**JUnit\_Basic Testing Exercises**

**Exercise 1: Setting Up JUnit**

### **Step-by-Step Setup of JUnit in a Java Project**

### **1. Create a New Java Project**

#### **In IntelliJ IDEA:**

* Open IntelliJ IDEA.
* Go to **File > New > Project**.
* Select **Maven** from the left sidebar (as we’re using Maven to manage dependencies).
* Click **Next**, enter:  
  + **GroupId:** com.example
  + **ArtifactId:** JUnitDemo
* Click **Finish**.

#### **In Eclipse:**

* Go to **File > New > Maven Project**.
* Choose **Create a simple project (skip archetype selection)**.
* Enter:  
  + **Group Id:** com.example
  + **Artifact Id:** JUnitDemo
* Click **Finish**.

### **2. Add JUnit Dependency to pom.xml**

Open the pom.xml file in your project and add the following inside the <dependencies> section:

<dependencies>

<!-- JUnit dependency -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

</dependencies>

#### **After adding:**

* In IntelliJ, **right-click on the pom.xml > Reload Maven Project**.
* In Eclipse, **Right-click on the project > Maven > Update Project**.

This will download and link the JUnit 4.13.2 library.

### **3. Create a New Test Class**

Create a simple Java class that you want to test. Example:

#### **File: Calculator.java**

package com.example;

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

Now create a test class to write unit tests using JUnit.

#### **File: CalculatorTest.java**

package com.example;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calc = new Calculator();

int result = calc.add(5, 3);

assertEquals(8, result);

}

}

### **Notes:**

* @Test: Marks the method as a test method.
* assertEquals(expected, actual): Checks if the expected and actual values are equal. If not, the test fails.
* The test class should be in the src/test/java directory, as Maven expects test files there by default.

Once done, **run the test**:

* In IntelliJ, right-click on the test method or class → **Run 'CalculatorTest'**.
* In Eclipse, right-click the test class → **Run As > JUnit Test**.

If everything is set up correctly, the test will pass.

### **Exercise 2: Writing Basic JUnit Tests**

### **1. Create a New Java Class with Some Methods to Test**

Let’s create a class called MathUtils which contains a few basic arithmetic methods:

#### **MathUtils.java**

package com.example;

public class MathUtils {

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;

}

}

This class includes:

* Addition
* Subtraction
* Multiplication
* Division (with exception handling for divide-by-zero)

### **2. Write JUnit Tests for These Methods**

Create a separate test class in the src/test/java directory.

#### **MathUtilsTest.java**

package com.example;

import org.junit.Test;

import static org.junit.Assert.\*;

public class MathUtilsTest {

MathUtils math = new MathUtils();

@Test

public void testAdd() {

assertEquals(10, math.add(7, 3));

}

@Test

public void testSubtract() {

assertEquals(4, math.subtract(9, 5));

}

@Test

public void testMultiply() {

assertEquals(20, math.multiply(4, 5));

}

@Test

public void testDivide() {

assertEquals(2, math.divide(10, 5));

}

@Test(expected = IllegalArgumentException.class)

public void testDivideByZero() {

math.divide(10, 0);

}

}

### **Explanation of Each Test:**

* testAdd(): Verifies that 7 + 3 equals 10.
* testSubtract(): Verifies 9 - 5 equals 4.
* testMultiply(): Verifies 4 × 5 equals 20.
* testDivide(): Verifies 10 ÷ 5 equals 2.
* testDivideByZero(): Expects an IllegalArgumentException to be thrown when dividing by zero.

### **Running the Tests**

* In IntelliJ or Eclipse, right-click the MathUtilsTest file > **Run**.
* All tests should pass if the implementation is correct.
* If any assertion fails, the test runner will show which test failed and why.

This completes writing and executing basic JUnit tests.

### **Exercise 3: Assertions in JUnit**

### **Objective:**

To demonstrate the use of **various JUnit assertions** to validate test outcomes.

JUnit provides a rich set of assertion methods to perform unit test validations. These assertions help confirm that code behaves as expected.

### **Solution Code with Explanation**

#### 

#### **AssertionsTest.java**

package com.example;

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testAssertions() {

// 1. Assert equals: checks if two values are equal

assertEquals("Sum should be 5", 5, 2 + 3);

// 2. Assert true: checks if a condition is true

assertTrue("5 is greater than 3", 5 > 3);

// 3. Assert false: checks if a condition is false

assertFalse("5 is not less than 3", 5 < 3);

// 4. Assert null: checks if a value is null

assertNull("Should be null", null);

// 5. Assert not null: checks if a value is not null

assertNotNull("Should not be null", new Object());

}

}

### **Explanation of Each Assertion**

| **Assertion Method** | **Purpose** | **Example Result** |
| --- | --- | --- |
| assertEquals(a, b) | Passes if a == b | assertEquals(5, 2 + 3) |
| assertTrue(condition) | Passes if condition is true | assertTrue(5 > 3) |
| assertFalse(condition) | Passes if condition is false | assertFalse(5 < 3) |
| assertNull(obj) | Passes if obj == null | assertNull(null) |
| assertNotNull(obj) | Passes if obj != null | assertNotNull(new Object()) |

You can optionally pass a message as the **first argument** in assertions, which helps during test failure analysis.

### **Running the Test**

* Run the test class as a **JUnit test** from your IDE.
* All assertions should pass if the logic is correct.
* If any assertion fails, the test runner will highlight the failure and the corresponding message (if given).

This exercise illustrates how to use various assertion types to validate different conditions in your unit tests.

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

### **Objective:**

To structure tests using the **AAA (Arrange-Act-Assert)** pattern and manage **test setup/cleanup** using @Before and @After annotations in JUnit 4.

### **Step-by-Step Breakdown**

### **1. Understanding the AAA Pattern**

| **Phase** | **Description** |
| --- | --- |
| Arrange | Set up any necessary data or test objects |
| Act | Execute the method or behavior under test |
| Assert | Verify the result or expected outcome |

### **2. Using @Before and @After Annotations**

* @Before: Method runs **before every test method** – used to initialize objects or data.
* @After: Method runs **after every test method** – used for cleanup (closing connections, clearing memory, etc.)

### **Example Code**

### **BankAccount.java (class to be tested)**

package com.example;

public class BankAccount {

private double balance;

public BankAccount(double initialBalance) {

this.balance = initialBalance;

}

public void deposit(double amount) {

balance += amount;

}

public boolean withdraw(double amount) {

if (amount > balance) return false;

balance -= amount;

return true;

}

public double getBalance() {

return balance;

}

}

#### **BankAccountTest.java (JUnit test using AAA and setup/teardown)**

package com.example;

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class BankAccountTest {

private BankAccount account;

@Before

public void setUp() {

// Arrange: initialize before each test

account = new BankAccount(100.0);

System.out.println("Setup completed");

}

@After

public void tearDown() {

// Cleanup after each test

account = null;

System.out.println("Teardown completed");

}

@Test

public void testDeposit() {

// Arrange is already done in setUp()

// Act

account.deposit(50.0);

// Assert

assertEquals(150.0, account.getBalance(), 0.001);

}

@Test

public void testWithdrawSuccess() {

// Act

boolean result = account.withdraw(40.0);

// Assert

assertTrue(result);

assertEquals(60.0, account.getBalance(), 0.001);

}

@Test

public void testWithdrawFailure() {

// Act

boolean result = account.withdraw(200.0);

// Assert

assertFalse(result);

assertEquals(100.0, account.getBalance(), 0.001);

}

}

### **Key Points:**

* All test methods reuse the account object initialized in @Before, ensuring **independent and isolated tests**.
* @After resets or releases resources, preventing data leakage between tests.
* Each test follows the AAA pattern:  
  + **Arrange**: Handled by @Before setup.
  + **Act**: Perform method call being tested.
  + **Assert**: Validate outcome using assertions.

### **Output Example When Running Tests:**

nginx

Setup completed

Teardown completed

Setup completed

Teardown completed

Setup completed

Teardown completed

This shows that @Before and @After run before and after each test case respectively, maintaining a clean test environment.