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.

Ans:

```
public class SimpleMath {
    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 double divide(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Division by zero is not allowed.");
        }
        return (double) a / b;
    }
}

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

public class SimpleMathTest {
```

```
@Test
public void testAdd() {
   SimpleMath math = new SimpleMath();
   assertEquals(5, math.add(2, 3));
   assertEquals(0, math.add(-1, 1));
   assertEquals(-3, math.add(-1, -2));
}

@Test
public void testSubtract() {
   SimpleMath math = new SimpleMath();
   assertEquals(1, math.subtract(3, 2));
   assertEquals(-2, math.subtract(-1, 1));
   assertEquals(1, math.subtract(-1, -2));
}

@Test
public void testMultiply() {
   SimpleMath math = new SimpleMath();
   assertEquals(6, math.multiply(2, 3));
   assertEquals(-1, math.multiply(-1, 1));
   assertEquals(2, math.multiply(-1, -2));
}

@Test
public void testDivide() {
   SimpleMath math = new SimpleMath();
   assertEquals(2.0, math.divide(6, 3), 0.0001);
   assertEquals(-1.0, math.divide(-3, 3), 0.0001);
```

```
        assertEquals(0.5, math.divide(1, 2), 0.0001);
    }

    @Test
    public void testDivideByZero() {
        SimpleMath math = new SimpleMath();
        assertThrows(IllegalArgumentException.class, () -> {
            math.divide(1, 0);
        });
    }
}
```

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

Ans:

```
public class SimpleMath {
    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 double divide(int a, int b) {
```

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

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

public class SimpleMathTest {

    private SimpleMath math;

    @BeforeAll
    public static void beforeAllTests() {
        System.out.println("This runs once before all tests.");
    }

    @AfterAll
    public static void afterAllTests() {
        System.out.println("This runs once after all tests.");
    }

    @BeforeEach
    public void setUp() {
        math = new SimpleMath();
        System.out.println("This runs before each test.");
    }
```

```
@AfterEach
public void tearDown() {
    System.out.println("This runs after each test.");
}

@Test
public void testAdd() {
    assertEquals(5, math.add(2, 3));
    assertEquals(0, math.add(-1, 1));
    assertEquals(-3, math.add(-1, -2));
}

@Test
public void testSubtract() {
    assertEquals(1, math.subtract(3, 2));
    assertEquals(-2, math.subtract(-1, 1));
    assertEquals(1, math.subtract(-1, -2));
}

@Test
public void testMultiply() {
    assertEquals(6, math.multiply(2, 3));
    assertEquals(-1, math.multiply(-1, 1));
    assertEquals(2, math.multiply(-1, -2));
}

@Test
public void testDivide() {
```

```
        assertEquals(2.0, math.divide(6, 3), 0.0001);
        assertEquals(-1.0, math.divide(-3, 3), 0.0001);
        assertEquals(0.5, math.divide(1, 2), 0.0001);
    }

    @Test
    public void testDivideByZero() {
        assertThrows(IllegalArgumentException.class, () -> {
            math.divide(1, 0);
        });
    }
}
```

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

Ans:

```
public class StringUtil {
    public static boolean isPalindrome(String str) {
        if (str == null || str.isEmpty()) {
            return false;
        }
        String reversed = new StringBuilder(str).reverse().toString();
        return str.equals(reversed);
    }

    public static String reverse(String str) {
        if (str == null) {
            return null;
```

```
        }
        return new StringBuilder(str).reverse().toString();
    }

    public static boolean containsOnlyDigits(String str) {
        if (str == null || str.isEmpty()) {
            return false;
        }
        return str.chars().allMatch(Character::isDigit);
    }
}

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

public class StringUtilTest {

    @BeforeAll
    public static void beforeAllTests() {
        System.out.println("This runs once before all tests.");
    }

    @AfterAll
    public static void afterAllTests() {
        System.out.println("This runs once after all tests.");
    }

    @BeforeEach
    public void setUp() {
```

```
    System.out.println("This runs before each test.");
}

@AfterEach
public void tearDown() {
    System.out.println("This runs after each test.");
}

@Test
public void testIsPalindrome() {
    assertTrue(StringUtil.isPalindrome("madam"));
    assertTrue(StringUtil.isPalindrome("racecar"));
    assertFalse(StringUtil.isPalindrome("hello"));
    assertFalse(StringUtil.isPalindrome(""));
    assertFalse(StringUtil.isPalindrome(null));
}

@Test
public void testReverse() {
    assertEquals("madam", StringUtil.reverse("madam"));
    assertEquals("racecar", StringUtil.reverse("racecar"));
    assertEquals("olleh", StringUtil.reverse("hello"));
    assertEquals("", StringUtil.reverse(""));
    assertNull(StringUtil.reverse(null));
}

@Test
public void testContainsOnlyDigits() {
    assertTrue(StringUtil.containsOnlyDigits("123456"));
```

```
        assertTrue(StringUtil.containsOnlyDigits("0000"));
        assertFalse(StringUtil.containsOnlyDigits("123a456"));
        assertFalse(StringUtil.containsOnlyDigits("hello"));
        assertFalse(StringUtil.containsOnlyDigits(""));
        assertFalse(StringUtil.containsOnlyDigits(null));
    }
}
```

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

Ans:

Garbage Collection (GC) in Java is a form of automatic memory management that helps in reclaiming memory used by objects that are no longer in use by the application. Different GC algorithms have been developed to cater to different application needs and workloads. Here is a comparison of some key garbage collection algorithms in Java: Serial GC, Parallel GC, Concurrent Mark-Sweep (CMS) GC, Garbage-First (G1) GC, and Z Garbage Collector (ZGC).

1. Serial GC

- **Description**: The Serial GC uses a single thread to perform all garbage collection work, including both minor and major collections.
- **Usage**: It is suitable for small applications with a single processor or applications with low memory footprints.
- **Advantages**:
    - Simple and easy to implement.
    - Low overhead and low latency due to single-threaded nature.
- **Disadvantages**:
    - Not scalable for multi-threaded applications.
    - Long pause times during garbage collection, which can affect application performance.

2. Parallel GC

- **Description**: The Parallel GC, also known as the throughput collector, uses multiple threads to speed up garbage collection processes.

- **Usage**: Suitable for applications running on multi-core processors where high throughput is more critical than low latency.
- **Advantages**:
    - Better performance on multi-core systems due to parallelism.
    - Higher throughput by utilizing multiple threads for garbage collection.
- **Disadvantages**:
    - Can cause longer pause times compared to other advanced collectors.
    - Not ideal for applications requiring low-latency guarantees.

3. Concurrent Mark-Sweep (CMS) GC

- **Description**: The CMS GC aims to minimize pause times by performing most of its work concurrently with the application threads.
- **Usage**: Suitable for applications that require low-latency garbage collection.
- **Advantages**:
    - Reduces long pause times by doing most of the garbage collection work concurrently.
    - Suitable for applications requiring responsiveness and low-latency.
- **Disadvantages**:
    - Requires more CPU resources, as it runs alongside application threads.
    - May suffer from fragmentation issues, leading to full garbage collection cycles.

4. Garbage-First (G1) GC

- **Description**: The G1 GC is designed for large heap applications and aims to provide predictable pause times. It divides the heap into regions and uses both parallel and concurrent phases to achieve its goals.
- **Usage**: Suitable for applications with large heaps and requiring predictable pause times.
- **Advantages**:
    - Predictable pause times by allowing the user to set desired pause time goals.
    - Better heap management by dividing the heap into regions.
    - Handles both young and old generations in a unified manner.
- **Disadvantages**:
    - More complex and may require tuning to achieve optimal performance.
    - Higher CPU usage compared to simpler collectors.

5. Z Garbage Collector (ZGC)

- **Description**: ZGC is designed for ultra-low latency and handles very large heaps. It performs most of the work concurrently and aims to keep pause times below 10 milliseconds.
- **Usage**: Suitable for applications that need extremely low-latency garbage collection and can operate on very large heaps (up to multiple terabytes).
- **Advantages**:
    - Extremely low pause times (sub-10ms) regardless of heap size.
    - Capable of handling very large heaps efficiently.
    - Minimal impact on application performance.
- **Disadvantages**:
    - Higher CPU and memory overhead compared to simpler collectors.
    - Relatively new and may not be as mature as other collectors.