CodInClub

Powered By BridgeLabz

1️⃣ Basic JUnit Test: Testing a Calculator Class

Problem:

Create a `Calculator` class with methods `add(int a, int b)`, `subtract(int a, int b)`, `multiply(int a, int b)`, and `divide(int a, int b)`. Write JUnit test cases for each method.

👉 **Bonus: Test for division by zero and handle exceptions properly.**

```
package JUnit;

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;
    }
}
```

```
package JUnit;

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

public class CalculatorTest {
    Calculator calculator;
```

```
    @BeforeEach
    public void setUp() {
        calculator = new Calculator();
    }

    @Test
    @DisplayName("Addition Test")
    public void testAdd() {
        assertEquals(5, calculator.add(2, 3));
    }

    @Test
    @DisplayName("Subtraction Test")
    public void testSubtract() {
        assertEquals(1, calculator.subtract(3, 2));
    }

    @Test
    @DisplayName("Multiplication Test")
    public void testMultiply() {
        assertEquals(6, calculator.multiply(2, 3));
    }

    @Test
    @DisplayName("Division Test")
    public void testDivide() {
        assertEquals(2, calculator.divide(6, 3));
    }

    @Test
    @DisplayName("Division by Zero Test")
    public void testDivideByZero() {
        assertThrows(ArithmeticException.class, () -> calculator.divide(6, 0));
    }
}
```

2️⃣ Testing String Utility Methods

Problem:

Create a `StringUtils` class with the following methods:

- **`reverse(String str)`: Returns the reverse of a given string.**
- **`isPalindrome(String str)`: Returns `true` if the string is a palindrome.**
- **`toUpperCase(String str)`: Converts a string to uppercase.**

Write JUnit test cases to verify that these methods work correctly.

```
package JUnit;

public class StringUtils {
    public String reverse(String str) {
        StringBuilder sb = new StringBuilder();
        for (int i = str.length() - 1; i >= 0; i--) {
            sb.append(str.charAt(i));
        }
        return sb.toString();
    }

    public boolean isPalindrome(String str) {
        return str.equals(reverse(str));
    }

    public String toUpperCase(String str) {
        String result = "";
        for (char c : str.toCharArray()) {
            result += Character.toUpperCase(c);
        }
        return result;
    }
}
```

```java
package JUnit;

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

public class StringUtilsTest {
    StringUtils stringUtils;

    @BeforeEach
    public void setUp() {
        stringUtils = new StringUtils();
    }

    @Test
    public void testReverse() {
        assertEquals("cba", stringUtils.reverse("abc"));
    }

    @Test
    public void testIsPalindrome() {
        assertTrue(stringUtils.isPalindrome("madam"));
        assertFalse(stringUtils.isPalindrome("hello"));
    }

    @Test
    public void testToUpperCase() {
        assertEquals("HELLO", stringUtils.toUpperCase("hello"));
    }
}
```

3️⃣ Testing List Operations

Problem:

Create a `ListManager` class that has the following methods:

- **`addElement(List<Integer> list, int element)`: Adds an element to a list.**
- **`removeElement(List<Integer> list, int element)`: Removes an element from a list.**
- **`getSize(List<Integer> list)`: Returns the size of the list.**

Write JUnit tests to verify that:

✅ Elements are correctly added.

✅ Elements are correctly removed.

✅ The size of the list is updated correctly.

```
package JUnit;

import java.util.List;

public class ListManager {

    public void addElement(List<Integer> list, int element) {
        list.add(element);
    }
    public void removeElement(List<Integer> list, int element) {
        list.remove((Integer) element);
    }
    public int getSize(List<Integer> list) {
        return list.size();
    }
}
```

```
package JUnit;

import org.junit.jupiter.api.*;
import java.util.*;
import static org.junit.jupiter.api.Assertions.*;
```

```java
public class ListManagerTest {
    ListManager listManager;
    List<Integer> list;

    @BeforeEach
    public void setUp() {
        listManager = new ListManager();
        list = new ArrayList<>();
    }

    @Test
    public void testAddElement() {
        listManager.addElement(list, 5);
        assertTrue(list.contains(5));
    }

    @Test
    public void testRemoveElement() {
        list.add(5);
        listManager.removeElement(list, 5);
        assertFalse(list.contains(5));
    }

    @Test
    public void testGetSize() {
        list.add(1);
        list.add(2);
        assertEquals(2, listManager.getSize(list));
    }
}
```

4️⃣ Testing Exception Handling

Problem:

Create a method `divide(int a, int b)` that throws an `ArithmeticException` if `b` is zero. Write a JUnit test to verify that the exception is thrown properly.

```
package JUnit;

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

public class DivideTest {
    public int divide(int a, int b) {
        if (b == 0)
            throw new ArithmeticException("Cannot divide by zero");
        return a / b;
    }

    @Test
    public void testExceptionHandling() {
        assertThrows(ArithmeticException.class, () -> divide(5, 0));
    }
}
```

5️⃣ Testing @BeforeEach and @AfterEach Annotations

Problem:

Create a class `DatabaseConnection` with a method `connect()` and `disconnect()`.

- **Use `@BeforeEach` to initialize a database connection before each test.**
- **Use `@AfterEach` to close the connection after each test.**

Write JUnit test cases to verify that the connection is established and closed correctly.

```java
package JUnit;

public class DatabaseConnection {
    public boolean connect() {
        return true;
    }

    public boolean disconnect() {
        return true;
    }
}
```

```java
package JUnit;

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

public class DatabaseConnectionTest {
    DatabaseConnection db;

    @BeforeEach
    public void setUp() {
        db = new DatabaseConnection();
        assertTrue(db.connect());
    }

    @AfterEach
    public void tearDown() {
        assertTrue(db.disconnect());
    }

    @Test
    public void testConnection() {
        assertTrue(db.connect());
    }
}
```

6️⃣ Testing Parameterized Tests

Problem:

Create a method `isEven(int number)` that returns `true` if a number is even.

- **Use `@ParameterizedTest` to test this method with multiple values like `2, 4, 6, 7, 9`.**

```
package JUnit;

public class NumberUtils {
    public boolean isEven(int number) {
        return number % 2 == 0;
    }
}
```

```
package JUnit;

NumberUtilsTest.java

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
import static org.junit.jupiter.api.Assertions.*;

public class NumberUtilsTest {
    NumberUtils numUtils = new NumberUtils();

    @ParameterizedTest
    @ValueSource(ints = { 2, 4, 6, 8, 10 })
    public void testIsEven(int number) {
        assertTrue(numUtils.isEven(number));
    }
}
```

7️⃣Performance Testing Using @Timeout

Problem:

Create a method `longRunningTask()` that sleeps for 3 seconds before returning a result.

- **Use `@Timeout(2)` in JUnit to fail the test if the method takes more than 2 seconds.**

```
package JUnit;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Timeout;
import static org.junit.jupiter.api.Assertions.*;
import java.util.concurrent.TimeUnit;

public class PerformanceTest {
    public void longRunningTask() throws InterruptedException {
        Thread.sleep(3000);
    }

    @Test
    @Timeout(value = 2, unit = TimeUnit.SECONDS)
    public void testLongRunningTask() {
        assertThrows(InterruptedException.class, () -> longRunningTask());
    }
}
```

8️⃣ Testing File Handling Methods

Problem:

Create a class `FileProcessor` with the following methods:

- **`writeToFile(String filename, String content)`: Writes content to a file.**
- **`readFromFile(String filename)`: Reads content from a file.**

Write JUnit tests to check if:

✅ **The content is written and read correctly.**

✅ **The file exists after writing.**

✅ **Handling of `IOException` when the file does not exist.**

```
package JUnit;

import java.io.*;

public class FileProcessor {
    public void writeToFile(String filename, String content) throws IOException
{
        BufferedWriter writer = new BufferedWriter(new FileWriter(filename));
        writer.write(content);
        writer.close();
    }

    public String readFromFile(String filename) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(filename));
        String line = reader.readLine();
        reader.close();
        return line;
    }
}
```

```java
package JUnit;

import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
import java.io.*;

public class FileProcessorTest {
    FileProcessor fileProcessor;

    @BeforeEach
    public void setUp() {
        fileProcessor = new FileProcessor();
    }

    @Test
    public void testFileWriteAndRead() throws IOException {
        String filename = "test.txt";
        String content = "Hello, World!";
        fileProcessor.writeToFile(filename, content);
        assertEquals(content, fileProcessor.readFromFile(filename));
    }
}
```

📝 Advanced JUnit Practice Problems

1️⃣ Testing Banking Transactions

📌 Problem:

Create a `BankAccount` class with:

- `deposit(double amount)`: Adds money to the balance.
- `withdraw(double amount)`: Reduces balance.
- `getBalance()`: Returns the current balance.

✅ **Write JUnit tests to check correct balance updates.**

✅ **Ensure withdrawals fail if funds are insufficient.**

```
package JUnit;

public class BankAccount {
    private double balance;

    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;
    }
}
```

```
package JUnit;

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

public class BankAccountTest {
    BankAccount account;

    @BeforeEach
    public void setUp() {
        account = new BankAccount();
    }

    @Test
    public void testDeposit() {
```

```
        account.deposit(100);
        assertEquals(100, account.getBalance());
    }

    @Test
    public void testWithdraw() {
        account.deposit(100);
        assertTrue(account.withdraw(50));
        assertFalse(account.withdraw(100));
    }
}
```

2️⃣ Testing Password Strength Validator

📌 **Problem:**

Create a `PasswordValidator` class with:

- **Password must have at least 8 characters, one uppercase letter, and one digit.**

✅ **Write JUnit tests for valid and invalid passwords.**

```
package JUnit;

public class PasswordValidator {
    public boolean isValid(String password) {
        return password.length() >= 8 && password.matches(".*[A-Z].*") &&
password.matches(".*\\d.*");
    }
}
```

```java
package JUnit;

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

public class PasswordValidatorTest {
    PasswordValidator validator = new PasswordValidator();

    @Test
    public void testValidPassword() {
        assertTrue(validator.isValid("Test1234"));
    }

    @Test
    public void testInvalidPassword() {
        assertFalse(validator.isValid("test"));
    }
}
```

3️⃣ Testing Temperature Converter

📌 **Problem:**

Create a `TemperatureConverter` class with:

- **`celsiusToFahrenheit(double celsius)`: Converts Celsius to Fahrenheit.**
- **`fahrenheitToCelsius(double fahrenheit)`: Converts Fahrenheit to Celsius.**

✅ **Write JUnit tests to validate conversions.**

```java
package JUnit;

public class TemperatureConverter {
    public double celsiusToFahrenheit(double celsius) {
        return (celsius * 9 / 5) + 32;
    }

    public double fahrenheitToCelsius(double fahrenheit) {
        return (fahrenheit - 32) * 5 / 9;
    }
}
```

```java
package JUnit;

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

public class TemperatureConverterTest {
    TemperatureConverter converter = new TemperatureConverter();

    @Test
    public void testCelsiusToFahrenheit() {
        assertEquals(32, converter.celsiusToFahrenheit(0));
    }

    @Test
    public void testFahrenheitToCelsius() {
        assertEquals(0, converter.fahrenheitToCelsius(32));
    }
}
```

4️⃣ Testing Date Formatter

📌 **Problem:**

Create a `DateFormatter` class with:

- **`formatDate(String inputDate)`: Converts `yyyy-MM-dd` format to `dd-MM-yyyy`.**

✅ **Write JUnit test cases for valid and invalid dates.**

```
package JUnit;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class DateFormatter {
    public String formatDate(String inputDate) {
        return
LocalDate.parse(inputDate).format(DateTimeFormatter.ofPattern("dd-MM-yyyy"));
    }
}
```

```
package JUnit;

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

public class DateFormatterTest {
    DateFormatter formatter = new DateFormatter();

    @Test
    public void testFormatDate() {
        assertEquals("15-08-2023", formatter.formatDate("2023-08-15"));
    }
}
```

5️⃣ Testing User Registration

📌 **Problem:**

Create a `UserRegistration` class with:

- **`registerUser(String username, String email, String password)`.**
- **Throws `IllegalArgumentException` for invalid inputs.**

✅ **Write JUnit tests to verify valid and invalid user registrations.**

```
package JUnit;

public class UserRegistration {
    public boolean registerUser(String username, String email, String password)
{
        if (username.isEmpty() || !email.contains("@") || password.length() <
8) {
            throw new IllegalArgumentException("Invalid input");
        }
        return true;
    }
}
```

```
package JUnit;

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

public class UserRegistrationTest {
    UserRegistration userReg = new UserRegistration();

    @Test
    public void testValidRegistration() {
        assertTrue(userReg.registerUser("JohnDoe", "john@example.com",
"Password1"));
    }

    @Test
```

```
    public void testInvalidRegistration() {
        assertThrows(IllegalArgumentException.class, () ->
userReg.registerUser("", "email.com", "123"));
    }
}
```