Day - 22

1] 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.

Solution :-

Code :-

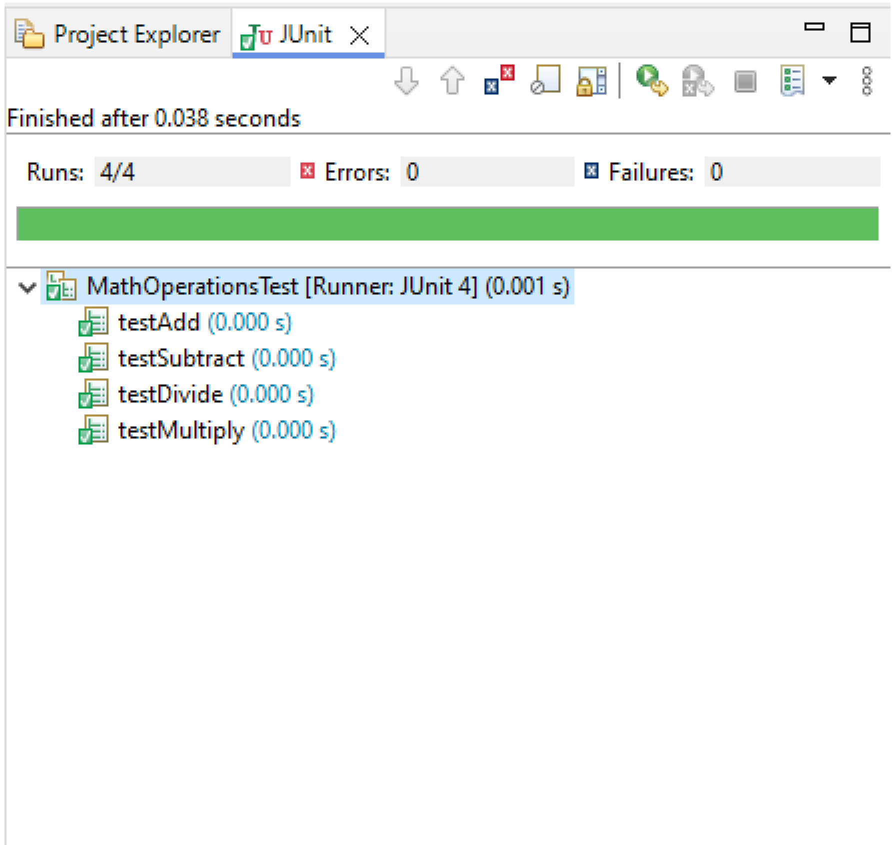
MathOperations.java | MathOperationsTest.java | AssignQues/pom.xml

```java
import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class MathOperationsTest {

    @Test
    public void testAdd() {
        MathOperations mathOperations = new MathOperations();
        assertEquals(4, mathOperations.add(2, 2));
    }

    @Test
    public void testSubtract() {
        MathOperations mathOperations = new MathOperations();
        assertEquals(0, mathOperations.subtract(2, 2));
    }

    @Test
    public void testMultiply() {
        MathOperations mathOperations = new MathOperations();
        assertEquals(4, mathOperations.multiply(2, 2));
    }

    @Test
    public void testDivide() {
        MathOperations mathOperations = new MathOperations();
        assertEquals(1, mathOperations.divide(2, 2));
    }
}
```

MathOperations.java MathOperationsTest.java AssignQues/pom.xml

```java
class MathOperations {
    int add(int a, int b) {
        return a + b;
    }

    int subtract(int a, int b) {
        return a - b;
    }

    int multiply(int a, int b) {
        return a * b;
    }

    int divide(int a, int b) {
        return a / b;
    }
}
```

Output :-



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

Solution :-
Code :-

CalculateTest.java | Calculate.java

```java
package testwithjunit;
public class Calculate {
    public int add(int... number) {
        int result =0;

        for(int i :number) {
            result = result+i;
        }

        return result;
    }

    public int sub(int... number) {
        int result =0;

        for(int i :number) {
            result = result+i;
        }

        return result;
    }
    public int multiply(int... number) {
        int result = 1;
        for(int i:number) {
            result = result*i;
        }
        return result;
    }
    public int divide(int a, int b) {
        int result=0;
```

CalculateTest.java | Calculate.java

```java
            result = result+i;
        }

        return result;
    }
    public int multiply(int... number) {
        int result = 1;
        for(int i:number) {
            result = result*i;
        }
        return result;
    }
    public int divide(int a, int b) {
        int result=0;
        result = a/b;

        return result;
    }
    public static void main(String[] args) {
        Calculate c = new Calculate();
        System.out.println(c.add(1,2));
        System.out.println(c.add(1,2,3));
        System.out.println(c.add(1,2,3,4));

        System.out.println(c.multiply(2,2));
      //System.out.println(c.divide(2, 0));
    }


}
```

CalculateTest.java   Calculate.java

```java
package testwithjunit;

import static org.junit.Assert.assertEquals;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

public class CalculateTest {

    Calculate cal;

    @Rule
    public ExpectedException ex= ExpectedException.none();
    @BeforeClass
    public static void setUpClass() {
        System.out.println("From static setupclass()");
    }

    @Before
    public void setUp() {
        System.out.println("Set up  before");
         cal = new Calculate();
    }

    @Test
```

CalculateTest.java Calculate.java

```java

    @Test
    public void testAdd() {
        System.out.println("From add() ");
        assertEquals(24, cal.add(10,10,4));
    }
    public void testsub() {
        System.out.println("From sub() ");
        assertEquals(24, cal.sub(10,10,4));
    }

    @Test
    public void testMultiply() {
        System.out.println("From Multiply() ");
        assertEquals(3, cal.multiply(1,3));
    }
    @Test
    public void testDivideWithZero() {
        System.out.println("From Divide() ");
        ex.expect(ArithmeticException.class);
        cal.divide(5, 5);
    }


    @After
    public void tearDown() {
        System.out.println("Tear down after ");
        cal=null;
    }
    @AfterClass
```

CalculateTest.java   Calculate.java

```java
    public void testsub() {
        System.out.println("From sub() ");
        assertEquals(24, cal.sub(10,10,4));
    }

    @Test
    public void testMultiply() {
        System.out.println("From Multiply() ");
        assertEquals(3, cal.multiply(1,3));
    }
    @Test
    public void testDivideWithZero() {
        System.out.println("From Divide() ");
        ex.expect(ArithmeticException.class);
        cal.divide(5, 5);
    }


    @After
    public void tearDown() {
        System.out.println("Tear down after ");
        cal=null;
    }
    @AfterClass
    public static void tearDownClass() {
        System.out.println("From static teardownclass()");
    }
}

```

Output :-

```
Console
<terminated> CalculateTest [JUnit] C:\Users\Skynet\.p2\pool\plugins\org.eclips
From static setupclass()
Set up  before
From add()
Tear down after
Set up  before
From Divide()
Tear down after
Set up  before
From Multiply()
Tear down after
From static teardownclass()
```

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

Solution :-
Code :-

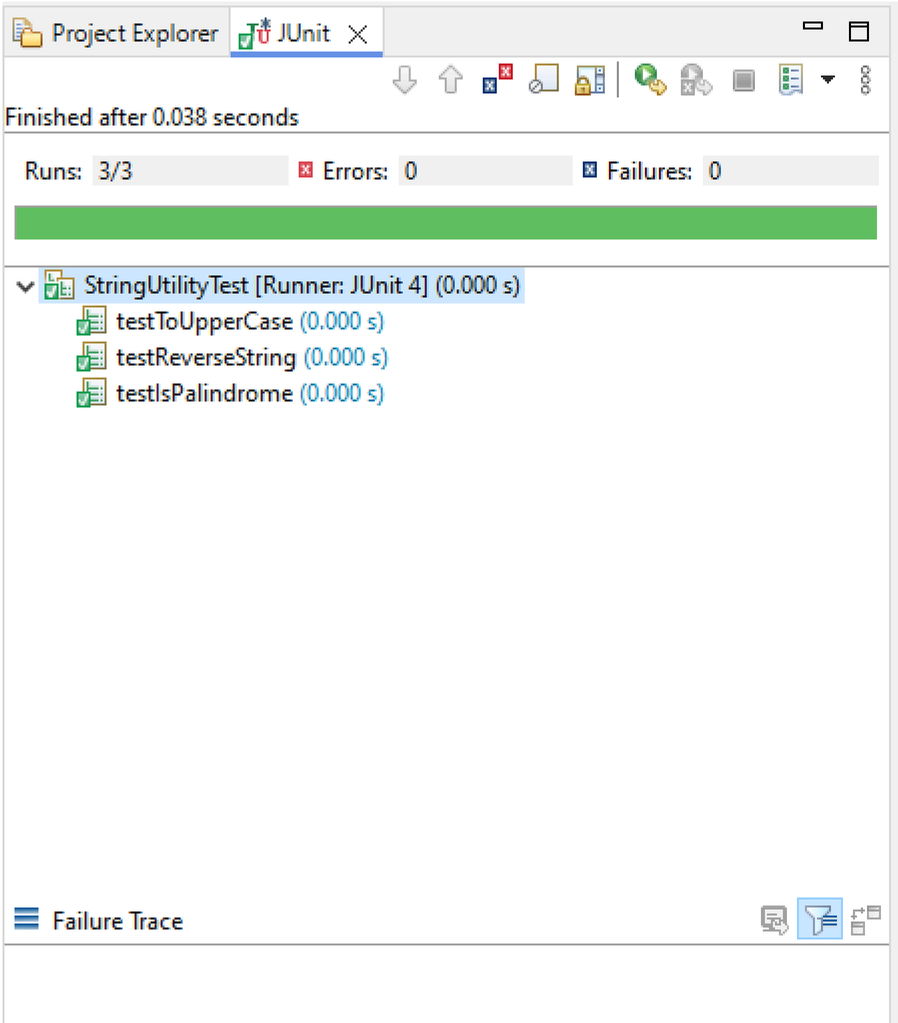
StringUtility.java | StringUtilityTest.java

```java

public class StringUtility {


        public static boolean isPalindrome(String s) {
            String cleaned = s.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();
            return cleaned.equals(new StringBuilder(cleaned).reverse().toString());
        }

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

        public static String toUpperCase(String s) {
            return s.toUpperCase();
        }
    }



```

StringUtility.java | StringUtilityTest.java

```java
import org.junit.Test;
import static org.junit.Assert.*;

    public class StringUtilityTest {

        @Test
        public void testIsPalindrome() {
            assertTrue(StringUtility.isPalindrome("A man, a plan, a canal, Panama"));
            assertTrue(StringUtility.isPalindrome("No 'x' in Nixon"));
            assertFalse(StringUtility.isPalindrome("This is not a palindrome"));
            assertTrue(StringUtility.isPalindrome("racecar"));
            assertFalse(StringUtility.isPalindrome("hello"));
        }

        @Test
        public void testReverseString() {
            assertEquals("olleh", StringUtility.reverseString("hello"));
            assertEquals("racecar", StringUtility.reverseString("racecar"));
            assertEquals("54321", StringUtility.reverseString("12345"));
            assertEquals("", StringUtility.reverseString(""));
        }

        @Test
        public void testToUpperCase() {
            assertEquals("HELLO", StringUtility.toUpperCase("hello"));
            assertEquals("HELLO WORLD", StringUtility.toUpperCase("HeLLo WoRLd"));
            assertEquals("12345", StringUtility.toUpperCase("12345"));
            assertEquals("", StringUtility.toUpperCase(""));
        }
    }
```

Output :-



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

Solution :-

Garbage collection (GC) is a form of automatic memory management in Java. The Java Virtual Machine (JVM) includes several garbage collection algorithms, each with different performance characteristics and use cases. Below is a comparison of some of the most commonly used garbage collection algorithms: Serial, Parallel, CMS, G1, and ZGC.

1. Serial Garbage Collector

Characteristics:

- **Single-threaded:** Uses a single thread for both minor and major garbage collection events.
- **Stop-the-world:** Pauses all application threads during garbage collection.
- **Use Case:** Best suited for small applications with relatively small heap sizes and limited hardware resources (single-processor machines).

Advantages:

- Simple implementation.
- Low overhead for small applications.

Disadvantages:

- Not suitable for large applications or those requiring low latency.
- Long pause times due to stop-the-world nature.

2. Parallel Garbage Collector

Characteristics:

- **Multi-threaded:** Uses multiple threads for both minor and major garbage collection events.
- **Throughput-focused:** Aims to maximize throughput by reducing the overall time spent in garbage collection.
- **Stop-the-world:** Still involves stop-the-world pauses, but with shorter durations compared to the Serial GC due to parallel processing.
- **Use Case:** Suitable for applications that prioritize high throughput over low pause times, often used in multiprocessor environments.

Advantages:

- Better performance on multi-core systems.
- Improved throughput compared to Serial GC.

Disadvantages:

- Longer pause times compared to more advanced collectors like CMS or G1.
- Not ideal for applications requiring low latency.

3. Concurrent Mark-Sweep (CMS) Garbage Collector

Characteristics:

- **Concurrent collection:** Performs most of its work concurrently with the application threads to minimize pause times.

- **Low-latency focus:** Designed to reduce pause times, making it suitable for applications requiring quick response times.
- **Phases:** Initial mark (stop-the-world), concurrent mark, remark (stop-the-world), and concurrent sweep.
- **Use Case:** Ideal for interactive applications where low pause times are critical.

Advantages:

- Shorter pause times compared to Serial and Parallel GCs.
- Better suited for applications with high responsiveness requirements.

Disadvantages:

- More complex implementation.
- Potential for fragmentation issues.
- Higher CPU usage due to concurrent processing.

4. Garbage-First (G1) Garbage Collector

Characteristics:

- **Region-based:** Divides the heap into regions and performs garbage collection on selected regions, aiming to meet a user-defined pause time goal.

- **Predictable pause times:** Designed to provide more predictable pause times by focusing on regions with the most garbage.
- **Mixed collections:** Can perform both young and old generation collections simultaneously.
- **Use Case:** Suitable for large applications requiring a balance between throughput and low pause times, often used in modern Java applications.

Advantages:

- More predictable and configurable pause times.
- Better handling of large heaps compared to CMS.
- Concurrent marking reduces pause times.

Disadvantages:

- Slightly higher overhead compared to simpler collectors.
- Requires tuning to achieve optimal performance.

5. Z Garbage Collector (ZGC)

Characteristics:

- **Low-latency:** Designed to maintain very low pause times (typically in the range of milliseconds) regardless of heap size.
- **Concurrent and scalable:** Most of the work is done concurrently, and it scales well with large heaps.

- **Region-based:** Uses regions similar to G1 but with different mechanisms for garbage collection.
- **Use Case:** Best suited for applications requiring ultra-low pause times and capable of handling very large heaps (multiple terabytes).

Advantages:

- Extremely low pause times, suitable for applications with stringent latency requirements.
- Handles large heaps efficiently.
- Concurrent compaction reduces fragmentation.

Disadvantages:

- Higher CPU usage due to concurrent processes.
- Relatively new, so it may have fewer optimizations and less maturity compared to other collectors.

Summary Comparison Table :-

Garbage Collector	Threads	Pause Times	Throughput	Latency	Suitable For
Serial	Single	Long	Low	High	Small applications with limited resources
Parallel	Multiple	Medium (shorter than Serial)	High	Medium	High-throughput applications on multi-core systems
CMS	Multiple	Short	Medium	Low	Interactive applications with low-latency needs
G1	Multiple	Predictable/Configurable	Medium	Medium/Low	Large applications needing balanced performance
ZGC	Multiple	Extremely short	Medium	Very Low	Applications requiring ultra-low latency and large heap management