Day 30 Assignment

Name: Mehul Anjikhane Email: mehulanjikhane13@gmail.com

# Task 1: Write a set of JUnit tests for a given class with simple mathematical operations (add, subtract, multiply, divide) using the basic @Test annotation.

**Calculator Class:**

**package** assignments;

**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** IllegalArgumentException("Division by zero is not

allowed");

}

}

**return** a / b;

}

# CalculatorTest Class:

**package** assignments;

**import** org.junit.Test;

**import static** org.junit.Assert.\*;

**public class** CalculatorTest {

**private** Calculator calculator = **new** Calculator(); @Test

**public void** testAdd() {

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

}

@Test

**public void** testSubtract() {

*assertEquals*(1, calculator.subtract(3, 2));

}

@Test

**public void** testMultiply() {

*assertEquals*(6, calculator.multiply(2, 3));

}

@Test(expected = IllegalArgumentException.**class**)

**public void** testDivideByZero() { calculator.divide(3, 0);

}

@Test

**public void** testDivide() {

*assertEquals*(2, calculator.divide(6, 3));

}

}

# Task 2: Extend the above JUnit tests to use @Before, @After, @BeforeClass, and @AfterClass annotations to manage test setup and teardown.

**CalculatorTest Class:**

**package** assignments;

**import static** org.junit.Assert.*assertEquals*;

**import** org.junit.\*;

**public class** CalculatorTest {

**private static** Calculator *calculator*; @BeforeClass

**public static void** setUpBeforeClass() **throws** Exception { System.***out***.println("Executed before all test cases"); *calculator* = **new** Calculator();

}

@AfterClass

**public static void** tearDownAfterClass() **throws** Exception { System.***out***.println("Executed after all test cases"); *calculator* = **null**;

}

@Before

**public void** setUp() **throws** Exception { System.***out***.println("Executed before each test case");

}

@After

**public void** tearDown() **throws** Exception { System.***out***.println("Executed after each test case");

}

@Test

**public void** testAdd() {

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

}

@Test

**public void** testSubtract() {

*assertEquals*(1, *calculator*.subtract(3, 2));

}

@Test

**public void** testMultiply() {

*assertEquals*(6, *calculator*.multiply(2, 3));

}

@Test(expected = IllegalArgumentException.**class**)

**public void** testDivideByZero() {

*calculator*.divide(3, 0);

}

@Test

**public void** testDivide() {

*assertEquals*(2, *calculator*.divide(6, 3));

}

}

# Output:

Executed before all test cases Executed before each test case Executed after each test case Executed before each test case Executed after each test case Executed before each test case Executed after each test case Executed before each test case Executed after each test case Executed before each test case Executed after each test case Executed after all test cases

# Task 3: Create test cases with assertEquals, assertTrue, and assertFalse to validate the correctness of a custom String utility class.

**StringUtil Class:**

**package** assignments;

**public class** StringUtil {

**public static boolean** isNullOrEmpty(String str) {

**return** str == **null** || str.isEmpty();

}

**public static boolean** isPalindrome(String str) {

**if** (str == **null**) {

## return false;

}

String reversed = **new** StringBuilder(str).reverse().toString();

**return** str.equals(reversed);

}

**public static** String reverse(String str) {

**if** (str == **null**) {

## return null;

}

**return new** StringBuilder(str).reverse().toString();

}

}

# StringUtilTest Class:

**package** assignments;

**import** org.junit.Test;

**import static** org.junit.Assert.\*;

**public class** StringUtilTest { @Test

**public void** testIsNullOrEmpty() { *assertTrue*(StringUtil.*isNullOrEmpty*(**null**)); *assertTrue*(StringUtil.*isNullOrEmpty*("")); *assertFalse*(StringUtil.*isNullOrEmpty*("abc"));

}

@Test

**public void** testIsPalindrome() { *assertTrue*(StringUtil.*isPalindrome*("madam")); *assertFalse*(StringUtil.*isPalindrome*("hello")); *assertFalse*(StringUtil.*isPalindrome*(**null**));

}

@Test

**public void** testReverse() {

*assertEquals*("cba", StringUtil.*reverse*("abc")); *assertEquals*("", StringUtil.*reverse*("")); *assertNull*(StringUtil.*reverse*(**null**));

}

}

# Task 4: Research and present a comparison of different garbage collection algorithms (Serial, Parallel, CMS, G1, ZGC) in Java.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **GC** | **Characteristics** | **Advantages** | **Disadvantages** | **Use Case** |
| **Serial GC** | Single-threaded, uses Mark-Copy for Young Generation and Mark- Sweep-Compact for Old Generation | Low overhead, simple implementation | Long stop-the-world (STW) pauses, not suitable for applications requiring high  responsiveness | Small applications or environments where pause times are not critical |
| **Parallel GC** | Multi-threaded, uses Mark-Copy for Young Generation and Mark- Sweep-Compact for Old Generation | Reduces pause times by utilizing multiple threads | Still involves lengthy STW pauses for larger heaps | Applications where throughput is more important than low latency  (e.g., batch processing) |
| **CMS GC** | Performs most work concurrently with application threads | Lower pause times compared to Serial and Parallel GC, suitable for applications requiring better responsiveness | Higher CPU usage, potential fragmentation issues, occasional full GC pauses | Interactive applications where response time is critical |
| **G1 GC** | Divides heap into regions, prioritizes regions with the most  garbage, designed for large heaps | Concurrent, compacting collector, predictable pause times | More complex tuning compared to simpler GCs | Applications with large heaps requiring  predictable, low- latency GC |
| **ZGC** | Designed for very large heaps, aims to keep pause times below 10ms | Handles very large heaps efficiently, extremely low pause times | Higher CPU usage, complexity | Real-time systems, applications requiring minimal pause times regardless of heap size |