Manjula Nannuri
Day 23_Assignment

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.

Create a Class with Mathematical Operations:

```
Packagejunit.com;

publicclassMathOperations {

        publicintadd(inta, intb) {
        returna + b;
           }
        publicintsubtract(inta, intb) {
        returna - b;
           }
        publicintmultiply(inta, intb) {
        returna * b;
           }
        publicdoubledivide(inta, intb) {
        if (b == 0) {
        thrownewIllegalArgumentException("Cannot divide by zero");
              }
        return (double) a / b;
           }

}
```

Create JUnit Tests for the MathOperations Class:

```
packagejunitAssignments;
```

```
importstaticorg.junit.jupiter.api.Assertions.*;
importorg.junit.jupiter.api.Test;
importstaticorg.junit.Assert.assertEquals;
classMathOperationTest {
        publicvoidtestAdd() {
        MathOperations math = newMathOperations();
        int result = math.add(4, 5);
        assertEquals(9, result);
          }
        publicvoidtestSubtract() {
        MathOperations math = newMathOperations();
        int result = math.subtract(10, 14);
        assertEquals(4, result);
          }
        publicvoidtestMultiply() {
        MathOperations math = newMathOperations();
        int result = math.multiply(7, 5);
        assertEquals(35, result);
          }
        publicvoidtestDivide() {
        MathOperations math = newMathOperations();
        double result = math.divide(10, 2);
        assertEquals(5.0, result, 0.001);
          }
```

```
    publicvoidtestDivideByZero() {
    MathOperations math = newMathOperations();
    Throwable exception = assertThrows(IllegalArgumentException.class, () ->math.divide(5, 0));
    assertEquals("Cannot divide by zero", exception.getMessage());
      }
}
```

Output:

All test cases succesufully executed

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

```
Packagejunit.com;
importstaticorg.junit.jupiter.api.Assertions.*;
importorg.junit.jupiter.api.Test;
importstaticorg.junit.jupiter.api.Assertions.assertEquals;
importstaticorg.junit.jupiter.api.Assertions.assertThrows;
importorg.junit.After;
importorg.junit.AfterClass;
importorg.junit.Before;
importorg.junit.BeforeClass;
class MathOperationTestTest1 {
publicclassMathOperationTest {
```

```
private MathOperations math;

public static void setUpBeforeClass() {

System.out.println("Setting up before the test class");

   }

public void setUp() {

      math = new MathOperations();

System.out.println("Setting up before each test method");

   }

public void testAdd() {

int result = math.add(9, 3);

assertEquals(12, result);

   }

public void testSubtract() {

int result = math.subtract(10, 16);

assertEquals(6, result);

   }

public void testMultiply() {

int result = math.multiply(7, 3);

assertEquals(21, result);

   }

public void testDivide() {

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

assertEquals(5.0, result, 0.001);

   }
```

```
    publicvoidtestDivideByZero() {
    Throwable exception = assertThrows(IllegalArgumentException.class, () ->math.divide(5, 0));
    assertEquals("Cannot divide by zero", exception.getMessage());
      }
    publicvoidtearDown() {
        math = null;
    System.out.println("Tearing down after each test method");
      }
    publicstaticvoidtearDownAfterClass() {
    System.out.println("Tearing down after the test class");
      }
    }
}
```

OUTPUT:

All test cases passed successfully.

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

Create the Custom String Utility Class:

```
package junitAssignments;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
class StringUtil {
```

```
    public static String reverse(String str) {
        return new StringBuilder(str).reverse().toString();
    }
    public static booleanisPalindrome(String str) {
        String reversed = reverse(str);
        return str.equals(reversed);
    }
    public static booleancontainsIgnoreCase(String str, String subStr) {
        return str.toLowerCase().contains(subStr.toLowerCase());
    }
}
```

Write Test Cases Using JUnit Assertions:

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import org.junit.Test;
public class StringUtilTest {
   public void testReverse() {
assertEquals("olleH", StringUtil.reverse("Hello"));
assertEquals("", StringUtil.reverse(""));
assertEquals("12345", StringUtil.reverse("54321"));
   }
   public void testIsPalindrome() {
```

```
assertTrue(StringUtil.isPalindrome("radar"));

assertTrue(StringUtil.isPalindrome("level"));

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

   }

   public void testContainsIgnoreCase() {

assertTrue(StringUtil.containsIgnoreCase("Hello World", "hello"));

assertTrue(StringUtil.containsIgnoreCase("OpenAI", "openai"));

assertFalse(StringUtil.containsIgnoreCase("Java Programming", "python"));

   }

}
```

Task 4:

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

1. Serial GC

Overview:

- **Algorithm Type:** Stop-the-world, single-threaded.
- **Phases:** Mark, Sweep, and Compact phases.
- **Use Case:** Suitable for single-threaded applications or small heaps.

Pros:

- Simple and easy to implement.
- Low overhead in terms of additional threads and complexity.

Cons:

- Not suitable for applications requiring high throughput or low pause times.
- Entire application is paused during garbage collection.

2. Parallel GC (also known as Throughput Collector)

Overview:

- **Algorithm Type:** Stop-the-world, multi-threaded.
- **Phases:** Mark, Sweep, and Compact phases, similar to Serial GC but with multiple threads.
- **Use Case:** Suitable for applications with large heaps and requiring high throughput.

Pros:

- Higher throughput compared to Serial GC due to multi-threaded garbage collection.
- Good for applications where raw throughput is more important than pause times.

Cons:

- Longer pause times compared to more advanced collectors like CMS and G1.
- Can cause significant pauses in applications with very large heaps.

3. Concurrent Mark-Sweep (CMS) GC

Overview:

- **Algorithm Type:** Mostly concurrent, with some stop-the-world phases.
- **Phases:** Initial Mark (stop-the-world), Concurrent Mark, Remark (stop-the-world), Concurrent Sweep.
- **Use Case:** Suitable for applications requiring low pause times and can tolerate some CPU overhead.

Pros:

- Lower pause times compared to Serial and Parallel GC.
- Concurrent phases reduce the time the application is paused.

Cons:

- Can lead to fragmentation since it doesn't compact the heap.
- Higher CPU usage due to concurrent phases.
- Risk of "Concurrent Mode Failure" if CMS cannot keep up with allocation rates, which triggers a fallback to stop-the-world full GC.

4. Garbage-First (G1) GC

Overview:

- **Algorithm Type:** Concurrent and parallel, region-based.
- **Phases:** Initial Mark (stop-the-world), Concurrent Mark, Remark (stop-the-world), Cleanup.
- **Use Case:** Suitable for large heaps with predictable pause times.

Pros:

- Designed to replace CMS with better performance and lower fragmentation.

- Compacts the heap, reducing fragmentation.
- Predictable pause times due to region-based collection and incremental compaction.

Cons:

- Can be more complex to tune compared to simpler collectors.
- May have higher overhead in terms of CPU and memory compared to Serial and Parallel GC.

5. Z Garbage Collector (ZGC)

Overview:

- **Algorithm Type:** Concurrent, region-based, low-latency.
- **Phases:** Concurrent Mark, Concurrent Relocate.
- **Use Case:** Suitable for applications requiring very low pause times and can tolerate some CPU overhead.

Pros:

- Ultra-low pause times (typically less than 10ms).
- Suitable for large heaps (up to terabytes).
- Mostly concurrent with minimal stop-the-world pauses.

Cons:

- Higher complexity and overhead compared to simpler GCs like Serial and Parallel GC.
- Requires more CPU and memory resources.

Comparison Table:

Feature	Serial GC	Parallel GC	CMS GC	G1 GC	ZGC
Algorithm Type	Stop-the-world	Stop-the-world	Mostly Concurrent	Concurrent and Parallel	Concurrent and Parallel
Threading	Single-threaded	Multi-threaded	Multi-threaded	Multi-threaded	Multi-threaded
Pause Time	High	High	Low	Predictable, Low	Ultra-low
Throughput	Low	High	Medium	High	High
Heap Size	Small	Medium to Large	Medium to Large	Large	Very Large (TBs)
Fragmentation	Low (Compact)	Low (Compact)	High	Low	Low
Use Case	Small applications	High throughput apps	Low-latency apps	Large heaps, predictable pause times	Ultra-low latency apps with very large heaps

Conclusion:

The choice of garbage collector depends on the specific requirements of your application:

- **Serial GC:** Best for small, single-threaded applications.
- **Parallel GC:** Best for applications needing high throughput.
- **CMS GC:** Best for applications needing low pause times and can tolerate some fragmentation.
- **G1 GC:** Best for large heaps with a need for predictable pause times.
- **ZGC:** Best for ultra-low pause times and very large heaps.

Each GC has its own advantages and trade-offs, and the best choice often depends on the application's performance characteristics and requirements.