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:

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

class SimpleMathOperationsTesting {

	CalculaterDemo cal = new CalculaterDemo();

	@Test
	void testAdd() {
		int actual = cal.add(4, 5);
		assertEquals(9, actual);
		System.out.println("Add tested..");
	}

	@Test
	void testSub() {
		int actual = cal.sub(6, 4);
		assertEquals(2, actual);
		System.out.println("Sub tested..");
	}

	@Test
	void testMul() {
		int actual = cal.mul(2, 4);
		assertEquals(8, actual);
		System.out.println("Mul tested..");

	}

	@Test
	void testDiv() {
		int actual = cal.div(5, 4);
		assertNotEquals(0, actual);
		System.out.println("Div tested..");
	}
}
```

Output:

```
Add tested..
Div tested..
Mul tested..
Sub tested..
```

Task 2:

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

Program:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.jupiter.api.Test;

class Testing {

	@BeforeClass
	static void testBeforeClass() {
		System.out.println("Before class...");
	}

	@AfterClass
	static void testAfterClass() {
		System.out.println("After class...");
	}
	@Before
	void testBeforeMethod() {
		System.out.println("Before annotation...");
	}

	@Test
	void addTest() {
		System.out.println("Add test...");
	}

	@Test
	void subTest() {
		System.out.println("sub test...");
	}


	@After
	void testAfterMethod() {
		System.out.println("After annotation...");
	}
}
```

Output:

```
Before class...
Before annotation...
sub test...
After annotation...
Before annotation...
Add test...
After annotation...
After class...
```

Task 3:

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

Program:

```
import static org.junit.Assert.assertTrue;
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class TestingAsserts {

	AssertsTesting at=new AssertsTesting();
	@Test
	void testAddString() {
		String str1=at.str1("prakash ", "manikanta");
		assertEquals("prakash manikanta", str1);
		System.out.println("Add string method is tested...");
	}
	@Test
	void testTrueString() {
		String str1=at.str1("prakash ", "manikanta");
		assertTrue(str1.length() > 10);
		System.out.println("True string method is tested");
	}

	@Test
	void testFalseString() {
		String str1=at.str1("prakash ", "manikanta");
		assertFalse(str1.length() > 30);
		System.out.println("False string method is tested");
	}
}
```

Output:

```
Add string method is tested...
False string method is tested
True string method is tested
```

Task 4:

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

Explanation:

Serial Garbage Collector:

- **Type:** Single-threaded, stop-the-world.
- **Usage:** Typically used for small applications or in scenarios where low memory footprint low is critical.
- **Performance:** Suitable for applications with small heaps (<1GB) and low pause time requirements. However, it may cause noticeable pauses in larger applications.

Parallel Garbage Collector:

- **Type:** Multi-threaded, stop-the-world.
- **Usage:** Suited for applications requiring higher throughput. It utilizes multiple threads for garbage collection, which can lead to shorter pause times compared to Serial GC.
- **Performance:** Best suited for applications with medium to large heaps (>1GB) where latency is not a critical concern. It can achieve higher throughput by utilizing multiple CPU cores.

Concurrent Mark-Sweep (CMS) Garbage Collector:

- **Type:** Concurrent, low-pause.
- **Usage:** Designed for applications where low-latency is critical. It performs most of its work concurrently with the application threads, reducing pause times.
- **Performance:** Provides shorter pause times compared to the Parallel GC for medium-sized heaps. However, it may not be suitable for very large heaps due to fragmentation issues and higher CPU overhead.

G1 (Garbage-First) Garbage Collector:

- **Type:** Region-based, low-pause.
- **Usage:** Intended for large heaps (>4GB) with stringent pause time requirements. G1 divides the heap into regions and performs garbage collection on those regions with the most garbage first.
- **Performance:** Offers better predictability in terms of pause times compared to CMS and Parallel GC for large heaps. It aims to maintain consistent pause times regardless of the heap size.

ZGC (Z Garbage Collector):

- **Type:** Low-latency, scalable.
- **Usage:** Optimized for applications requiring very low pause times (<10ms) and scalability. It's designed to handle heaps ranging from a few hundred megabytes to multi-terabyte heaps.
- **Performance:** Provides ultra-low pause times, making it suitable for latency-sensitive applications such as financial trading systems or gaming servers. It achieves this by using concurrent and parallel algorithms for marking and relocation phases.

Comparison:

- **Pause Times:** Serial and Parallel GCs have longer pause times compared to CMS, G1, and ZGC. Among low-pause collectors, CMS and G1 offer shorter pause times for medium to large heaps, while ZGC provides the lowest pause times across all heap sizes.
- **Throughput:** Parallel GC generally offers the highest throughput, followed by G1. CMS and ZGC prioritize low-pause times over throughput.
- **Fragmentation:** CMS may suffer from heap fragmentation issues, especially with large heaps, leading to longer pause times for compaction. G1 and ZGC mitigate fragmentation issues through region-based management and concurrent compaction.
- **Heap Size:** While Serial and Parallel GCs are suitable for smaller heaps, CMS, G1, and ZGC are designed for larger heaps, with G1 and ZGC being particularly effective for heaps larger than 4GB.
- **Latency:** For applications with strict latency requirements, CMS, G1, and ZGC are preferred choices due to their low-pause characteristics. Among them, ZGC provides the lowest latency.