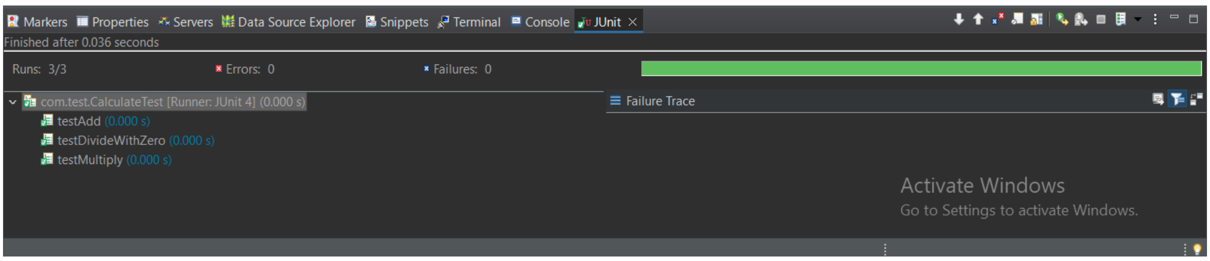
Task 1: Write a set of JUnit tests for a given class with simple mathematical operations (add, subtract, multiply, divide) using the basic @Test annotation.

```java
package com.test;
import static org.junit.Assert.assertEquals;

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

TestWithJunit/... emsmvn/pom.xml Calculate.java CalculateTest.j... AssertionsTest... Validate.java Validate

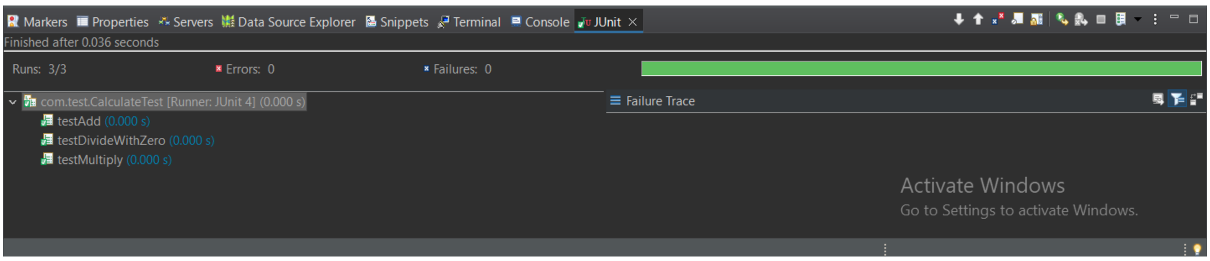
```java
    @Before
    public void setUp() {
        System.out.println("Set up  before");
         cal = new Calculate();
    }

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

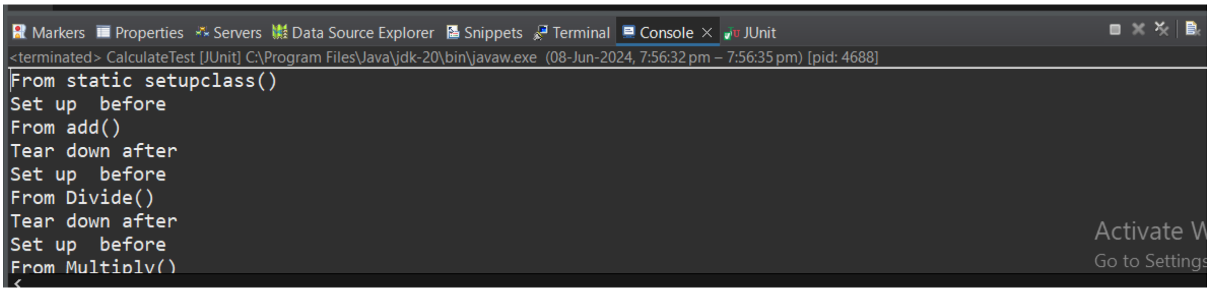


Task 2: Extend the above JUnit tests to use @Before, @After, @BeforeClass, and @AfterClass annotations to manage test setup and teardown.

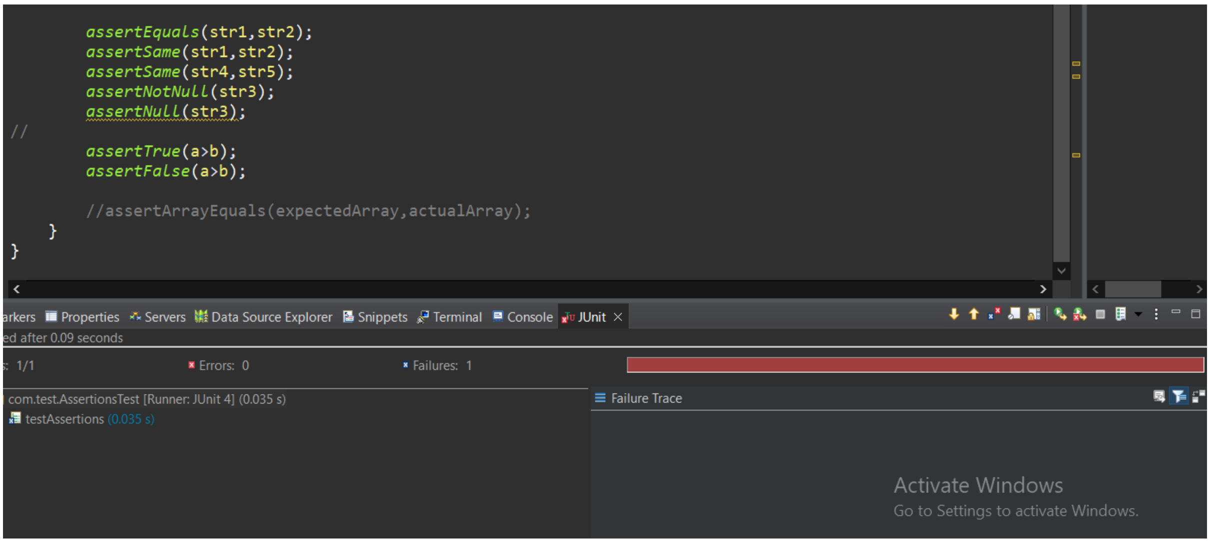
```java
package com.test;
import static org.junit.Assert.assertEquals;

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

```

```java
        assertEquals(24, cal.add(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()");
    }
}
```





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



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

1. Serial Garbage Collector:

- The Serial Garbage Collector, also known as the Serial Collector, is a simple garbage collector that operates using a single thread.
- It is suitable for applications with small to medium-sized heaps and low pause time requirements.
- The Serial Collector uses a "stop-the-world" approach, where all application threads are paused during garbage collection.
- It is generally used for single-threaded applications or applications with low latency requirements.
- While it can achieve compact heap layouts and low overhead, it may not be suitable for multi-threaded applications due to its stop-the-world nature.

2. Parallel Garbage Collector:

- The Parallel Garbage Collector, also known as the Throughput Collector, is designed for multi-threaded applications with medium to large heaps.
- It uses multiple threads to perform garbage collection in parallel, which can lead to improved throughput by utilizing multiple CPU cores.
- Like the Serial Collector, it also uses a "stop-the-world" approach but benefits from parallelism for faster garbage collection.
- The Parallel Collector is suitable for batch processing and applications where throughput is prioritized over pause times.
- While it can provide better throughput compared to the Serial Collector, it may introduce longer pause times due to its stop-the-world nature during garbage collection.

3. Concurrent Mark-Sweep (CMS) Garbage Collector:

- The CMS Garbage Collector, also known as the Concurrent Collector, is designed to minimize pause times for applications with large heaps and strict latency requirements.
- It uses multiple threads to perform most of the garbage collection work concurrently with the application threads, reducing pause times.
- The CMS Collector operates in multiple phases, including initial mark, concurrent marking, remark, and concurrent sweep.
- It is suitable for applications where low latency is critical, such as web servers and interactive applications.
- However, the CMS Collector may suffer from fragmentation issues and can lead to longer pause times during the remark and compaction phases.

4. G1 (Garbage-First) Garbage Collector:

- The G1 Garbage Collector is designed to provide both high throughput and low latency for large heaps (typically 4GB or larger).
- It divides the heap into regions and uses a combination of parallel, concurrent, and incremental approaches for garbage collection.
- The G1 Collector focuses on minimizing pause times by dynamically adjusting the size of the regions and prioritizing regions with the most garbage for collection.
- It is suitable for applications with large heaps and varying workload patterns, such as mixed workloads or applications with fluctuating memory demands.
- The G1 Collector aims to provide more predictable pause times compared to the CMS Collector while still achieving high throughput.

5. ZGC (Z Garbage Collector):

- The ZGC is a low-latency garbage collector introduced in JDK 11.
- It is designed to handle heaps ranging from a few megabytes to multiple terabytes with pause times not exceeding 10 milliseconds.
- The ZGC uses colored pointers and a load-barrier-based approach to enable concurrent marking, relocation, and compaction of objects.
- It is suitable for applications with stringent latency requirements, such as financial trading systems, gaming servers, and real-time analytics platforms.
- The ZGC aims to provide consistently low pause times, even with large heap sizes and high allocation rates.