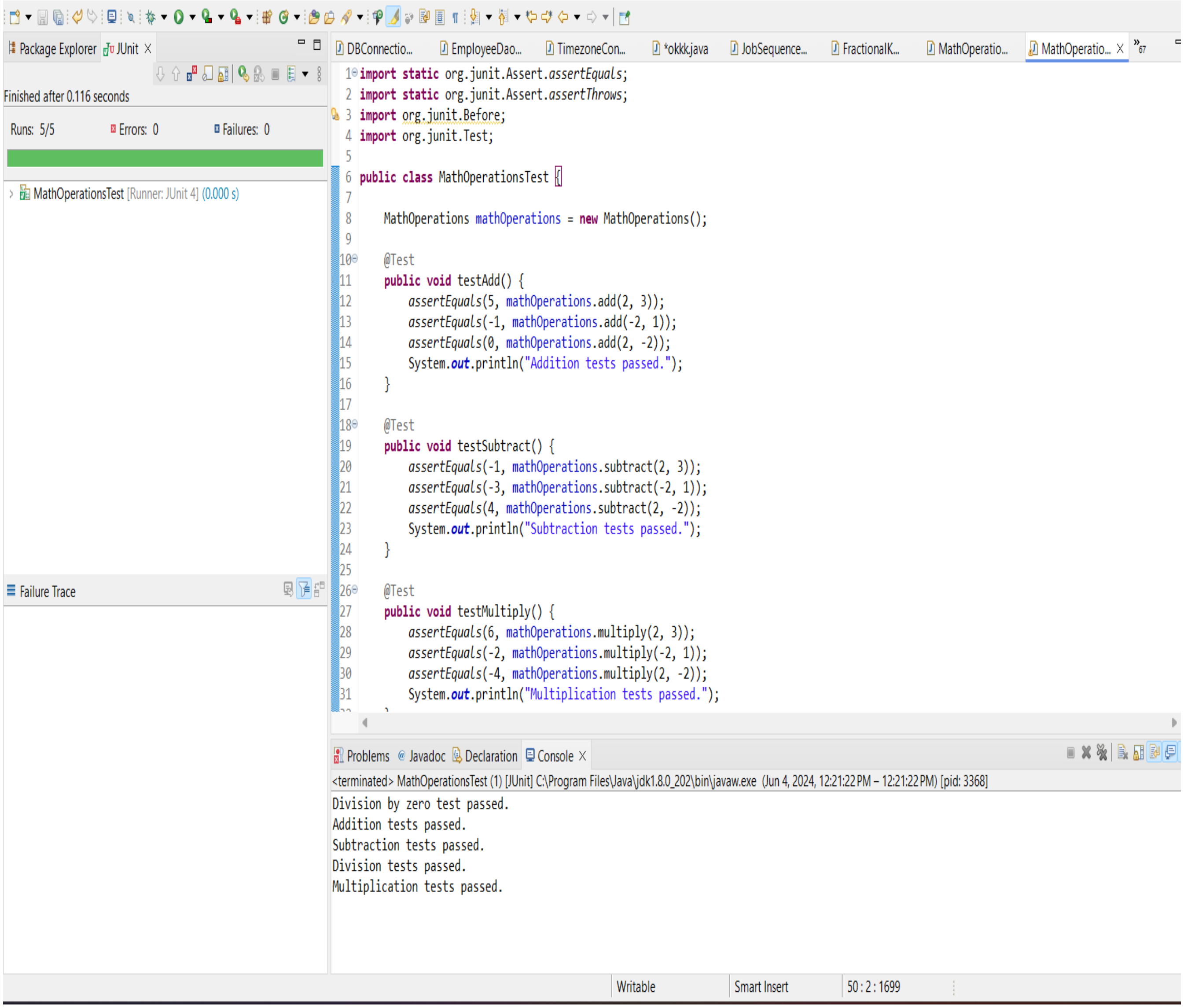
Day 22:

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.

Solution:

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


import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThrows;
import org.junit.Before;
import org.junit.Test;

public class MathOperationsTest {

    MathOperations mathOperations = new MathOperations();

    @Test
    public void testAdd() {
        assertEquals(5, mathOperations.add(2, 3));
        assertEquals(-1, mathOperations.add(-2, 1));
        assertEquals(0, mathOperations.add(2, -2));
        System.out.println("Addition tests passed.");
    }

    @Test
    public void testSubtract() {
        assertEquals(-1, mathOperations.subtract(2, 3));
        assertEquals(-3, mathOperations.subtract(-2, 1));
        assertEquals(4, mathOperations.subtract(2, -2));
        System.out.println("Subtraction tests passed.");
    }

    @Test
    public void testMultiply() {
        assertEquals(6, mathOperations.multiply(2, 3));
        assertEquals(-2, mathOperations.multiply(-2, 1));
        assertEquals(-4, mathOperations.multiply(2, -2));
        System.out.println("Multiplication tests passed.");
    }

    @Test
    public void testDivide() {
        assertEquals(2, mathOperations.divide(6, 3));
        assertEquals(-2, mathOperations.divide(-4, 2));
        assertEquals(-1, mathOperations.divide(2, -2));
        System.out.println("Division tests passed.");
    }

    @Test
    public void testDivideByZero() {
        Exception exception = assertThrows(IllegalArgumentException.class, () -> {
            mathOperations.divide(1, 0);
        });
        assertEquals("Division by zero is not allowed.", exception.getMessage());
        System.out.println("Division by zero test passed.");
    }
}
```

Output:



Finished after 0.116 seconds

Runs: 5/5 Errors: 0 Failures: 0

MathOperationsTest [Runner: JUnit 4] (0.000 s)

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThrows;
import org.junit.Before;
import org.junit.Test;

public class MathOperationsTest {

    MathOperations mathOperations = new MathOperations();

    @Test
    public void testAdd() {
        assertEquals(5, mathOperations.add(2, 3));
        assertEquals(-1, mathOperations.add(-2, 1));
        assertEquals(0, mathOperations.add(2, -2));
        System.out.println("Addition tests passed.");
    }

    @Test
    public void testSubtract() {
        assertEquals(-1, mathOperations.subtract(2, 3));
        assertEquals(-3, mathOperations.subtract(-2, 1));
        assertEquals(4, mathOperations.subtract(2, -2));
        System.out.println("Subtraction tests passed.");
    }

    @Test
    public void testMultiply() {
        assertEquals(6, mathOperations.multiply(2, 3));
        assertEquals(-2, mathOperations.multiply(-2, 1));
        assertEquals(-4, mathOperations.multiply(2, -2));
        System.out.println("Multiplication tests passed.");
```

```
<terminated> MathOperationsTest (1) [JUnit] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Jun 4, 2024, 12:21:22 PM – 12:21:22 PM) [pid: 3368]
Division by zero test passed.
Addition tests passed.
Subtraction tests passed.
Division tests passed.
Multiplication tests passed.
```

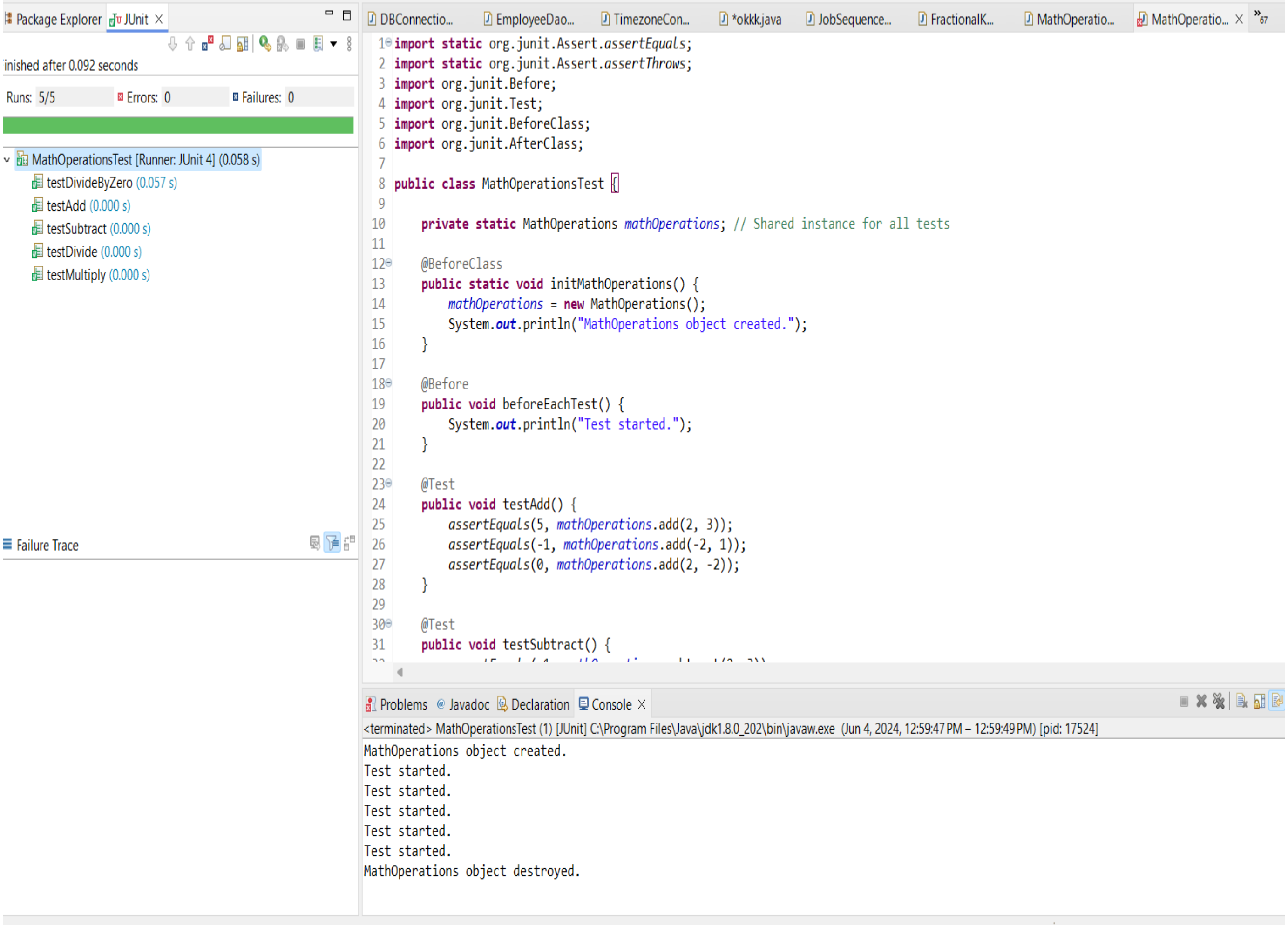
Task 2: Extend the above JUnit tests to use @Before, @After, @BeforeClass, and @AfterClass annotations to manage test setup and teardown.

Solution:

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThrows;
import org.junit.Before;
import org.junit.Test;
import org.junit.BeforeClass;
import org.junit.AfterClass;

public class MathOperationsTest {

    private static MathOperations mathOperations; // Shared instance for all tests
```

```
@BeforeClass
public static void initMathOperations() {
    mathOperations = new MathOperations();
    System.out.println("MathOperations object created.");
}

@Before
public void beforeEachTest() {
    System.out.println("Test started.");
}

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

@Test
public void testSubtract() {
    assertEquals(-1, mathOperations.subtract(2, 3));
    assertEquals(-3, mathOperations.subtract(-2, 1));
    assertEquals(4, mathOperations.subtract(2, -2));
}

@Test
public void testMultiply() {
    assertEquals(6, mathOperations.multiply(2, 3));
    assertEquals(-2, mathOperations.multiply(-2, 1));
    assertEquals(-4, mathOperations.multiply(2, -2));
}

@Test
public void testDivide() {
    assertEquals(2, mathOperations.divide(6, 3));
    assertEquals(-2, mathOperations.divide(-4, 2));
    assertEquals(-1, mathOperations.divide(2, -2));
}

@Test
public void testDivideByZero() {
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        mathOperations.divide(1, 0);
    });
    assertEquals("Division by zero is not allowed.", exception.getMessage());
}

@After
public void afterEachTest() {
    System.out.println("Test finished.");
}

@AfterClass
public static void tearDownMathOperations() {
        mathOperations = null;
        System.out.println("MathOperations object destroyed.");
    }
}
```

Output:



inished after 0.092 seconds

Runs: 5/5 Errors: 0 Failures: 0

- MathOperationsTest [Runner: JUnit 4] (0.058 s)
  - testDivideByZero (0.057 s)
  - testAdd (0.000 s)
  - testSubtract (0.000 s)
  - testDivide (0.000 s)
  - testMultiply (0.000 s)

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThrows;
import org.junit.Before;
import org.junit.Test;
import org.junit.BeforeClass;
import org.junit.AfterClass;

public class MathOperationsTest {

    private static MathOperations mathOperations; // Shared instance for all tests

    @BeforeClass
    public static void initMathOperations() {
        mathOperations = new MathOperations();
        System.out.println("MathOperations object created.");
    }

    @Before
    public void beforeEachTest() {
        System.out.println("Test started.");
    }

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

    @Test
    public void testSubtract() {
```

```
<terminated> MathOperationsTest (1) [JUnit] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe  (Jun 4, 2024, 12:59:47 PM – 12:59:49 PM) [pid: 17524]
MathOperations object created.
Test started.
Test started.
Test started.
Test started.
Test started.
MathOperations object destroyed.
```

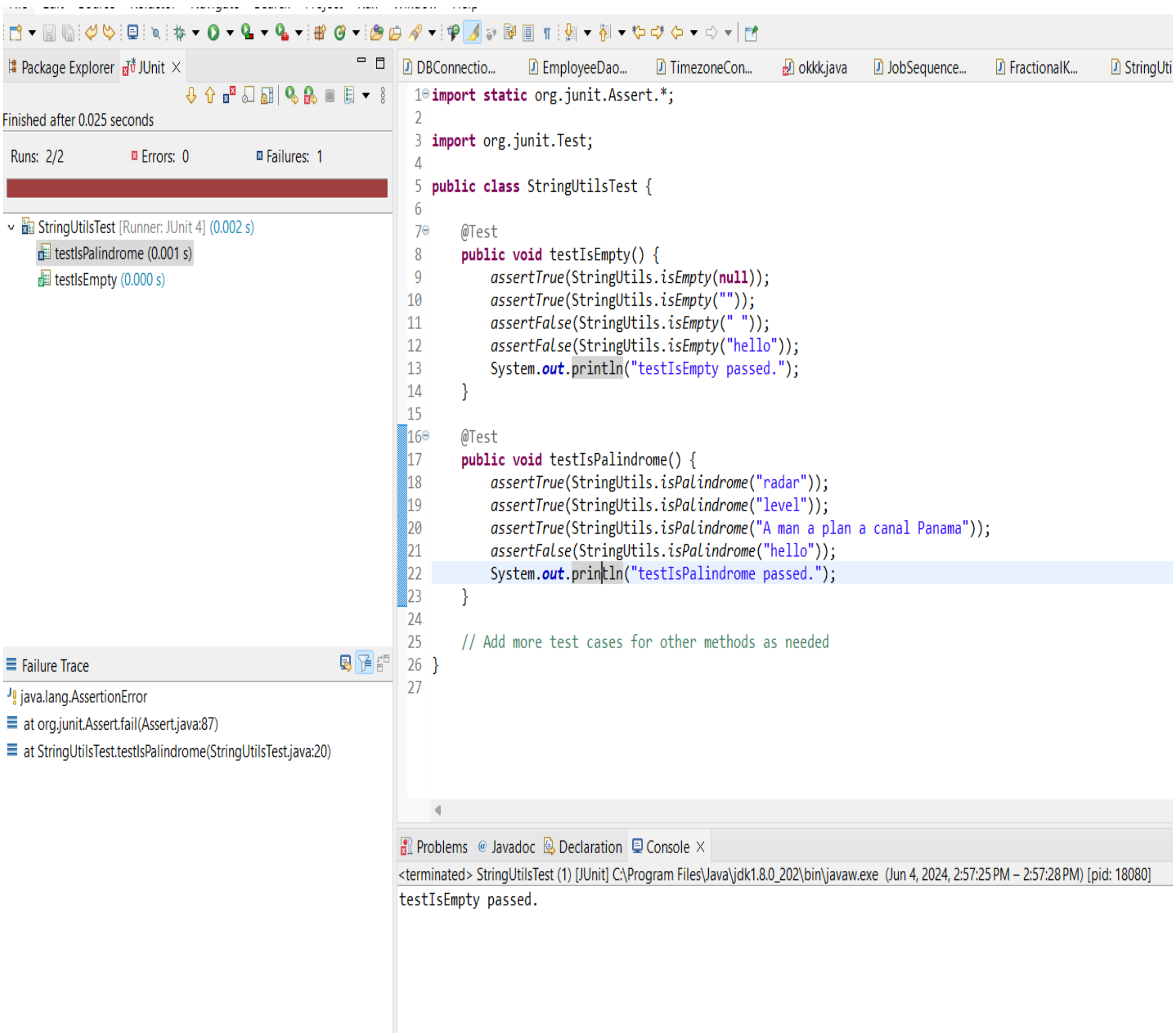
Task 3: Create test cases with assertEquals, assertTrue, and assertFalse to validate the correctness of a custom String utility class.

Solution:

```
public class StringUtils {

    public static boolean isEmpty(String str) {
        return str == null || str.isEmpty();
    }
```

```java
    public static boolean isPalindrome(String str) {
        if (str == null)
            return false;

        String reversed = new StringBuilder(str).reverse().toString();
        return str.equals(reversed);
    }

    // Add more methods as needed
}


import static org.junit.Assert.*;

import org.junit.Test;

public class StringUtilsTest {

    @Test
    public void testIsEmpty() {
        assertTrue(StringUtils.isEmpty(null));
        assertTrue(StringUtils.isEmpty(""));
        assertFalse(StringUtils.isEmpty(" "));
        assertFalse(StringUtils.isEmpty("hello"));
        System.out.println("testIsEmpty passed.");
    }

    @Test
    public void testIsPalindrome() {
        assertTrue(StringUtils.isPalindrome("radar"));
        assertTrue(StringUtils.isPalindrome("level"));
        assertTrue(StringUtils.isPalindrome("A man a plan a canal Panama"));
        assertFalse(StringUtils.isPalindrome("hello"));
        System.out.println("testIsPalindrome passed.");
    }

    // Add more test cases for other methods as needed
}
```

Output:



```
Finished after 0.025 seconds
Runs: 2/2    Errors: 0    Failures: 1

StringUtilsTest [Runner: JUnit 4] (0.002 s)
    testIsPalindrome (0.001 s)
    testIsEmpty (0.000 s)
```

```java
import static org.junit.Assert.*;

import org.junit.Test;

public class StringUtilsTest {

    @Test
    public void testIsEmpty() {
        assertTrue(StringUtils.isEmpty(null));
        assertTrue(StringUtils.isEmpty(""));
        assertFalse(StringUtils.isEmpty(" "));
        assertFalse(StringUtils.isEmpty("hello"));
        System.out.println("testIsEmpty passed.");
    }

    @Test
    public void testIsPalindrome() {
        assertTrue(StringUtils.isPalindrome("radar"));
        assertTrue(StringUtils.isPalindrome("level"));
        assertTrue(StringUtils.isPalindrome("A man a plan a canal Panama"));
        assertFalse(StringUtils.isPalindrome("hello"));
        System.out.println("testIsPalindrome passed.");
    }

    // Add more test cases for other methods as needed
}
```

```
java.lang.AssertionError
at org.junit.Assert.fail(Assert.java:87)
at StringUtilsTest.testIsPalindrome(StringUtilsTest.java:20)
```

```
<terminated> StringUtilsTest (1) [JUnit] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Jun 4, 2024, 2:57:25 PM – 2:57:28 PM) [pid: 18080]
testIsEmpty passed.
```

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

Solution:

Garbage collection (GC) in Java is a critical aspect of memory management, and several algorithms are available, each with its unique characteristics and use cases. Below is a comparison of the most common garbage collection algorithms in Java: Serial, Parallel, Concurrent Mark-Sweep (CMS), Garbage-First (G1), and Z Garbage Collector (ZGC).

1. Serial Garbage Collector

Overview:

- The Serial GC is the simplest garbage collector, designed for single-threaded environments.
- It is most suitable for small applications with single processors.

Characteristics:

- **Single-threaded**: Both minor and major garbage collections are performed by a single thread.
- **Stop-the-world**: Application execution pauses during garbage collection.
- **Low overhead**: Minimal resource usage, suitable for environments with limited resources.

Use Cases:

- Client machines or small applications.
- Environments where pause times are not critical.

Advantages:

- Simplicity and ease of implementation.
- Low overhead.

Disadvantages:

- Not suitable for multi-threaded applications.
- Can cause noticeable pause times in larger applications.

2. Parallel Garbage Collector

Overview:

- The Parallel GC, also known as the throughput collector, is designed for maximizing application throughput by using multiple threads for garbage collection.

Characteristics:

- **Multi-threaded**: Uses multiple threads for both minor and major collections.
- **Stop-the-world**: Similar to Serial GC, but leverages multiple threads to minimize pause times.

Use Cases:

- Server-side applications or large-scale applications.
- Environments where throughput is more critical than low pause times.

Advantages:

- Improved throughput due to multi-threading.
- Suitable for applications with large datasets.

Disadvantages:

- Still causes stop-the-world pauses, which may not be suitable for applications requiring low-latency.

3. Concurrent Mark-Sweep (CMS) Garbage Collector

Overview:

- The CMS GC is designed to minimize pause times and provide better responsiveness.

Characteristics:

- **Multi-threaded**: Uses multiple threads for concurrent marking and sweeping phases.
- **Concurrent Phases**: Most of the work is done concurrently with the application threads.
- **Old Generation Focused**: Primarily targets the old generation to reduce pause times.

Use Cases:

- Applications where low pause times and high responsiveness are critical, such as web servers and real-time applications.

Advantages:

- Low pause times due to concurrent execution.
- Suitable for applications requiring quick response times.

Disadvantages:

- Higher CPU usage due to concurrent operations.
- Fragmentation issues, which can lead to Full GC cycles.

4. Garbage-First (G1) Garbage Collector

Overview:

- The G1 GC is designed as a replacement for the CMS GC, providing better control over pause times and handling large heaps more efficiently.

Characteristics:

- **Region-based**: Divides the heap into regions and performs incremental garbage collection.

- **Predictable Pause Times**: Can be tuned to meet specific pause time goals.
- **Concurrent Phases**: Similar to CMS, G1 performs many operations concurrently.

Use Cases:

- Large applications requiring predictable pause times.
- Environments with large heaps and high throughput requirements.

Advantages:

- Predictable and configurable pause times.
- Handles large heaps efficiently with region-based management.

Disadvantages:

- Can be complex to tune and optimize.
- Higher overhead compared to simpler collectors like Serial or Parallel GC.

5. Z Garbage Collector (ZGC)

Overview:

- The ZGC is a low-latency garbage collector designed to handle large heaps with minimal pause times.

Characteristics:

- **Region-based**: Similar to G1, it uses regions but focuses on minimizing pauses.
- **Concurrent Phases**: Performs most of the garbage collection work concurrently.
- **Low Pause Times**: Pauses are typically in the range of a few milliseconds, regardless of heap size.

Use Cases:

- Applications with very large heaps and stringent latency requirements.
- Real-time applications and high-frequency trading systems.

Advantages:

- Extremely low pause times.
- Scalable to very large heaps (multi-terabyte).

Disadvantages:

- Higher CPU and memory overhead.
- Requires more recent Java versions (JDK 11 and later).