**Student Name:** KARTHYAYENI P

**Registration No:** 22CSR089

**Course/Batch:** KONGU ENGINEERING COLLEGE (B.E COMPUTER SCIENCE AND ENGINEERING)

**EXERCISE 1: SETTING UP JUnit**

**Introduction:**

This Java project demonstrates how to set up and use JUnit for unit testing a simple Calculator class. It verifies arithmetic operations like addition, subtraction, multiplication, division, and other logic using automated test cases.

**Objective:**

* **Ensure Code Accuracy:** Validate that each method in the Calculator class performs the expected operation correctly.
* **Handle Edge Cases:** Test for exceptional conditions like division by zero to ensure application robustness.
* **Automate Testing:** Use JUnit to automate the testing process, enabling faster development and reliable code changes.

**Implementation Breakdown:**

**Calculator.java:**

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 ArithmeticException("Cannot divide by zero");

return a / b;

}

public int square(int a) {

return a \* a;

}

public int max(int a, int b) {

return a > b ? a : b;

}

public boolean isEven(int a) {

return a % 2 == 0;

}

}

**CalculatorTest.java:**

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

Calculator calc = new Calculator();

@Test

public void testAddPositive() {

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

}

@Test

public void testAddNegative() {

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

}

@Test

public void testSubtract() {

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

}

@Test

public void testMultiply() {

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

}

@Test

public void testMultiplyByZero() {

assertEquals(0, calc.multiply(0, 10));

}

@Test

public void testDivide() {

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

}

@Test(expected = ArithmeticException.class)

public void testDivideByZero() {

calc.divide(10, 0);

}

@Test

public void testSquare() {

assertEquals(16, calc.square(4));

}

@Test

public void testMax() {

assertEquals(9, calc.max(7, 9));

}

@Test

public void testIsEvenTrue() {

assertTrue(calc.isEven(6));

}

@Test

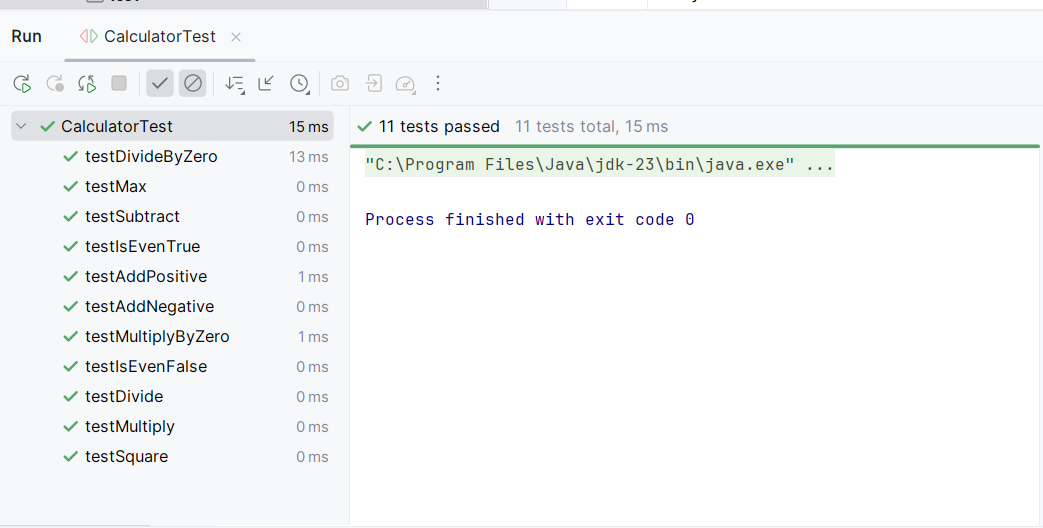
public void testIsEvenFalse() {

assertFalse(calc.isEven(5));

}

}

**Output:**

****

**Conclusion:**

The JUnit test setup successfully verifies the correctness of various calculator functions. It ensures code reliability through automated checks and promotes good testing practices in software development.

**EXERCISE 3: ASSERTIONS IN JUnit**

**Introduction:**

This Java project showcases the use of **JUnit 5** assertions to test the functionality of a BankAccount class, verifying deposit, withdrawal, balance handling, and validation logic through unit testing.

**Objective:**

* **Verify Core Functionalities:** Ensure correct behavior of account operations like deposit, withdrawal, and balance tracking.
* **Handle Invalid Scenarios:** Use assertions to confirm that exceptions are properly thrown for invalid operations (e.g., overdrafts, negative deposits).
* **Assess Logical Conditions:** Test conditions such as checking account owner details and evaluating wealth status (isRich() method).

**Implementation Breakdown:**

**BankAccount.java:**

public class BankAccount {

private String owner;

private double balance;

public BankAccount(String owner, double initialAmount) {

this.owner = owner;

this.balance = initialAmount;

}

public void deposit(double amount) {

if (amount <= 0) throw new IllegalArgumentException("Deposit must be positive");

balance += amount;

}

public void withdraw(double amount) {

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

balance -= amount;

}

public double getBalance() {

return balance;

}

public String getOwner() {

return owner;

}

public boolean isRich() {

return balance >= 1\_00\_000;

}

}

**BankAccountTest.java:**

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

public class BankAccountTest {

@Test

public void testInitialBalance() {

BankAccount acc = new BankAccount("Karthi", 5000);

System.out.println("Initial balance: ₹" + acc.getBalance());

assertEquals(5000, acc.getBalance());

}

@Test

public void testDeposit() {

BankAccount acc = new BankAccount("Karthi", 2000);

acc.deposit(3000);

System.out.println("After deposit, balance: ₹" + acc.getBalance());

assertEquals(5000, acc.getBalance());

}

@Test

public void testWithdraw() {

BankAccount acc = new BankAccount("Karthi", 7000);

acc.withdraw(2000);

System.out.println("After withdrawal, balance: ₹" + acc.getBalance());

assertEquals(5000, acc.getBalance());

}

@Test

public void testOverWithdraw() {

BankAccount acc = new BankAccount("Karthi", 1000);

System.out.println("Trying to withdraw ₹2000 from ₹1000 balance");

assertThrows(IllegalArgumentException.class, () -> acc.withdraw(2000));

}

@Test

public void testNegativeDeposit() {

BankAccount acc = new BankAccount("Karthi", 1000);

System.out.println("Trying to deposit -100");

assertThrows(IllegalArgumentException.class, () -> acc.deposit(-100));

}

@Test

public void testOwnerName() {

BankAccount acc = new BankAccount("Karthi", 1000);

System.out.println("Account belongs to: " + acc.getOwner());

assertEquals("Karthi", acc.getOwner());

}

@Test

public void testMultipleDeposits() {

BankAccount acc = new BankAccount("Karthi", 1000);

acc.deposit(1000);

acc.deposit(2000);

acc.deposit(3000);

System.out.println("After multiple deposits, balance: ₹" + acc.getBalance());

assertEquals(7000, acc.getBalance());

}

@Test

public void testMultipleWithdrawals() {

BankAccount acc = new BankAccount("Karthi", 10000);

acc.withdraw(2000);

acc.withdraw(3000);

System.out.println("After multiple withdrawals, balance: ₹" + acc.getBalance());

assertEquals(5000, acc.getBalance());

}

@Test

public void testIsRichFalse() {

BankAccount acc = new BankAccount("Karthi", 90000);

System.out.println("Is account rich? " + acc.isRich());

assertFalse(acc.isRich());

}

@Test

public void testIsRichTrue() {

BankAccount acc = new BankAccount("Karthi", 100000);

System.out.println("Is account rich? " + acc.isRich());

assertTrue(acc.isRich());

}

@Test

public void testExactWithdraw() {

BankAccount acc = new BankAccount("Karthi", 5000);

acc.withdraw(5000);

System.out.println("Balance after exact withdrawal: ₹" + acc.getBalance());

assertEquals(0, acc.getBalance());

}

@Test

public void testBalanceNotNegative() {

BankAccount acc = new BankAccount("Karthi", 3000);

try {

acc.withdraw(5000);

} catch (Exception e) {

System.out.println("Exception caught: " + e.getMessage());

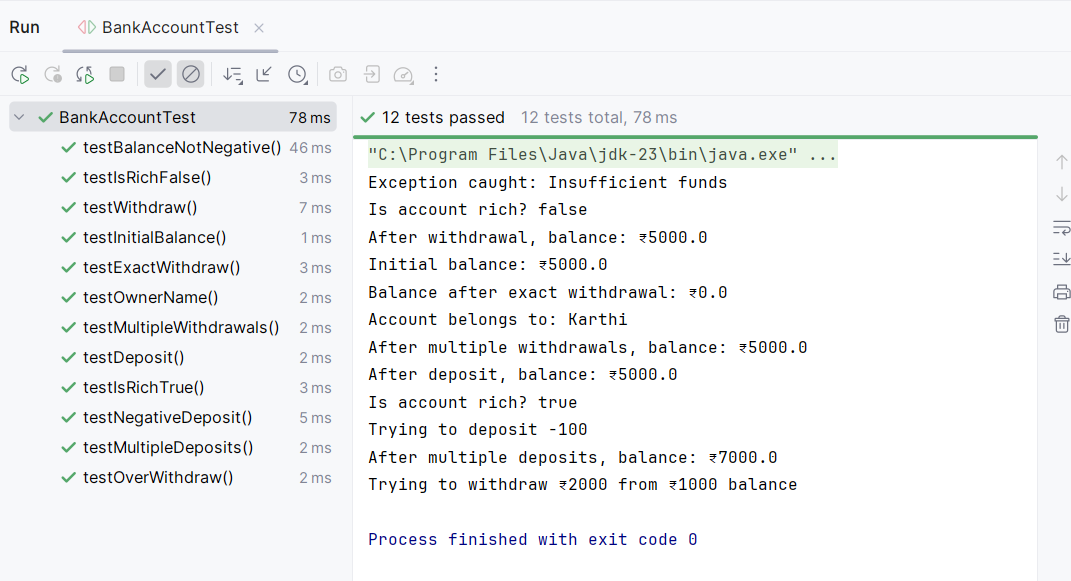
}

assertTrue(acc.getBalance() >= 0);

}

}

**Output:**

****

**Conclusion:**

The test suite effectively validates the BankAccount class using JUnit assertions, ensuring both correctness and robustness of financial operations. This promotes reliable and error-resistant code through automated testing.

**EXERCISE 4: ARRANGE-ACT-ASSERT(AAA) PATTERN, TEST FIXTURES, SETUP AND TEARDOWN METHODS IN JUnit**

**Introduction:**

This Java project uses JUnit to test a ShoppingCart class by applying the Arrange-Act-Assert (AAA) pattern. It demonstrates the use of test fixtures, including @Before for setup and @After for teardown, to ensure clean, consistent testing for shopping cart operations.

**Objective:**

* **Follow AAA Pattern:** Clearly separate test logic into three stages – Arrange (prepare data), Act (execute methods), and Assert (verify outcomes).
* **Use Setup/Teardown Methods:** Apply @Before to initialize the cart before each test and @After to clean up after each test to maintain test isolation.
* **Verify Functional Accuracy:** Test shopping cart operations like adding, removing, clearing items, and calculating totals to ensure expected behavior.

**Implementation:**

**ShoppingCart.java:**

import java.util.\*;

public class ShoppingCart {

private List<String> items = new ArrayList<>();

private Map<String, Double> priceList = new HashMap<>();

public void addItem(String item, double price) {

items.add(item);

priceList.put(item, price);

}

public void removeItem(String item) {

items.remove(item);

priceList.remove(item);

}

public double calculateTotal() {

return items.stream().mapToDouble(item -> priceList.getOrDefault(item, 0.0)).sum();

}

public int getItemCount() {

return items.size();

}

public void clear() {

items.clear();

priceList.clear();

}

public boolean isEmpty() {

return items.isEmpty();

}

}

**ShoppingCartTest.java:**

import static org.junit.Assert.\*;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

public class ShoppingCartTest {

private ShoppingCart cart;

@Before

public void setUp() {

cart = new ShoppingCart();

System.out.println("New cart created.");

}

@After

public void tearDown() {

cart.clear();

System.out.println("Cart cleared.\n");

}

@Test

public void testAddSingleItem() {

String item = "Book";

cart.addItem(item, 150.0);

assertEquals(1, cart.getItemCount());

assertEquals(150.0, cart.calculateTotal(), 0.001);

System.out.println("Added 1 item. Total: ₹" + cart.calculateTotal());

}

@Test

public void testAddMultipleItems() {

cart.addItem("Pen", 10.0);

cart.addItem("Notebook", 40.0);

cart.addItem("Pencil", 5.0);

assertEquals(3, cart.getItemCount());

assertEquals(55.0, cart.calculateTotal(), 0.001);

System.out.println("3 items added. Total: ₹" + cart.calculateTotal());

}

@Test

public void testRemoveItem() {

cart.addItem("Perfume", 500.0);

cart.addItem("Soap", 50.0);

cart.removeItem("Perfume");

assertEquals(1, cart.getItemCount());

assertEquals(50.0, cart.calculateTotal(), 0.001);

System.out.println("Removed 'Perfume'. Remaining total: ₹" + cart.calculateTotal());

}

@Test

public void testClearCart() {

cart.addItem("Bag", 800.0);

cart.addItem("Shoes", 1200.0);

cart.clear();

assertTrue(cart.isEmpty());

assertEquals(0.0, cart.calculateTotal(), 0.001);

System.out.println("Cart cleared manually. Total: ₹" + cart.calculateTotal());

}

@Test

public void testIsEmptyInitially() {

assertTrue(cart.isEmpty());

System.out.println("Cart is empty initially.");

}

@Test

public void testItemCountAfterAddAndRemove() {

cart.addItem("Charger", 300.0);

cart.addItem("Mouse", 700.0);

cart.removeItem("Charger");

assertEquals(1, cart.getItemCount());

System.out.println("Added 2, removed 1. Items left: " + cart.getItemCount());

}

@Test

public void testTotalCalculationAccuracy() {

cart.addItem("Pen Drive", 999.99);

cart.addItem("Headphones", 2000.50);

double total = cart.calculateTotal();

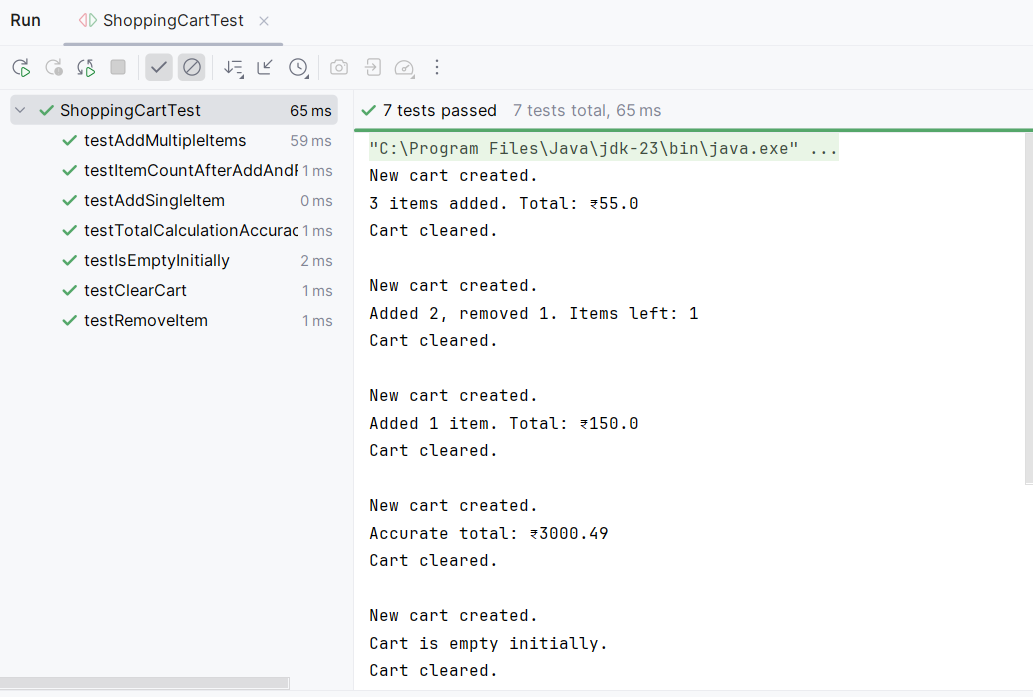
assertEquals(3000.49, total, 0.01);

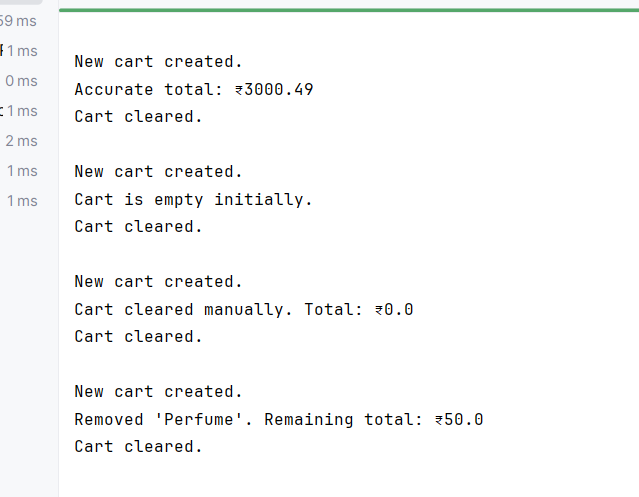
System.out.println("Accurate total: ₹" + total);

}

}

**Output:**

****

****

**Conclusion:**

The test suite successfully applies structured testing using the AAA pattern and lifecycle methods (@Before and @After). It ensures reliable and maintainable unit tests, validating the correctness of shopping cart functionalities across various scenarios.