K DEEPAK **ASSIGNMENT –(DAY-29)**

DAY-22 JUNIT CORE JAVA

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.

```
package junitpackage;

public class Calculator {

    public int sum(int x, int y) {
        return x + y;
    }

    public int difference(int x, int y) {
        return x - y;
    }

    public int product(int x, int y) {
        return x * y;
    }

    public double quotient(int x, int y) {
        if (y == 0) {
            throw new IllegalArgumentException("Cannot divide by zero");
        }
        return (double) x / y;
```

```
    }
}

package junitpackage;

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

class CalculatorTest {

    @Test
    public void testSum() {
        Calculator calc = new Calculator();
        int result = calc.sum(3, 7);
        assertEquals(10, result);
    }

    @Test
    public void testDifference() {
        Calculator calc = new Calculator();
        int result = calc.difference(15, 8);
        assertEquals(7, result);
    }

    @Test
    public void testProduct() {
```

```
        Calculator calc = new Calculator();
        int result = calc.product(6, 9);
        assertEquals(54, result);
    }

    @Test
    public void testQuotient() {
        Calculator calc = new Calculator();
        double result = calc.quotient(20, 4);
        assertEquals(5.0, result, 0.001);
    }

    @Test
    public void testQuotientByZero() {
        Calculator calc = new Calculator();
        Throwable exception = assertThrows(IllegalArgumentException.class, () ->
calc.quotient(5, 0));
        assertEquals("Cannot divide by zero", exception.getMessage());
    }
}
```

Output:

```
Test run started
Test run finished after 93 ms
[         5 tests started      ]
[         5 tests successful   ]
[         0 tests failed       ]
```

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

```
package junitpackage;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.AfterEach;

class CalculatorTest {
    private Calculator calculator;

    @BeforeAll
    public static void setUpBeforeClass() {
        System.out.println("Setting up before the test class");
    }

    @BeforeEach
    public void setUp() {
        calculator = new Calculator();
        System.out.println("Setting up before each test method");
    }

    @Test
    public void testSum() {
```

```
        int result = calculator.sum(9, 3);
        assertEquals(12, result);
    }

    @Test
    public void testDifference() {
        int result = calculator.difference(20, 16);
        assertEquals(4, result);
    }

    @Test
    public void testProduct() {
        int result = calculator.product(7, 3);
        assertEquals(21, result);
    }

    @Test
    public void testQuotient() {
        double result = calculator.quotient(10, 2);
        assertEquals(5.0, result, 0.001);
    }

    @Test
    public void testQuotientByZero() {
        Throwable exception = assertThrows(IllegalArgumentException.class, () ->
calculator.quotient(5, 0));
        assertEquals("Cannot divide by zero", exception.getMessage());
    }
```

```
    @AfterEach
    public void tearDown() {
        calculator = null;
        System.out.println("Tearing down after each test method");
    }

    @AfterAll
    public static void tearDownAfterClass() {
        System.out.println("Tearing down after the test class");
    }
}
```

Output:

All successfully Executed!

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

```
package mystringutils;

public class StringUtils {

    public static String reverseString(String str) {
        return new StringBuilder(str).reverse().toString();
    }

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

```
        String reversed = reverseString(str);
        return str.equalsIgnoreCase(reversed);
    }

    public static boolean containsIgnoreCase(String str, String subStr) {
        return str.toLowerCase().contains(subStr.toLowerCase());
    }
}
package mystringutils;

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

public class StringUtilsTest {

    @Test
    public void testReverseString() {
        assertEquals("olleH", StringUtils.reverseString("Hello"));
        assertEquals("", StringUtils.reverseString(""));
        assertEquals("12345", StringUtils.reverseString("54321"));
    }

    @Test
    public void testIsPalindrome() {
        assertTrue(StringUtils.isPalindrome("radar"));
        assertTrue(StringUtils.isPalindrome("level"));
        assertFalse(StringUtils.isPalindrome("hello"));
    }
```

```
    @Test
    public void testContainsIgnoreCase() {
        assertTrue(StringUtils.containsIgnoreCase("Hello World", "hello"));
        assertTrue(StringUtils.containsIgnoreCase("OpenAI", "openai"));
        assertFalse(StringUtils.containsIgnoreCase("Java Programming",
"python"));
    }
}
```

Output:

```
Test run started
Test run finished after 123 ms
[         3 tests started      ]
[         3 tests successful   ]
[         0 tests failed       ]
```

Task 4:

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

Serial Garbage Collector:

Overview: The Serial Garbage Collector is a stop-the-world, single-threaded algorithm used in Java Virtual Machines (JVMs). It operates in three main phases: Mark, Sweep, and Compact. This collector is most suitable for smaller applications or those running on systems with limited resources.

Pros: Simple and straightforward implementation due to its single-threaded nature. Low overhead in terms of additional threads and complexity.

Cons: Halts the entire application during garbage collection, leading to noticeable pause times. Not ideal for applications requiring high throughput or low pause times.

Parallel Garbage Collector:

Overview: The Parallel Garbage Collector, also known as the Throughput Collector, is designed for applications needing high throughput. It operates similarly to the Serial GC but uses multiple threads for garbage collection tasks.

Pros: Utilizes multiple threads to achieve higher garbage collection efficiency and throughput. Suitable for applications with medium to large heaps that prioritize overall throughput.

Cons: Can cause longer pauses compared to more advanced collectors, which can affect application responsiveness. Not suitable for applications sensitive to pause times and latency.

Concurrent Mark-Sweep (CMS) Garbage Collector:

Overview: The Concurrent Mark-Sweep GC combines concurrent and stop-the-world phases to minimize application pause times. It performs Initial Mark and Remark phases in stop-the-world mode while executing Mark and Sweep concurrently with the application threads.

Pros: Reduces application pause times by executing garbage collection concurrently with the application threads. Suitable for applications requiring low latency and responsiveness.

Cons: May lead to heap fragmentation since it does not compact the entire heap during garbage collection. Requires additional CPU resources for concurrent phases, which can impact application performance.

Garbage-First (G1) Garbage Collector:

Overview: The Garbage-First GC is a region-based collector designed to replace CMS with better performance and lower fragmentation. It divides the heap into regions and collects those with the most garbage first, aiming for predictable pause times.

Pros: Predictable pause times suitable for applications with large heaps and strict latency requirements. Reduces heap fragmentation through region-based collection and incremental compaction.

Cons: More complex to tune and configure compared to simpler collectors like Serial and Parallel GC. May introduce higher CPU and memory overhead due to its region-based approach.

Z Garbage Collector (ZGC):

Overview: The Z Garbage Collector is a concurrent, low-latency collector designed for applications requiring very low pause times, even with very large heap sizes up to terabytes. It uses concurrent Mark and Concurrent Relocate phases to minimize stop-the-world pauses.

Pros: Achieves ultra-low pause times (typically less than 10ms), suitable for latency-sensitive applications. Supports very large heaps, making it scalable for applications with massive memory requirements.

Cons: Higher complexity and resource requirements compared to simpler GC algorithms. Requires more CPU and memory resources to operate efficiently.

Conclusion:

Choosing the right garbage collector depends on your application's specific requirements. Serial GC is suitable for smaller applications with limited resources, while Parallel GC offers higher throughput for medium to large applications. CMS GC reduces pause times but may cause fragmentation, G1 GC provides predictable pauses for large heaps, and ZGC ensures ultra-low latency for very large applications. Each GC algorithm has its trade-offs, and selecting the best one involves considering factors like application size, throughput needs, latency sensitivity, and pause time predictability.