1.A Test Annotation

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.Test;

public class MathOperationsTest {

@Test

public void testAddition() {

MathOperations math = new MathOperations();

int result = math.add(3, 5);

Assertions.assertEquals(8, result);

}

@Test

public void testSubtraction() {

MathOperations math = new MathOperations();

int result = math.subtract(10, 4);

Assertions.assertEquals(6, result);

}

@Test

public void testMultiplication() {

MathOperations math = new MathOperations();

int result = math.multiply(2, 6);

Assertions.assertEquals(12, result);

}

@Test

public void testDivision() {

MathOperations math = new MathOperations();

double result = math.divide(10, 2);

Assertions.assertEquals(5.0, result);

}

@Test

public void testDivisionByZero() {

MathOperations math = new MathOperations();

Assertions.assertThrows(ArithmeticException.class, () -> {

math.divide(10, 0);

});

}

}

2.A Before and After Classes

import org.junit.After;

import org.junit.AfterClass;

import org.junit.Before;

import org.junit.BeforeClass;

import org.junit.Test;

import static org.junit.Assert.\*;

public class MathOperationsTest {

private static MathOperations math;

@BeforeClass

public static void setUpBeforeClass() throws Exception {

math = new MathOperations();

}

@AfterClass

public static void tearDownAfterClass() throws Exception {

math = null;

}

@Before

public void setUp() throws Exception {

System.out.println("Setting up for a test...");

}

@After

public void tearDown() throws Exception {

System.out.println("Cleaning up after a test...");

}

@Test

public void testAddition() {

int result = math.add(3, 5);

assertEquals(8, result);

}

@Test

public void testSubtraction() {

int result = math.subtract(10, 4);

assertEquals(6, result);

}

@Test

public void testMultiplication() {

int result = math.multiply(2, 6);

assertEquals(12, result);

}

@Test

public void testDivision() {

double result = math.divide(10, 2);

assertEquals(5.0, result, 0.0001);

}

@Test(expected = ArithmeticException.class)

public void testDivisionByZero() {

math.divide(10, 0);

}

}

3.A Asser equals,true

import org.junit.Test;

import static org.junit.Assert.\*;

public class StringUtilsTest {

@Test

public void testConcatenate() {

String result = StringUtils.concatenate("Hello", "World");

assertEquals("HelloWorld", result);

}

@Test

public void testIsNullOrEmpty() {

assertTrue(StringUtils.isNullOrEmpty(null));

assertTrue(StringUtils.isNullOrEmpty(""));

assertFalse(StringUtils.isNullOrEmpty("Hello"));

}

@Test

public void testIsPalindrome() {

assertTrue(StringUtils.isPalindrome("radar"));

assertTrue(StringUtils.isPalindrome("madam"));

assertFalse(StringUtils.isPalindrome("hello"));

}

@Test

public void testReverse() {

assertEquals("olleh", StringUtils.reverse("hello"));

assertEquals("racecar", StringUtils.reverse("racecar"));

}

}

4.A Research and Presentation

Garbage collection (GC) algorithms in Java play a crucial role in managing memory and ensuring optimal performance of Java applications. Here's a comparison of several popular garbage collection algorithms used in Java:

1. Serial Garbage Collector

Description: Also known as the Serial Collector, it is a simple, single-threaded garbage collector primarily intended for small-scale applications or single-threaded environments.

Usage: Default collector for client-style applications on JVMs prior to JDK 9.

Algorithm: Uses a mark-sweep-compact algorithm where garbage collection is performed sequentially. It stops all application threads during garbage collection.

Advantages:

Simplicity in implementation and low overhead.

Suitable for applications with small heaps and where responsiveness is not critical.

Disadvantages:

Pause times can be significant for larger heaps since it's single-threaded.

Not suitable for multi-threaded applications or large-scale enterprise applications.

2. Parallel Garbage Collector

Description: Also known as the throughput collector, it uses multiple threads to perform garbage collection, thus reducing pause times by leveraging available CPU cores.

Usage: Default collector for server-style applications on JVMs prior to JDK 9.

Algorithm: Similar to the Serial Collector but operates with multiple threads for major collections. Uses mark-sweep-compact like Serial GC.

Advantages:

- Improves application throughput by utilizing multiple CPU cores.

- Suitable for multi-threaded applications with medium to large heaps.

Disadvantages:

- Can cause longer pause times compared to newer collectors like G1 or ZGC.

- Pause times can still be noticeable for large heaps.

3. Concurrent Mark-Sweep (CMS) Garbage Collector

Description: Designed to minimize pause times by doing most of the garbage collection work concurrently with the application threads.

Usage: Deprecated since JDK 14 and removed in JDK 15 in favor of G1 and other collectors.

Algorithm: Uses multiple threads for the initial mark phase but most of the work is done concurrently with the application threads during the sweep and compact phases.

Advantages:

- Shorter pause times compared to Serial and Parallel GC for large heaps.

- Suitable for applications requiring low-latency response times.

Disadvantages:

- Can lead to higher CPU usage due to concurrent activity.

- May suffer from fragmentation issues due to its concurrent nature.

4. G1 (Garbage-First) Garbage Collector

Description: Designed for applications requiring large heaps with minimal pause times. It divides the heap into regions and performs garbage collection based on the region rather than the entire heap.

Usage: Default garbage collector introduced in JDK 9 for large-scale applications.

Algorithm: Uses a combination of generational and concurrent garbage collection techniques. It divides the heap into regions and uses a concurrent marking phase followed by a mixed collection phase to reclaim garbage.

Advantages:

- Predictable pause times even with large heaps.

- Suitable for large-scale multi-threaded applications where response times are critical.

- Reduces fragmentation compared to CMS.

Disadvantage:

- May have slightly higher overhead compared to Parallel GC in terms of throughput.

- Requires tuning to achieve optimal performance for specific applications.

5. ZGC (Z Garbage Collector)

Description: A low-latency garbage collector designed for applications requiring very short pause times, typically less than 10ms.

Usage: Introduced in JDK 11 and is suitable for large-scale applications where ultra-low pause times are critical.

Algorithm: Uses a concurrent garbage collection algorithm where all stages of GC (mark, relocate, and reference processing) are performed concurrently with application threads.

Advantages:

- Extremely low pause times even with very large heaps (tens of terabytes).

- Minimal impact on application throughput.

- Handles fragmentation well.

Disadvantages:

- Higher CPU overhead compared to other collectors.

- Requires newer hardware and may not be suitable for all applications.

.