**TDD using JUnit5 and Mockito**

**1. JUnit Basic Testing Exercises**

**Exercise 1: Setting Up Junit**

**Scenario:**

* You need to set up JUnit in your Java project to start writing unit tests.

**Steps:**

1. **Create a Java Project**
   * In Eclipse: File > New > Java Project
   * Project name: JUnitSetup
   * Click Finish
2. **Add JUnit Library**
   * Right-click project > Build Path > Configure Build Path
   * Go to Libraries tab
   * Click Add Library > JUnit > JUnit 4 > Finish
   * Click Apply and Close
3. **Delete module-info.java (if present)**
   * Right-click on it > Delete (This avoids module system conflicts with JUnit
4. **Create a Java Class: Calculator.java**

**CODE:**

**package** com.example.junitsetup;

**public** **class** Calculator {

**public** **int** add(**int** a, **int** b) {

**return** a + b;

}

**public** **int** subtract(**int** a, **int** b) {

**return** a - b;

}

**public** **int** multiply(**int** a, **int** b) {

**return** a \* b;

}

**public** **int** divide(**int** a, **int** b) {

**if** (b == 0) **throw** **new** IllegalArgumentException("Cannot divide by zero");

**return** a / b;

}

}

1. **Create a Test Class: CalculatorTest.java**

**CODE:**

**package** com.example.junitsetup;

**import** **static** org.junit.Assert.\*;

**import** org.junit.Test;

**public** **class** CalculatorTest {

@Test

**public** **void** testAdd() {

Calculator calc = **new** Calculator();

*assertEquals*(5, calc.add(2, 3));

}

@Test

**public** **void** testSubtract() {

Calculator calc = **new** Calculator();

*assertEquals*(1, calc.subtract(4, 3));

}

@Test

**public** **void** testMultiply() {

Calculator calc = **new** Calculator();

*assertEquals*(12, calc.multiply(3, 4));

}

@Test

**public** **void** testDivide() {

Calculator calc = **new** Calculator();

*assertEquals*(2, calc.divide(6, 3));

}

@Test(expected = IllegalArgumentException.**class**)

**public** **void** testDivideByZero() {

Calculator calc = **new** Calculator();

calc.divide(5, 0); // This should throw an exception

}

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 3: Assertions in JUnit**

**Scenario:**

* You need to use different types of assertions in JUnit to validate test results. In this exercise, we will test a basic Student class using various assertion methods.

**Step 1 : Create a Java Class – Student.java**

**CODE:**

**package** com.example.junitassertions;

**public** **class** Student {

**private** String name;

**private** **int** age;

**public** Student(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

**public** String getName() {

**return** name;

}

**public** **boolean** isAdult() {

**return** age >= 18;

}

**public** **static** Student getNullStudent() {

**return** **null**;

}

}

**Step 2 : Create a Test Class – StudentTest.java**

**CODE:**

**package** com.example.junitassertions;

**import** **static** org.junit.Assert.\*;

**import** org.junit.Test;

**public** **class** StudentTest {

@Test

**public** **void** testStudentAssertions() {

Student s = **new** Student("Kaviya", 20);

// assertEquals: Check if name is correct

*assertEquals*("Kaviya", s.getName());

// assertTrue: Student should be adult

*assertTrue*(s.isAdult());

// assertFalse: Student should not be adult

Student teen = **new** Student("Mini", 16);

*assertFalse*(teen.isAdult());

// assertNull: Static method should return null

*assertNull*(Student.*getNullStudent*());

// assertNotNull: Ensure object is not null

*assertNotNull*(s);

}

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

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

**Scenario:**

* You need to organize your unit test logic using the Arrange-Act-Assert (AAA) pattern. You should also use @Before and @After annotations to prepare and clean up resources before and after each test.

**Step 1 : Create a Java Class – BankAccount.java**

**CODE:**

**package** com.example.aaapattern;

**public** **class** BankAccount {

**private** **int** balance;

**public** BankAccount(**int** initialBalance) {

**this**.balance = initialBalance;

}

**public** **void** deposit(**int** amount) {

balance += amount;

}

**public** **void** withdraw(**int** amount) {

**if** (amount > balance) **throw** **new** IllegalArgumentException("Insufficient funds");

balance -= amount;

}

**public** **int** getBalance() {

**return** balance;

}

}

**Step 2 : Create a Test Class – BankAccountTest.java**

**CODE:**

**package** com.example.aaapattern;

**import** **static** org.junit.Assert.\*;

**import** org.junit.Before;

**import** org.junit.After;

**import** org.junit.Test;

**public** **class** BankAccountTest {

**private** BankAccount account;

// Setup method runs before each test

@Before

**public** **void** setUp() {

account = **new** BankAccount(1000); // Arrange

System.***out***.println("Setup complete");

}

// Teardown method runs after each test

@After

**public** **void** tearDown() {

account = **null**;

System.***out***.println("Teardown complete");

}

@Test

**public** **void** testDeposit() {

// Act

account.deposit(500);

// Assert

*assertEquals*(1500, account.getBalance());

}

@Test

**public** **void** testWithdraw() {

// Act

account.withdraw(200);

// Assert

*assertEquals*(800, account.getBalance());

}

@Test(expected = IllegalArgumentException.**class**)

**public** **void** testWithdrawMoreThanBalance() {

// Act

account.withdraw(2000);

}

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**2. Mockito exercises**

**Exercise 1: Mocking and Stubbing**

**Scenario:**

* You need to test a service that depends on an external API. Using Mockito, you'll mock the external API and stub its method to return predefined values.

**Step 1: Create a New Java Project**

1. Open Eclipse
2. Click on File > New > Java Project
3. Name the project: JUnitMockitoMockAndStub
4. Click Finish

**Step 2: Add Required JARs**

1. Download JAR files from these official links:
   * [Mockito Core 4.0.0](https://repo1.maven.org/maven2/org/mockito/mockito-core/4.0.0/mockito-core-4.0.0.jar)
   * [JUnit 4.13.2](https://repo1.maven.org/maven2/junit/junit/4.13.2/junit-4.13.2.jar)
   * [Hamcrest Core 1.3](https://repo1.maven.org/maven2/org/hamcrest/hamcrest-core/1.3/hamcrest-core-1.3.jar)
   * [Byte Buddy 1.12.8](https://repo1.maven.org/maven2/net/bytebuddy/byte-buddy/1.12.8/)
   * [Objenesis 3.2](https://repo1.maven.org/maven2/org/objenesis/objenesis/3.2/)
2. Right-click on the project → Build Path > Configure Build Path
3. Go to Libraries tab
4. Click on Classpath, then click Add External JARs...
5. Select all downloaded .jar files
6. Click Apply and Close

**Step 3: Create Interface – ExternalApi.java**

**CODE:**

**package** com.example.mockitomocking;

**public** **interface** ExternalApi {

String getData();

}

**Step 4: Create Service Class – MyService.java**

**CODE:**

**package** com.example.mockitomocking;

**public** **class** MyService {

**private** ExternalApi api;

**public** MyService(ExternalApi api) {

**this**.api = api;

}

**public** String fetchData() {

**return** api.getData();

}

}

**Step 5: Create Test Class – MyServiceTest.java**

**CODE:**

**package** com.example.mockitomocking;

**import** **static** org.mockito.Mockito.\*;

**import** **static** org.junit.Assert.\*;

**import** org.junit.Test;

**import** org.mockito.Mockito;

**public** **class** MyServiceTest {

@Test

**public** **void** testExternalApi() {

// Mocking the ExternalApi

ExternalApi mockApi = Mockito.*mock*(ExternalApi.**class**);

// Stubbing getData() to return a fake value

*when*(mockApi.getData()).thenReturn("Mock Data");

// Injecting the mock into the service

MyService service = **new** MyService(mockApi);

// Calling the method under test

String result = service.fetchData();

// Asserting the expected result

*assertEquals*("Mock Data", result);

}

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 2: Verifying Interactions**

**Scenario:**

* You want to ensure that a specific method (e.g., getData()) is actually called during the execution of your service logic.  
  Mockito helps verify this using the verify() method.

**Step 1: Create a Java Project**

1. Open Eclipse → File > New > Java Project
2. Name it: MockitoVerifying
3. Click Finish

**Step 2: Add External JARs**

1. Right-click the project → Build Path > Configure Build Path
2. Go to the Libraries tab → Click Add External JARs
3. Add all the required .jar files (listed above)
4. Click Apply and Close

**Step 3: Create an Interface – ExternalApi.java**

**CODE:**

**package** com.example.mockitoverifying;

**public** **interface** ExternalApi {

String getData();

}

**Step 4: Create a Java Class – MyService.java**

**CODE:**

**package** com.example.mockitoverifying;

**public** **class** MyService {

**private** ExternalApi api;

**public** MyService(ExternalApi api) {

**this**.api = api;

}

**public** **void** fetchData() {

api.getData(); // method we will verify

}

}

**Step 5: Create a Test Class – MyServiceTest.java**

**CODE:**

**package** com.example.mockitoverifying;

**import** **static** org.mockito.Mockito.\*;

**import** **static** org.junit.Assert.\*;

**import** org.junit.Test;

**import** org.mockito.Mockito;

**public** **class** MyServiceTest {

@Test

**public** **void** testVerifyInteraction() {

// Step 1: Create a mock of ExternalApi

ExternalApi mockApi = Mockito.*mock*(ExternalApi.**class**);

// Step 2: Create service and call fetchData()

MyService service = **new** MyService(mockApi);

service.fetchData();

// Step 3: Verify that getData() was called once

*verify*(mockApi).getData();

}

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.