Day 22

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.

```
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) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Division by zero");
        }
        return a / b;
    }
}
```

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

```
public class MathOperationsTest {

    private final MathOperations mathOperations = new MathOperations();

    @Test
    public void testAdd() {
        assertEquals(5, mathOperations.add(2, 3));
        assertEquals(-1, mathOperations.add(-2, 1));
        assertEquals(0, mathOperations.add(0, 0));
    }

    @Test
    public void testSubtract() {
        assertEquals(1, mathOperations.subtract(3, 2));
        assertEquals(-3, mathOperations.subtract(-2, 1));
        assertEquals(0, mathOperations.subtract(0, 0));
    }

    @Test
    public void testMultiply() {
        assertEquals(6, mathOperations.multiply(2, 3));
        assertEquals(-2, mathOperations.multiply(-2, 1));
        assertEquals(0, mathOperations.multiply(0, 5));
    }

    @Test
    public void testDivide() {
        assertEquals(2, mathOperations.divide(6, 3));
        assertEquals(-2, mathOperations.divide(-4, 2));
```

```
        Exception exception = assertThrows(ArithmeticException.class, () -> {
            mathOperations.divide(1, 0);
        });
        assertEquals("Division by zero", exception.getMessage());
    }
}
```

Explanation

Imports:

- import static org.junit.jupiter.api.Assertions.*; - Import static methods for assertions.
- import org.junit.jupiter.api.Test; - Import the @Test annotation.

Test Class:

- The test class MathOperationsTest contains individual test methods for each operation in the MathOperations class

Test Methods:

- **testAdd**(): Tests the add method with various inputs.
- **testSubtract**(): Tests the subtract method with various inputs.
- **testMultiply**(): Tests the multiply method with various inputs.
- **testDivide**(): Tests the divide method with valid inputs and checks for division by zero.

Assertions:

- **assertEquals(expected, actual, message):** Checks if the actual value matches the expected value.
- assertThrows(expectedType, executable): Checks if the provided executable throws an exception of the expectedType.

Division by Zero:

The test for division by zero checks if the divide method throws an ArithmeticException with the message "Division by zero".

Output:

Running MathOperationsTest

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.021 s - in MathOperationsTest

Results:

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0

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

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
public class MathOperationsTest {
   private MathOperations mathOperations;
   @BeforeClass
   public static void setUpBeforeClass() {
      System.out.println("Before all tests");
   }

   @AfterClass
   public static void tearDownAfterClass()  {
      System.out.println("After all tests");
   }

   @Before
   public void setUp()  {
      mathOperations = new MathOperations();
      System.out.println("Before each test");
   }
```

```
@After
public void tearDown() throws Exception {
   System.out.println("After each test");
}

@Test
public void testAdd() {
   assertEquals(5, mathOperations.add(2, 3));
   assertEquals(-1, mathOperations.add(-2, 1));
   assertEquals(0, mathOperations.add(0, 0));
}

@Test
public void testSubtract() {
   assertEquals(1, mathOperations.subtract(3, 2));
   assertEquals(-3, mathOperations.subtract(-2, 1));
   assertEquals(0, mathOperations.subtract(0, 0));
}

@Test
public void testMultiply() {
   assertEquals(6, mathOperations.multiply(2, 3));
   assertEquals(-2, mathOperations.multiply(-2, 1));
   assertEquals(0, mathOperations.multiply(0, 5));
}

@Test
public void testDivide() {
   assertEquals(2, mathOperations.divide(6, 3));
```

```
        assertEquals(-2, mathOperations.divide(-4, 2));


        Exception exception = assertThrows(ArithmeticException.class, () -> {
            mathOperations.divide(1, 0);
        });
        assertEquals("Division by zero", exception.getMessage());
    }
}
```

Explanation

Imports:

- import static org.junit.jupiter.api.Assertions.*; - Import static methods for assertions.
- import org.junit.jupiter.api.*; - Import JUnit annotations.

Test Class:

The test class MathOperationsTest now includes setup and teardown methods using JUnit lifecycle annotations.

Annotations:

- @**BeforeClass**: Static method annotated with @BeforeClass to run before all test methods in the class. Suitable for one-time setup.
- @**AfterClass**: Static method annotated with @AfterClass to run after all test methods in the class. Suitable for one-time teardown.
- @**Before**: Method annotated with @Before to run before each test method. Suitable for setup needed before each test.
- @**After**: Method annotated with @After to run after each test method. Suitable for cleanup needed after each test.

Setup and Teardown Methods:

- **setUpBeforeClass**(): Prints "Before all tests". Runs once before all test cases.
- **tearDownAfterClass**(): Prints "After all tests". Runs once after all test cases.
- **setUp**(): Instantiates a new MathOperations object and prints "Before each test". Runs before each test case.
- **tearDown**(): Prints "After each test". Runs after each test case.

Test Methods:

testAdd(), testSubtract(), testMultiply(), testDivide(): Same as before but now benefit from the setup and teardown provided by @BeforeEach and @AfterEach.

Output:

Before all tests

Before each test

After each test

Before each test

After each test

Before each test

After each test

Before each test

After each test

After all tests

Running MathOperationsTest

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.021 s - in MathOperationsTest

Results:

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0

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

```
public class StringUtil {

  public static boolean isNullOrEmpty(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 isPalindrome(String str) {
        if (str == null) {
            return false;
        }
        String reversed = reverse(str);
        return str.equals(reversed);
    }
}
```

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

public class StringUtilTest {

    @Test
    public void testIsNullOrEmpty() {
        assertTrue(StringUtil.isNullOrEmpty(null));
        assertTrue(StringUtil.isNullOrEmpty(""));
        assertFalse(StringUtil.isNullOrEmpty("abc"));
    }
```

```
    @Test
    public void testReverse() {
        assertEquals(null, StringUtil.reverse(null));
        assertEquals("", StringUtil.reverse(""));
        assertEquals("cba", StringUtil.reverse("abc"));
    }

    @Test
    public void testIsPalindrome() {
        assertFalse(StringUtil.isPalindrome(null));
        assertTrue(StringUtil.isPalindrome(""));
        assertTrue(StringUtil.isPalindrome("madam"))l
        assertFalse(StringUtil.isPalindrome("hello"));
    }
}
```

Explanation

Imports:

- import static org.junit.jupiter.api.Assertions.*; - Import static methods for assertions.
- import org.junit.jupiter.api.Test; - Import the @Test annotation.

Test Class:

- The test class StringUtilTest contains individual test methods for each method in the StringUtil class.

Test Methods:

testIsNullOrEmpty():

- Tests the isNullOrEmpty method with null, empty string, and a non-empty string.
- Uses assertTrue and assertFalse for boolean assertions.

testReverse():

- Tests the reverse method with null, empty string, and a regular string.
- Uses assertEquals to check if the reversed string matches the expected value.

testIsPalindrome():

- Tests the isPalindrome method with null, empty string, palindrome string, and non-palindrome string.
- Uses assertTrue and assertFalse for boolean assertions**.**

Output:

Running StringUtilTest

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.015 s - in StringUtilTest

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.

1. Serial Garbage Collector

Description:

- The simplest garbage collector.
- Uses a single thread for both minor and major garbage collections.
- Best suited for single-threaded applications or applications with small heaps.

Advantages:

- Simplicity and low overhead.
- Predictable pauses, making it suitable for applications where long pause times are acceptable.

Disadvantages:

- Not scalable due to single-threaded nature.
- Can lead to long pause times for large heaps or multi-threaded applications.

Use Cases:

- Applications with small heaps.
- Single-threaded applications or applications with less emphasis on throughput.

JVM Option:

-XX:+UseSerialGC

2. Parallel Garbage Collector

Description:

- Also known as the Throughput Collector.
- Uses multiple threads for minor garbage collection and a single thread for major garbage collection.

Advantages:

- Improves throughput by using multiple threads for minor collections.
- Reduces the duration of minor GC pauses.

Disadvantages:

- Full GCs are still single-threaded, which can lead to long pause times.
- Not ideal for applications requiring low latency.

Use Cases:

- Applications that prioritize high throughput over low latency.
- Multi-threaded applications with significant processing.

JVM Option:

-XX:+UseParallelGC

3. Concurrent Mark-Sweep (CMS) Garbage Collector

Description:

- Focuses on low pause times.
- Uses multiple threads for both minor and major garbage collections.
- The major GC is performed concurrently with application threads.

Advantages:

- Reduces pause times by performing most of the GC work concurrently.
- Suitable for applications requiring low latency.

Disadvantages:

- Higher CPU usage due to concurrent operations.

- Can suffer from fragmentation and "Concurrent Mode Failure" which triggers a fallback to a stop-the-world full GC.

Use Cases;

- Applications requiring low latency and minimal pause times.
- Web servers and interactive applications.

JVM Option:

-XX:+UseConcMarkSweepGC

4. Garbage First (G1) Garbage Collector

Description:

- Designed to replace CMS.
- Divides the heap into regions and performs GC in a region-based manner.
- Performs both concurrent and parallel garbage collection.

Advantages:

- Balances low pause times with high throughput.
- More predictable pause times by controlling the number of regions collected within a given time budget.
- Handles large heaps more efficiently than CMS.

Disadvantages:

Can be more complex to tune compared to simpler collectors.

Use Cases:

- Applications with large heaps requiring a balance of throughput and low latency.
- Applications that previously used CMS but need better handling of large heaps.

JVM Option:

-XX:+UseG1GC

5. Z Garbage Collector (ZGC)

Description:

- Aimed at ultra-low pause times.

- Uses a load barrier and colored pointers to achieve concurrent compaction and marking.
- Designed to handle multi-terabyte heaps.

Advantages:

- Extremely low pause times (typically in the range of sub-millisecond).
- Can handle very large heaps (multi-terabyte) efficiently.
- Scales well with the number of cores.

Disadvantage

- Higher memory usage due to load barriers and colored pointers.
- Requires a 64-bit system.

Use Cases:

- Large-scale applications requiring minimal pause times.
- Real-time systems and latency-sensitive applications.

JVM Option:

-XX:+UseZGC