**JUNIT BASIC TESTING EXERCISES**

**Exercise 1: Setting Up JUnit in IntelliJ IDEA**

**STEP 1 : Create a new Java project in your IDE :**

* Open IntelliJ IDEA.
* Go to **File > New > Project**.
* Select **Java**, choose the SDK, click **Next**, and finish the setup.

**STEP 2 : Add JUnit dependency to your project:**

* Right-click your project > **Add Framework Support** > Select **Maven**.
* IntelliJ will generate a pom.xml file.
* Add the following dependency inside the <dependencies> section:

<dependency>

<groupId>junit</groupId>

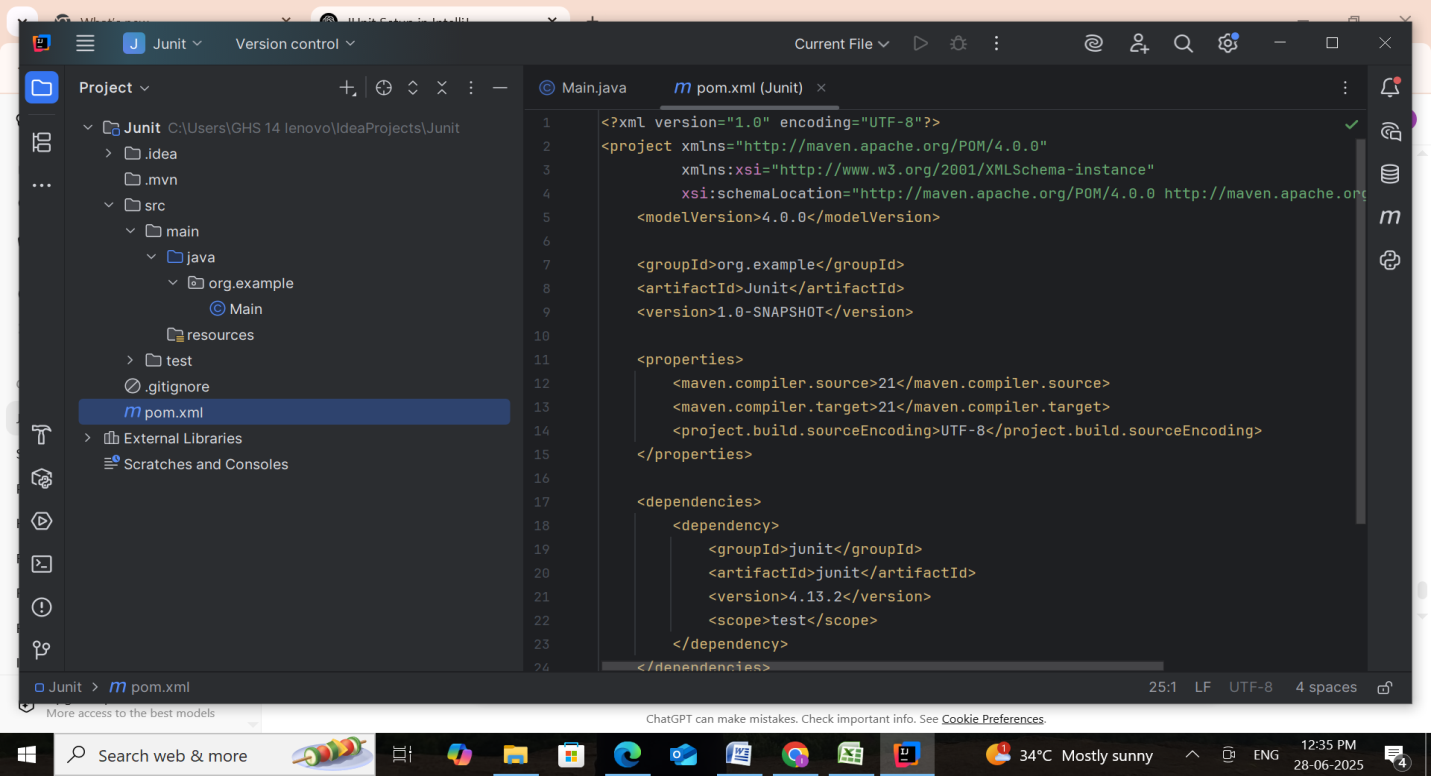
<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

* Save the file and let IntelliJ auto-download the dependency.



**Step 3: Create a Test Class**

* Right-click on src/test/java > **New > Java Class**.
* Name it something like MyTest.
* Add the following sample code:

**Main.java**

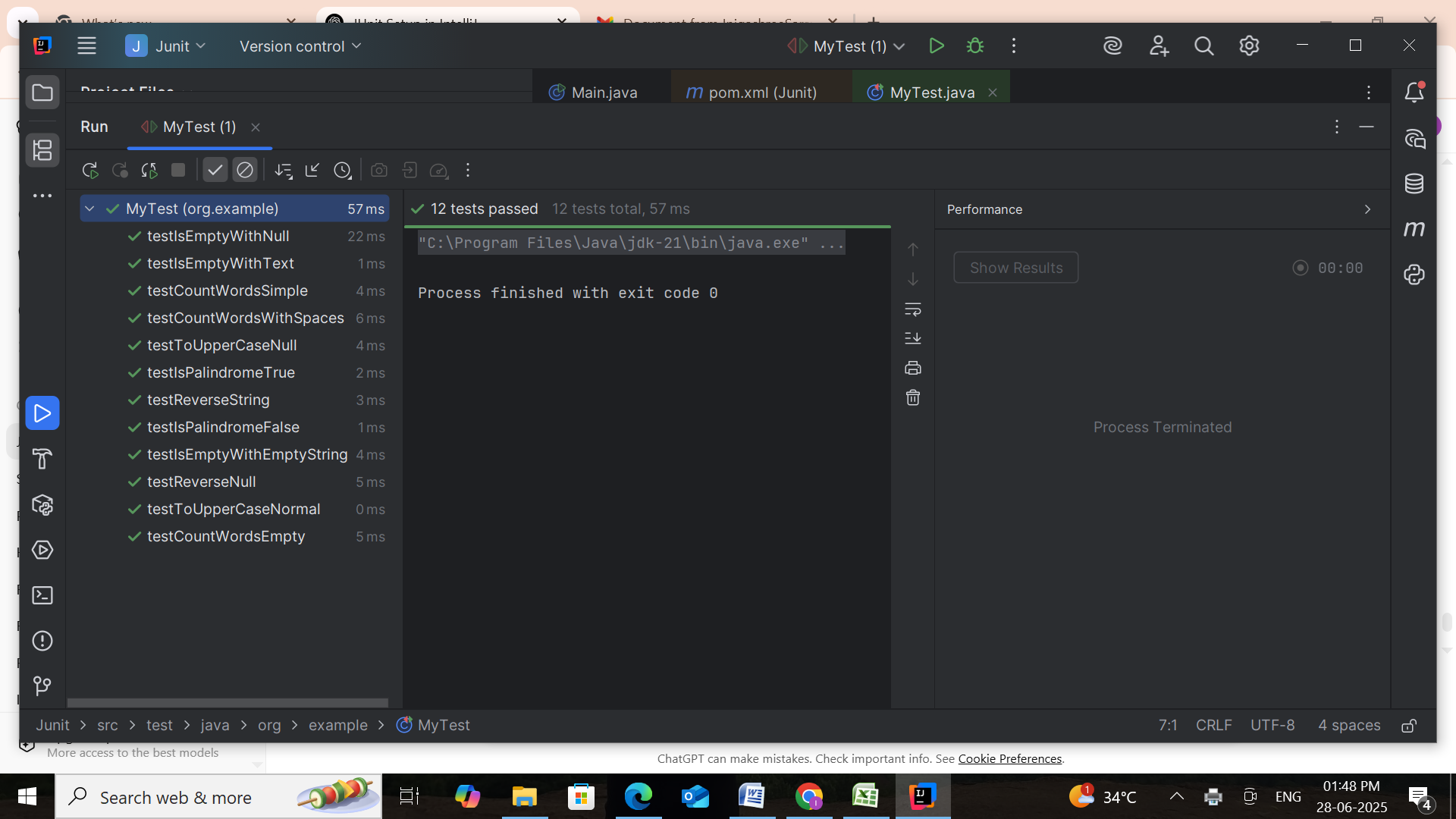
package org.example;  
  
public class Main {  
  
 public boolean isEmpty(String input) {  
 return input == null || input.isEmpty();  
 }  
  
 public boolean isPalindrome(String input) {  
 if (input == null) return false;  
 String clean = input.replaceAll("[\\W\_]", "").toLowerCase();  
 return new StringBuilder(clean).reverse().toString().equals(clean);  
 }  
  
 public String reverse(String input) {  
 if (input == null) return null;  
 return new StringBuilder(input).reverse().toString();  
 }  
  
 public int countWords(String input) {  
 if (input == null || input.trim().isEmpty()) return 0;  
 return input.trim().split("\\s+").length;  
 }  
  
 public String toUpperCase(String input) {  
 return input == null ? null : input.toUpperCase();  
 }  
  
 public static void main(String[] args) {  
 Main main = new Main();  
 System.*out*.println("Is 'madam' a palindrome? " + main.isPalindrome("madam"));  
 System.*out*.println("Reversed of 'hello' = " + main.reverse("hello"));  
 System.*out*.println("Word count: " + main.countWords("JUnit is simple to learn"));  
 }  
}

**MyTest.java**

package org.example;  
  
import org.junit.Test;  
import static org.junit.Assert.\*;  
  
public class MyTest {  
  
 Main main = new Main();  
  
 @Test  
 public void testIsEmptyWithNull() {  
 *assertTrue*(main.isEmpty(null));  
 }  
  
 @Test  
 public void testIsEmptyWithEmptyString() {  
 *assertTrue*(main.isEmpty(""));  
 }  
  
 @Test  
 public void testIsEmptyWithText() {  
 *assertFalse*(main.isEmpty("JUnit"));  
 }  
  
 @Test  
 public void testIsPalindromeTrue() {  
 *assertTrue*(main.isPalindrome("Madam"));  
 *assertTrue*(main.isPalindrome("A man, a plan, a canal: Panama"));  
 }  
  
 @Test  
 public void testIsPalindromeFalse() {  
 *assertFalse*(main.isPalindrome("OpenAI"));  
 }  
  
 @Test  
 public void testReverseString() {  
 *assertEquals*("tac", main.reverse("cat"));  
 }  
  
 @Test  
 public void testReverseNull() {  
 *assertNull*(main.reverse(null));  
 }  
  
 @Test  
 public void testCountWordsSimple() {  
 *assertEquals*(3, main.countWords("I love Java"));  
 }  
  
 @Test  
 public void testCountWordsWithSpaces() {  
 *assertEquals*(2, main.countWords(" Hello World "));  
 }  
  
 @Test  
 public void testCountWordsEmpty() {  
 *assertEquals*(0, main.countWords(" "));  
 }  
  
 @Test  
 public void testToUpperCaseNormal() {  
 *assertEquals*("HELLO", main.toUpperCase("hello"));  
 }  
  
 @Test  
 public void testToUpperCaseNull() {  
 *assertNull*(main.toUpperCase(null));  
 }  
}

### ****Step 4: Run the Test****

1. Right-click anywhere in the MyTest.java file.
2. Click **Run 'MyTest'**.
3. You should see the **test pass** with a green checkmark in the Run panel.



**Exercise 3: Assertions in JUnit**

This Java project uses **JUnit 5** to perform unit testing on a Student class. It focuses on validating academic logic like grade calculation, average computation, malfunction detection, and identifying highest and lowest marks.

Through structured tests, the project ensures correctness in normal and edge-case scenarios.

### ****Goals:****

#### Verify Core Functionalities

* Test accurate computation of:
  + **Average marks**
  + **Grade classification** (A, B, C, etc.)
  + **Highest and lowest subject marks**

#### Handle Invalid & Edge Scenarios

* Validate behavior when:
  + Marks list is **empty**
  + A subject is **failed** (score < 40)
  + All subjects are passed with various grades

#### Assess Logical Conditions

* Test logic-driven methods like:
  + hasFailedAnySubject() – detects any failing score
  + Grading thresholds – verifies if correct grade is assigned based on average

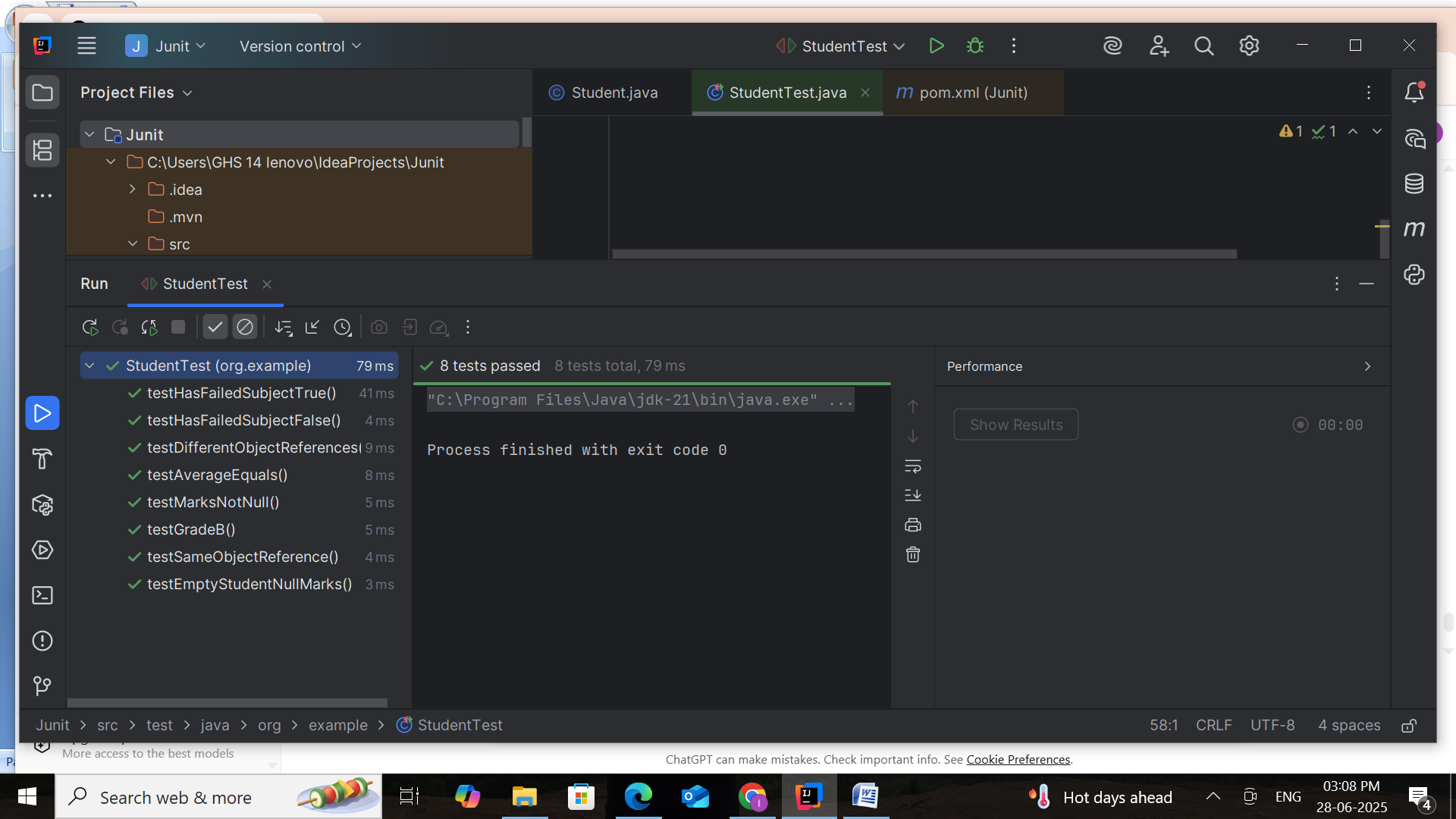
**Code & Output :**

Student.java

package org.example;  
  
public class Student {  
 private String name;  
 private int[] marks;  
  
 public Student(String name, int[] marks) {  
 this.name = name;  
 this.marks = marks;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public int[] getMarks() {  
 return marks;  
 }  
  
 public double getAverage() {  
 if (marks == null || marks.length == 0) return 0.0;  
 int sum = 0;  
 for (int mark : marks) {  
 sum += mark;  
 }  
 return (double) sum / marks.length;  
 }  
  
 public String getGrade() {  
 double avg = getAverage();  
 if (avg >= 90) return "A";  
 if (avg >= 75) return "B";  
 if (avg >= 50) return "C";  
 return "F";  
 }  
  
 public boolean hasFailedAnySubject() {  
 if (marks == null) return false;  
 for (int mark : marks) {  
 if (mark < 40) return true;  
 }  
 return false;  
 }  
  
 public int getHighestMark() {  
 if (marks == null || marks.length == 0) return 0;  
 int max = marks[0];  
 for (int mark : marks) {  
 if (mark > max) max = mark;  
 }  
 return max;  
 }  
  
 public int getLowestMark() {  
 if (marks == null || marks.length == 0) return 0;  
 int min = marks[0];  
 for (int mark : marks) {  
 if (mark < min) min = mark;  
 }  
 return min;  
 }  
}

**StudentTest.java**

package org.example;  
  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
  
public class StudentTest {  
  
 @Test  
 public void testAverageEquals() {  
 Student s = new Student("Arun", new int[]{80, 90, 100});  
 *assertEquals*(90.0, s.getAverage(), "Average should be 90.0");  
 }  
  
 @Test  
 public void testGradeB() {  
 Student s = new Student("Kavi", new int[]{80, 75, 78});  
 *assertEquals*("B", s.getGrade(), "Grade should be B");  
 }  
  
 @Test  
 public void testHasFailedSubjectTrue() {  
 Student s = new Student("Ravi", new int[]{45, 35, 60});  
 *assertTrue*(s.hasFailedAnySubject(), "Should return true for failed subject");  
 }  
  
 @Test  
 public void testHasFailedSubjectFalse() {  
 Student s = new Student("Anu", new int[]{70, 75, 80});  
 *assertFalse*(s.hasFailedAnySubject(), "Should return false if no failed subjects");  
 }  
  
 @Test  
 public void testMarksNotNull() {  
 Student s = new Student("Divya", new int[]{50, 60});  
 *assertNotNull*(s.getMarks(), "Marks should not be null");  
 }  
  
 @Test  
 public void testEmptyStudentNullMarks() {  
 Student s = new Student("Ghost", null);  
 *assertNull*(s.getMarks(), "Marks should be null");  
 }  
  
 @Test  
 public void testSameObjectReference() {  
 Student s1 = new Student("Arun", new int[]{90, 95});  
 Student s2 = s1;  
 *assertSame*(s1, s2, "Both should refer to the same object");  
 }  
  
 @Test  
 public void testDifferentObjectReferences() {  
 Student s1 = new Student("A", new int[]{50});  
 Student s2 = new Student("B", new int[]{60});  
 *assertNotSame*(s1, s2, "Should not refer to the same object");  
 }  
}



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

This Java project uses **JUnit 5** to test a BankAccount class. It applies the **Arrange-Act-Assert (AAA) pattern** for test clarity and uses @BeforeEach and @AfterEach to manage reusable test setup and cleanup.

The tests validate the correctness of banking operations **like deposit, withdrawal, balance tracking** and **error handling** for invalid transactions.

**Goals:**

**1. Organized Testing with AAA Pattern**

* Tests follow Arrange → Act → Assert structure for clarity and maintainability.
* Common setup and cleanup are handled with @BeforeEach and @AfterEach.

**2. Thorough Functional Verification**

Verifies core behaviors like:

* Deposit and Withdrawal operations
* Accurate balance calculation
* Transaction success/failure logic

### 3. Robust Handling of Edge Cases

Covers scenarios like:

* Overdraft attempts (withdrawal greater than balance)
* Negative deposits or withdrawals
* Zero transaction values

**Code & Output:**

**BankAccount.java**

package org.example;  
  
public class BankAccount {  
 private String owner;  
 private double balance;  
  
 public BankAccount(String owner, double initialAmount) {  
 this.owner = owner;  
 this.balance = initialAmount;  
 }  
  
 public String getOwner() {  
 return owner;  
 }  
  
 public double getBalance() {  
 return balance;  
 }  
  
 public void deposit(double amount) {  
 if (amount <= 0) {  
 throw new IllegalArgumentException("Deposit must be positive.");  
 }  
 balance += amount;  
 }  
  
 public void withdraw(double amount) {  
 if (amount <= 0) {  
 throw new IllegalArgumentException("Withdrawal must be positive.");  
 }  
 if (amount > balance) {  
 throw new IllegalArgumentException("Insufficient funds.");  
 }  
 balance -= amount;  
 }  
  
 public boolean isRich() {  
 return balance >= 100000; // Example threshold  
 }  
}

**BankAccountTest.java**

package org.example;  
  
import org.junit.jupiter.api.\*;  
import static org.junit.jupiter.api.Assertions.\*;  
  
public class BankAccountTest {  
  
 BankAccount account;  
  
 @BeforeEach  
 public void setUp() {  
 System.*out*.println("🔧 Setting up test account...");  
 account = new BankAccount("Alice", 1000.0);  
 }  
  
 @AfterEach  
 public void tearDown() {  
 System.*out*.println("🧹 Cleaning up after test...");  
 account = null;  
 }  
  
 @Test  
 public void testDepositIncreasesBalance() {  
   
 account.deposit(500);  
  
   
 *assertEquals*(1500.0, account.getBalance());  
 }  
  
 @Test  
 public void testWithdrawDecreasesBalance() {  
   
 account.withdraw(300);  
 *assertEquals*(700.0, account.getBalance());  
 }  
  
 @Test  
 public void testWithdrawMoreThanBalanceThrows() {  
   
 *assertThrows*(IllegalArgumentException.class, () -> account.withdraw(2000));  
 }  
  
 @Test  
 public void testNegativeDepositThrows() {  
 *assertThrows*(IllegalArgumentException.class, () -> account.deposit(-100));  
 }  
  
 @Test  
 public void testIsRichReturnsFalse() {  
 *assertFalse*(account.isRich());  
 }  
  
 @Test  
 public void testIsRichReturnsTrue() {  
 BankAccount rich = new BankAccount("Bob", 150000);  
 *assertTrue*(rich.isRich());  
 }  
  
 @Test  
 public void testAccountOwnerNotNull() {  
 *assertNotNull*(account.getOwner());  
 }  
  
 @Test  
 public void testZeroWithdrawalThrows() {  
 *assertThrows*(IllegalArgumentException.class, () -> account.withdraw(0));  
 }  
}

