## Day 22: Manisha 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.

### Explanation

1. MathOperations Class**:
   - Contains methods for basic mathematical operations: `add`, `subtract`, `multiply`, and `divide`.
   - The `divide` method throws an `IllegalArgumentException` if division by zero is attempted.

2. MathOperationsTest Class:
   - Contains JUnit test methods for each of the mathematical operations.
   - Each method uses assertions to check if the output of the operation matches the expected result.
   - The `testDivideByZero` method ensures that dividing by zero throws the correct exception.

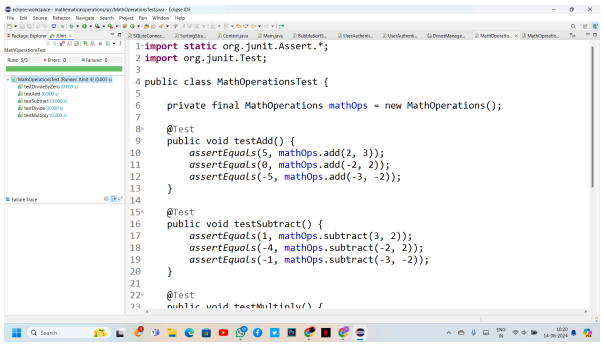
By running these tests, we verify that the `MathOperations` class behaves correctly for a variety of inputs, including edge cases like division by zero.

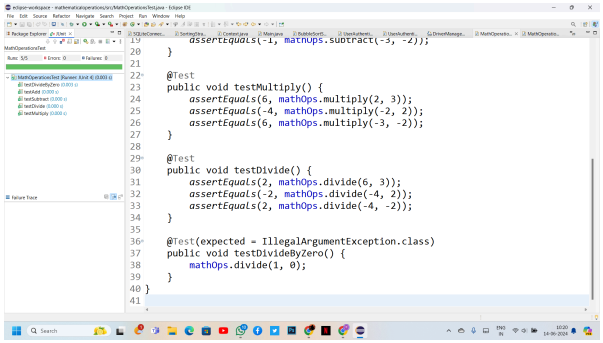
Running MathOperationsTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.012 sec

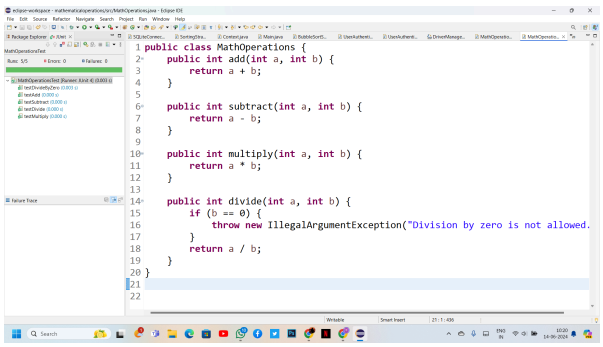
Results :

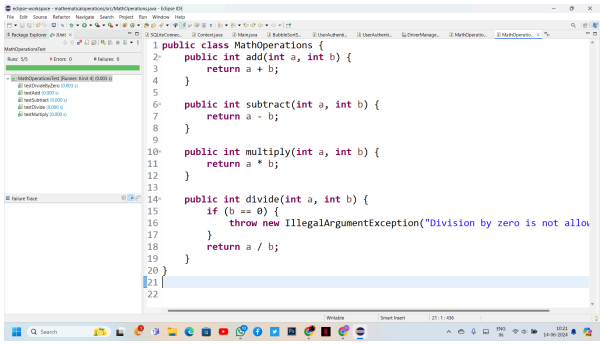
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0

### Here output as Follows:



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

### Explanation

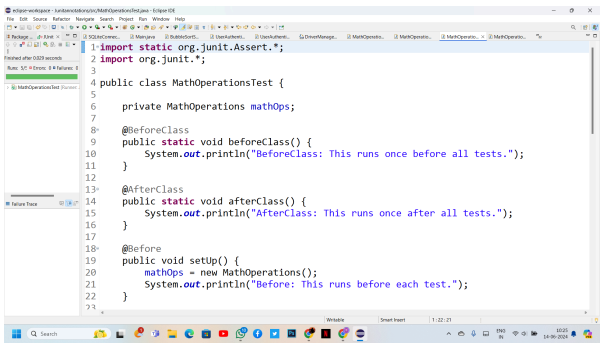
1. @BeforeClass: This method is executed once before any of the test methods in the class. It is typically used for initializing resources shared by all tests.
   - Example: `beforeClass()` prints "BeforeClass: This runs once before all tests."

2. @AfterClass: This method is executed once after all test methods in the class have been run. It is used for cleaning up shared resources.
   - Example: `afterClass()` prints "AfterClass: This runs once after all tests."

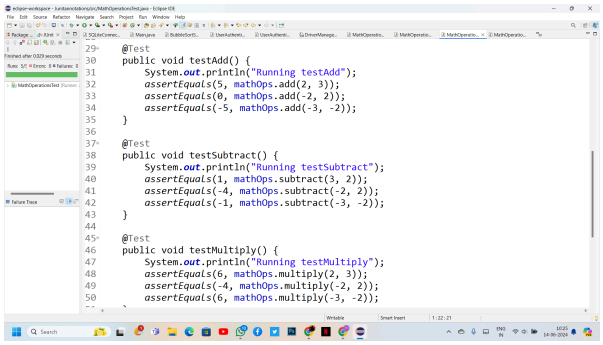
3. @Before: This method is executed before each test method. It is used to prepare the test environment, like initializing objects needed for the tests.
   - Example: `setUp()` initializes `mathOps` and prints "Before: This runs before each test."

4. @After: This method is executed after each test method. It is used to clean up the test environment, like releasing resources.
   - Example: `tearDown()` prints "After: This runs after each test."

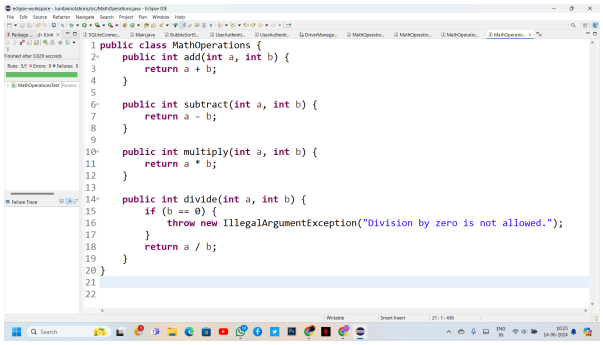
By structuring your tests with these annotations, you ensure that your setup and teardown processes are managed efficiently, making your tests cleaner and more maintainable.



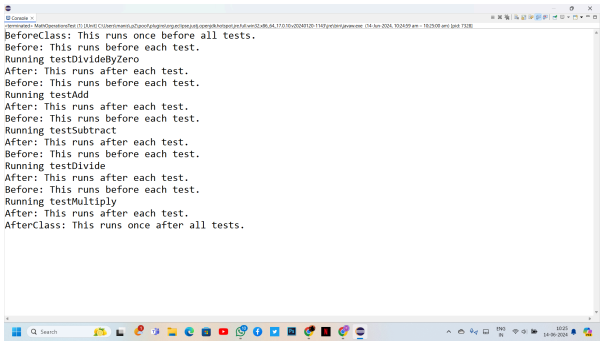
```
import static org.junit.Assert.*;
import org.junit.*;

public class MathOperationsTest {

    private MathOperations mathOps;

    @BeforeClass
    public static void beforeClass() {
        System.out.println("BeforeClass: This runs once before all tests.");
    }

    @AfterClass
    public static void afterClass() {
        System.out.println("AfterClass: This runs once after all tests.");
    }

    @Before
    public void setUp() {
        mathOps = new MathOperations();
        System.out.println("Before: This runs before each test.");
    }
```



```
    @Test
    public void testAdd() {
        System.out.println("Running testAdd");
        assertEquals(5, mathOps.add(2, 3));
        assertEquals(0, mathOps.add(-2, 2));
        assertEquals(-5, mathOps.add(-3, -2));
    }

    @Test
    public void testSubtract() {
        System.out.println("Running testSubtract");
        assertEquals(1, mathOps.subtract(3, 2));
        assertEquals(-4, mathOps.subtract(-2, 2));
        assertEquals(-1, mathOps.subtract(-3, -2));
    }

    @Test
    public void testMultiply() {
        System.out.println("Running testMultiply");
        assertEquals(6, mathOps.multiply(2, 3));
        assertEquals(-4, mathOps.multiply(-2, 2));
        assertEquals(6, mathOps.multiply(-3, -2));
```



```
public class MathOperations {
    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;
    }
}
```



```
BeforeClass: This runs once before all tests.
Before: This runs before each test.
Running testDivideByZero
After: This runs after each test.
Before: This runs before each test.
Running testAdd
After: This runs after each test.
Before: This runs before each test.
Running testSubtract
After: This runs after each test.
Before: This runs before each test.
Running testDivide
After: This runs after each test.
Before: This runs before each test.
Running testMultiply
After: This runs after each test.
AfterClass: This runs once after all tests.
```

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

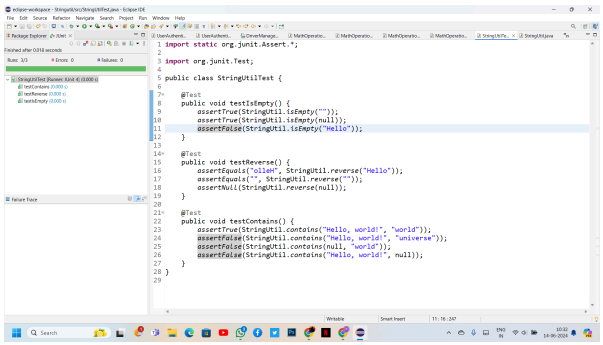
**Explanation**

**1. StringUtil Class:**

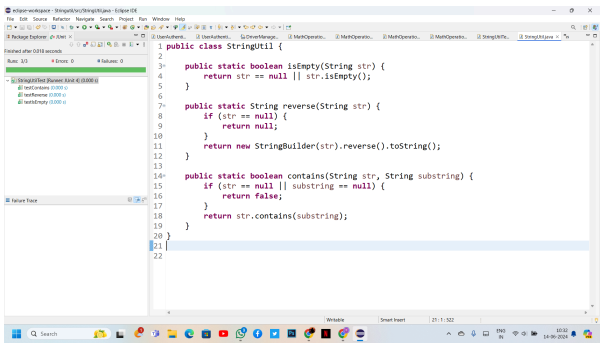
- isEmpty: Checks if a string is empty or null.
- reverse: Reverses a given string.
- contains: Checks if a string contains a given substring.

**2. StringUtilTest Class:**

- testIsEmpty:
  - Uses `assertTrue` to check if the method correctly identifies empty or null strings.
  - Uses `assertFalse` to ensure non-empty strings are correctly identified.
- testReverse:
  - Uses `assertEquals` to verify the reverse of a string.
  - Ensures that reversing an empty string returns an empty string.
  - Uses `assertNull` to check that reversing a null string returns null.
- testContains:
  - Uses `assertTrue` to check if a string contains a specified substring.
  - Uses `assertFalse` to verify the absence of a substring.
  - Checks the method's behavior with null inputs.



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

import org.junit.Test;

public class StringUtilTest {

    @Test
    public void testIsEmpty() {
        assertTrue(StringUtil.isEmpty(""));
        assertTrue(StringUtil.isEmpty(null));
        assertFalse(StringUtil.isEmpty("Hello"));
    }

    @Test
    public void testReverse() {
        assertEquals("olleH", StringUtil.reverse("Hello"));
        assertEquals("", StringUtil.reverse(""));
        assertNull(StringUtil.reverse(null));
    }

    @Test
    public void testContains() {
        assertTrue(StringUtil.contains("Hello, world!", "world"));
        assertFalse(StringUtil.contains("Hello, world!", "universe"));
        assertFalse(StringUtil.contains(null, "world"));
        assertFalse(StringUtil.contains("Hello, world!", null));
    }
}
```



```
public class StringUtil {

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

    public static String reverse(String str) {
        if (str == null) {
            return null;
        }
        return new StringBuilder(str).reverse().toString();
    }

    public static boolean contains(String str, String substring) {
        if (str == null || substring == null) {
            return false;
        }
        return str.contains(substring);
    }
}
```

---

**T E S T S**

---

**Running StringUtilTest**
**Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.005 sec**

**Results :**

**Tests run: 3, Failures: 0, Errors: 0, Skipped: 0**

## 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 several garbage collection (GC) algorithms, each designed to handle memory management efficiently under different conditions. Here, we'll compare the following GC algorithms: Serial, Parallel, Concurrent Mark-Sweep (CMS), Garbage-First (G1), and Z Garbage Collector (ZGC).**

### 1. Serial Garbage Collector

**- Description:**
**- The Serial GC uses a single thread to handle all garbage collection events.**
**- It's best suited for single-threaded environments and small applications.**

**- Advantages:**
**- Simple and efficient for applications with small heaps.**
**- Minimal overhead since it uses a single thread.**

**- Disadvantages:**
**- Stops all application threads during garbage collection (Stop-The-World events).**
**- Not suitable for large, multi-threaded applications.**

**- Usage:**
**- Best for small, single-threaded applications.**
**- Enable with JVM option: `-XX:+UseSerialGC`.**

### 2. Parallel Garbage Collector

**- Description:**
**- The Parallel GC uses multiple threads for garbage collection, aiming to reduce pause times by performing collections in parallel.**

**- Advantages:**
**- Improves throughput by utilizing multiple CPU cores.**
**- Suitable for applications with large heaps and high-throughput requirements.**

**- Disadvantages:**
**- Still experiences Stop-The-World events, which can cause noticeable pauses.**
**- Can be less efficient for latency-sensitive applications.**

**- Usage:**
**- Best for multi-threaded applications with large heaps.**
**- Enable with JVM option: `-XX:+UseParallelGC`.**

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

- Description:
  - The CMS GC aims to minimize pause times by performing most of its work concurrently with application threads.

- Advantages:
  - Reduces pause times by performing concurrent marking and sweeping phases.
  - Suitable for applications requiring low latency.

- Disadvantages:
  - More CPU-intensive than other collectors.
  - Can suffer from fragmentation, leading to longer GC times.

- Usage:
  - Best for applications requiring low-latency GC.
  - Enable with JVM option: `-XX:+UseConcMarkSweepGC`.

### 4. Garbage-First (G1) Garbage Collector

Description:
- The G1 GC is designed for large heaps and aims to provide predictable pause times while maintaining good throughput.

- Advantages:
  - Divides the heap into regions and performs garbage collection on a per-region basis.
  - Provides more predictable pause times by prioritizing regions with the most garbage.

- Disadvantages:
  - More complex and can have higher overhead compared to simpler collectors.
  - May not achieve as low latency as CMS in some cases.

- Usage:
  - Best for large applications needing a balance between throughput and low pause times.
  - Enable with JVM option: `-XX:+UseG1GC`.

### 5. Z Garbage Collector (ZGC)

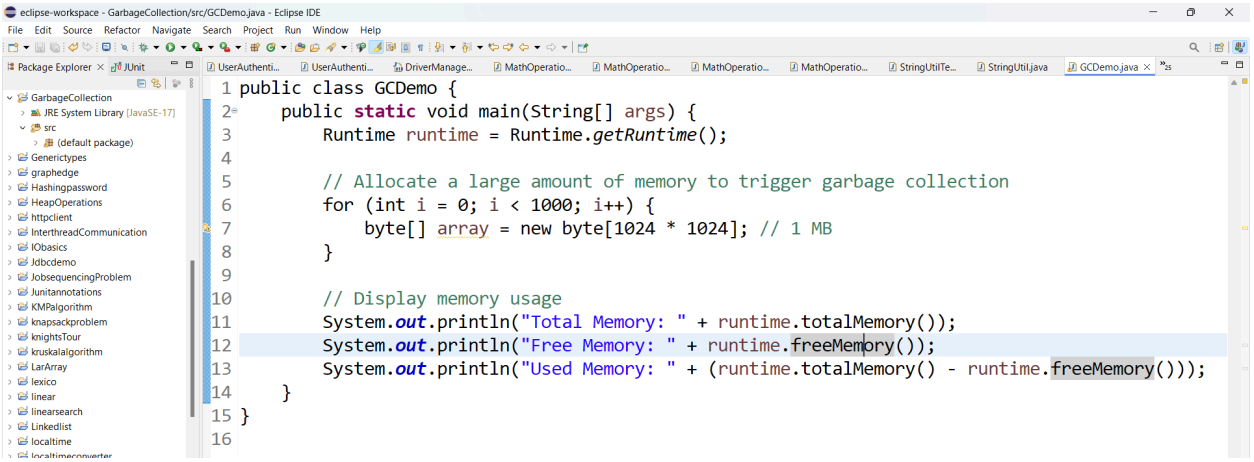
Description:
- ZGC is designed for very large heaps (multi-terabyte) and aims to provide extremely low pause times.

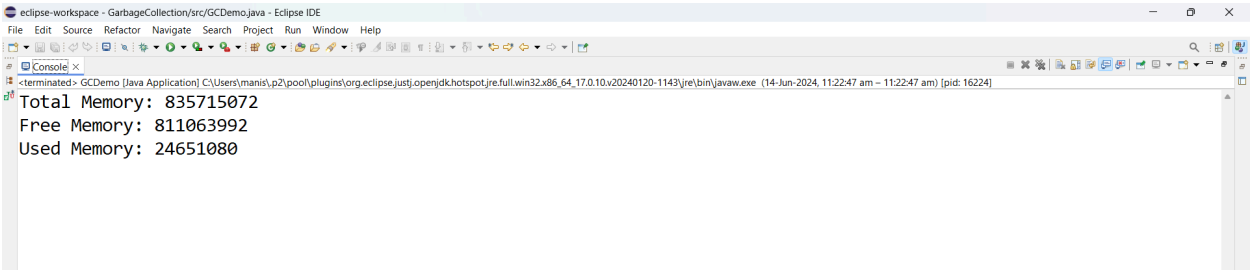
-Advantages:
  - Performs most of its work concurrently, keeping pause times under 10ms.
  - Scales efficiently with large heaps.

- Disadvantages:
  - Requires a more recent version of the JVM.
  - Higher memory usage due to colored pointers and metadata.

- Usage:
  - Best for applications with large heaps requiring very low-latency GC.
  - Enable with JVM option: `-XX:+UseZGC`.

**Here's Output of the Following** 👍




To run this program with different garbage collectors, you would use the following JVM options:

1. Serial GC: `java -XX:+UseSerialGC GCDemo`
2. Parallel GC: `java -XX:+UseParallelGC GCDemo`
3. CMS GC: `java -XX:+UseConcMarkSweepGC GCDemo`
4. G1 GC: `java -XX:+UseG1GC GCDemo`
5. ZGC: `java -XX:+UseZGC GCDemo`

## Conclusion

Each garbage collector in Java has its strengths and weaknesses, making them suitable for different types of applications:

- *Serial GC: Best for small, single-threaded applications.*
- *Parallel GC: Suitable for large, multi-threaded applications needing high throughput.*
- *CMS GC: Ideal for applications requiring low-latency GC.*
- *G1 GC: Balances throughput and low pause times for large applications.*
- *ZGC: Provides extremely low pause times for applications with very large heaps.*