Write a set of JUnit tests for a given class with simple mathematical operations (add, subtract, multiply, divide) using the basic @Test annotation.

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MathOperations {

    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) throws IllegalArgumentException {
        if (b == 0) {
            throw new IllegalArgumentException("Division by zero is not allowed.");
        }
        return a / b;
    }

    public static class MathOperationsTest {

        @Test
        public void testAdd() {
            MathOperations mathOps = new MathOperations();
            assertEquals(5, mathOps.add(2, 3), "2 + 3 should equal 5");
            assertEquals(-1, mathOps.add(-2, 1), "-2 + 1 should equal -1");
        }

        @Test
        public void testSubtract() {
            MathOperations mathOps = new MathOperations();
            assertEquals(1, mathOps.subtract(3, 2), "3 - 2 should equal 1");
            assertEquals(-3, mathOps.subtract(-2, 1), "-2 - 1 should equal -3");
```

```java
        }

        @Test
        public void testMultiply() {
            MathOperations mathOps = new MathOperations();
            assertEquals(6, mathOps.multiply(2, 3), "2 * 3 should equal 6");
            assertEquals(-2, mathOps.multiply(-1, 2), "-1 * 2 should equal -2");
        }

        @Test
        public void testDivide() {
            MathOperations mathOps = new MathOperations();
            assertEquals(2, mathOps.divide(6, 3), "6 / 3 should equal 2");
            assertEquals(-2, mathOps.divide(-4, 2), "-4 / 2 should equal -2");

            Exception exception = assertThrows(IllegalArgumentException.class, () -> {
                mathOps.divide(1, 0);
            });

            assertEquals("Division by zero is not allowed.", exception.getMessage());
        }
    }
}
```

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

```java
import org.junit.jupiter.api.*;

import static org.junit.jupiter.api.Assertions.*;

public class MathOperations {

    public int add(int a, int b) {
        return a + b;
    }
```

```java
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) throws IllegalArgumentException {
    if (b == 0) {
        throw new IllegalArgumentException("Division by zero is not allowed.");
    }
    return a / b;
}

public static class MathOperationsTest {

    private MathOperations mathOps;

    @BeforeAll
    public static void initAll() {
        System.out.println("Starting tests...");
    }

    @BeforeEach
    public void init() {
        mathOps = new MathOperations();
        System.out.println("Starting a test...");
    }

    @Test
    public void testAdd() {
        assertEquals(5, mathOps.add(2, 3), "2 + 3 should equal 5");
        assertEquals(-1, mathOps.add(-2, 1), "-2 + 1 should equal -1");
    }

    @Test
    public void testSubtract() {
        assertEquals(1, mathOps.subtract(3, 2), "3 - 2 should equal 1");
        assertEquals(-3, mathOps.subtract(-2, 1), "-2 - 1 should equal -3");
```

```
        }

        @Test
        public void testMultiply() {
            assertEquals(6, mathOps.multiply(2, 3), "2 * 3 should equal 6");
            assertEquals(-2, mathOps.multiply(-1, 2), "-1 * 2 should equal -2");
        }

        @Test
        public void testDivide() {
            assertEquals(2, mathOps.divide(6, 3), "6 / 3 should equal 2");
            assertEquals(-2, mathOps.divide(-4, 2), "-4 / 2 should equal -2");

            Exception exception = assertThrows(IllegalArgumentException.class, () -> {
                mathOps.divide(1, 0);
            });

            assertEquals("Division by zero is not allowed.", exception.getMessage());
        }

        @AfterEach
        public void tearDown() {
            System.out.println("Finished a test.");
        }

        @AfterAll
        public static void tearDownAll() {
            System.out.println("Finished all tests.");
        }
    }
}
```

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

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class StringUtil {

    public static boolean isEmpty(String str) {
```

```java
        return str == null || str.isEmpty();
    }

    public static String reverse(String str) {
        if (str == null) {
            return null;
        }
        return new StringBuilder(str).reverse().toString();
    }

    public static boolean isPalindrome(String str) {
        if (str == null) {
            return false;
        }
        String reversed = reverse(str);
        return str.equals(reversed);
    }

    public static String toUpperCase(String str) {
        if (str == null) {
            return null;
        }
        return str.toUpperCase();
    }

    public static class StringUtilTest {

        @Test
        public void testIsEmpty() {
            assertTrue(StringUtil.isEmpty(null), "String should be considered empty if it is
null");
            assertTrue(StringUtil.isEmpty(""), "String should be considered empty if it is an
empty string");
            assertFalse(StringUtil.isEmpty("Hello"), "String 'Hello' should not be considered
empty");
        }

        @Test
        public void testReverse() {
```

```
            assertEquals("olleH", StringUtil.reverse("Hello"), "The reverse of 'Hello' should
be 'olleH'");
            assertEquals("", StringUtil.reverse(""), "The reverse of an empty string should be
an empty string");
            assertNull(StringUtil.reverse(null), "The reverse of null should be null");
        }

        @Test
        public void testIsPalindrome() {
            assertTrue(StringUtil.isPalindrome("madam"), "'madam' should be identified as a
palindrome");
            assertFalse(StringUtil.isPalindrome("hello"), "'hello' should not be identified as a
palindrome");
            assertFalse(StringUtil.isPalindrome(null), "null should not be identified as a
palindrome");
        }

        @Test
        public void testToUpperCase() {
            assertEquals("HELLO", StringUtil.toUpperCase("hello"), "'hello' should be
converted to 'HELLO'");
            assertEquals("", StringUtil.toUpperCase(""), "An empty string should remain an
empty string when converted to uppercase");
            assertNull(StringUtil.toUpperCase(null), "Converting null to uppercase should
return null");
        }
    }

    public static void main(String[] args) {
        org.junit.platform.console.ConsoleLauncher.main(
            new String[] { "--select-class", "StringUtil$StringUtilTest" });
    }
}
```

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

Garbage collection (GC) in Java is a critical component for managing memory. Different GC algorithms have been designed to balance trade-offs between application throughput, latency, and memory footprint. Here's a comparison of some of the main garbage collection algorithms in Java: Serial, Parallel, Concurrent Mark-Sweep (CMS), Garbage-First (G1), and Z Garbage Collector (ZGC).

1. Serial Garbage Collector

Description:

- The simplest GC algorithm.
- Uses a single thread to perform all garbage collection work.
- Best suited for single-threaded applications or environments with small heap sizes.

Advantages:

- Simple and easy to implement.
- Minimal overhead due to the use of a single thread.

Disadvantages:

- Pauses all application threads during GC (stop-the-world pauses).
- Not suitable for multi-threaded applications or large heap sizes due to long pause times.

Use Case:

- Best for small applications with low memory requirements and single-threaded environments.

JVM Option:

- `-XX:+UseSerialGC`

2. Parallel Garbage Collector

Description:

- Also known as the "Throughput Collector".
- Uses multiple threads to perform GC work.
- Focuses on maximizing application throughput by minimizing the total time spent on GC.

Advantages:

- Efficient for applications with high throughput requirements.
- Can handle larger heap sizes better than the Serial GC.

Disadvantages:

- Still has stop-the-world pauses.
- Less focus on minimizing pause times, which can be problematic for latency-sensitive applications.

Use Case:

- Suitable for multi-threaded applications where throughput is more important than low latency.

JVM Option:

- `-XX:+UseParallelGC`

3. Concurrent Mark-Sweep (CMS) Garbage Collector

Description:

- Designed to minimize pause times by performing most of the GC work concurrently with application threads.
- Uses multiple threads for GC and runs in phases to reduce stop-the-world pauses.

Advantages:

- Reduced pause times compared to Serial and Parallel GCs.
- Suitable for applications requiring low latency.

Disadvantages:

- Can cause fragmentation, leading to inefficient memory use.
- Requires more CPU resources compared to Serial and Parallel GCs.
- Phases where stop-the-world pauses still occur.

Use Case:

- Applications that require low pause times and can tolerate some overhead from concurrent GC work.

JVM Option:

- `-XX:+UseConcMarkSweepGC`

4. Garbage-First (G1) Garbage Collector

Description:

- Aims to achieve both high throughput and low pause times.
- Divides the heap into regions and performs GC in a controlled manner.
- Uses a mix of concurrent and stop-the-world phases to manage memory.

Advantages:

- Predictable pause times, making it suitable for latency-sensitive applications.
- Handles large heaps more efficiently than CMS.
- Reduces fragmentation through region-based memory management.

Disadvantages:

- More complex to configure and tune.
- Can require significant CPU resources.

Use Case:

- Large applications needing a balance between low pause times and high throughput.

JVM Option:

- `-XX:+UseG1GC`

5. Z Garbage Collector (ZGC)

Description:

- Aims to achieve very low pause times (usually in the range of milliseconds).
- Performs most of the GC work concurrently with application threads.
- Designed for large heaps and memory-intensive applications.

Advantages:

- Extremely low pause times, typically sub-millisecond.
- Scales well with large heaps (terabyte-sized).

- Efficient concurrent compaction to reduce fragmentation.

Disadvantages:

- Relatively new and may not be as mature as other GCs.
- Can have higher CPU overhead due to extensive concurrent operations.

Use Case:

- Applications requiring minimal pause times and handling very large heaps.

JVM Option:

- `-XX:+UseZGC`

Summary

Garbage Collector	Description	Advantages	Disadvantages	Use Cases	JVM Option
Serial	Single-threaded GC	Simple, minimal overhead	Long pause times, not for large heaps	Small, single-threaded applications	`-XX:+UseSerialGC`
Parallel	Multi-threaded GC for throughput	High throughput	Longer pause times	Multi-threaded, throughput-critical	`-XX:+UseParallelGC`
CMS	Concurrent Mark-Sweep	Low pause times	Fragmentation, more CPU use	Latency-sensitive applications	`-XX:+UseConcMarkSweepGC`
G1	Garbage-First, region-based	Predictable pauses, handles large heaps	Complex tuning, more CPU use	Large applications, balanced needs	`-XX:+UseG1GC`

ZGC	Ultra-low pause time, concurrent	Very low pauses, handles very large heaps	Higher CPU overhead	Large, latency-critical applications	`-XX:+UseZGC`