Day 22_Assignment

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.

Code: -

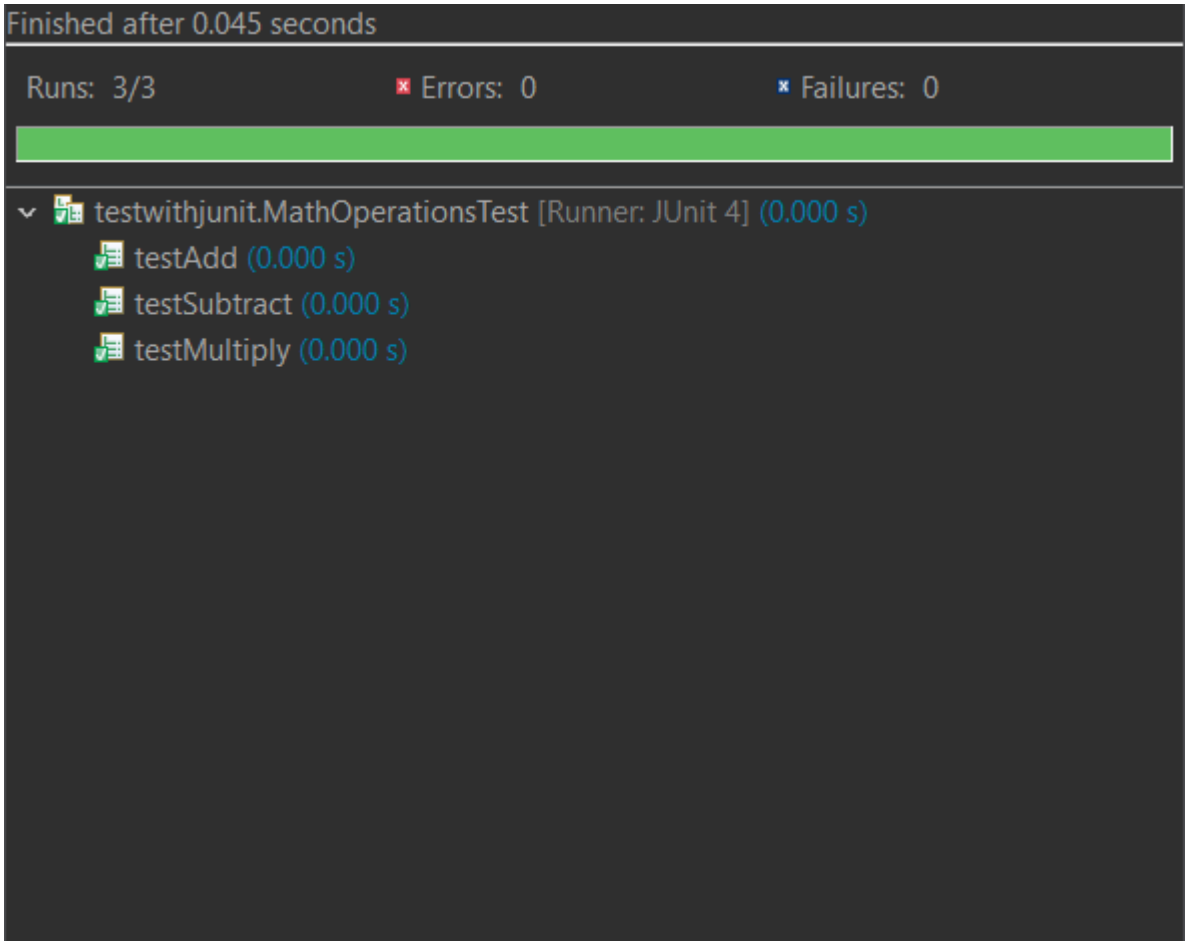
Class

```
1 package testwithjunit;
2 class MathOperations {
3     int add(int a, int b) {
4         return a + b;
5     }
6
7     int subtract(int a, int b) {
8         return a - b;
9     }
0
1     int multiply(int a, int b) {
2         return a * b;
3     }
4
5     int divide(int a, int b) {
6         return a / b;
7     }
8 }
9
```

TestClass

```java
package testwithjunit;
import static org.junit.Assert.assertEquals;

import org.junit.Test;

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));
    }
}
```

Output: -



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

Code: -

```
package testwithjunit;
import static org.junit.Assert.assertEquals;
import org.junit.*;
import org.junit.rules.ExpectedException;
public class MathServiceTest {
     private static MathService cal;
     @Rule
     public ExpectedException ex =
ExpectedException.none();
     @BeforeClass
     public static void setUpClass() {
          System.out.println("BeforeClass: Initializing
Calculator instance...");
```

```
          cal = new MathService();
     }
     @Before
     public void setUp() {
          System.out.println("Before: Setting up for the
test...");
     }
     @Test
     public void testAdd() {
          System.out.println("Test Add...");
          assertEquals(20, cal.add(10, 10));
     }
     @Test
     public void testSub() {
          System.out.println("Test Sub...");
          assertEquals(12, cal.subtract(18, 6));
     }
     @Test
     public void testMul() {
          System.out.println("Test Mul...");
          assertEquals(24, cal.multiply(12, 2));
     }
     @Test
     public void testDivideWithZero() {
          System.out.println("Test Divide by Zero...");
          ex.expect(ArithmeticException.class);
          cal.divide(5, 0);
     }
     @After
     public void tearDown() {
          System.out.println("After: Tearing down...");
     }
     @AfterClass
     public static void tearDownClass() {
          System.out.println("AfterClass: Cleaning up...");
          cal = null;
     }
}
```

Output: -

```
<terminated> MathServiceTest [JUnit] C:\Users\user\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.
BeforeClass: Initializing Calculator instance...
Before: Setting up for the test...
Test Add...
After: Tearing down...
Before: Setting up for the test...
Test Mul...
After: Tearing down...
Before: Setting up for the test...
Test Sub...
After: Tearing down...
Before: Setting up for the test...
Test Divide by Zero...
After: Tearing down...
AfterClass: Cleaning up...
```

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

Code: -

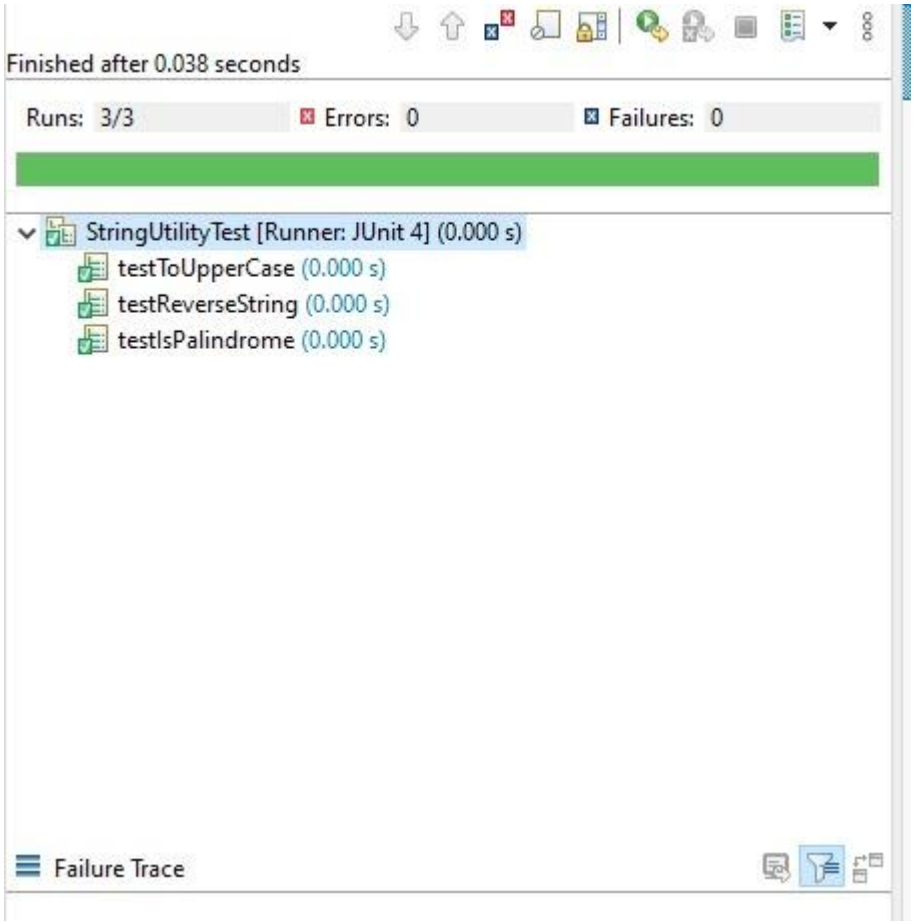
```java
package testwithjunit;
public class StringUtility {

    public static boolean isNotEmpty(String str) {
        return str != null && !str.isEmpty();
    }

    public static boolean isBlank(String str) {
        return str == null || str.isBlank();
    }

    public static String capitalizeFirstLetter(String str) {
        if (str == null || str.isEmpty()) {
            return str;
        }
        return str.substring(0, 1).toUpperCase() + str.substring(1);
    }
}
```

TestClass: -

```
package testwithjunit;
import org.junit.jupiter.api.Test;

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

public class StringUtilityTest {

    @Test
    public void testIsNotEmpty_EmptyString() {
        assertFalse(StringUtility.isNotEmpty(""));
    }

    @Test
    public void testIsNotEmpty_NullString() {
        assertFalse(StringUtility.isNotEmpty(null));
    }

    @Test
    public void testIsNotEmpty_ValidString() {
        assertTrue(StringUtility.isNotEmpty("Hello"));
    }

    @Test
    public void testIsBlank_EmptyString() {
        assertTrue(StringUtility.isBlank(""));
    }

    @Test
    public void testIsBlank_WhitespaceString() {
        assertTrue(StringUtility.isBlank("   "));
    }

    @Test
    public void testIsBlank_NullString() {
        assertTrue(StringUtility.isBlank(null));
    }

    @Test
    public void testIsBlank_ValidString() {
        assertFalse(StringUtility.isBlank("Hello"));
```

```
    }

    @Test
    public void testCapitalizeFirstLetter_EmptyString() {
        assertEquals("",
StringUtility.capitalizeFirstLetter(""));
    }

    @Test
    public void testCapitalizeFirstLetter_NullString() {
        assertEquals(null,
StringUtility.capitalizeFirstLetter(null));
    }

    @Test
    public void
testCapitalizeFirstLetter_SingleCharacter() {
        assertEquals("A",
StringUtility.capitalizeFirstLetter("a"));
    }

    @Test
    public void testCapitalizeFirstLetter_ValidString() {
        assertEquals("Hello",
StringUtility.capitalizeFirstLetter("hello"));
    }
}
```

Output: -



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

Comparison of Garbage Collection Algorithms in Java:

Java offers various garbage collection (GC) algorithms to manage memory automatically. Each algorithm has its own strengths and weaknesses, making the choice dependent on your application's specific needs. Here's a breakdown of some common algorithms:

1. Serial Garbage Collector (Serial GC):

Description: The simplest GC algorithm. It uses a single thread to perform collection, stopping all application threads during the process.

Strengths:

Simple and easy to implement.

Predictable performance.

Weaknesses:

Causes pauses in application execution due to "stop-the-world" behavior.

Not suitable for multi-core systems or applications sensitive to pauses.

2. Parallel Garbage Collector (Parallel GC):

Description: An extension of Serial GC that utilizes multiple threads to improve performance. It divides the heap into regions and collects them concurrently.

Strengths:

Faster collection than Serial GC due to parallelism.

Better suited for multi-core systems.

Weaknesses:

Still has "stop-the-world" pauses, although shorter than Serial GC.

Increased complexity compared to Serial GC.

3. Concurrent Mark Sweep Collector (CMS):

Description: A low-pause collector designed to minimize application pauses during garbage collection. It runs concurrently with application threads, marking reachable objects and sweeping unreachable ones later.

Strengths:

Aims for minimal application pauses.

Good for interactive applications sensitive to pauses.

Weaknesses:

May not reclaim all garbage in a single cycle, leading to potential full GCs.

Throughput (overall memory management speed) can be lower than Parallel GC.

4. Garbage-First Collector (G1):

Description: A more advanced collector introduced in Java 7. It divides the heap into regions of different sizes and ages. G1 prioritizes collecting regions with more short-lived objects, aiming for predictable pauses regardless of heap size.

Strengths:

Targets short-lived objects for efficient collection.

Provides predictable pauses suitable for real-time applications.

Adapts to changing memory usage patterns.

Weaknesses:

More complex than other collectors.

May not be ideal for applications with very large heaps.

5. Z Garbage Collector (ZGC):

Description: An experimental collector introduced in Java 11, aiming for low-latency garbage collection with minimal pauses. It utilizes color-based marking and relocation techniques for efficient memory management.

Strengths:

Targets minimal garbage collection pauses (designed for sub-millisecond pauses).

Potential for significant performance improvements in latency-sensitive applications.

Weaknesses:

Still under development and considered experimental.

Might not be suitable for all applications due to its experimental nature.

Choosing the Right GC Algorithm:

The optimal GC algorithm depends on your application's needs. Here are some general guidelines:

Simple applications: Serial GC might be sufficient.

Multi-core applications: Consider Parallel GC for balanced performance.

Interactive applications sensitive to pauses: CMS collector could be a good choice.

Real-time applications requiring predictable pauses: G1 offers better control over pauses.

Latency-sensitive applications (experimental): ZGC might be worth exploring.

Additional factors to consider:

Heap size

Object allocation patterns

Application performance requirements