# 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.

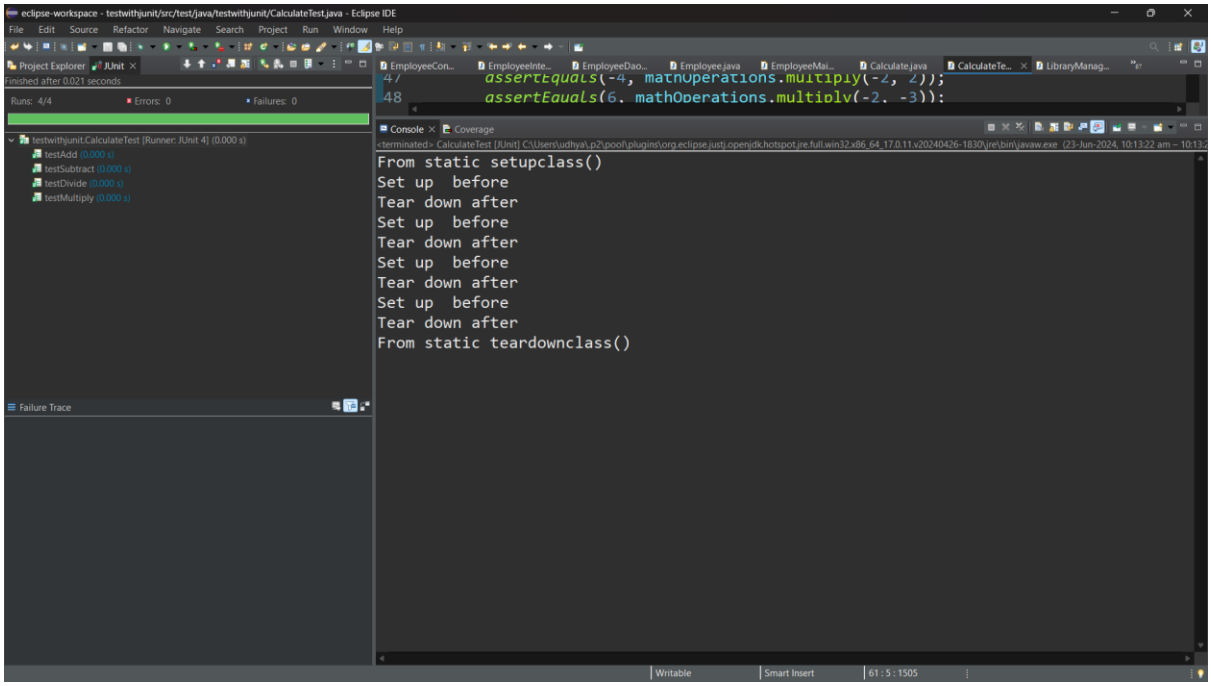
```
package testwithjunit;


public class Calculate {


        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;
  }
```

```
}

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.Test;



public class CalculateTest {
	private final Calculate mathOperations = new Calculate();
	Calculate cal;
	@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() {
   assertEquals(5, mathOperations.add(2, 3));
   assertEquals(0, mathOperations.add(-2, 2));
   assertEquals(-5, mathOperations.add(-2, -3));
}

@Test
public void testSubtract() {
   assertEquals(1, mathOperations.subtract(3, 2));
   assertEquals(-4, mathOperations.subtract(-2, 2));
   assertEquals(1, mathOperations.subtract(-2, -3));
}

@Test
public void testMultiply() {
   assertEquals(6, mathOperations.multiply(2, 3));
   assertEquals(-4, mathOperations.multiply(-2, 2));
   assertEquals(6, mathOperations.multiply(-2, -3));
   assertEquals(0, mathOperations.multiply(2, 0));
}
```

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



    }



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

## OUTPUT:




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

```
package testwithjunit;


public class Calculate {


        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;
    }
}
```

```
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.Test;



public class CalculateTest {
        private final Calculate mathOperations = new Calculate();
        Calculate cal;
```

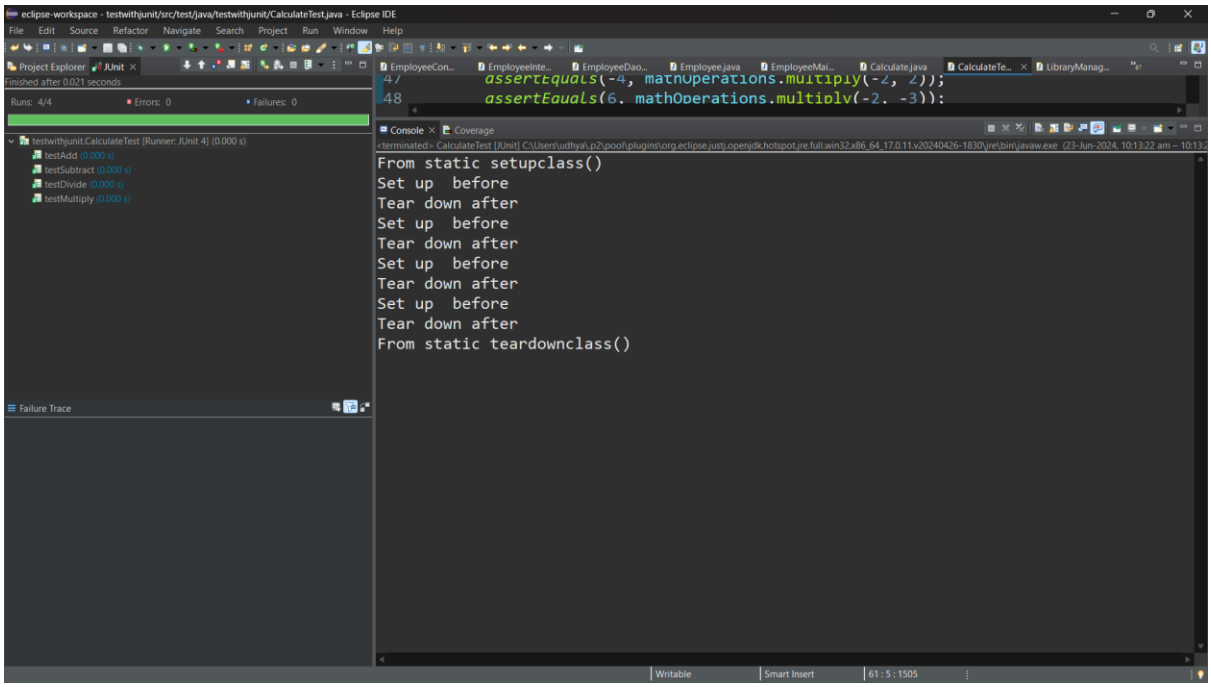
```
	@BeforeClass
	public static void setUpClass() {
		System.out.println("From static setupclass()");
	}

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

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

@Test
public void testSubtract() {
   assertEquals(1, mathOperations.subtract(3, 2));
   assertEquals(-4, mathOperations.subtract(-2, 2));
   assertEquals(1, mathOperations.subtract(-2, -3));
}
```

```
@Test
public void testMultiply() {
   assertEquals(6, mathOperations.multiply(2, 3));
   assertEquals(-4, mathOperations.multiply(-2, 2));
   assertEquals(6, mathOperations.multiply(-2, -3));
   assertEquals(0, mathOperations.multiply(2, 0));
}

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



}


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

```
        System.out.println("From static teardownclass()");
    }
}
```

**OUTPUT:**




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

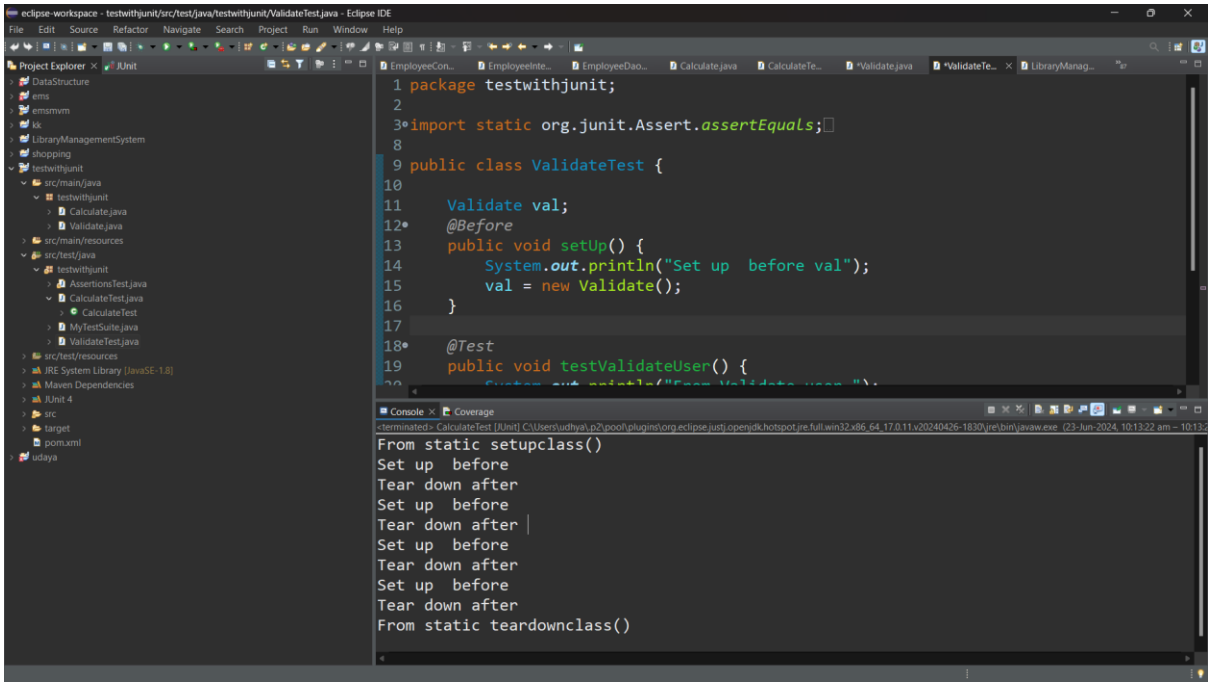
```
package testwithjunit;


public class Validate {


public String validateUser(String un, String pwd) {
```

```
		String user = null;

		if(un.equals("uday") && pwd.equals("pass")) {
			user = "Welcome " +un;
		}else {
			user = "Welcome Guest";
		}
		return user;
	}

	public static void main(String[] args) {
		Validate v= new Validate();
		System.out.println(v.validateUser("uday", "pass"));

	}

}
```

```
package testwithjunit;

import static org.junit.Assert.assertEquals;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
```

```
public class ValidateTest {

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

	@Test
	public void testValidateUser() {
		System.out.println("From Validate user ");
		assertEquals("Welcome Deeps",val.validateUser("uday", "pass") );
	}
	@After
	public void tearDown() {
		System.out.println("Tear down after val");
		val=null;
	}
}
```

**OUTPUT:**




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

1. Serial Garbage Collector

Algorithm: Uses a single thread for garbage collection.

Generations: Young and Old (Tenured).

Characteristics:

Stop-the-world pauses: The application is paused during GC.

Efficiency: Simple and efficient for single-threaded applications.

Overhead: Minimal overhead due to lack of multi-threading.

Use Cases: Suitable for small applications or single-threaded environments where pause times are not critical.

Pros: Low overhead, simple implementation.

Cons: Long pause times in multi-threaded applications, not scalable.

2. Parallel Garbage Collector

Algorithm: Uses multiple threads for garbage collection.

Generations: Young and Old (Tenured).

Characteristics:

Stop-the-world pauses: The application is paused during GC, but multiple threads work simultaneously to reduce pause time.

Throughput-focused: Aims to maximize application throughput by reducing GC overhead.

Use Cases: Suitable for applications where high throughput is essential and pause times are acceptable.

Pros: Better performance in multi-threaded environments, reduced pause times compared to Serial GC.

Cons: Still experiences significant pauses, can be inefficient for applications requiring low latency.

3. Concurrent Mark-Sweep (CMS) Garbage Collector

Algorithm: Uses multiple threads and works concurrently with the application.

Generations: Young and Old (Tenured).

Characteristics:

Concurrent phases: Includes concurrent marking and sweeping phases to minimize pause times.

Stop-the-world phases: Short initial mark and remark phases still pause the application.

Low-latency: Designed to reduce pause times and provide low-latency GC.

Use Cases: Suitable for applications requiring low pause times and low latency, such as interactive applications.

Pros: Shorter pause times compared to Serial and Parallel GC.

Cons: Higher CPU usage due to concurrent threads, potential for "floating garbage" (objects that become unreachable during the concurrent phase but are not collected until the next cycle), fragmentation issues.

4. Garbage-First (G1) Garbage Collector

Algorithm: Divides the heap into regions and uses a concurrent marking cycle to identify live data.

Generations: Does not explicitly divide into young and old generations but has logical regions.

Characteristics:

Region-based: Collects regions with the most garbage first.

Concurrent phases: Includes concurrent marking and compaction phases.

Predictable pauses: Provides more predictable pause times by focusing on regions with the most garbage.

Use Cases: Suitable for large heap applications requiring predictable pause times.

Pros: Predictable and configurable pause times, better handling of large heaps.

Cons: More complex tuning required compared to other collectors, overhead of maintaining regions.

5. Z Garbage Collector (ZGC)

Algorithm: Highly concurrent, region-based collector designed for scalability and low-latency.

Generations: Does not explicitly divide into young and old generations but uses regions.

Characteristics:

Concurrent phases: Almost all phases are concurrent with minimal stop-the-world pauses.

Low-latency: Aims for sub-millisecond pause times.

Scalability: Designed to handle large heaps (multi-terabyte).

Use Cases: Suitable for applications requiring ultra-low latency and scalability, such as real-time systems and large-scale applications.

Pros: Extremely low pause times, scalable to very large heaps, minimal impact on application throughput.

Cons: Relatively new and evolving, can be more complex to tune, higher CPU and memory overhead due to concurrent nature.