**Performing Unit Testing Using JUnit 4**

JUnit 4 is a widely-used framework for unit testing in Java. It provides a simple way to test and validate your code. Here's an overview of key concepts and features:

### *1. Overview*

JUnit 4 allows you to write and run repeatable tests. It provides annotations to define tests, setup and teardown methods, and more. It uses assertions to verify expected outcomes and can be integrated into build tools and CI/CD pipelines.

### *2. Tests, Assertions, and Fixtures*

- **Tests**: Methods annotated with @Test. Each test method should test a single aspect of the code.
- **Assertions**: Methods used to check if the actual outcomes match the expected results. Common assertions include assertEquals, assertTrue, assertFalse, etc.
- **Fixtures**: Setup and teardown methods to initialize and clean up resources needed for tests.

### *3. Writing and Running Tests*

**Writing Tests:**

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

    public class MyTests {

        @Test
        public void testAddition() {
            int result = 2 + 3;
            assertEquals(5, result);
        }
    }
```

**Running Tests:**

You can run tests in various ways:

- **IDE**: Most IDEs (like IntelliJ IDEA, Eclipse) have built-in support to run JUnit tests.
- **Command Line**: Use tools like Maven or Gradle.
- **Build Tools**: Integrate tests into your build process using Maven (mvn test) or Gradle (gradle test).

### *4. Assertions*

JUnit provides several assertion methods to check various conditions:

- **assertEquals(expected, actual)**: Checks if expected equals actual.
- **assertNotEquals(unexpected, actual)**: Checks if unexpected does not equal actual.
- **assertTrue(condition)**: Checks if condition is true.
- **assertFalse(condition)**: Checks if condition is false.
- **assertNull(object)**: Checks if object is null.
- **assertNotNull(object)**: Checks if object is not null.
- **assertArrayEquals(expectedArray, actualArray)**: Checks if arrays are equal.

**Example:**

```
import org.junit.Test;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertArrayEquals;

public class AssertionTests {

    @Test
    public void testArrayEquals() {
        int[] expected = {1, 2, 3};
        int[] actual = {1, 2, 3};
        assertArrayEquals(expected, actual);
```

```
    }

    @Test
    public void testTrueCondition() {
        assertTrue(5 > 3);
    }
}
```

### *5. Test Fixtures: @Before and @After, @BeforeClass and @AfterClass*

**Test Fixtures** allow you to set up and clean up resources for your tests.

- **@Before**: Runs before each test method.
- **@After**: Runs after each test method.
- **@BeforeClass**: Runs once before all test methods in the class.
- **@AfterClass**: Runs once after all test methods in the class.

**Example:**

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

public class FixtureTests {

    @BeforeClass
    public static void setUpBeforeClass() {
        System.out.println("Before all tests");
    }

    @AfterClass
    public static void tearDownAfterClass() {
        System.out.println("After all tests");
    }

    @Before
    public void setUp() {
        System.out.println("Before each test");
    }
```

```
    @After
    public void tearDown() {
        System.out.println("After each test");
    }

    @Test
    public void testMethod1() {
        System.out.println("Test 1");
    }

    @Test
    public void testMethod2() {
        System.out.println("Test 2");
    }
}
```

### *6. Test Cases for Exception and Timeout*

**Exception Testing**: To test if a specific exception is thrown, use @Test(expected = Exception.class).

**Timeout Testing**: To specify a time limit for a test, use timeout attribute in the @Test annotation.

**Example:**

```
import org.junit.Test;

public class ExceptionAndTimeoutTests {

    @Test(expected = ArithmeticException.class)
    public void testException() {
        int result = 1 / 0;  // This will throw
ArithmeticException
    }

    @Test(timeout = 1000)  // 1 second timeout
    public void testTimeout() throws
InterruptedException {
        Thread.sleep(500);  // Sleeps for 500ms,
within the timeout limit
    }
}
```

## 7. Parameterized Tests

Parameterized tests allow you to run the same test with different inputs. You need to use the @RunWith(Parameterized.class) annotation and provide data through a static method.

**Example:**

```
import static org.junit.Assert.assertEquals;
import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

@RunWith(Parameterized.class)
public class ParameterizedTests {

    private int input;
    private int expected;

    public ParameterizedTests(int input, int expected) {
        this.input = input;
        this.expected = expected;
    }

    @Parameterized.Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
            {1, 2}, {2, 4}, {3, 6}
        });
    }

    @Test
    public void testMultiplication() {
        assertEquals(expected, input * 2);
    }
}
```

## 8. Test Suites

Test suites allow you to group multiple test classes together. Use @RunWith(Suite.class) and @Suite.SuiteClasses.

**Example:**

**TestSuite.java** can include TestClass1 and TestClass2, and running TestSuite will execute all tests in both classes.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({TestClass1.class,
TestClass2.class})
public class TestSuite {
    // No code needed
}
```

By using these concepts, you can structure and manage your unit tests effectively with JUnit 4.