**Week-2**

**Junit Basic Testing**

**Exercise 1: Setting Up JUnit**

JUnit is one of the most widely used unit testing frameworks for Java. It allows developers to test individual pieces of code (called units) such as methods or classes to ensure they behave as expected. Setting up JUnit is the first step toward implementing Test-Driven Development (TDD) and improving code reliability.

**Steps:**

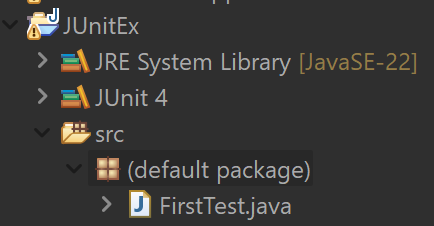
**1.Create a Java Project**  
Open Eclipse → Go to File → New → Java Project → Name it (e.g., JUnitDemo) → Click **Finish**.

**2.Add JUnit Library**  
Right-click on the project → Build Path → Add Libraries → Select **JUnit** → Choose **JUnit 4** → Click **Finish**.

**3.Create a JUnit Test Class**  
Right-click the src package → New → JUnit Test Case → Name it FirstTest → Select **JUnit 4** → Click **Finish**.

**4.Verify Sample Test Method**  
A test class with a sample @Test method will be created.

**Folder Structure:**



**FirstTest.java**

//FirstTest.java

import static org.junit.Assert.\*;

import org.junit.Test;

public class FirstTest {

*@Test*

public void sampleTest() {

*assertEquals*(2 + 2, 4);

}

}

**Exercise 2: Writing Basic JUnit Tests**

JUnit is a widely used testing framework in Java that allows developers to create and run repeatable tests. The purpose of this exercise is to:

* Learn how to write test cases using @Test annotations.
* Use assertEquals to validate output against expected results.
* Verify the correctness of small methods in isolation.

By writing basic unit tests, developers can catch bugs early, improve code quality, and make future changes with more confidence.

**Steps:**

**1.Create a Java Class with Methods**

Create a class MathUtils with methods like add(int a, int b) and multiply(int a, int b).

**2.Create a JUnit Test Class**

Right-click on your package → New → JUnit Test Case → Name it MathUtilsTest and select **JUnit**

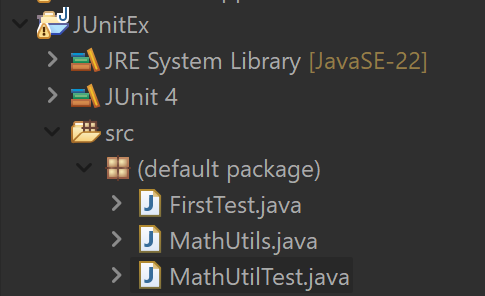
**3.Write Test Methods Using @Test Annotation**

* Use @Test to define test cases.
* Use assertions like assertEquals(expected, actual) to check results.

**4.Run the JUnit Test**

* Right-click the test class → Click **Run As → JUnit Test** to view the results (green = pass, red = fail).

**Folder Structure:**



**MathUtils.java**

**//** MathUtils.java

public class MathUtils {

public int add(int a, int b) {

return a + b;

}

public int multiply(int a, int b) {

return a \* b;

}

}

**MathUtilsTest.java**

**//** MathUtilsTest.java

import org.junit.Test;

import static org.junit.Assert.\*;

public class MathUtilTest {

*@Test*

public void testAdd() {

MathUtils math = new MathUtils();

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

}

*@Test*

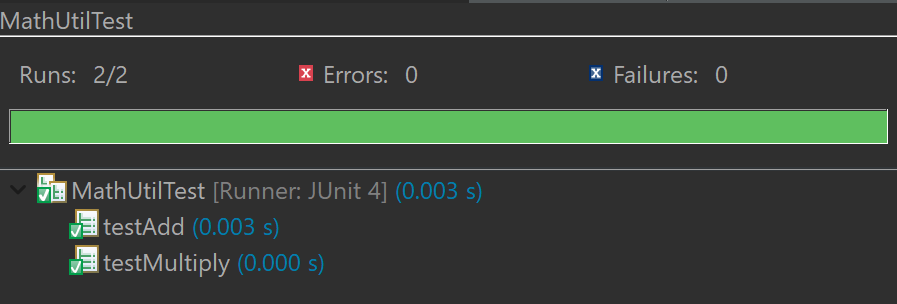
public void testMultiply() {

MathUtils math = new MathUtils();

*assertEquals*(15, math.multiply(3, 5));

}

}



**Exercise 3: Assertions in JUnit**

Assertions are the heart of unit testing in JUnit. They are used to check whether the actual output of your code matches the expected result. This helps developers verify the correctness of their logic during testing.

In this exercise, we explore different types of JUnit assertions such as:

* assertEquals() – for comparing expected and actual values
* assertTrue() / assertFalse() – for testing boolean conditions
* assertNull() / assertNotNull() – for checking object references

Using assertions ensures that our test cases are precise, meaningful, and can detect issues early in the development cycle.

**Steps:**

**1.Create a Java Class with Methods**

Create a class TemperatureConverter with methods like toFahrenheit and toCelsius.

**2.Create a JUnit Test Class**

Right-click on your package → New → JUnit Test Case → Name it TemperatureConverterTest and select **Junit4.**

**3.Write Test Methods Using @Test Annotation**

* Use @Test to define test cases.
* Use assertions like assertEquals, assertTrue and assertNull etc to check results.

**4.Run the JUnit Test**

* Right-click the test class → Click **Run As → JUnit Test** to view the results (green = pass, red = fail).

**TemperatureConverter.java**

//TemperatureConverter.java

public class TemperatureConverter {

// Convert Celsius to Fahrenheit

public double toFahrenheit(double celsius) {

return (celsius \* 9 / 5) + 32;

}

// Convert Fahrenheit to Celsius

public double toCelsius(double fahrenheit) {

return (fahrenheit - 32) \* 5 / 9;

}

}

**TemperatureConverterTest.java**

//TemperatureConverterTest.java

import org.junit.Test;

import static org.junit.Assert.\*;

public class TemperatureConverterTest {

*@Test*

public void testToFahrenheit() {

TemperatureConverter converter = new TemperatureConverter();

double result = converter.toFahrenheit(0);

*assertEquals*(32.0, result, 0.001); // Expected value

*assertTrue*(result > 30); // Logical check

*assertFalse*(result < 0); // Should not be negative

*assertNotEquals*(100.0, result, 0.001); // Not 100°F

}

*@Test*

public void testToCelsius() {

TemperatureConverter converter = new TemperatureConverter();

double result = converter.toCelsius(212);

*assertEquals*(100.0, result, 0.001);

*assertTrue*(result >= 100);

*assertFalse*(result < 0);

}

*@Test*

public void testNotNullConversion() {

TemperatureConverter converter = new TemperatureConverter();

Double result = converter.toFahrenheit(10);

*assertNotNull*(result); // Object is not null

}

*@Test*

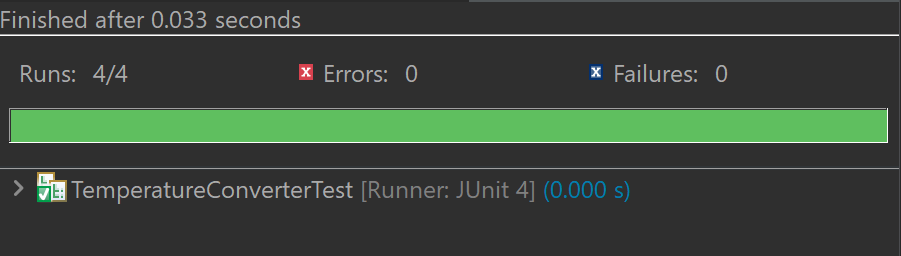
public void testNullCheck() {

String str = null;

*assertNull*(str); // Intentionally checking null

}

}



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

In unit testing, it's important to **organize your test code** for readability and reusability. The **AAA Pattern** helps with this by dividing test logic into three clear sections:

1. **Arrange** – Set up the test data and environment
2. **Act** – Call the method under test
3. **Assert** – Check that the result is as expected

Additionally, JUnit provides **test fixtures** via the @Before and @After annotations:

* @Before: Runs **before each test method** – useful for common setup code
* @After: Runs **after each test method** – useful for cleanup

These features help make your tests **cleaner, less repetitive, and easier to maintain**.

**Steps:**

**1.Create the Main Class**   
Define methods like donate(int), receive(int), and isEligibleToDonate(age, weight) to simulate a blood bank system.

**2.Create JUnit Test Class**   
Write test cases for donating, receiving blood, and checking donor eligibility using the Arrange-Act-Assert (AAA) pattern.

**3.Use @Before and @After Annotations**

* Use @Before to initialize a new BloodBank object before each test.
* Use @After to clean up or log messages after each test runs.

**4.Run the JUnit Test Class**Right-click on BloodBankTest.java → Choose Run As → JUnit Test → Ensure all tests pass (green bar).

**BloodBank.java**

//BloodBank.java

public class BloodBank {

private int bloodUnits = 0;

// Method to donate blood

public void donate(int units) {

if (units > 0) {

bloodUnits += units;

}

}

// Method to receive blood

public boolean receive(int units) {

if (units <= bloodUnits) {

bloodUnits -= units;

return true;

} else {

return false;

}

}

// Check eligibility to donate based on age and weight

public boolean isEligibleToDonate(int age, double weight) {

return age >= 18 && weight >= 50.0;

}

public int getTotalUnits() {

return bloodUnits;

}

}

**BloodBankTest.java**

//BloodBankTest.java

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class BloodBankTest {

private BloodBank bloodBank;

// Runs before each test (Arrange step)

*@Before*

public void setUp() {

bloodBank = new BloodBank();

System.***out***.println("Setup: New blood bank initialized");

}

// Runs after each test (Teardown step)

*@After*

public void tearDown() {

System.***out***.println("Teardown: Test completed\n");

}

*@Test*

public void testDonate() {

// Act

bloodBank.donate(3);

// Assert

*assertEquals*(3, bloodBank.getTotalUnits());

}

*@Test*

public void testReceiveSuccess() {

bloodBank.donate(5);

boolean result = bloodBank.receive(3);

*assertTrue*(result); // Assert True

*assertEquals*(2, bloodBank.getTotalUnits()); // Assert Equals

}

*@Test*

public void testReceiveFail() {

bloodBank.donate(2);

boolean result = bloodBank.receive(5);

*assertFalse*(result); // Not enough blood to receive

*assertEquals*(2, bloodBank.getTotalUnits()); // Units unchanged

}

*@Test*

public void testEligibility() {

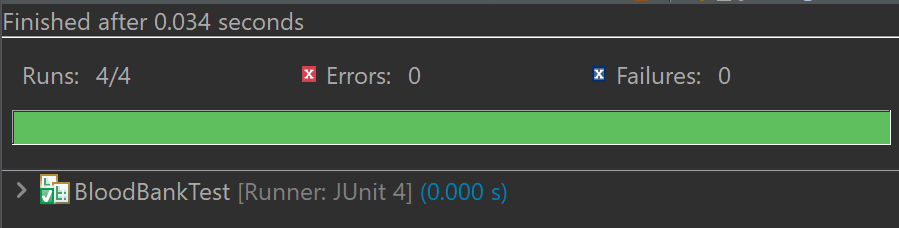
*assertTrue*(bloodBank.isEligibleToDonate(25, 60.0));

*assertFalse*(bloodBank.isEligibleToDonate(16, 45.0)); // Underage and underweight

*assertFalse*(bloodBank.isEligibleToDonate(20, 40.0)); // Only underweight

}

}



**Conclusion:**

With these exercises, we learned how to install and configure JUnit and write unit tests in Java. We learned how to write simple tests, with different kinds of assertions, and the way we could structure the logic of the test, following the AAA pattern.  
  
It was made possible thanks to test fixtures with @Before and @After to avoid duplications and make the code more maintainable. As we learned by doing with examples like temperature conversion and blood donation, we could see practical applications of testing.  
  
In general, JUnit can improve code quality, make debugging easier, and encourage test-driven development.

**-- THE END --**