**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.

**Program:**

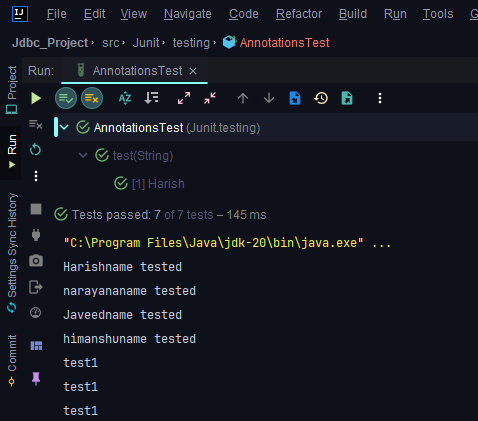
**Step 1:** Create a new Class called Calculator

*package* Junit.calsi;  
  
*public class* Calculator {  
  
 *public int* add(*int* a, *int* b) {  
  
 *return* a + b;  
  
 }  
  
 *public int* sub(*int* a, *int* b) {  
  
 *return* a - b;  
  
 }  
  
 *public int* mul(*int* a, *int* b) {  
  
 *return* a \* b;  
  
 }  
  
}

**Step 2:** Create a CalculatorTest class for Junit testing

*package* Junit.testing;  
  
*import static* org.junit.jupiter.api.Assertions.\*;  
  
*import* org.junit.jupiter.api.AfterAll;  
*import* org.junit.jupiter.api.AfterEach;  
*import* org.junit.jupiter.api.BeforeAll;  
*import* org.junit.jupiter.api.BeforeEach;  
*import* org.junit.jupiter.api.RepeatedTest;  
*import* org.junit.jupiter.params.ParameterizedTest;  
*import* org.junit.jupiter.params.provider.ValueSource;  
  
*class* CalculatorTest {  
  
 @BeforeAll  
 *static void* setUpBeforeClass() *throws* Exception {  
 }  
  
 @AfterAll  
 *static void* tearDownAfterClass() *throws* Exception {  
 }  
  
 @BeforeEach  
 *void* setUp() *throws* Exception {  
 }  
  
 @AfterEach  
 *void* tearDown() *throws* Exception {  
 }  
  
 @ParameterizedTest  
 @ValueSource(strings = { "Harish", "narayana", "Javeed", "himanshu" })  
  
 *void* test(String name) {  
  
 System.***out***.println(name + "name tested");  
 *assertTrue*(name.length() > 5);  
  
 }  
   
 @RepeatedTest(3)  
   
 *void* test1() {  
   
 System.***out***.println("test1");  
   
 *assertTrue*(*true*);  
 }  
  
}

**Output:**

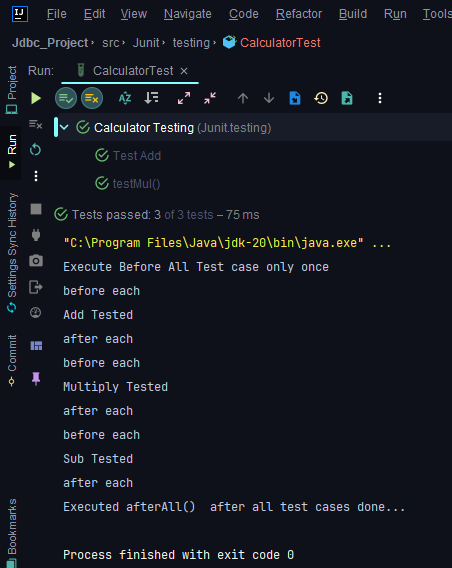
****

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

**Program:**

*package* Junit.testing;  
  
*import static* org.junit.jupiter.api.Assertions.*assertEquals*;  
*import static* org.junit.jupiter.api.Assertions.*assertNotEquals*;  
*import static* org.junit.jupiter.api.Assertions.*assertNotNull*;  
*import static* org.junit.jupiter.api.Assertions.*assertTrue*;  
  
*import* Junit.calsi.Calculator;  
*import* org.junit.jupiter.api.AfterAll;  
*import* org.junit.jupiter.api.AfterEach;  
*import* org.junit.jupiter.api.BeforeAll;  
*import* org.junit.jupiter.api.BeforeEach;  
*import* org.junit.jupiter.api.DisplayName;  
*import* org.junit.jupiter.api.Tag;  
*import* org.junit.jupiter.api.Test;  
  
  
@DisplayName("Calculator Testing")  
*class* AnnotationTest {  
  
 *static* Calculator *cal* = *null*;  
  
 @BeforeAll  
 *public static void* beforeAll() {  
  
 System.***out***.println("Execute Before All Test case only once");  
  
 *cal* = *new* Calculator();  
  
 }  
   
   
 @BeforeEach  
 *public void* before() {  
   
 System.***out***.println("before each");  
   
 }  
   
 @AfterEach  
 *public void* after() {  
   
 System.***out***.println("after each");  
   
   
 }  
   
   
  
 @Tag("G1")  
 @DisplayName("Test Add")  
 @Test  
 *void* testAdd() {  
  
 *assertNotNull*(*cal*);  
  
 *int* actual = *cal*.add(5,4);  
  
 *assertEquals*(9, actual);  
   
 System.***out***.println("Add Tested");  
  
 }  
  
 *//@Disabled* @Tag("G1")  
 @Test  
 *void* testSub() {  
  
 *int* actual = *cal*.sub(10, 4);  
  
 *assertEquals*(6, actual);  
  
 *assertNotEquals*(0, actual);  
   
 System.***out***.println("Sub Tested");  
  
 }  
  
 @Test  
 *void* testMul() {  
  
 *int* actual = *cal*.mul(5, 4);  
  
 *assertTrue*(actual > 0);  
   
 System.***out***.println("Multiply Tested");  
  
 }  
  
 @AfterAll  
 *public static void* afterAll() {  
  
 System.***out***.println("Executed afterAll() after all test cases done...");  
  
 }  
  
}

**Output:**

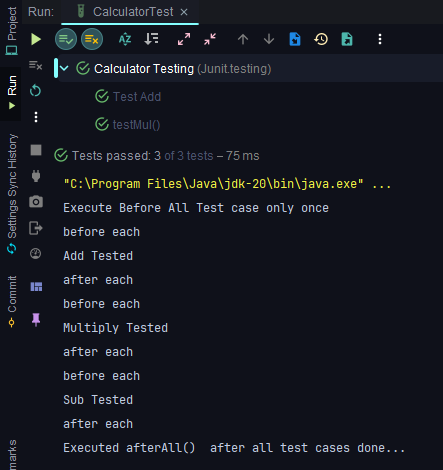
****

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

**Program:**

*package* Junit.testing;  
  
*import static* org.junit.jupiter.api.Assertions.\*;  
  
*import* org.junit.jupiter.api.AfterAll;  
*import* org.junit.jupiter.api.AfterEach;  
*import* org.junit.jupiter.api.BeforeAll;  
*import* org.junit.jupiter.api.BeforeEach;  
*import* org.junit.jupiter.api.RepeatedTest;  
*import* org.junit.jupiter.params.ParameterizedTest;  
*import* org.junit.jupiter.params.provider.ValueSource;  
  
*class* AnnotationsTest {  
  
 @BeforeAll  
 *static void* setUpBeforeClass() *throws* Exception {  
 }  
  
 @AfterAll  
 *static void* tearDownAfterClass() *throws* Exception {  
 }  
  
 @BeforeEach  
 *void* setUp() *throws* Exception {  
 }  
  
 @AfterEach  
 *void* tearDown() *throws* Exception {  
 }  
  
 @ParameterizedTest  
 @ValueSource(strings = { "Harish", "narayana", "Javeed", "himanshu" })  
  
 *void* test(String name) {  
  
 System.***out***.println(name + "name tested");  
 *assertTrue*(name.length() > 5);  
  
 }  
   
 @RepeatedTest(3)  
   
 *void* test1() {  
   
 System.***out***.println("test1");  
   
 *assertTrue*(*true*);  
 }  
  
}

**Output:**

****

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

**Answer:**

**GC algorithms:**

1. **Serial Garbage Collector:**
   * Type: Single-threaded, stop-the-world collector.
   * Use Case: Suitable for small applications or single-threaded environments.
   * Pros: Simple, low overhead.
   * Cons: High pause times during major collections.
   * Command Line Option: -XX:+UseSerialGC.
2. **Parallel Garbage Collector (Parallel GC):**
   * Type: Multi-threaded, stop-the-world collector.
   * Use Case: General-purpose applications with multi-core CPUs.
   * Pros: Better throughput than Serial GC.
   * Cons: Longer pause times than G1 or ZGC.
   * Command Line Option: -XX:+UseParallelGC.
3. **Concurrent Mark-Sweep (CMS) Garbage Collector:**
   * Type: Concurrent collector (low pause times).
   * Use Case: Applications requiring low-latency response times.
   * Pros: Low pause times, good for interactive applications.
   * Cons: Fragmentation, may cause CPU overhead.
   * Command Line Option: -XX:+UseConcMarkSweepGC.
4. **G1 Garbage Collector:**
   * Type: Region-based, parallel collector.
   * Use Case: Large heaps, balanced throughput and low-latency requirements.
   * Pros: Predictable pause times, adaptive.
   * Cons: Higher CPU usage than CMS.
   * Command Line Option: -XX:+UseG1GC.
5. **Z Garbage Collector (ZGC):**
   * Type: Low-latency, region-based collector.
   * Use Case: Applications requiring ultra-low pause times.
   * Pros: Extremely short pause times, suitable for large heaps.
   * Cons: May have slightly lower throughput.
   * Command Line Option: -XX:+UseZGC.