

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.

```
1 package com.test;
2 import static org.junit.Assert.assertEquals;
11
12 public class CalculateTest {
13
14     Calculate cal;
15
16     // @Rule
17     // public ExpectedException ex= ExpectedException.none();
18     // @BeforeClass
19     // public static void setUpClass() {
20     //     System.out.println("From static setUpClass()");
21     // }
22
23     @Before
24     public void setUp() {
25         System.out.println("Set up before");
26         cal = new Calculate();
27     }
28
29     @Test
30     public void testAdd() {
31         System.out.println("From add() ");
32         assertEquals(24, cal.add(10,10,4));
33     }
34 }
```

```
TestWithJUnit/...  emsmvn/pom.xml  Calculate.java  CalculateTestj...  AssertionsTest...  Validate.java  Validate
23 @Before
24 public void setUp() {
25     System.out.println("Set up before");
26     cal = new Calculate();
27 }
28
29 @Test
30 public void testAdd() {
31     System.out.println("From add() ");
32     assertEquals(24, cal.add(10,10,4));
33 }
34
35 @Test
36 public void testMultiply() {
37     System.out.println("From Multiply() ");
38     assertEquals(3, cal.multiply(1,3));
39 }
40 @Test
41 public void testDivideWithZero() {
42     System.out.println("From Divide() ");
43     //ex.expect(ArithmeticException.class);
44     cal.divide(5, 5);
45 }
46 }
```

Markers Properties Servers Data Source Explorer Snippets Terminal Console JUnit x

Finished after 0.036 seconds

Runs: 3/3 Errors: 0 Failures: 0

com.test.CalculateTest (Runner: JUnit 4) (0.000 s)

- testAdd (0.000 s)
- testDivideWithZero (0.000 s)
- testMultiply (0.000 s)

Failure Trace

Activate Windows
Go to Settings to activate Windows.

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

```
1 package com.test;
2 import static org.junit.Assert.assertEquals;
11
12 public class CalculateTest {
13
14     Calculate cal;
15
16     @Rule
17     public ExpectedException ex = ExpectedException.none();
18     @BeforeClass
19     public static void setUpClass() {
20         System.out.println("From static setupclass()");
21     }
22
23     @Before
24     public void setUp() {
25         System.out.println("Set up before");
26         cal = new Calculate();
27     }
28
29     @Test
30     public void testAdd() {
31         System.out.println("From add() ");
32         assertEquals(24, cal.add(10,10,4));
33     }
34
35     @Test
36     public void testMultiply() {
37         System.out.println("From Multiply() ");
38         assertEquals(3, cal.multiply(1,3));
39     }
40     @Test
41     public void testDivideWithZero() {
42         System.out.println("From Divide() ");
43         ex.expect(ArithmeticException.class);
44         cal.divide(5, 5);
45     }
46
47
48     @After
49     public void tearDown() {
50         System.out.println("Tear down after ");
51         cal=null;
52     }
53     @AfterClass
54     public static void tearDownClass() {
55         System.out.println("From static teardownclass()");
56     }
57 }
```

JUnit Test Results:

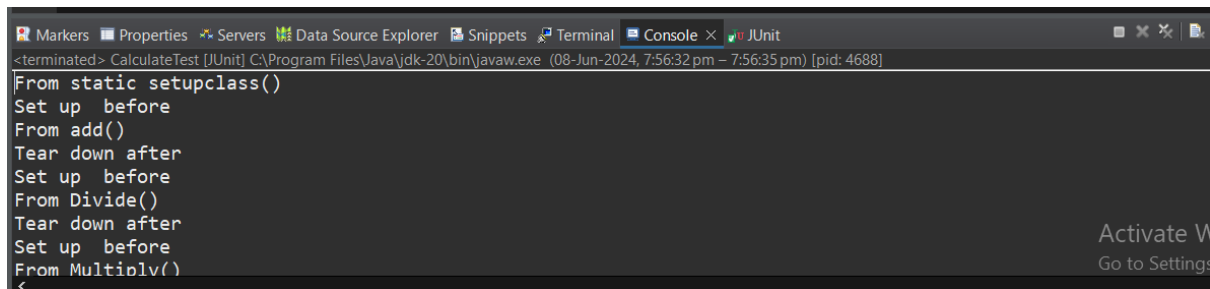
Finished after 0.036 seconds

Runs: 3/3 Errors: 0 Failures: 0

com.test.CalculateTest [Runner: JUnit 4] (0.000 s)

- testAdd (0.000 s)
- testDivideWithZero (0.000 s)
- testMultiply (0.000 s)

Activate Windows
Go to Settings to activate Windows.

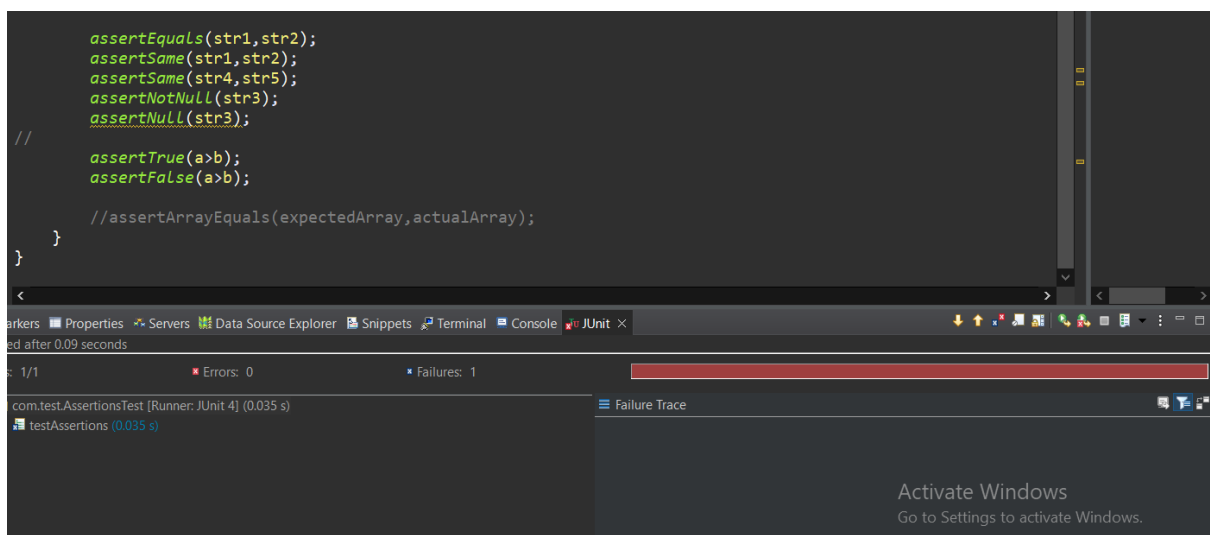


```
<terminated> CalculateTest [JUnit] C:\Program Files\Java\jdk-20\bin\javaw.exe (08-Jun-2024, 7:56:32 pm - 7:56:35 pm) [pid: 4688]
From static setupclass()
Set up before
From add()
Tear down after
Set up before
From Divide()
Tear down after
Set up before
From Multinlv()
Activate Windows
Go to Settings
```

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



```
1 package com.test;
2
3 import static org.junit.Assert.*;
4
12
13 public class AssertionsTest {
14
15     @Test
16     public void testAssertions() {
17
18         String str1=new String("deeps");
19         String str2=new String("deeps");
20
21         String str3=null;
22
23         String str4="deeps";
24         String str5="deeps";
25
26         int a=5;
27         int b=6;
28
29         String [] expectedArray= {"one","two","three"};
30         String [] actualArray= {"one","two","three"};
31
32         assertEquals(str1,str2);
33         assertEquals(str1,str2);
34         assertEquals(str4,str5);
35
36     }
37 }
```



```
assertEquals(str1,str2);
assertEquals(str1,str2);
assertEquals(str4,str5);
assertNotNull(str3);
assertNull(str3);
//
assertTrue(a>b);
assertFalse(a>b);
//assertEquals(expectedArray,actualArray);
}
}

com.test.AssertionsTest [Runner: JUnit 4] (0.035 s)
testAssertions (0.035 s)
Failure Trace
Activate Windows
Go to Settings to activate Windows.
```

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

1. Serial Garbage Collector:

- The Serial Garbage Collector, also known as the Serial Collector, is a simple garbage collector that operates using a single thread.
- It is suitable for applications with small to medium-sized heaps and low pause time requirements.
- The Serial Collector uses a "stop-the-world" approach, where all application threads are paused during garbage collection.
- It is generally used for single-threaded applications or applications with low latency requirements.
- While it can achieve compact heap layouts and low overhead, it may not be suitable for multi-threaded applications due to its stop-the-world nature.

2. Parallel Garbage Collector:

- The Parallel Garbage Collector, also known as the Throughput Collector, is designed for multi-threaded applications with medium to large heaps.
- It uses multiple threads to perform garbage collection in parallel, which can lead to improved throughput by utilizing multiple CPU cores.
- Like the Serial Collector, it also uses a "stop-the-world" approach but benefits from parallelism for faster garbage collection.
- The Parallel Collector is suitable for batch processing and applications where throughput is prioritized over pause times.
- While it can provide better throughput compared to the Serial Collector, it may introduce longer pause times due to its stop-the-world nature during garbage collection.

3. Concurrent Mark-Sweep (CMS) Garbage Collector:

- The CMS Garbage Collector, also known as the Concurrent Collector, is designed to minimize pause times for applications with large heaps and strict latency requirements.
- It uses multiple threads to perform most of the garbage collection work concurrently with the application threads, reducing pause times.
- The CMS Collector operates in multiple phases, including initial mark, concurrent marking, remark, and concurrent sweep.
- It is suitable for applications where low latency is critical, such as web servers and interactive applications.
- However, the CMS Collector may suffer from fragmentation issues and can lead to longer pause times during the remark and compaction phases.

4. G1 (Garbage-First) Garbage Collector:

- The G1 Garbage Collector is designed to provide both high throughput and low latency for large heaps (typically 4GB or larger).
- It divides the heap into regions and uses a combination of parallel, concurrent, and incremental approaches for garbage collection.
- The G1 Collector focuses on minimizing pause times by dynamically adjusting the size of the regions and prioritizing regions with the most garbage for collection.
- It is suitable for applications with large heaps and varying workload patterns, such as mixed workloads or applications with fluctuating memory demands.
- The G1 Collector aims to provide more predictable pause times compared to the CMS Collector while still achieving high throughput.

5. ZGC (Z Garbage Collector):

- The ZGC is a low-latency garbage collector introduced in JDK 11.
- It is designed to handle heaps ranging from a few megabytes to multiple terabytes with pause times not exceeding 10 milliseconds.
- The ZGC uses colored pointers and a load-barrier-based approach to enable concurrent marking, relocation, and compaction of objects.
- It is suitable for applications with stringent latency requirements, such as financial trading systems, gaming servers, and real-time analytics platforms.
- The ZGC aims to provide consistently low pause times, even with large heap sizes and high allocation rates.