**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.**

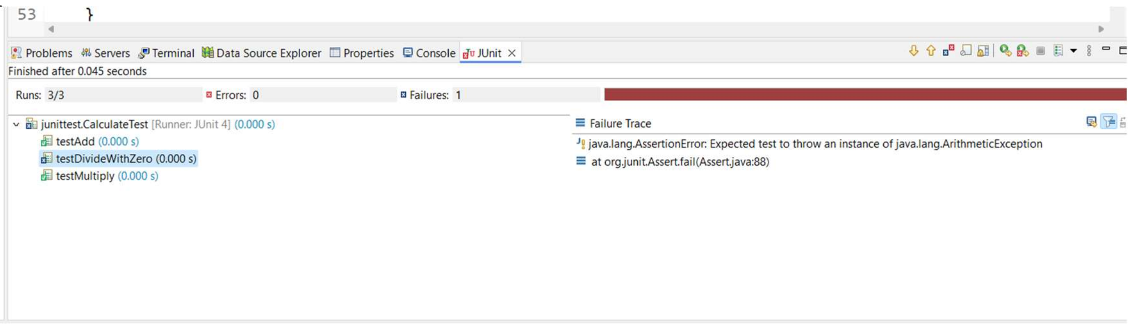
```
package junittest;


public class Calculate {

	public int add(int... number) {
		int result =0;

		for(int i :number) {
			result = result+i;
		}

		return result;
	}

	public int multiply(int... number) {

		int result =1;

		for(int i :number) {
			result = result*i;
		}

		return result;
	}

	public int divide(int a, int b) {
		int result=0;
		result = a/b;

		return result;
	}

	public static void main(String[] args) {
		Calculate c = new Calculate();
		System.out.println(c.add(1,2));
		System.out.println(c.add(1,2,3));
		System.out.println(c.add(1,2,3,4));
		//System.out.println(c.divide(2, 0));
		System.out.println(c.multiply(2,2));
	}


}
```

CalculateTest.java

```
package junittest;

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.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

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));
	}
```

```
        @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()");
        }
}
```

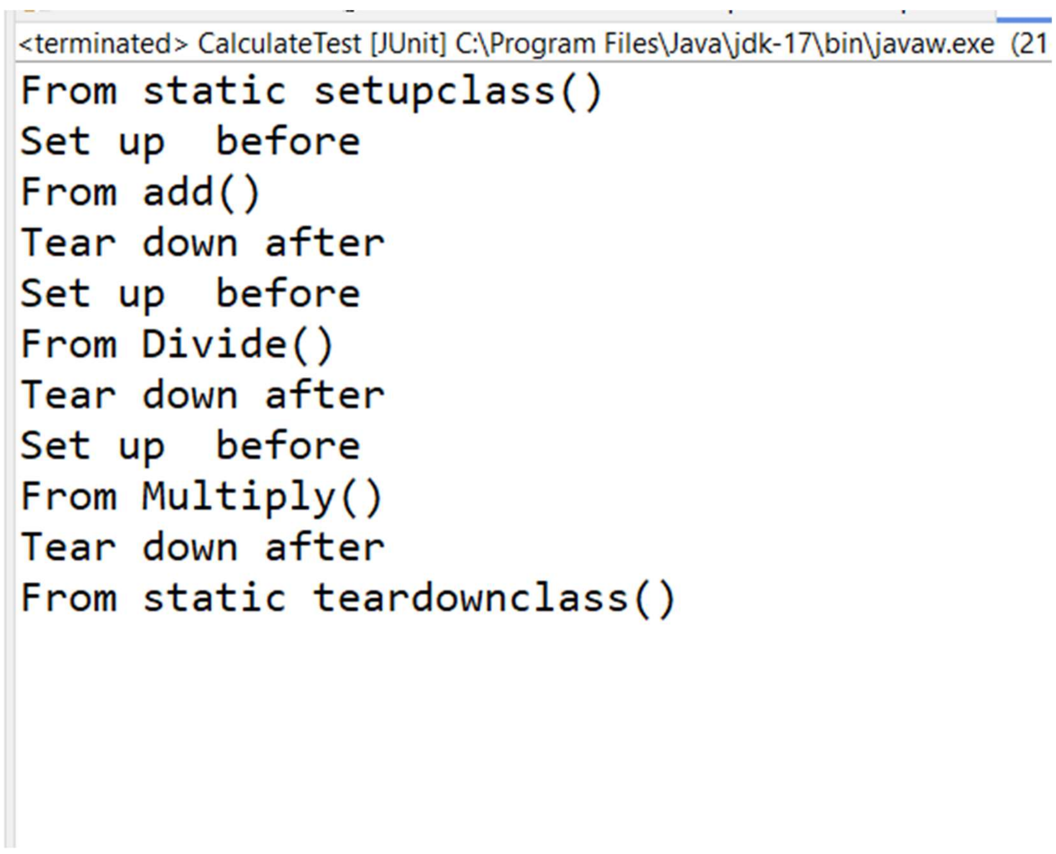
Output:



```
53      }
```

Problems | Servers | Terminal | Data Source Explorer | Properties | Console | JUnit

```
Finished after 0.045 seconds

Runs: 3/3        Errors: 0        Failures: 1

junittest.CalculateTest [Runner: JUnit 4] (0.000 s)
    testAdd (0.000 s)
    testDivideWithZero (0.000 s)
    testMultiply (0.000 s)

Failure Trace
java.lang.AssertionError: Expected test to throw an instance of java.lang.ArithmeticException
at org.junit.Assert.fail(Assert.java:88)
```



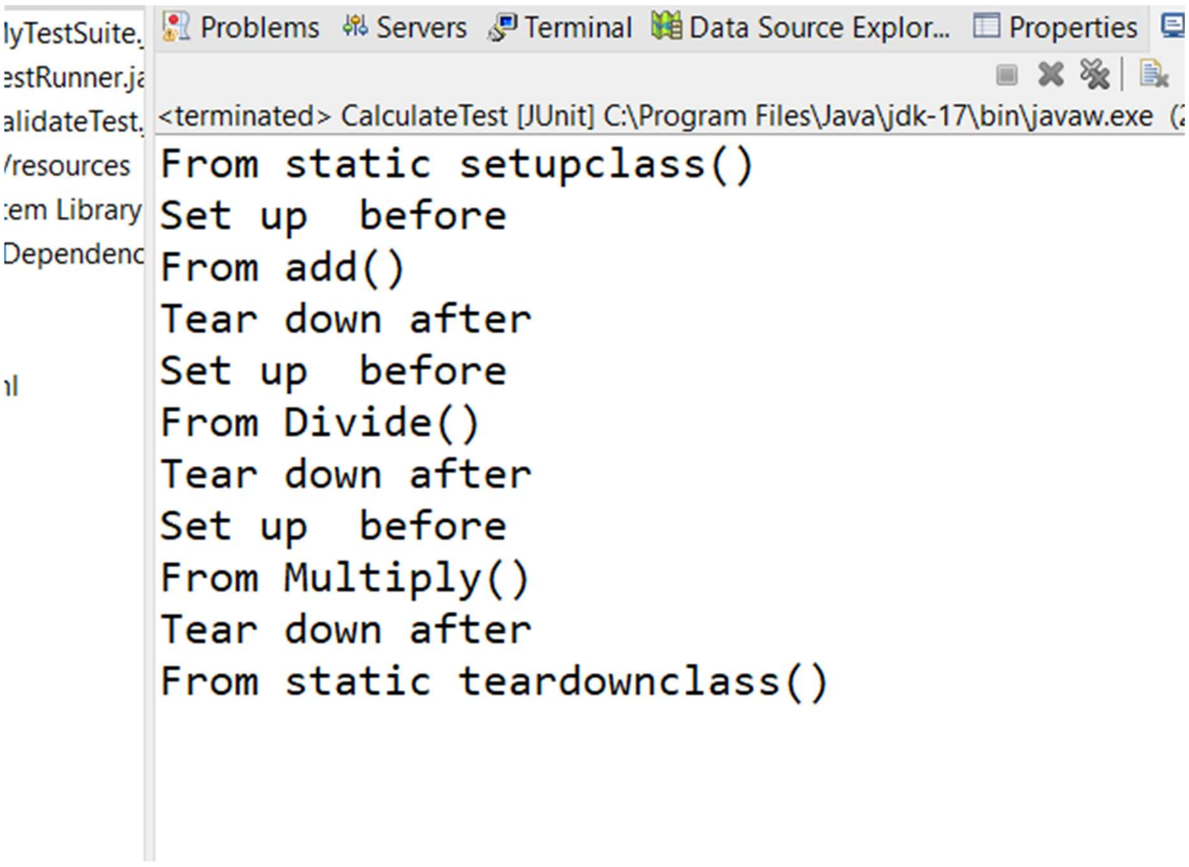
```
<terminated> CalculateTest [JUnit] C:\Program Files\Java\jdk-17\bin\javaw.exe  (21
From static setupclass()
Set up  before
From add()
Tear down after
Set up  before
From Divide()
Tear down after
Set up  before
From Multiply()
Tear down after
From static teardownclass()
```

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

```
package junittest;

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.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

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));
	}

	@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(15, 5);
	}


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

Output:



```
<terminated> CalculateTest [JUnit] C:\Program Files\Java\jdk-17\bin\javaw.exe
From static setupclass()
Set up  before
From add()
Tear down after
Set up  before
From Divide()
Tear down after
Set up  before
From Multiply()
Tear down after
From static teardownclass()
```

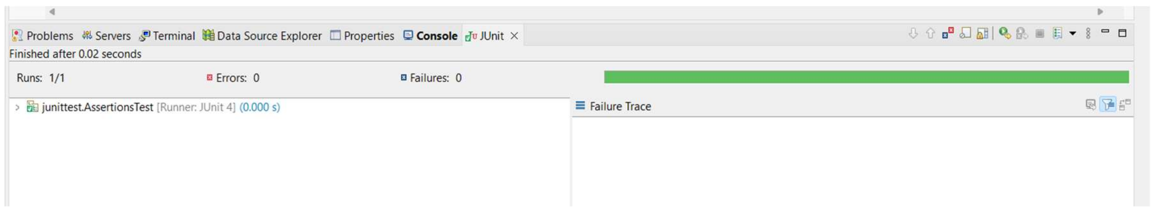
**Task 3: Create test cases with assertEquals, assertTrue, and assertFalse to validate the correctness of a custom String utility class.**

AssertionsTest

```
package junittest;

import static org.junit.Assert.assertArrayEquals;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNull;
import static org.junit.Assert.assertSame;
import static org.junit.Assert.assertTrue;

import org.junit.Test;

public class AssertionsTest {

	@Test
```

```
        public void testAssertions()
        {
                String str1=new String("bunny");
                String str2=new String("bunny");
                String str3=null;
                String str4="bunny";
                String str5="bunny";

                int a=5;
                int b=6;

                String[] expectedArray= {"one","two","three"};
                String[] actualArray= {"one","two","three"};


                        assertEquals(str1, str2);
                        assertSame(expectedArray, actualArray);
                        assertNull(str3);
                        assertSame(str4, str5);
                        assertNotNull(str3);

                        assertTrue(a>b);
                        assertFalse(a>b);
                        assertArrayEquals(expectedArray, actualArray);
        }
}
```

Output:



```
Finished after 0.02 seconds
Runs: 1/1          Errors: 0          Failures: 0
> junittest.AssertionsTest [Runner: JUnit 4] (0.000 s)
```

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

Java uses several garbage collection (GC) algorithms to manage memory efficiently. Here's a detailed comparison of the most commonly used ones: Serial, Parallel, Concurrent Mark-Sweep (CMS), Garbage-First (G1), and Z Garbage Collector (ZGC).

### Serial Garbage Collector:

- Single-threaded.
- Stop-the-world pauses during garbage collection.

- Suitable for small data sets and single-threaded environments.
- Prioritizes simplicity and smaller memory footprints.
- Pros: Simple, low overhead for small apps.
- Cons: Unsuitable for multi-threaded apps, long pauses.

**Parallel Garbage Collector:**

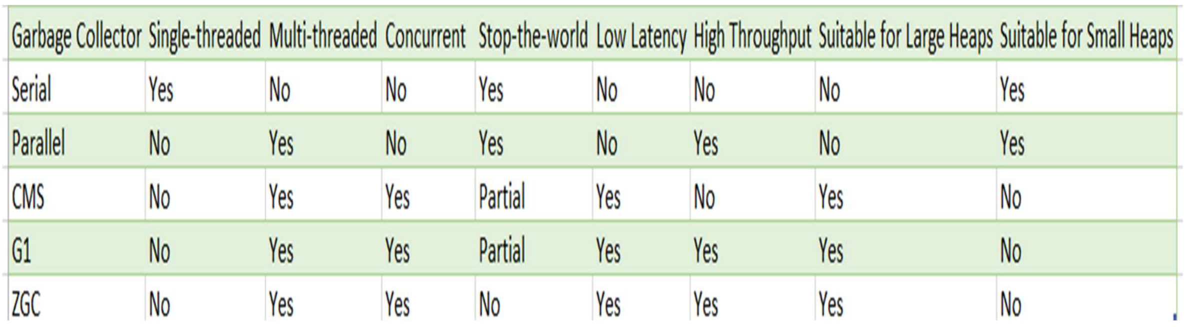
- Multi-threaded.
- Stop-the-world pauses.
- Suitable for medium to large data sets and multi-threaded environments.
- Prioritizes high throughput.
- Pros: Better performance in multi-threaded environments.
- Cons: Still has stop-the-world pauses, not ideal for low-latency apps.
- Concurrent Mark-Sweep (CMS) Garbage Collector:
- Concurrent (minimizes pause times).
- Suitable for low-latency apps.
- Pros: Reduced pause times, suitable for high response time requirements.
- Cons: Fragmentation, complexity, higher CPU usage.

**Garbage-First (G1) Garbage Collector:**

- Region-based, concurrent, and parallel.
- Predictable pause times.
- Suitable for large heaps.
- Balances latency and throughput.
- Pros: Predictable pause times, efficient for both young and old generation.
- Cons: Complexity, additional overhead.

**Z Garbage Collector (ZGC):**

- Region-based and concurrent.
- Scalable for large heaps.
- Very low pause times (minimal latency impact).
- Pros: Extremely low pause times, efficient handling of large heaps.
- Cons: Higher CPU and memory overhead, relatively new.



| Garbage Collector | Single-threaded | Multi-threaded | Concurrent | Stop-the-world | Low Latency | High Throughput | Suitable for Large Heaps | Suitable for Small Heaps |
|---|---|---|---|---|---|---|---|---|
| Serial | Yes | No | No | Yes | No | No | No | Yes |
| Parallel | No | Yes | No | Yes | No | Yes | No | Yes |
| CMS | No | Yes | Yes | Partial | Yes | No | Yes | No |
| G1 | No | Yes | Yes | Partial | Yes | Yes | Yes | No |
| ZGC | No | Yes | Yes | No | Yes | Yes | Yes | No |

Each garbage collector has its own strengths and weaknesses, making them suitable for different types of applications and requirements. Selecting the right GC depends on the specific needs of the application, such as latency requirements, heap size, and throughput goals.