40-JUnit5-Jupiter

Author: Vincent Lau

Learning Objectives

- Understand the purpose of using the JUnit5 framework
- Understand different APIs under jupiter-api

Introduction

JUnit 5 is a popular testing framework for Java that is used by developers to write and execute unit tests for their Java applications. It is the successor to JUnit 4 and provides several improvements and new features to make testing in Java more powerful and flexible.

JUnit 4 vs JUnit 5

JUnit 4 and JUnit 5 are both popular Java testing frameworks, but they have some key differences and are associated with different Java versions. Here's a breakdown of their relationships with Java versions:

- `JUnit 4` was released **before Java 8**; `JUnit 5`, also known as **JUnit Jupiter**, was released **after Java 8**.
- `JUnit 4` is compatible with Java 5 and later versions; `JUnit 5` requires Java 8 as a minimum runtime environment.
- `Junit 4` was the dominant version of JUnit for many years and is still in use in many projects; `JUnit 5` introduced significant changes and improvements over JUnit 4, including new architecture and features like parameterized tests and nested tests.
- `JUnit 5` is designed to work with the Java 8+ features, including lambdas and the Stream API, to make testing more flexible and expressive.

When choosing between JUnit 4 and JUnit 5, your choice may depend on your project's Java version requirements and whether you want to take advantage of the newer testing capabilities provided by JUnit 5.

Your First Test Case

As a beginner, you can start by setting up a simple JUnit 5 test project and writing your first test case. I'll walk you through the steps and provide a basic code example to get you started.

Step 1: Set Up Your Project

You can use a build tool like Maven or Gradle to set up your JUnit 5 project. Here, I'll provide an example using Maven.

1. Create a new Maven project or add the following dependencies to your existing Maven project's `pom.xml`:

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <!--  Junit5 requires Java 8 or above -->
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.10.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

2. Make sure your IDE (e.g., VSCode, IntelliJ IDEA, Eclipse) recognizes your project as a Maven project and downloads the dependencies.

Step 2: Write Your First Test

Now, let's write a simple JUnit 5 test case.

Create a Java class for your test. For example, `MyTest.java`:

```
1  import org.junit.jupiter.api.Test;
2  import static org.junit.jupiter.api.Assertions.assertEquals;
3
4  public class MyTest {
5      @Test
6      void additionTest() {
7          int result = 2 + 2;
8          assertEquals(4, result);
9      }
10 }
```

In this example, we have a test method named `additionTest`. This test checks if the result of the additional operation `2 + 2` is equal to `4`.

Step 3: Run Your Tests

You can run your tests using your IDE or from the command line using Maven.

- **Using IDE**: Most modern IDEs provide built-in support for running JUnit tests. Simply right-click on your test class or test method and choose "Run" or "Run as JUnit test."
- **Using Maven**: Open a terminal or command prompt, navigate to your project directory, and run the following command:

```
1 mvn test
```

Maven will execute your JUnit tests and display the results in the console. You should see output similar to this if your test passes:

```
1 -------------------------------------------------------
2  T E S T S
3 -------------------------------------------------------
4 Running MyTest
```

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.05 s - in MyTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

You can now continue to add more test methods to the `MyTest` class and explore additional features of JUnit 5 as you become more comfortable with writing tests for your Java code.

Fail Test Cases

Test Fail != Compile Fail

In JUnit 5, test cases that fail during execution **do not cause a compile failure**. The compiler and runtime errors are separate concerns in the software development process.

When you write a test case that fails during execution (i.e., a test assertion fails or an exception is thrown that wasn't expected), it will not prevent your code from compiling. Instead, it will be reported as a failed test when you run your test suite, but your code can still be compiled and executed.

Example of Failed Test Case

Here's an example of a failing test case in JUnit 5:

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class FailingTest {

    @Test
    void failingTestCase() {
        int result = 2 + 2;
        // This assertion will fail because 2 + 2 is not equal to 5.
        assertEquals(5, result);
    }
}
```

In this example, the `assertEquals` method is used to check if the result of the addition operation is equal to 5, which it is not. When you run this test, it will fail, and you will see an output indicating the failure:

```
org.opentest4j.AssertionFailedError: expected: <5> but was: <4>
```

However, despite the test failure, your code can still be compiled successfully. The compilation process and the execution of tests are separate stages in the development workflow.

Compile failures are typically related to issues like syntax errors, missing dependencies, or type mismatches in your code. Failing tests, on the other hand, are related to the behavior of your code during execution and are intended to help you identify and fix bugs and issues in your application.

Inlcude Test Files

In a Maven project, when you create a test class, it's **conventionally** named with the suffix `Test`, like `AppTest.java`. Maven's Surefire Plugin, which is commonly used for running tests in Maven projects, follows certain naming conventions and includes classes with names ending in `Test` by default when searching for tests to run **(surefire-plugin is a default plugin in maven project, refer to the above).**

If you've developed other Java files for testing and they don't follow the naming convention of ending in `Test`, they won't be automatically included and executed as tests by Maven's Surefire Plugin. To include and run your custom test classes that don't follow the naming convention, you have a few options:

Rename your test classes

You can rename your test classes to follow the naming convention. For example, if you have a class named `MyCustomTests.java`, you can rename it to `MyCustomTestsTest.java`, and it will be picked up by Surefire.

Configure Surefire (Not Recommand)

You can configure the Surefire Plugin in your `pom.xml` to include specific classes or patterns that should be considered as tests.

Modify the `<include>` elements to match your specific naming conventions or patterns for test classes.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.2</version> <!-- Adjust the version as needed -->
```

```
                <configuration>
                    <includes>
                        <include>**/*Test.java</include> <!-- Include classes ending with "Test" -->
                        <include>**/*CustomTests.java</include> <!-- Include your custom test classes -->
                    </includes>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

JUnit 5 Tags (Not Recommand)

If you're using JUnit 5, you can use tags to organize and run specific tests, regardless of their naming conventions. This approach allows you to annotate your test classes and methods with custom tags and then run tests based on those tags using Maven's filtering options.

For example, in JUnit 5, you can use the `@Tag` annotation to tag your tests and then run tests with specific tags using Maven's filtering. Here's an example of how to do this:

Then, you can run tests with the `myCustomTests` tag using Maven's filtering.

```
@Tag("myCustomTests")
public class MyCustomTests {
    // Your test methods here
}
```

Static Import

Static import is a feature in Java that allows you to import and use static members (fields and methods) of a class directly without specifying the class name. This feature was introduced in Java 5 (J2SE 5.0) and is useful when you have a class with many static members that you frequently use in your code. Static imports make your code more concise and readable by eliminating the need to prefix static members with the class name.

Importing Static Members

In JUnit, there are several assertion methods provided by the `Assertions` class. You can import these methods using static import to make your test code more concise and readable. Here's how you can do it:

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

import org.junit.jupiter.api.Test;

public class MyJUnitTest {

    @Test
    void testAssertions() {
        // assertEquals
        assertEquals(4, 2 + 2);
        // assertTrue and assertFalse
        assertTrue(5 > 3);
        assertFalse(2 == 5);
    }
}
```

In this example:

- We've imported all assertion methods from `Assertions` using `import static org.junit.jupiter.api.Assertions.*;`.
- Inside the `testAssertions` method, we can use various assertion methods like `assertEquals`, `assertTrue`, `assertFalse`, `assertNull`, `assertNotNull`, and `assertArrayEquals` without specifying `Assertions` as a prefix.

Importing Specific Static Members

If you prefer to import specific assertion methods rather than all of them, you can do so like this:

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Assertions.assertFalse;

import org.junit.jupiter.api.Test;

public class MyJUnitTest {

    @Test
    void testAssertions() {
        // assertEquals
        assertEquals(4, 2 + 2);
        // assertTrue and assertFalse
        assertTrue(5 > 3);
        assertFalse(2 == 5);
    }
```

```
}
```

In this case, we've imported only the `assertEquals`, `assertTrue` & `assertFalse` methods from `Assertions`, which makes your imports more specific.

Using static imports for assertion methods can make your JUnit test code cleaner and easier to read, especially when you have multiple assertions within a single test method.

Jupiter API

Test

This annotation marks a method as a test method that JUnit should run.

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

public class AssertionsGuide {
    @Test
    void testAssertEquals() {
        // This method is marked as a test case that JUnit should cover.
    }
}
```

Assertions

In this session, we explore the various assertion methods provided by JUnit Jupiter's `org.junit.jupiter.api.Assertions` class one by one with code examples for each method. JUnit Jupiter provides a rich set of assertion methods to verify different conditions in your test cases.

assertEquals

Checks if two values are equal.

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

public class AssertionsGuide {
    @Test
    void testAssertEquals() {
        // Checks if two values are equal
```

```
        assertEquals(4, 2 + 2);
    }
}
```

assertNotEquals

Checks if two values are not equal.

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

public class AssertionsGuide {
    @Test
    void testAssertNotEquals() {
        // Checks if two values are not equal
        assertNotEquals(5, 2 + 2);
    }
}
```

assertTrue

Checks if a condition is true.

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

public class AssertionsGuide {
    @Test
    void testAssertTrue() {
        // Checks if a condition is true
        assertTrue(5 > 3);
    }
}
```

assertFalse

Checks if a condition is false.

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

public class AssertionsGuide {
```

```
    @Test
    void testAssertFalse() {
        // Checks if a condition is false
        assertFalse(2 == 5);
    }
}
```

assertNull

Checks if a value is null.

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

public class AssertionsGuide {
    @Test
    void testAssertNull() {
        // Checks if a value is null
        String str = null;
        assertNull(str);
    }
}
```

assertNotNull

Checks if a value is not null.

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

public class AssertionsGuide {
    @Test
    void testAssertNotNull() {
        // Checks if a value is not null
        String nonNullStr = "Hello";
        assertNotNull(nonNullStr);
    }
}
```

assertArrayEquals

Checks if two arrays are equal.

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

public class AssertionsGuide {
    @Test
    void testAssertArrayEquals() {
        // Checks if two arrays are equal
        int[] expectedArray = {1, 2, 3};
        int[] actualArray = {1, 2, 3};
        assertArrayEquals(expectedArray, actualArray);
    }
}
```

assertSame

Checks if two references point to the same object.

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

public class AssertionsGuide {
    @Test
    void testAssertSame() {
        // Checks if two references point to the same object
        String str1 = "JUnit";
        String str2 = str1;
        assertSame(str1, str2);
    }
}
```

assertNotSame

Checks if two references do not point to the same object.

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

public class AssertionsGuide {
    @Test
    void testAssertNotSame() {
        // Checks if two references do not point to the same object
        String str1 = new String("JUnit");
        String str2 = new String("JUnit");
        assertNotSame(str1, str2);
```

```
    }
}
```

assertThrows

Checks if a specific exception is thrown.

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

public class AssertionsGuide {
    @Test
    void testAssertThrows() {
        // Checks if a specific exception is thrown
        assertThrows(ArithmeticException.class, () -> {
            int result = 1 / 0;
        });
    }
}
```

assertDoesNotThrow

Checks if a specific exception is not thrown.

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

public class AssertionsGuide {
    @Test
    void testAssertDoesNotThrow() {
        // Checks if a specific exception is not thrown
        assertDoesNotThrow(() -> {
            int result = 2 + 2;
        });
    }
}
```

assertAll

Groups multiple assertions and reports all failures.

```
import org.junit.jupiter.api.Test;
```

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

public class AssertionsGuide {
    @Test
    void testAssertAll() {
        // Checks multiple assertions and reports all failures
        assertAll(
            () -> assertTrue(2 > 1),
            () -> assertEquals(4, 2 + 2),
            () -> assertNotNull("JUnit")
        );
    }
}
```

assertTimeout

The `assertTimeout` method in JUnit Jupiter is used to check if a specific block of code executes within a specified time limit. It's particularly useful for testing the performance and responsiveness of your code.

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

public class AssertionsGuide {
    @Test
    void testAssertTimeout() {
        // This test checks if a piece of code runs within a specified time limit.
        assertTimeout(Duration.ofMillis(100), () -> {
            // Simulate a time-consuming operation (e.g., a computation or a sleep)
            Thread.sleep(50);
        });
    }
}
```

TestInstance

`@TestInstance` annotation is used to configure how test instances are created and managed for test classes. It allows you to control whether a new instance of the test class is created for each test method or if a single instance should be reused across multiple test methods within the same test class. This provides flexibility in managing the state of test classes.

PER_METHOD (Default)

```java
import org.junit.jupiter.api.*;

// @TestInstance(TestInstance.Lifecycle.PER_METHOD)
public class TestInstanceTest {

    private int x;

    @Test
    void testMethod1() {
        x++;
        Assertions.assertEquals(1, x);
    }

    @Test
    void testMethod2() {
        x++;
        Assertions.assertEquals(1, x);
    }
}
```

PER_CLASS

`@TestInstance(TestInstance.Lifecycle.PER_CLASS)` , JUnit creates a single instance of the test class for the entire test class, and this instance is reused for all test methods within the class. This can be useful when you want to share state across multiple test methods in the same class, reducing the overhead of creating a new instance for each method.

```java
import org.junit.jupiter.api.*;

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class TestInstanceTest {

    private int x;

    @Test
    void testMethod1() {
        x++;
        System.out.println("testMethod1");
        Assertions.assertEquals(1, x);
    }

```

```
    @Test
    void testMethod2() {
        x++;
        System.out.println("testMethod2");
        Assertions.assertEquals(2, x);
    }
}
```

BeforeEach

The `@BeforeEach` annotation is used to mark a method that should run before each test method in a test class. It is typically used for setting up common resources or state required for each test.

In this example, the `setUp()` method marked with `@BeforeEach` is executed before each test method (`testIncrement` and `testDecrement`), ensuring that the `count` variable is initialized to 5 before each test.

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class BeforeEachTest {

    private int count;

    @BeforeEach
    void setUp() {
        // This method will run before each test method
        count = 5;
    }

    @Test
    void testIncrement() {
        count++;
        assertEquals(6, count);
    }

    @Test
    void testDecrement() {
        count--;
        assertEquals(4, count);
    }
}
```

BeforeAll

The `@BeforeAll` annotation is used to mark a method that should run once before all the test methods in a test class. It is often used for one-time setup tasks or resource allocation.

In this example, the `setUp()` method marked with `@BeforeAll` is executed once before all the test methods in the test class. The `@TestInstance` annotation is used to specify the test instance lifecycle as `PER_CLASS`.

```java
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInstance;
import static org.junit.jupiter.api.Assertions.assertEquals;

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class BeforeAllExample {

    private int total;

    @BeforeAll
    void setUp() {
        // This method will run once before all test methods
        total = 0;
    }

    @Test
    void testAddition() {
        total += 5;
        assertEquals(5, total);
    }

    @Test
    void testSubtraction() {
        total -= 3;
        assertEquals(2, total);
    }
}
```

AfterEach

The `@AfterEach` annotation is used to mark a method that should run after each test method in a test class. It is typically used for cleaning up resources or resetting state after each test.

In this example, the `tearDown()` method marked with `@AfterEach` is executed after each test method (`testAppend` and `testClear`), ensuring that the `stringBuilder` is reset

to `null` after each test.

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Test;

public class AfterEachTest {

    private StringBuilder stringBuilder;

    @AfterEach
    void tearDown() {
        // This method will run after each test method
        stringBuilder = null;
    }

    @Test
    void testAppend() {
        stringBuilder = new StringBuilder("Hello");
        stringBuilder.append(" World");
        // Assertions or test logic here
    }

    @Test
    void testClear() {
        stringBuilder = new StringBuilder("JUnit");
        stringBuilder.setLength(0);
        // Assertions or test logic here
    }
}
```

AfterAll

The `@AfterAll` annotation is used to mark a method that should run once after all the test methods in a test class. It is often used for one-time cleanup tasks or resource deallocation.

In this example, the `tearDown()` method marked with `@AfterAll` is executed once after all the test methods in the test class. The `@TestInstance` annotation specifies the test instance lifecycle as `PER_CLASS`.

```
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInstance;

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
```

```
6  public class AfterAllExample {
7
8      private int count = 0;
9
10     @AfterAll
11     void tearDown() {
12         count = 0;
13     }
14
15     @Test
16     void testIncrement() {
17         count++;
18         assertEquals(1, count);
19     }
20
21     @Test
22     void testDecrement() {
23         count--;
24         assertEquals(0, count);
25     }
26 }
```

These annotations (`@BeforeEach` , `@BeforeAll` , `@AfterEach` , and `@AfterAll`) are essential for managing the setup and teardown of resources in your test classes, ensuring that your tests run in a clean and predictable environment.

DisplayName

The `@DisplayName` annotation in JUnit Jupiter is used to provide custom display names for your test classes and test methods. This annotation allows you to make your test output more descriptive and meaningful, improving the readability of your test reports and test execution results.

For example:

```
1  import static org.junit.jupiter.api.Assertions.assertEquals;
2
3  import org.junit.jupiter.api.DisplayName;
4  import org.junit.jupiter.api.Test;
5
6  @DisplayName("Calculator Tests")
7  class CalculatorTest {
8
9      private int x;
10
```

```
    @Test
    @DisplayName("Addition Test")
    void testAddition() {
        x++;
        assertEquals(1, x);
    }

    @Test
    @DisplayName("Subtraction Test")
    void testSubtraction() {
        x++;
        assertEquals(1, x);
    }
}
```

`@DisplayName` can be combined with other annotations to provide both a meaningful name and additional context for your tests. In this example, the test class has a custom name "Calculator Tests" using `@DisplayName`.

TestMethodOrder

Annotate your test class with `@TestMethodOrder` and specify `OrderAnnotation.class` to enable the order based on the `@Order` annotations.

With this configuration, JUnit will execute the test methods in the order specified by the `@Order` annotations.

When you run your tests, JUnit will use the `OrderAnnotation` to determine the order in which the test methods are executed.

In this case, the test methods will be executed in the order specified by the `@Order` annotations: `testMethod3`, `testMethod2`, and then `testMethod1`.

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.TestMethodOrder;
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;

@TestMethodOrder(OrderAnnotation.class)
public class CustomTestMethodOrderExample {

    @Test
    @Order(3)
    void testMethod1() {
        // Test logic for method 1
    }
```

```
14
15      @Test
16      @Order(2)
17      void testMethod2() {
18          // Test logic for method 2
19      }
20
21      @Test
22      @Order(1)
23      void testMethod3() {
24          // Test logic for method 3
25      }
26  }
```