  
We create a mock object of the ExternalApi interface/class.

2. Stub the API Method to Return a Predefined Value:  
when(mockApi.getData()).thenReturn("Mock Data");  
We stub the method getData() to return "Mock Data" whenever it is called during the test.

3. Create Service and Inject Mock:  
MyService service = new MyService(mockApi);  
Inject the mock API into the service class.

4. Call the Method Under Test and Assert:  
String result = service.fetchData();  
assertEquals("Mock Data", result);  
We call the service method and check that the result matches the expected value.

### Full Example Code:

import static org.mockito.Mockito.\*;  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
import org.mockito.Mockito;  
  
public class MyServiceTest {  
  
 @Test  
 public void testExternalApi() {  
 // 1. Create Mock  
 ExternalApi mockApi = Mockito.mock(ExternalApi.class);  
  
 // 2. Stub method  
 when(mockApi.getData()).thenReturn("Mock Data");  
  
 // 3. Inject mock into service  
 MyService service = new MyService(mockApi);  
  
 // 4. Call method and assert  
 String result = service.fetchData();  
 assertEquals("Mock Data", result);  
 }  
}

### Mockito Key Methods Used:

Mockito Method: Mockito.mock(Class.class)  
Purpose: Create a mock object  
  
Mockito Method: when(...).thenReturn(...)  
Purpose: Stub a method  
  
JUnit Method: assertEquals(expected, actual)  
Purpose: Assert and validate the output

### Why Use Mocking & Stubbing?

- Faster tests: No need to wait for real API calls.  
- Isolated testing: Test your logic without dependency on external systems.  
- Controlled test conditions: Simulate API failures, timeouts, etc.

# TDD using JUnit 5 and Mockito 5

## Mockito Exercises - Exercise 2: Verifying Interactions

### What is Interaction Verification in Mockito?

In unit testing, you often want to check whether a method was called, how many times it was called, and with what arguments it was called. This is known as Interaction Verification. Mockito provides the verify() method for this purpose.

### When Do We Use Interaction Verification?

- To ensure that your code under test is correctly communicating with its dependencies (like APIs, services, repositories, etc.).  
- Example scenarios include:  
 • Checking if a method was called at least once.  
 • Checking if a method was called with the correct arguments.  
 • Verifying that a method was never called.  
 • Verifying the exact number of invocations.

### Scenario Explanation:

Goal: Ensure that a specific method of the external API (getData()) was called when the service method fetchData() was executed.

### Steps Followed in the Example:

1. Create a Mock Object:  
ExternalApi mockApi = Mockito.mock(ExternalApi.class);  
Creates a mock for the external dependency ExternalApi.

2. Call the Method on Service:  
MyService service = new MyService(mockApi);  
service.fetchData();  
Instantiate the service and pass the mock dependency. Call the method fetchData on the service.

3. Verify the Interaction:  
verify(mockApi).getData();  
This line verifies that the method getData() was called exactly once on mockApi during this test run. If it was not called, the test will fail.

### Full Example Code:

import static org.mockito.Mockito.\*;  
import org.junit.jupiter.api.Test;  
import org.mockito.Mockito;  
  
public class MyServiceTest {  
  
 @Test  
 public void testVerifyInteraction() {  
 // 1. Create Mock  
 ExternalApi mockApi = Mockito.mock(ExternalApi.class);  
  
 // 2. Call method on service  
 MyService service = new MyService(mockApi);  
 service.fetchData();  
  
 // 3. Verify interaction  
 verify(mockApi).getData();  
 }  
}

### Mockito Verify Methods:

Mockito Method: verify(mock).methodName()  
Purpose: Verify that a method was called once.  
  
Mockito Method: verify(mock, times(n)).methodName()  
Purpose: Verify that a method was called exactly n times.  
  
Mockito Method: verify(mock, never()).methodName()  
Purpose: Verify that a method was never called.  
  
Mockito Method: verify(mock, atLeast(n)).methodName()  
Purpose: Verify that a method was called at least n times.  
  
Mockito Method: verify(mock, atMost(n)).methodName()  
Purpose: Verify that a method was called at most n times.

### Why Use Interaction Verification?

- To ensure correct business logic flow.  
- To check that methods are called when expected, and not called when not expected.  
- To improve test coverage and confidence in your code.