JUnit 5

Md. Mohsin Uddin

East West University

mmuddin@ewubd.edu

July 17, 2021

JUnit 5

- Unlike previous versions of JUnit, **JUnit 5** is composed of several different modules from three different sub-projects.
- **JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage**
- The JUnit Platform serves as a foundation for launching testing frameworks on the JVM.
- It also defines the TestEngine API for developing a testing framework that runs on the platform.

JUnit 5

- **JUnit Jupiter** is the combination of the new programming model and extension model for writing tests and extensions in JUnit 5.
- The Jupiter sub-project provides a TestEngine for running Jupiter based tests on the platform.
- **JUnit Vintage** provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform.
- **JUnit 5** requires **Java 8 (or higher)** at runtime.

Writing Tests: A First Test Case

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import example.util.Calculator;
import org.junit.jupiter.api.Test;
class MyFirstJUnitJupiterTests {
    private final Calculator calculator = new Calculator();
    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }

}
```

A Standard Test Class: Part I

```
import static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;
class StandardTests {
    @BeforeAll
    static void initAll() {
    }
    @BeforeEach
    void init() {
    }
    @Test
    void succeedingTest() {}
    @Test
    void failingTest() { fail("a failing test");
```

A Standard Test Class: Part II

```
    }
    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }
    @Test
    void abortedTest() {
        assumeTrue("abc".contains("Z"));
        fail("test should have been aborted");
    }
    @AfterEach
    void tearDown() {
    }
    @AfterAll
    static void tearDownAll() {
    }
}
```

Display Names

- Test classes and test methods can declare custom display names via **@DisplayName** - with spaces, special characters, and even emojis
- These will be displayed in test reports and by test runners and IDEs.

Display Names: Sample Test Class

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

@DisplayName("A special test case")
class DisplayNameDemo {
    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() {
    }
    @Test
    @DisplayName("&4 8 %83$ 3@")
    void testWithDisplayNameContainingSpecialCharacters() {
    }
    @Test
    @DisplayName(":)")
    void testWithDisplayNameContainingEmoji() {
    }
}
```

Assertions

- JUnit Jupiter comes with many of the assertion methods that JUnit 4 has and adds a few that lend themselves well to being used with Java 8 lambdas.
- All JUnit Jupiter assertions are static methods in the org.junit.jupiter.api.Assertions class.

Assertions: Sample Class Part I

```
import static java.time.Duration.ofMillis;
import static java.time.Duration.ofMinutes;
import static org.junit.jupiter.api.Assertions.*;
import java.util.concurrent.CountDownLatch;
import example.domain.Person;
import example.util.Calculator;
import org.junit.jupiter.api.Test;
class AssertionsDemo {
    private final Calculator calculator = new Calculator();
    private final Person person = new Person("Jane", "Doe");
    @Test
    void standardAssertions() {
        assertEquals(2, calculator.add(1, 1));
        assertEquals(4, calculator.multiply(2, 2),
        "optional␣failure␣message␣is␣now␣the␣last␣parameter");
        assertTrue('a' < 'b', "Failure␣Message");
    }
    @Test
    void groupedAssertions() {
```

Assertions: Sample Class Part II

```
        // In a grouped assertion all assertions are executed,
        //and all failures will be reported together.
        assertAll("person",
            () -> assertEquals("Jane", person.getFirstName()),
            () -> assertEquals("Doe", person.getLastName())
        );
    }
    @Test
    void exceptionTesting() {
        Exception exception =
        assertThrows(ArithmeticException.class,
        ()->calculator.divide(1, 0));
        assertEquals("/␣by␣zero", exception.getMessage());
    }
    @Test
    void timeoutNotExceeded() {
        // The following assertion succeeds.
        assertTimeout(ofMinutes(2), () -> {
            // Perform task that takes less than 2 minutes.
```

Assertions: Sample Class Part III

```
        });
    }
    @Test
    void timeoutNotExceededWithResult() {
        // The following assertion succeeds, and returns the su
        String actualResult = assertTimeout(ofMinutes(2),
        () -> {
            return "a␣result";
        });
        assertEquals("a␣result", actualResult);
    }
}
```

Disabling Test Classes

Entire test classes or individual test methods may be disabled via the **@Disabled annotation**, via one of the annotations discussed in **Conditional Test Execution**, or via a **custom ExecutionCondition**.

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

@Disabled("Disabled until bug #99 has been fixed")
class DisabledClassDemo {

    @Test
    void testWillBeSkipped() {
    }

}
```

Disabling Test Methods

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

class DisabledTestsDemo {

    @Disabled("Disabled until bug #42 has been resolved")
    @Test
    void testWillBeSkipped() {
    }

    @Test
    void testWillBeExecuted() {
    }
}
```

Conditional Test Execution: Operating System Conditions

```
@Test
@EnabledOnOs(MAC)
void onlyOnMacOs() {
    // ...
}
@TestOnMac
void testOnMac() {
    // ...
}
@Test
@EnabledOnOs({ LINUX, MAC })
void onLinuxOrMac() {
    // ...
}
@Test
@DisabledOnOs(WINDOWS)
void notOnWindows() {
    // ...
}
```

Conditional Test Execution: Java Runtime Environment Conditions: Part I

```
@Test
@EnabledOnJre(JAVA_8)
void onlyOnJava8() {
    // ...
}
@Test
@EnabledOnJre({ JAVA_9, JAVA_10 })
void onJava9Or10() {
    // ...
}
@Test
@EnabledForJreRange(min = JAVA_9, max = JAVA_11)
void fromJava9to11() {
    // ...
}
@Test
@EnabledForJreRange(min = JAVA_9)
```

Conditional Test Execution: Java Runtime Environment Conditions: Part II

```
void fromJava9toCurrentJavaFeatureNumber() {
    // ...
}
@Test
@EnabledForJreRange(max = JAVA_11)
void fromJava8To11() {
    // ...
}
@Test
@DisabledOnJre(JAVA_9)
void notOnJava9() {
    // ...
}
@Test
@DisabledForJreRange(min = JAVA_9, max = JAVA_11)
void notFromJava9to11() {
    // ...
```

Conditional Test Execution: Java Runtime Environment Conditions: Part III

```
}
@Test
@DisabledForJreRange(min = JAVA_9)
void notFromJava9toCurrentJavaFeatureNumber(){
    // ...
}
@Test
@DisabledForJreRange(max = JAVA_11)
void notFromJava8to11() {
    // ...
}
```

Conditional Test Execution: Custom Conditions

A test may be enabled or disabled based on the boolean return of a method via the **@EnabledIf** and **@DisabledIf** annotations.

```
@Test
@EnabledIf("customCondition")
void enabled() {
    // ...
}

@Test
@DisabledIf("customCondition")
void disabled() {
    // ...
}

boolean customCondition() {
    return true;
}
```

Test Execution Order

- Although true unit tests typically should not rely on the order in which they are executed, there are times when it is necessary to enforce a specific test method execution order.
- For example, when writing **integration tests** where the sequence of the tests is important.
- To control the order in which test methods are executed, annotate your test class or test interface with **@TestMethodOrder** and specify the desired **MethodOrderer** implementation.

Test Execution Order

- You can implement your own **custom MethodOrderer** or use one of the following **built-in MethodOrderer** implementations.
- **DisplayName:** sorts test methods alphanumerically based on their display names.
- **MethodName:** sorts test methods alphanumerically based on their method name and formal parameter lists.
- **OrderAnnotation:** sorts test methods numerically based on values specified via the @Order annotation.
- **Alphanumeric:** sorts test methods alphanumerically based on their names and formal parameter lists.

Test Execution Order

```
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;
@TestMethodOrder(OrderAnnotation.class)
class OrderedTestsDemo {
    @Test
    @Order(1)
    void nullValues(){//perform assertions against null values}
    @Test
    @Order(2)
    void emptyValues(){
    //perform assertions against empty values }
    @Test
    @Order(3)
    void validValues(){
    //perform assertions against valid values }
}
```

Repeated Tests

JUnit Jupiter provides the ability to repeat a test a specified number of times by annotating a method with **@RepeatedTest** and specifying the total number of repetitions desired.

```
@RepeatedTest(10)
void repeatedTest() {
    // ...
}
```

Parameterized Tests

- **Parameterized tests** make it possible to run a test multiple times with different arguments.
- They are declared just like regular @Test methods but use the **@ParameterizedTest** annotation instead.
- In addition, you must declare at least one source that will provide the arguments for each invocation and then consume the arguments in the test method.
- The following example demonstrates a parameterized test that uses the **@ValueSource** annotation to specify a String array as the source of arguments.

```java
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar",
"able␣was␣I␣ere␣I␣saw␣elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

Parameterized Tests: @ValueSource

The following types of literal values are supported by **@ValueSource**.

- short
- byte
- int
- long
- float
- double
- char
- boolean
- java.lang.String
- java.lang.Class

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertTrue(argument > 0 && argument < 4);
}
```

Parameterized Tests: @ValueSource

```
@ParameterizedTest
@NullSource
@EmptySource
@ValueSource(strings = { "␣", "␣␣␣", "\t", "\n" })
void nullEmptyAndBlankStrings(String text) {
    assertTrue(text == null || text.trim().isEmpty());
}
@ParameterizedTest
@NullAndEmptySource
@ValueSource(strings = { "␣", "␣␣␣", "\t", "\n" })
void nullEmptyAndBlankStrings(String text) {
    assertTrue(text == null || text.trim().isEmpty());
}
@ParameterizedTest
@EnumSource(ChronoUnit.class)
void testWithEnumSource(TemporalUnit unit) {
    assertNotNull(unit);
}
```

Parameterized Tests: @MethodSource

```
@ParameterizedTest
@MethodSource("stringProvider")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull(argument);
}
static Stream<String> stringProvider() {
    return Stream.of("apple", "banana");
}
@ParameterizedTest
@MethodSource
void testWithDefaultLocalMethodSource(String argument) {
    assertNotNull(argument);
}
static Stream<String> testWithDefaultLocalMethodSource() {
    return Stream.of("apple", "banana");
}
```

Parameterized Tests: @CsvSource

```
@ParameterizedTest
@CsvSource({
    "apple,1",
    "banana,2",
    "'lemon,␣lime',0xF1"
})
void testWithCsvSource(String fruit, int rank) {
    assertNotNull(fruit);
    assertNotEquals(0, rank);
}
```

Parameterized Tests: @CsvFileSource

Suppose, the contents of the two-column.csv file are as follows:

Country, reference

Sweden, 1

Poland, 2

"United States of America", 3

```
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv",
numLinesToSkip = 1)
void testWithCsvFileSourceFromClasspath(String country,
int reference) {
    assertNotNull(country);
    assertNotEquals(0, reference);
}
@ParameterizedTest
@CsvFileSource(files = "src/test/resources/two-column.csv",
numLinesToSkip = 1)
void testWithCsvFileSourceFromFile(String country,int ref){
    assertNotNull(country);
    assertNotEquals(0, ref); }
```

Timeouts

- The **@Timeout** annotation allows one to declare that a test method should fail if its execution time exceeds a given duration.
- The time unit for the duration **defaults to seconds** but is configurable.

```
class TimeoutDemo {
    @BeforeEach
    @Timeout(5)
    void setUp() {
        // fails if execution time exceeds 5 seconds
    }
    @Test
    @Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
    void failsIfExecutionTimeExceeds100Milliseconds() {
        // fails if execution time exceeds 100 milliseconds
    }
}
```

References

https://junit.org/junit5/docs/current/user-guide/#overview