Junit Framework

What is 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***

TestEngine for running JUnit 3 and JUnit 4

foundation for launching testing frameworks on the JVM.

combination of the programming model and extension model for writing tests and extensions

Sample POM

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>birzeit.edu</groupId>
<artifactId>tdd-example</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>tdd-example </name>
<description> </description>
<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<maven.compiler.source>1.8</maven.compiler.source>
<maven.compiler.target>${maven.compiler.source}</maven.compiler.target>
<junit.jupiter.version>5.8.1</junit.jupiter.version>
<junit.platform.version>1.8.1</junit.platform.version>
</properties>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.13.2</version>
<scope>test</scope>
</dependency>
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-engine</artifactId>
<version>${junit.jupiter.version}</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-api</artifactId>
<version>${junit.jupiter.version}</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.junit.platform</groupId>
<artifactId>junit-platform-runner</artifactId>
<version>${junit.platform.version}</version>
<scope>test</scope>
</dependency>
</dependencies>
</project>
```

Test Fixture

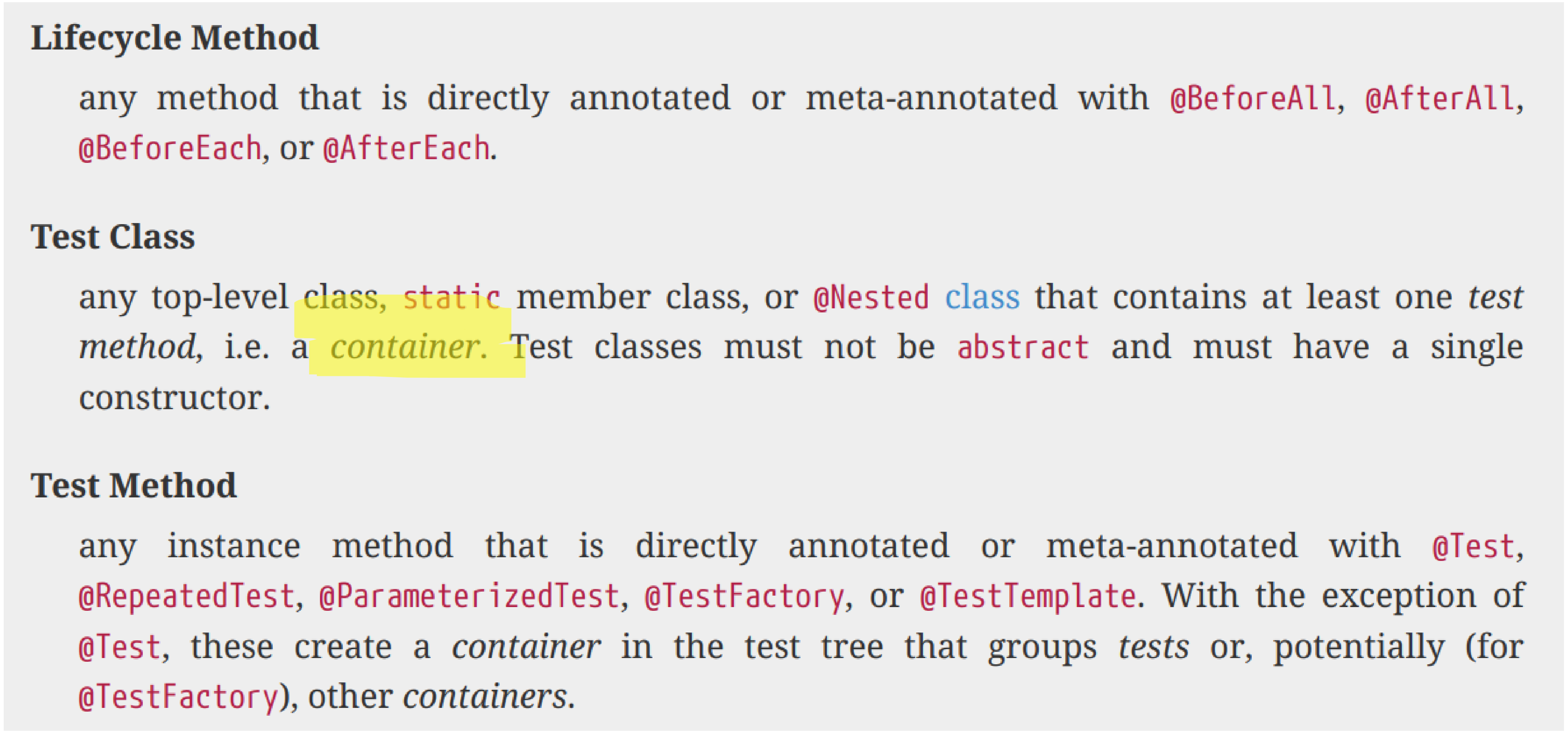
A test fixture is a context where a JUnit Test Case runs. Typically, test fixtures include:

- Objects or resources that are available for any test case.
- Activities required that makes these objects/resources available.
- These activities are
  - allocation (**setup**)
  - de-allocation (**teardown**).

Setup and Teardown

- Usually, there are some repeated tasks that must be done prior to each test case. **Example:** create a database connection.
- Likewise, at the end of each test case, there may be some repeated tasks. **Example:** to clean up once test execution is over.
- JUnit provides annotations that help in setup and teardown. It ensures that resources are released, and the test system is in a ready state for next test case.
- These JUnit annotations are discussed below-

Jupiter Concepts



Lifecycle Method

any method that is directly annotated or meta-annotated with `@BeforeAll`, `@AfterAll`, `@BeforeEach`, or `@AfterEach`.

Test Class

any top-level class, `static` member class, or `@Nested class` that contains at least one *test method*, i.e. a *container*. Test classes must not be `abstract` and must have a single constructor.

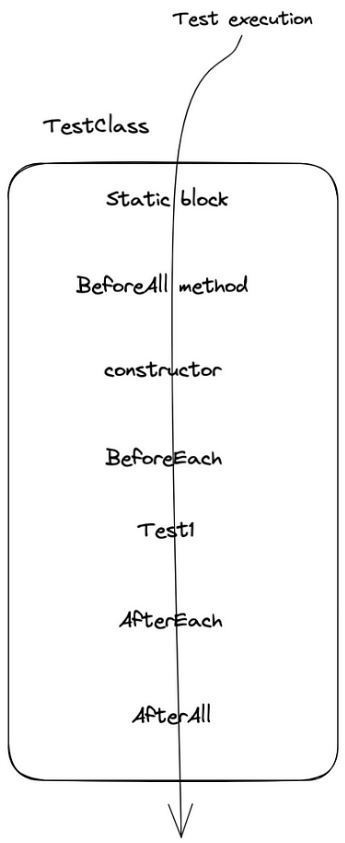
Test Method

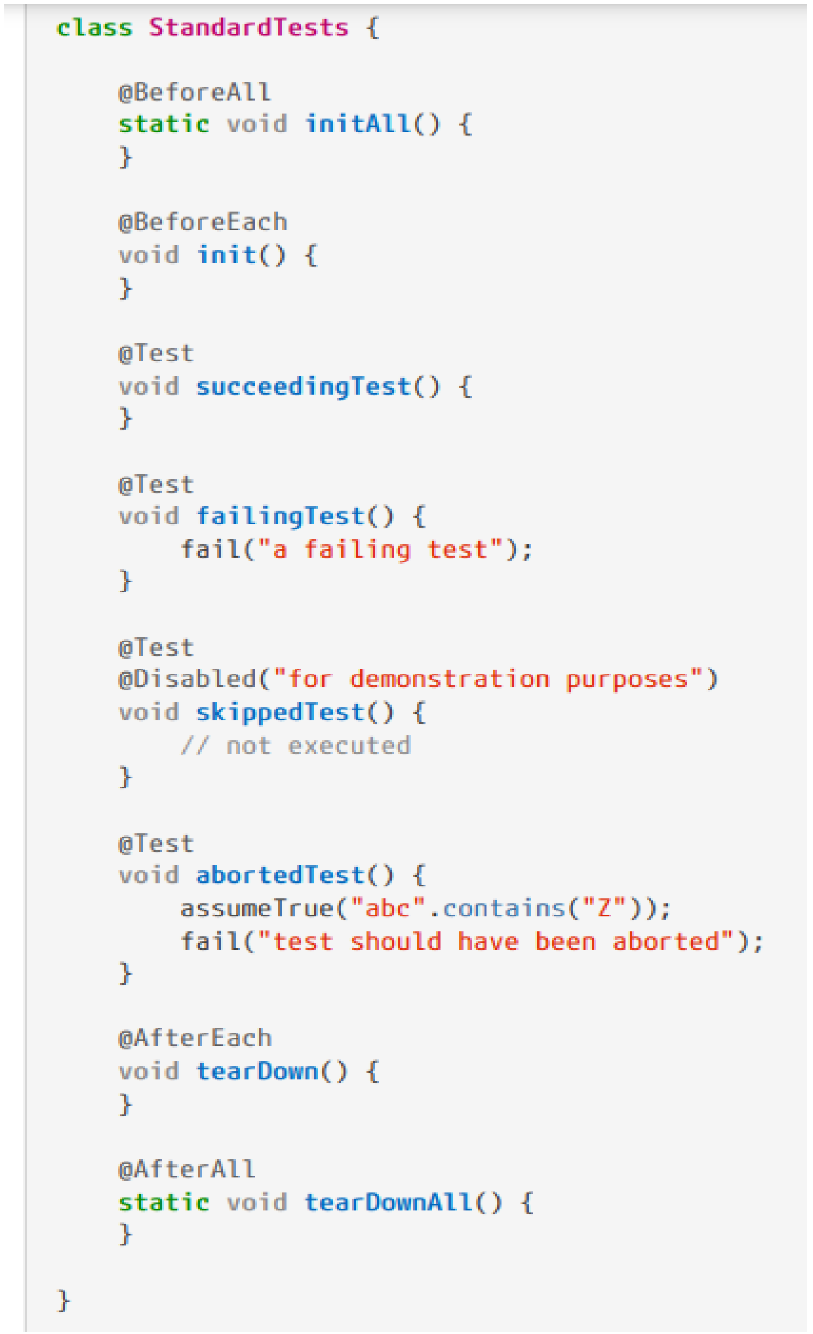
any instance method that is directly annotated or meta-annotated with `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, or `@TestTemplate`. With the exception of `@Test`, these create a *container* in the test tree that groups *tests* or, potentially (for `@TestFactory`), other *containers*.

Test Classes and Methods

- Test methods and lifecycle methods may be declared locally within the current test class, inherited from superclasses, or inherited from interfaces (see Test Interfaces and Default Methods).
- In addition, test methods and lifecycle methods must not be abstract and must not return a value (except @TestFactory methods which are required to return a value).

Class and method visibility Test classes, test methods, and lifecycle methods are not required to be public, but they must not be private.

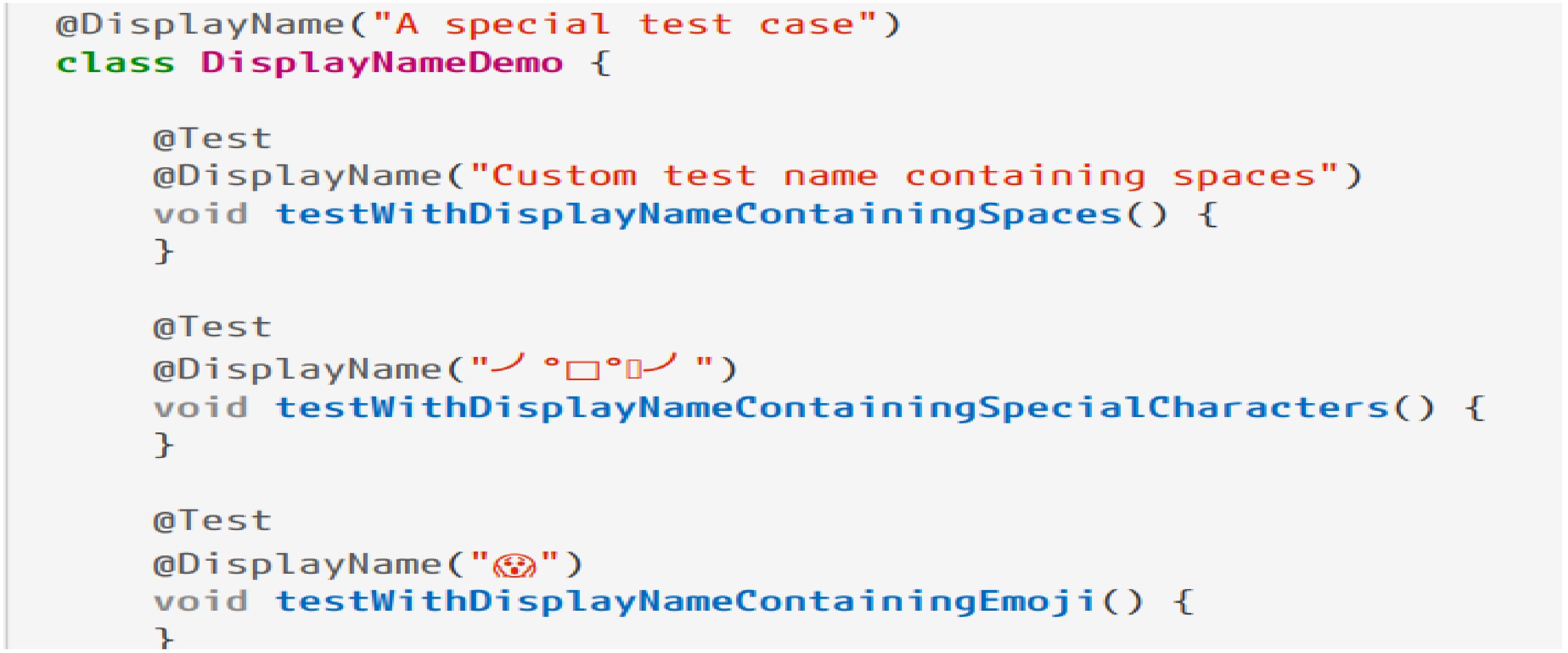




```
class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

    @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 — that will be displayed in test reports and by test runners and IDEs



```
@DisplayName("A special test case")
class DisplayNameDemo {

    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() {
    }

    @Test
    @DisplayName("╯°□°▯╯")
    void testWithDisplayNameContainingSpecialCharacters() {
    }

    @Test
    @DisplayName("😱")
    void testWithDisplayNameContainingEmoji() {
    }
```

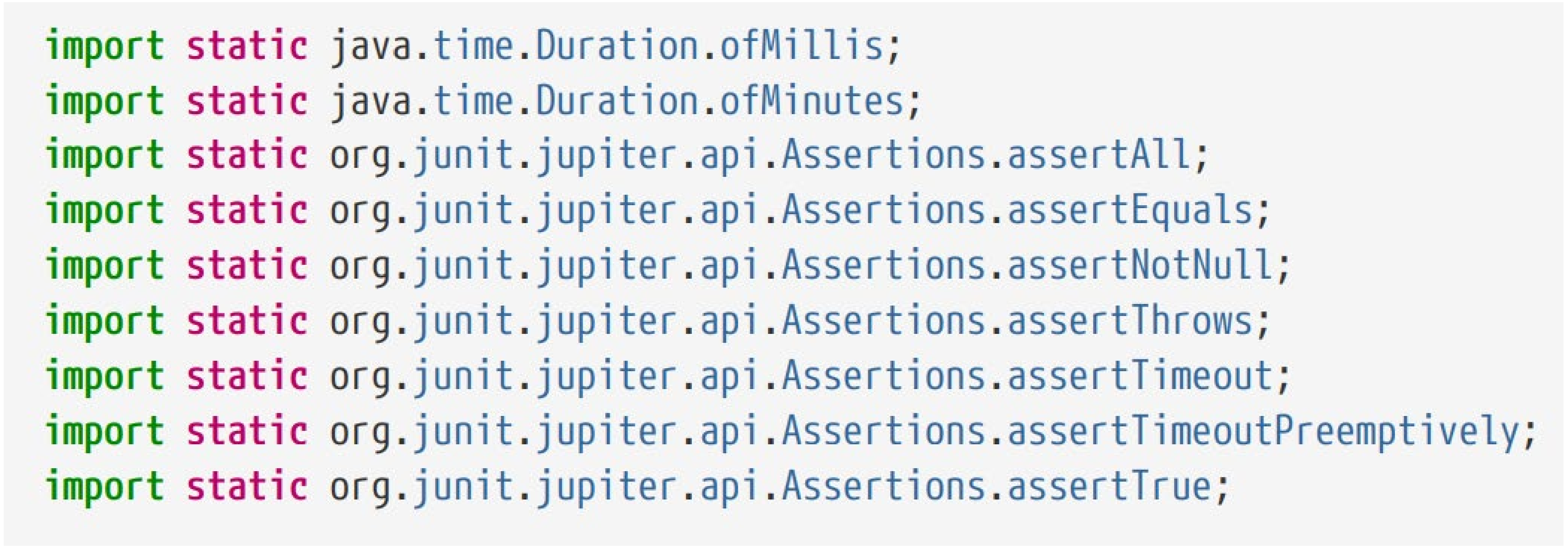
Write test cases that generates the following output?

```
+-- DisplayNameGeneratorDemo [OK]
 +-- A year is not supported [OK]
 | +-- A negative value for year is not supported by the leap year computation. [OK]
 | | +-- For example, year -1 is not supported. [OK]
 | | '-- For example, year -4 is not supported. [OK]
 | '-- if it is zero() [OK]
 '-- A year is a leap year [OK]
   +-- A year is a leap year -> if it is divisible by 4 but not by 100. [OK]
   '-- A year is a leap year -> if it is one of the following years. [OK]
     +-- Year 2016 is a leap year. [OK]
     +-- Year 2020 is a leap year. [OK]
     '-- Year 2048 is a leap year. [OK]
```

```
    @Nested
    @DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
    class A_year_is_not_supported {

        @Test
        void if_it_is_zero() {
        }

        @DisplayName("A negative value for year is not supported by the leap year
computation.")
        @ParameterizedTest(name = "For example, year {0} is not supported.")
        @ValueSource(ints = { -1, -4 })
        void if_it_is_negative(int year) {
        }

    }

    @Nested
    @IndicativeSentencesGeneration(separator = " -> ", generator =
DisplayNameGenerator.ReplaceUnderscores.class)
    class A_year_is_a_leap_year {

        @Test
        void if_it_is_divisible_by_4_but_not_by_100() {
        }

        @ParameterizedTest(name = "Year {0} is a leap year.")
        @ValueSource(ints = { 2016, 2020, 2048 })
        void if_it_is_one_of_the_following_years(int year) {
        }

    }
```

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.



```
import static java.time.Duration.ofMillis;
import static java.time.Duration.ofMinutes;
import static org.junit.jupiter.api.Assertions.assertAll;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTimeout;
import static org.junit.jupiter.api.Assertions.assertTimeoutPreemptively;
import static org.junit.jupiter.api.Assertions.assertTrue;
```

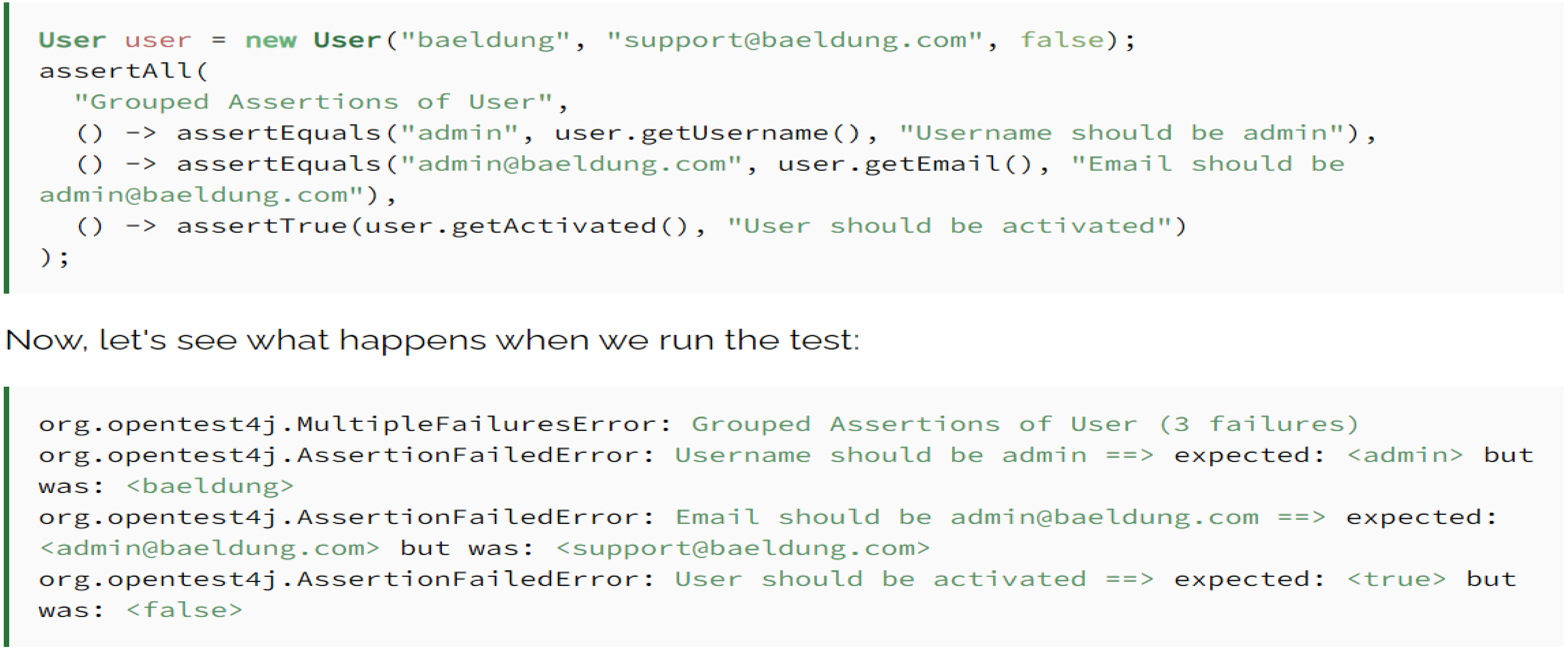
Using Multiple Assertions

```
User user = new User("baeldung", "support@baeldung.com", false);
assertEquals("admin", user.getUsername(), "Username should be admin");
assertEquals("admin@baeldung.com", user.getEmail(), "Email should be admin@baeldung.com");
assertTrue(user.getActivated(), "User should be activated");
```

- all of our assertions would fail:After running the test only the first assertion fails:
- org.opentest4j.AssertionFailedError: Username should be admin ==>
- Expected :admin
- Actual   :baeldung

Let's say we fix the failing code or test and re-run the test. We'd then get a second failure, and so on. **It would be better, in this situation, to group all these assertions into a single pass/failure.**

assertAll



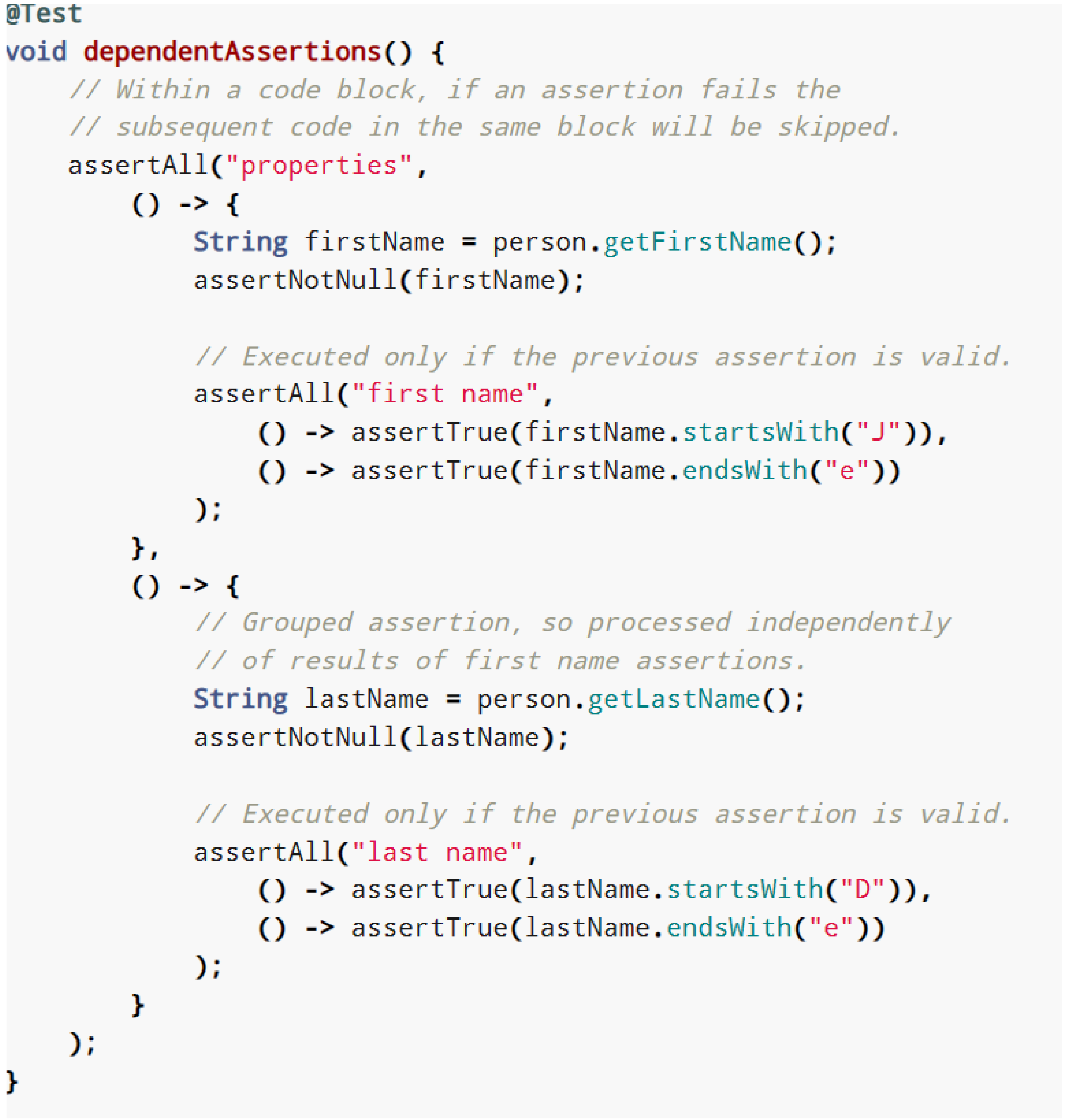
```
User user = new User("baeldung", "support@baeldung.com", false);
assertAll(
  "Grouped Assertions of User",
  () -> assertEquals("admin", user.getUsername(), "Username should be admin"),
  () -> assertEquals("admin@baeldung.com", user.getEmail(), "Email should be
admin@baeldung.com"),
  () -> assertTrue(user.getActivated(), "User should be activated")
);
```

Now, let's see what happens when we run the test:

```
org.opentest4j.MultipleFailuresError: Grouped Assertions of User (3 failures)
org.opentest4j.AssertionFailedError: Username should be admin ==> expected: <admin> but
was: <baeldung>
org.opentest4j.AssertionFailedError: Email should be admin@baeldung.com ==> expected:
<admin@baeldung.com> but was: <support@baeldung.com>
org.opentest4j.AssertionFailedError: User should be activated ==> expected: <true> but
was: <false>
```

https://www.baeldung.com/junit5-assertall-vs-multiple-assertions

Dependent Assertion



```java
@Test
void dependentAssertions() {
    // Within a code block, if an assertion fails the
    // subsequent code in the same block will be skipped.
    assertAll("properties",
        () -> {
            String firstName = person.getFirstName();
            assertNotNull(firstName);

            // Executed only if the previous assertion is valid.
            assertAll("first name",
                () -> assertTrue(firstName.startsWith("J")),
                () -> assertTrue(firstName.endsWith("e"))
            );
        },
        () -> {
            // Grouped assertion, so processed independently
            // of results of first name assertions.
            String lastName = person.getLastName();
            assertNotNull(lastName);

            // Executed only if the previous assertion is valid.
            assertAll("last name",
                () -> assertTrue(lastName.startsWith("D")),
                () -> assertTrue(lastName.endsWith("e"))
            );
        }
    );
}
```

```java
    @Test
    void timeoutNotExceededWithMethod() {
        // The following assertion invokes a method reference and returns an object.
        String actualGreeting = assertTimeout(ofMinutes(2), AssertionsDemo::greeting);
        assertEquals("Hello, World!", actualGreeting);
    }

    @Test
    void timeoutExceeded() {
        // The following assertion fails with an error message similar to:
        // execution exceeded timeout of 10 ms by 91 ms
        assertTimeout(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms.
            Thread.sleep(100);
        });
    }

    @Test
    void timeoutExceededWithPreemptiveTermination() {
        // The following assertion fails with an error message similar to:
        // execution timed out after 10 ms
        assertTimeoutPreemptively(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms.
            new CountDownLatch(1).await();
        });
    }
```

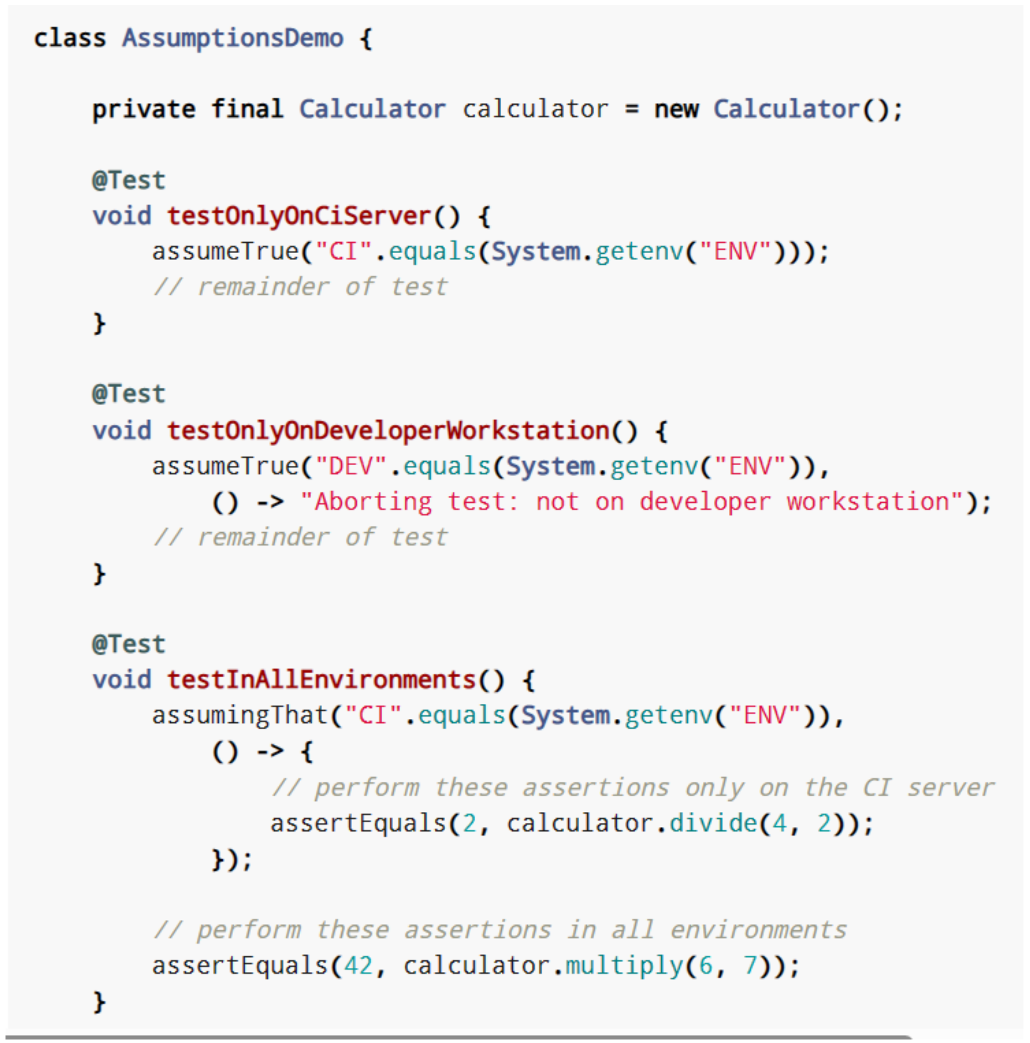
Assumptions

JUnit 5 assumptions class provides **static** methods to support conditional test execution based on assumptions. A failed assumption results in a test being aborted.
Assumptions are typically used whenever it does not make sense to continue the execution of a given test method. In the test report, these tests will be marked as passed.
JUnit Jupiter **Assumptions** class has the following methods:

- assumeFalse()
- assumeTrue()
- assumingThat()

Assumptions

Failed assumptions do not result in a test *failure*; rather, a failed assumption results in a test being *aborted*.

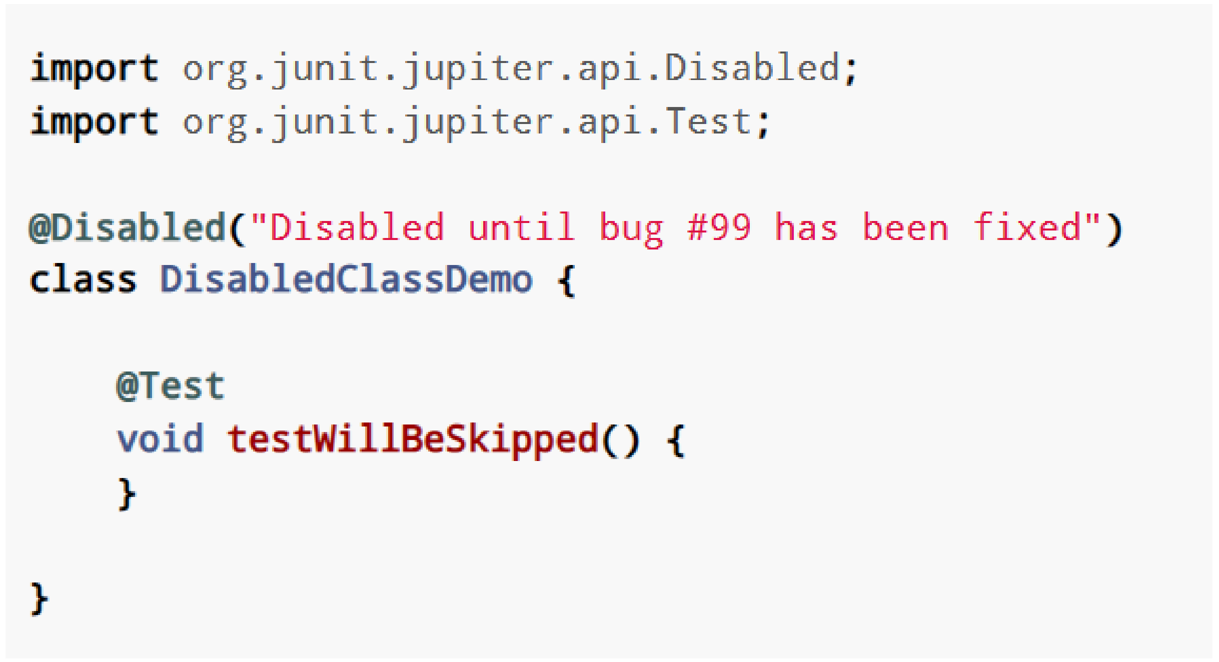


```
class AssumptionsDemo {

    private final Calculator calculator = new Calculator();

    @Test
    void testOnlyOnCiServer() {
        assumeTrue("CI".equals(System.getenv("ENV")));
        // remainder of test
    }

    @Test
    void testOnlyOnDeveloperWorkstation() {
        assumeTrue("DEV".equals(System.getenv("ENV")),
            () -> "Aborting test: not on developer workstation");
        // remainder of test
    }

    @Test
    void testInAllEnvironments() {
        assumingThat("CI".equals(System.getenv("ENV")),
            () -> {
                // perform these assertions only on the CI server
                assertEquals(2, calculator.divide(4, 2));
            });

        // perform these assertions in all environments
        assertEquals(42, calculator.multiply(6, 7));
    }
```

1. `Assumptions.assumeTrue()` – If the condition is true, then run the test, else aborting the test.
2. `Assumptions.false()` – If the condition is false, then run the test, else aborting the test.
3. `Assumptions.assumingThat()` – is much more flexible, If condition is true then executes, else do not abort test continue rest of code in test.

Disabling Tests

@Disabled may be declared without providing a *reason*; however, the JUnit team recommends that developers provide a short explanation for why a test class or test method has been disabled. Some development teams even require the presence of **issue tracking numbers** in the *reason* for automated traceability, etc.



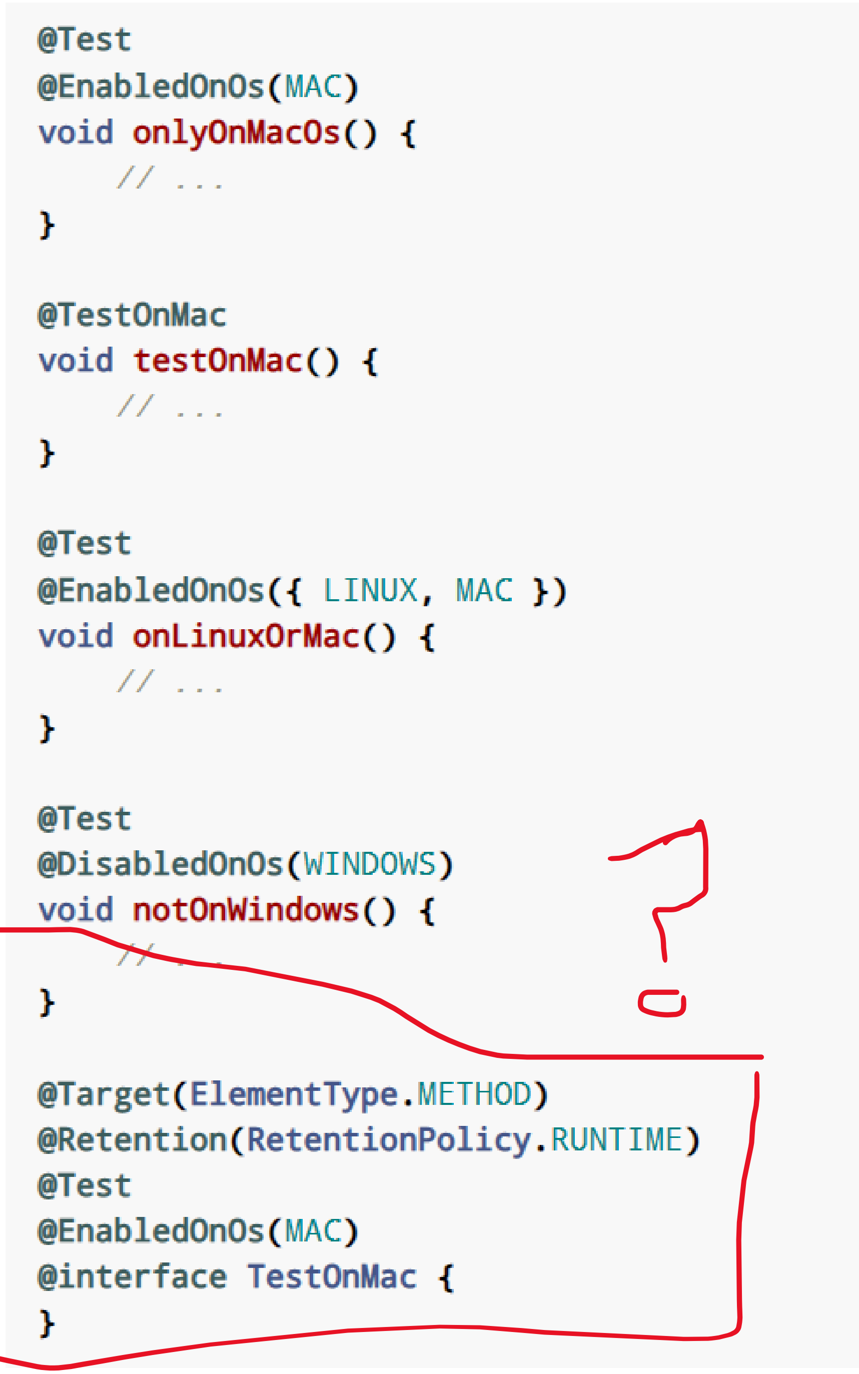
```java
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

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

    @Test
    void testWillBeSkipped() {
    }

}
```

Conditional Test Execution

The ExecutionCondition extension API in JUnit Jupiter allows developers to either *enable* or *disable* a container or test based on certain conditions *programmatically*. The simplest example of such a condition is the built-in DisabledCondition which supports the @Disabled annotation (see Disabling Tests). In addition to @Disabled, JUnit Jupiter also supports several other annotation-based conditions in the org.junit.jupiter.api.condition package that allow developers to enable or disable containers and tests *declaratively*. When multiple ExecutionCondition extensions are registered, a container or test is disabled as soon as one of the conditions returns *disabled*.

Operating System Conditions

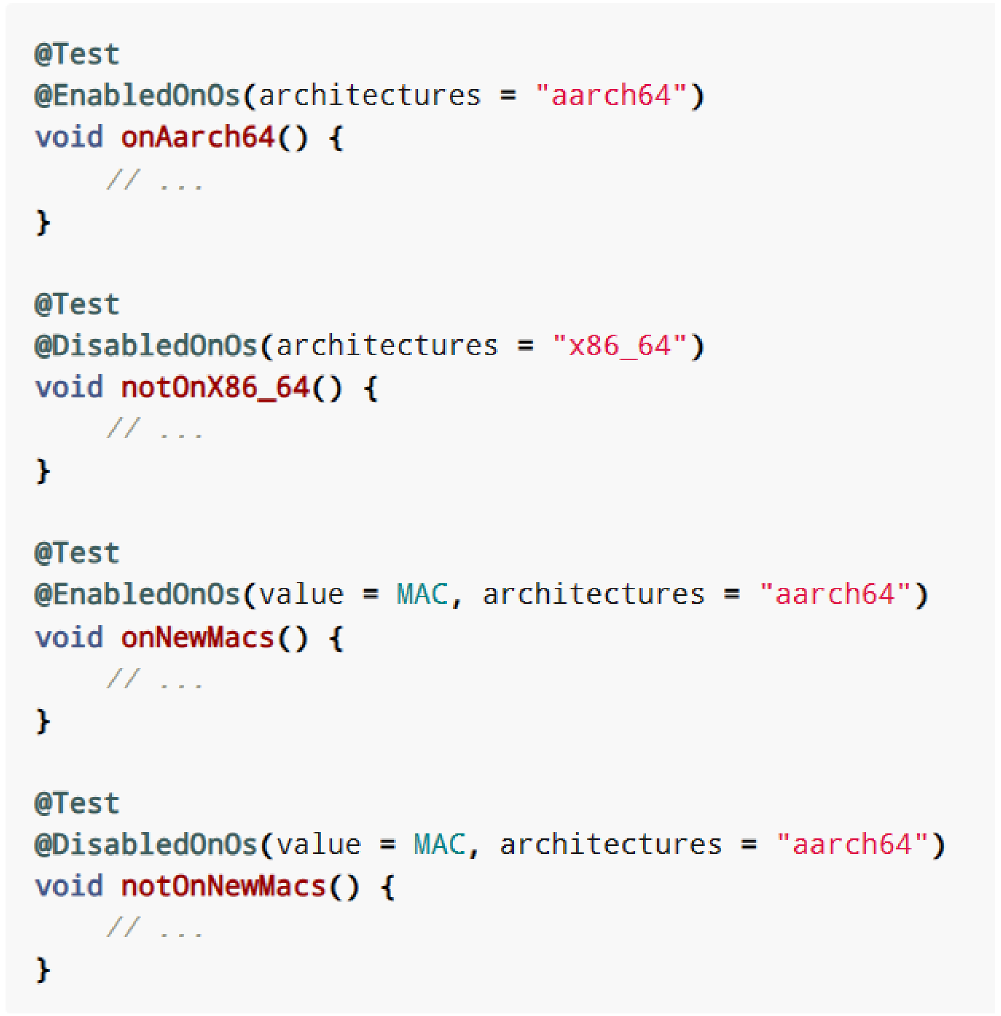
- As an exercise, understand find the custom annotation and understand how it is defined!



```
@Test
@EnabledOnOs(MAC)
void onlyOnMacOs() {
    // ...
}

@TestOnMac
void testOnMac() {
    // ...
}

@Test
@EnabledOnOs({ LINUX, MAC })
void onLinuxOrMac() {
    // ...
}

@Test
@DisabledOnOs(WINDOWS)
void notOnWindows() {
    // ...
}

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Test
@EnabledOnOs(MAC)
@interface TestOnMac {
}
```

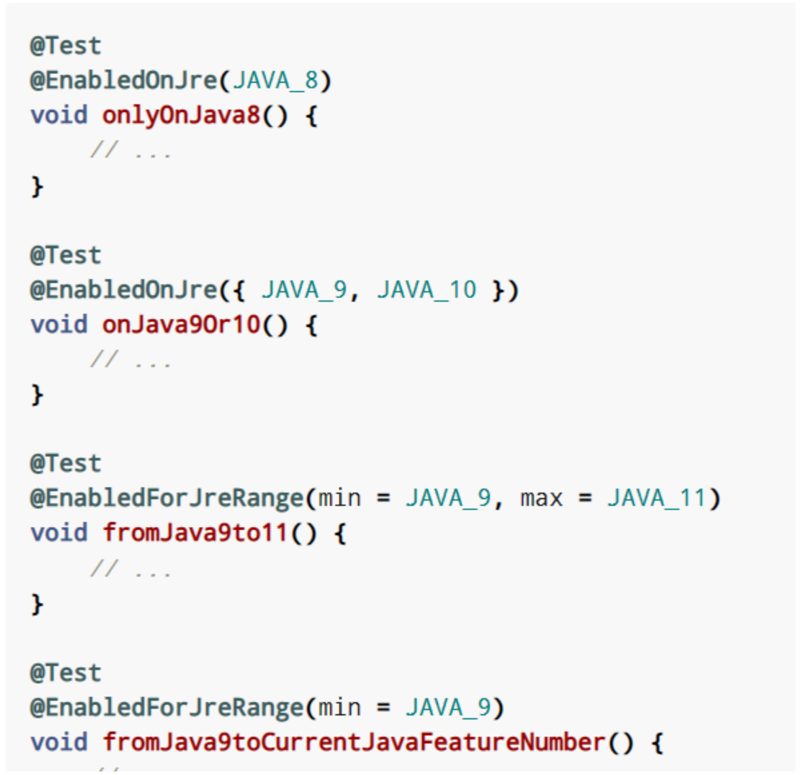
?

Architecture Conditions



```
@Test
@EnabledOnOs(architectures = "aarch64")
void onAarch64() {
    // ...
}

@Test
@DisabledOnOs(architectures = "x86_64")
void notOnX86_64() {
    // ...
}

@Test
@EnabledOnOs(value = MAC, architectures = "aarch64")
void onNewMacs() {
    // ...
}

@Test
@DisabledOnOs(value = MAC, architectures = "aarch64")
void notOnNewMacs() {
    // ...
}
```

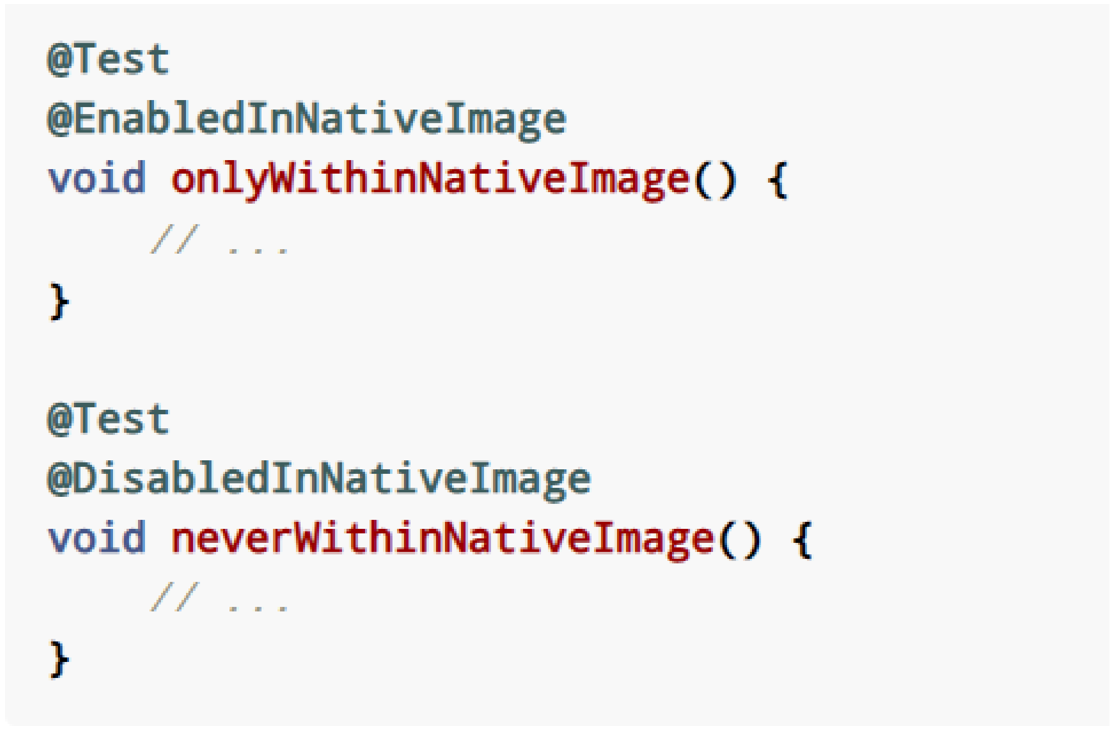
Java Runtime Environment Conditions



```java
@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)
void fromJava9toCurrentJavaFeatureNumber() {
```

Native Image Conditions

- What is a native executable?
- A native executable is created by the Native Image builder or native-image that processes your application classes and other metadata to create a binary for a specific operating system and architecture.

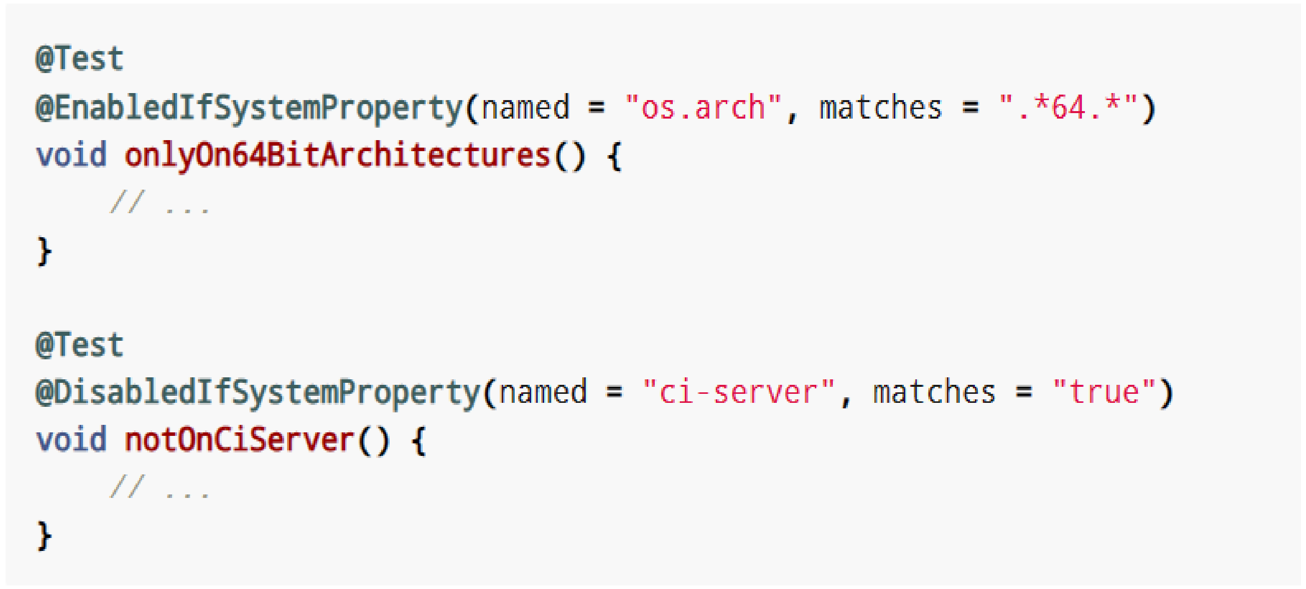


```
@Test
@EnabledInNativeImage
void onlyWithinNativeImage() {
    // ...
}

@Test
@DisabledInNativeImage
void neverWithinNativeImage() {
    // ...
}
```

System Property Conditions

You should be able to differentiate between

- Environmental variable (OS)
- System property for VM (VM)
- Program arguments (Program)



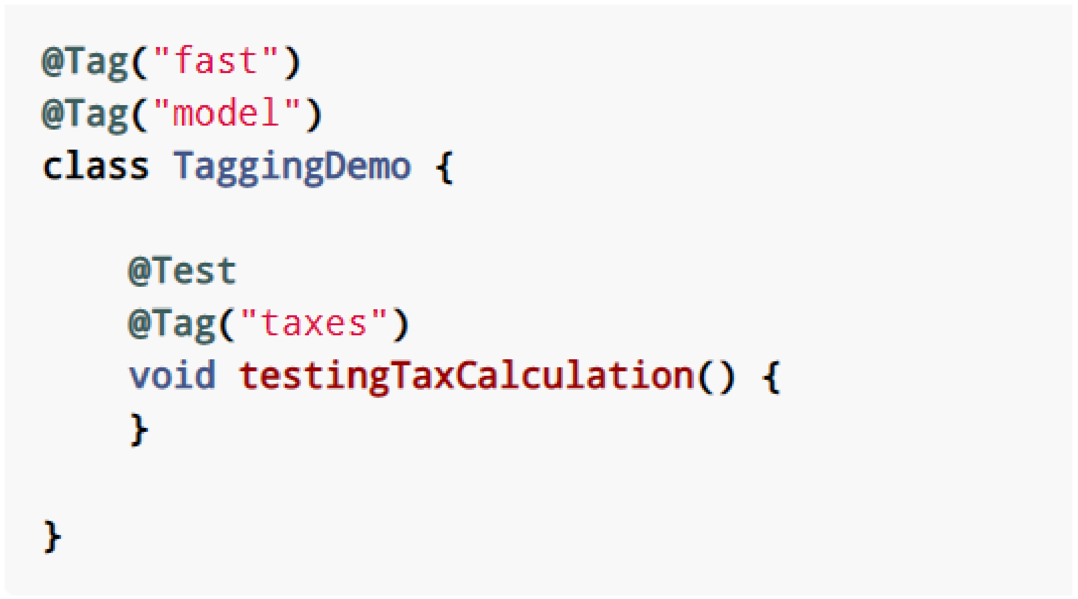
```
@Test
@EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
void onlyOn64BitArchitectures() {
    // ...
}

@Test
@DisabledIfSystemProperty(named = "ci-server", matches = "true")
void notOnCiServer() {
    // ...
}
```

Environment Variable Conditions

```java
@Test
@EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server")
void onlyOnStagingServer() {
    // ...
}

@Test
@DisabledIfEnvironmentVariable(named = "ENV", matches = ".*development.*")
void notOnDeveloperWorkstation() {
    // ...
}
```

Tagging and Filtering

Test classes and methods can be tagged via the @Tag annotation. Those tags can later be used to filter test discovery and execution. Please refer to the Tags section for more information about tag support in the JUnit Platform.



```java
@Tag("fast")
@Tag("model")
class TaggingDemo {

    @Test
    @Tag("taxes")
    void testingTaxCalculation() {
    }

}
```

Test Execution Order

- Method Order

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. You can implement your own custom MethodOrderer or use one of the following built-in MethodOrderer implementations.

- MethodOrderer.DisplayName: sorts test methods *alphanumerically* based on their display names (see display name generation precedence rules)
- MethodOrderer.MethodName: sorts test methods *alphanumerically* based on their names and formal parameter lists
- MethodOrderer.OrderAnnotation: sorts test methods *numerically* based on values specified via the @Order annotation
- MethodOrderer.Random: orders test methods *pseudo-randomly* and supports configuration of a custom *seed*
- MethodOrderer.Alphanumeric: sorts test methods *alphanumerically* based on their names and formal parameter lists; **deprecated in favor of** MethodOrderer.MethodName**, to be removed in 6.0**

```java
@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
    }

}
```

Class Order

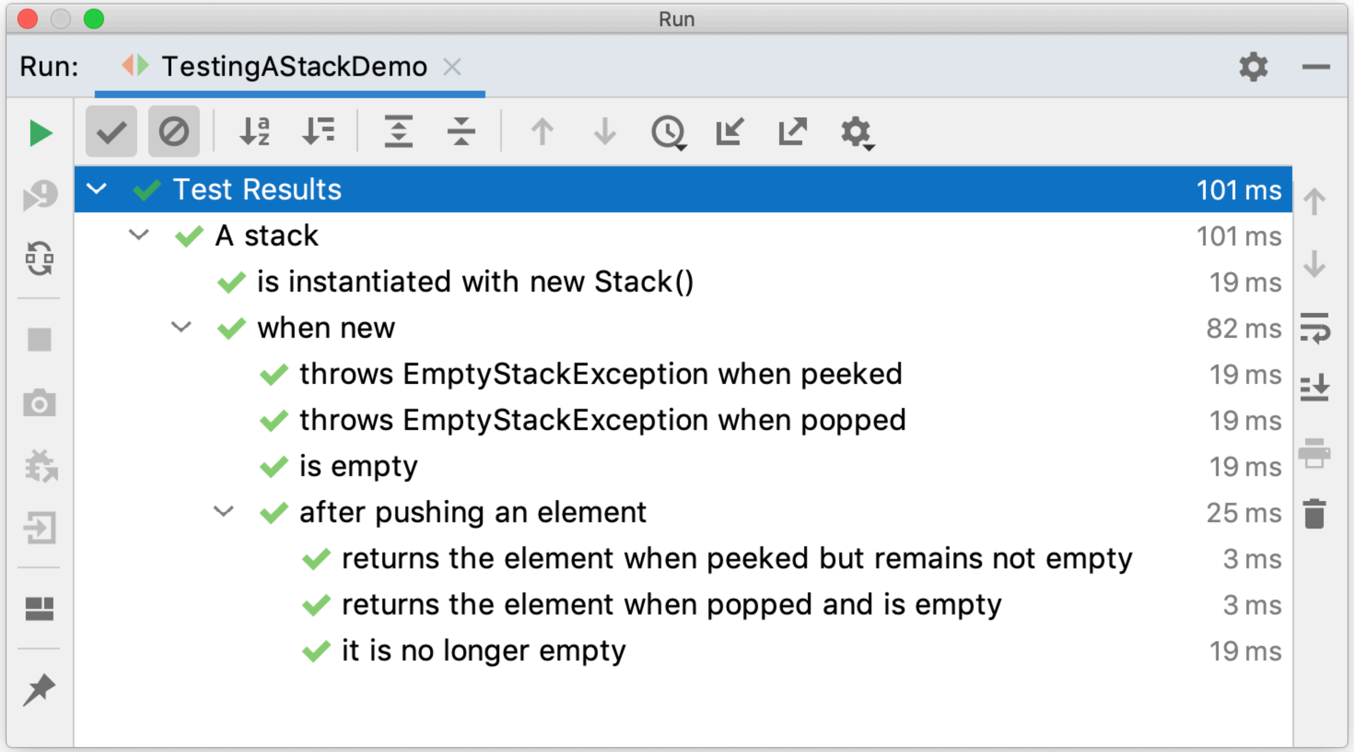
Although test classes typically should not rely on the order in which they are executed, there are times when it is desirable to enforce a specific test class execution order. You may wish to execute test classes in a random order to ensure there are no accidental dependencies between test classes, or you may wish to order test classes to optimize build time as outlined in the following scenarios.

- Run previously failing tests and faster tests first: "fail fast" mode
- With parallel execution enabled, run longer tests first: "shortest test plan execution duration" mode
- Various other use cases

```java
@TestClassOrder(ClassOrderer.OrderAnnotation.class)
class OrderedNestedTestClassesDemo {

    @Nested
    @Order(1)
    class PrimaryTests {

        @Test
        void test1() {
        }
    }

    @Nested
    @Order(2)
    class SecondaryTests {

        @Test
        void test2() {
        }
    }
}
```

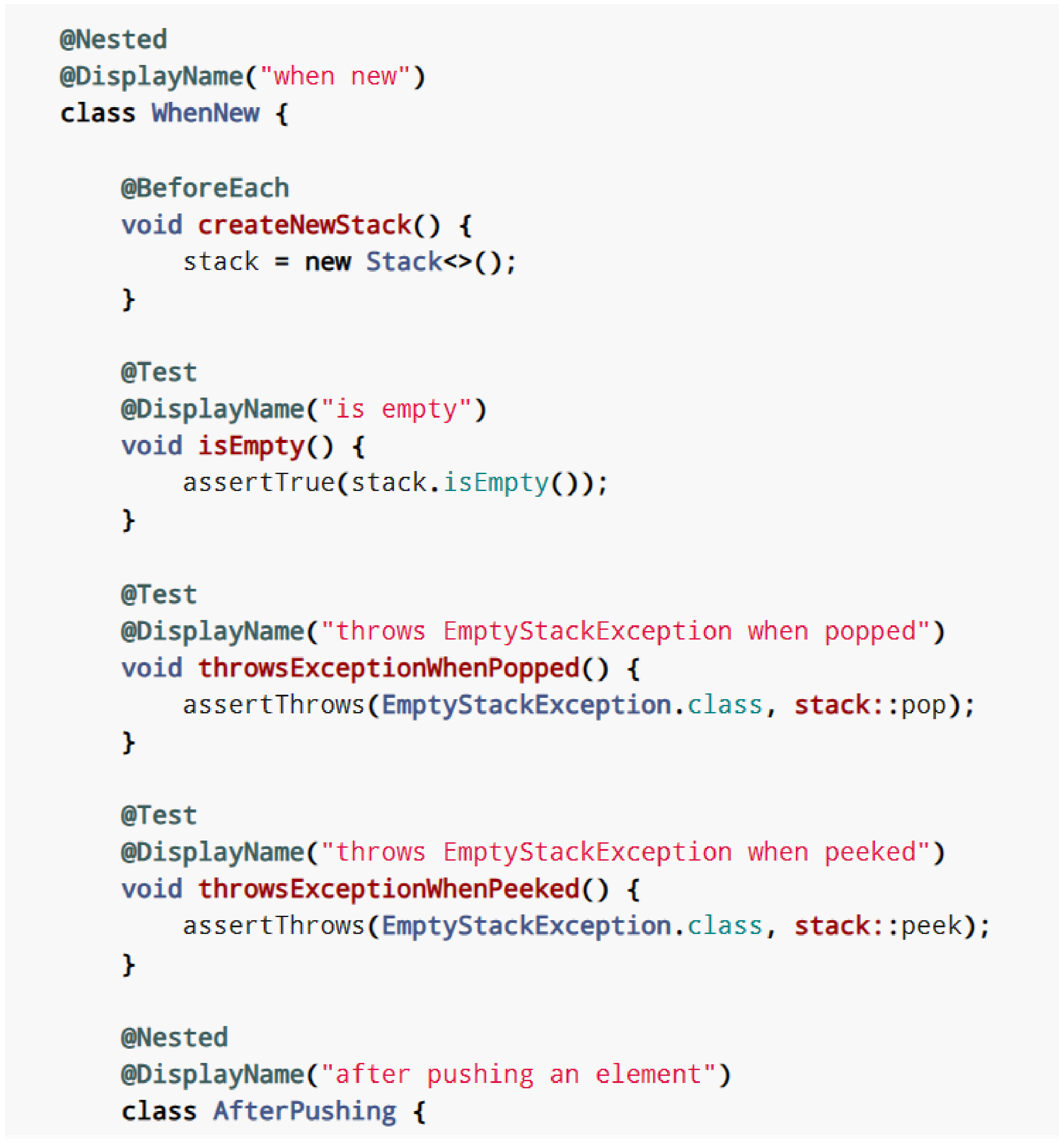
Nested Tests

@Nested tests give the test writer more capabilities to express the relationship among several groups of tests. Such nested tests make use of Java's nested classes and facilitate hierarchical thinking about the test structure.



```
Test Results                                                101 ms
  A stack                                                   101 ms
    is instantiated with new Stack()                         19 ms
    when new                                                 82 ms
      throws EmptyStackException when peeked                 19 ms
      throws EmptyStackException when popped                 19 ms
      is empty                                               19 ms
      after pushing an element                               25 ms
        returns the element when peeked but remains not empty  3 ms
        returns the element when popped and is empty          3 ms
        it is no longer empty                                19 ms
```

Nested Example



```
@Nested
@DisplayName("when new")
class WhenNew {

    @BeforeEach
    void createNewStack() {
        stack = new Stack<>();
    }

    @Test
    @DisplayName("is empty")
    void isEmpty() {
        assertTrue(stack.isEmpty());
    }

    @Test
    @DisplayName("throws EmptyStackException when popped")
    void throwsExceptionWhenPopped() {
        assertThrows(EmptyStackException.class, stack::pop);
    }

    @Test
    @DisplayName("throws EmptyStackException when peeked")
    void throwsExceptionWhenPeeked() {
        assertThrows(EmptyStackException.class, stack::peek);
    }

    @Nested
    @DisplayName("after pushing an element")
    class AfterPushing {
```

Setup and Teardown:Syntax

- **Setup**
- **@Before** annotation in JUnit is used on a method containing Java code to run before each test case. i.e it runs before each test execution.
- **Teardown (regardless of the verdict)**
- **@After** annotation is used on a method containing java code to run after each test case. These methods will run even if any exceptions are thrown in the test case or in the case of assertion failures

Common Use Cases for beforeEach and afterEach

- **Mocking out external services**
- **Resetting application state**

One common use case for the **beforeEach** and **afterEach** methods is mocking out external services that are used by the component or service being tested.
This can be useful for isolating the unit under test (also called the code under test) and preventing it from making actual network requests or interacting with real database systems

Another common use case for the **beforeEach** and **afterEach** methods is resetting the state of the application before and after each test. This can be useful for ensuring that each unit test is running in a clean environment and is not affected by the state of previous tests.

Example

```
private List<String> list;
@BeforeEach
void init() {
LOG.info("startup");
list = new ArrayList<>(Arrays.asList("test1", "test2"));
}

@AfterEach void teardown() { LOG.info("teardown");
list.clear();
}
```

@BeforAll

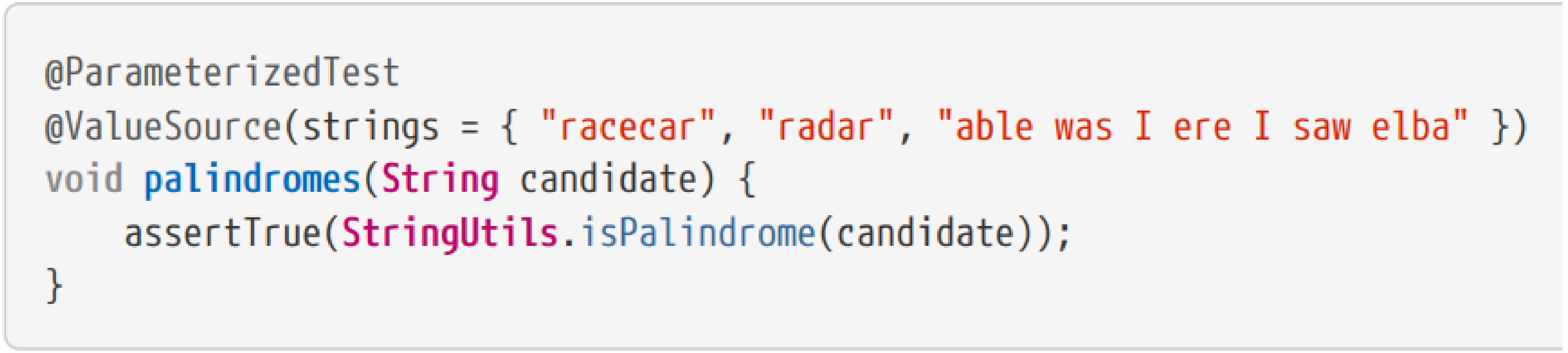
```
@BeforeAll
public static void init(){
System.out.println("BeforeAll init() method called");
}
```

- Why the BeforeAll is static?
- Notice the naming convention : Pascal not Camel!

Parametrized Tests

- This feature enables us to **execute a single test method multiple times with different parameters.**
- You must declare at least one source that will provide the arguments for each invocation and then consume the arguments in the test method.

@ValueSource



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

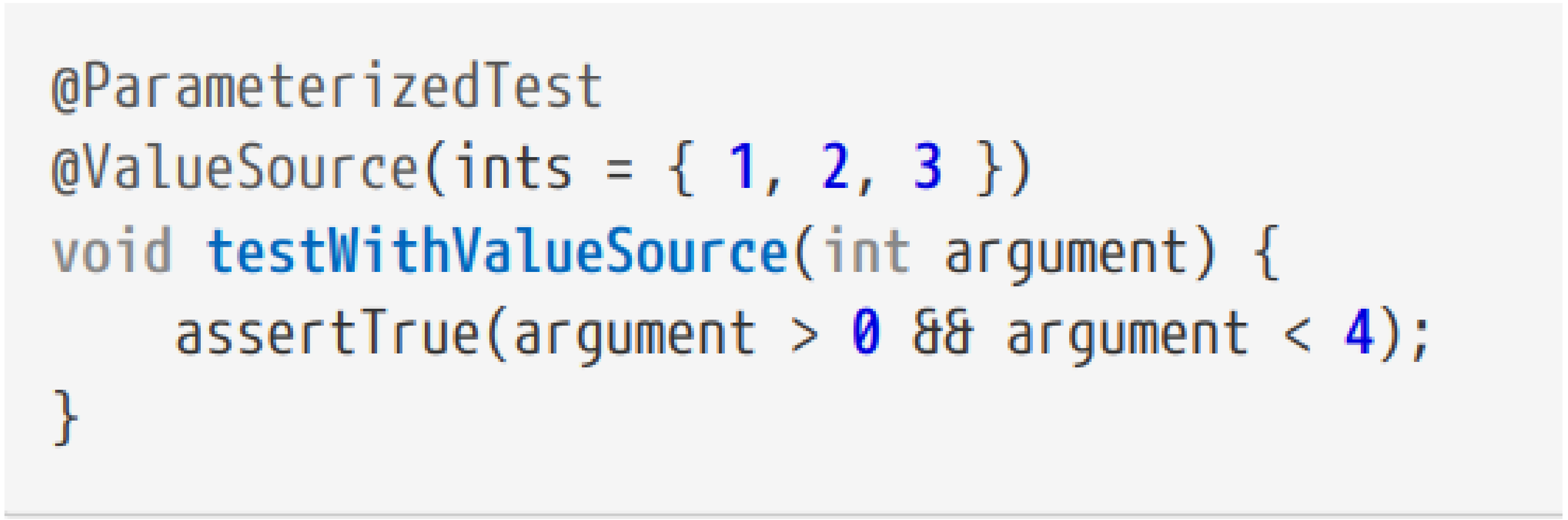
- Given the test case, write the implementation code?

A phrase is a **palindrome** if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers

@ValueSource

@ValueSource is one of the simplest possible sources. It lets you specify a single array of literal values and can only be used for providing a single argument per parameterized test invocation. The following types of literal values are supported by @ValueSource.

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

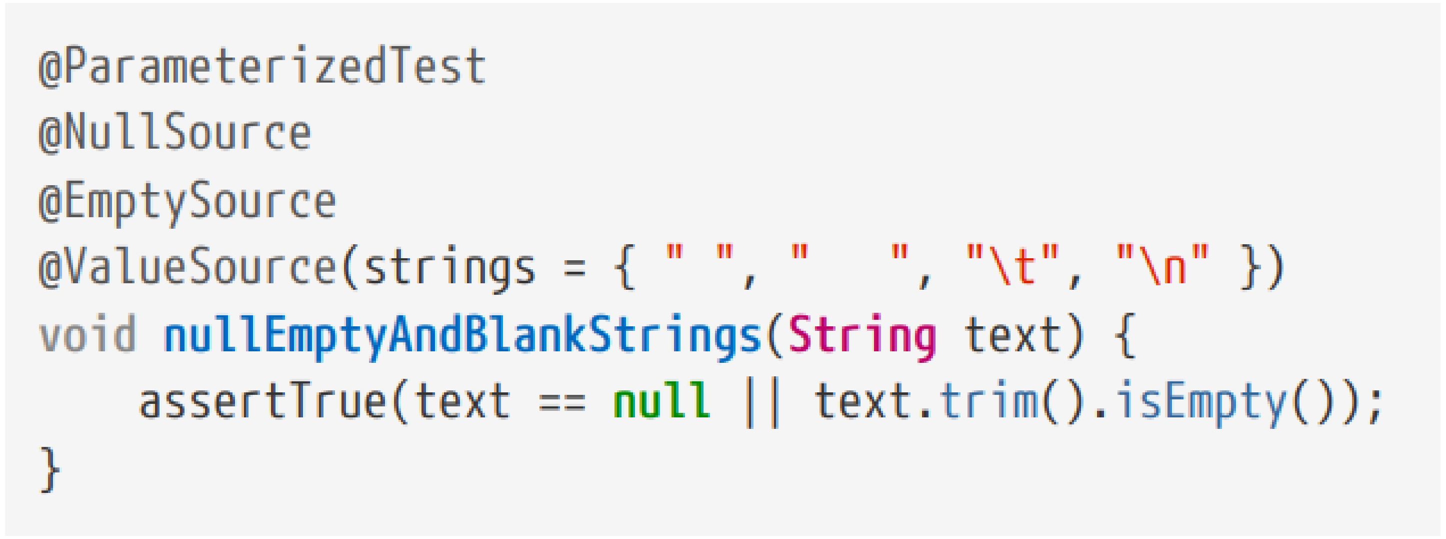


```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertTrue(argument > 0 && argument < 4);
}
```

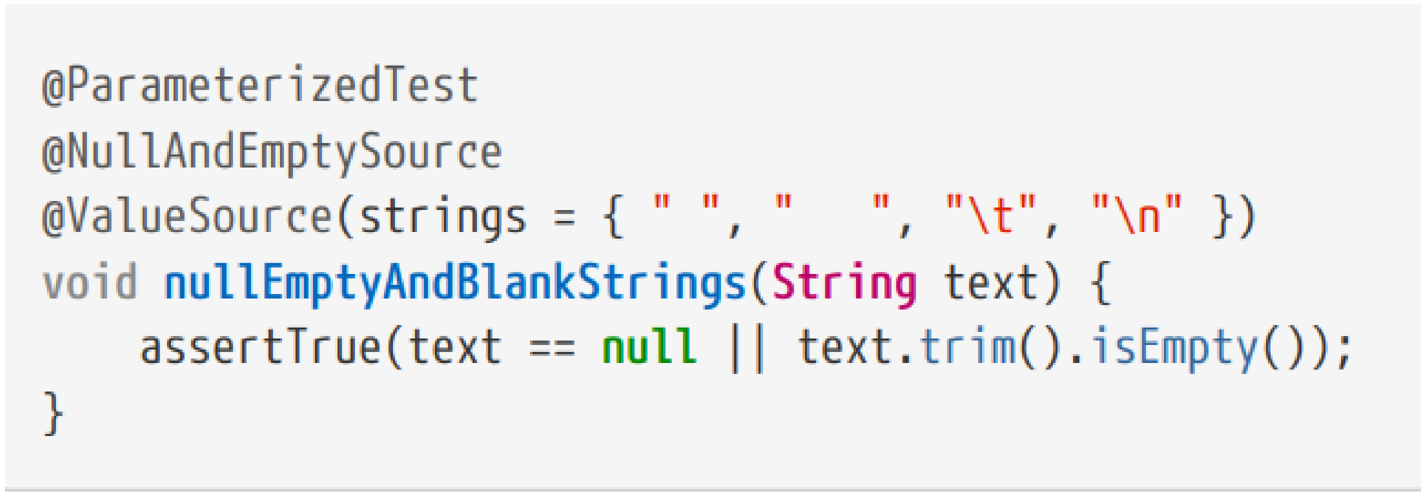
Null and Empty Values

- @NullSource: provides a single null argument to the annotated @ParameterizedTest method. @NullSource cannot be used for a parameter that has a primitive type.
- • @EmptySource: provides a single empty argument to the annotated @ParameterizedTest method for parameters of the following types: java.lang.String, java.util.List, java.util.Set java.util.Map, primitive arrays (e.g., int[], char[][], etc.), object arrays (e.g.,String[], Integer[][], etc.)
- • @NullAndEmptySource: a composed annotation that combines the functionality of @NullSource and @EmptySource.

Example

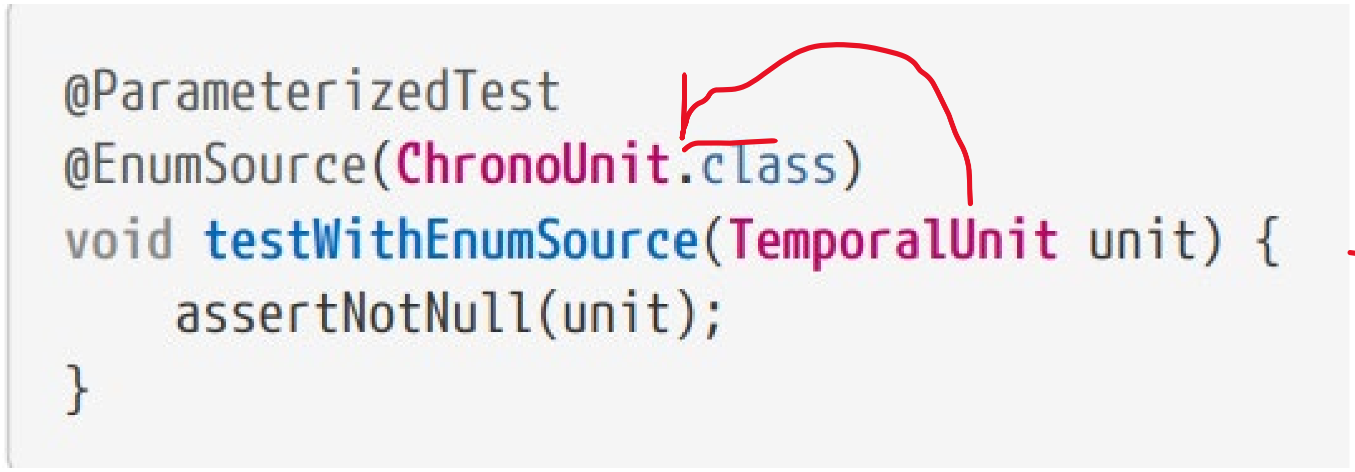


```java
@ParameterizedTest
@NullSource
@EmptySource
@ValueSource(strings = { " ", "   ", "\t", "\n" })
void nullEmptyAndBlankStrings(String text) {
    assertTrue(text == null || text.trim().isEmpty());
}
```

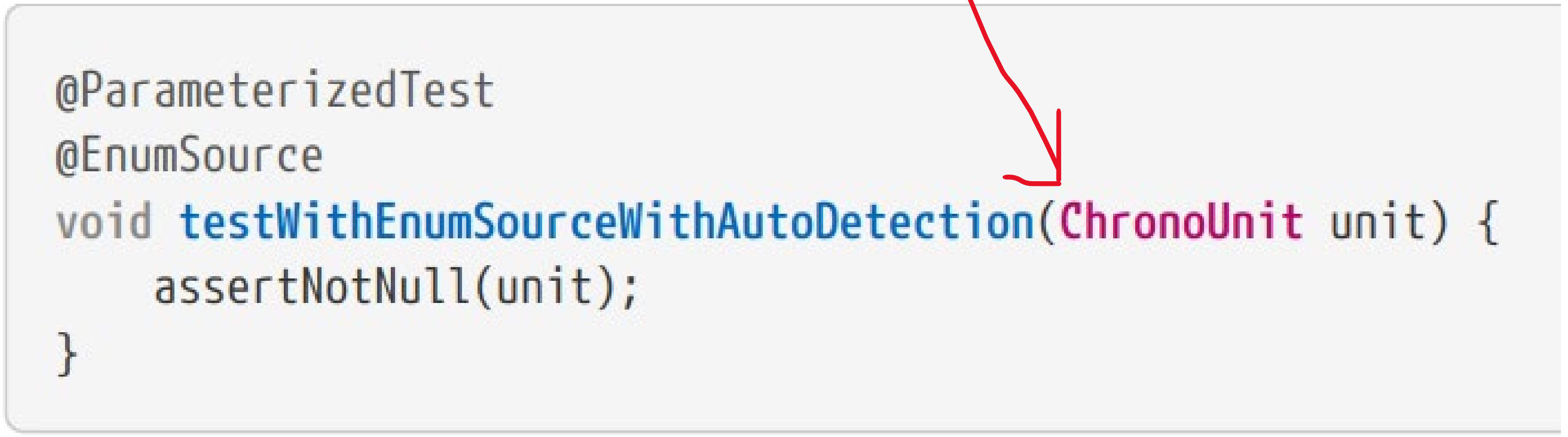


```java
@ParameterizedTest
@NullAndEmptySource
@ValueSource(strings = { " ", "   ", "\t", "\n" })
void nullEmptyAndBlankStrings(String text) {
    assertTrue(text == null || text.trim().isEmpty());
}
```

@EnumSource



```java
@ParameterizedTest
@EnumSource(ChronoUnit.class)
void testWithEnumSource(TemporalUnit unit) {
    assertNotNull(unit);
}
```



```java
@ParameterizedTest
@EnumSource
void testWithEnumSourceWithAutoDetection(ChronoUnit unit) {
    assertNotNull(unit);
}
```

@EnumSource

The annotation provides an optional `names` attribute that lets you specify which constants shall be used, like in the following example. If omitted, all constants will be used.

```java
@ParameterizedTest
@EnumSource(names = { "DAYS", "HOURS" })
void testWithEnumSourceInclude(ChronoUnit unit) {
    assertTrue(EnumSet.of(ChronoUnit.DAYS, ChronoUnit.HOURS).contains(unit));
}
```

Use these only

The `@EnumSource` annotation also provides an optional `mode` attribute that enables fine-grained control over which constants are passed to the test method. For example, you can exclude names from the enum constant pool or specify regular expressions as in the following examples.

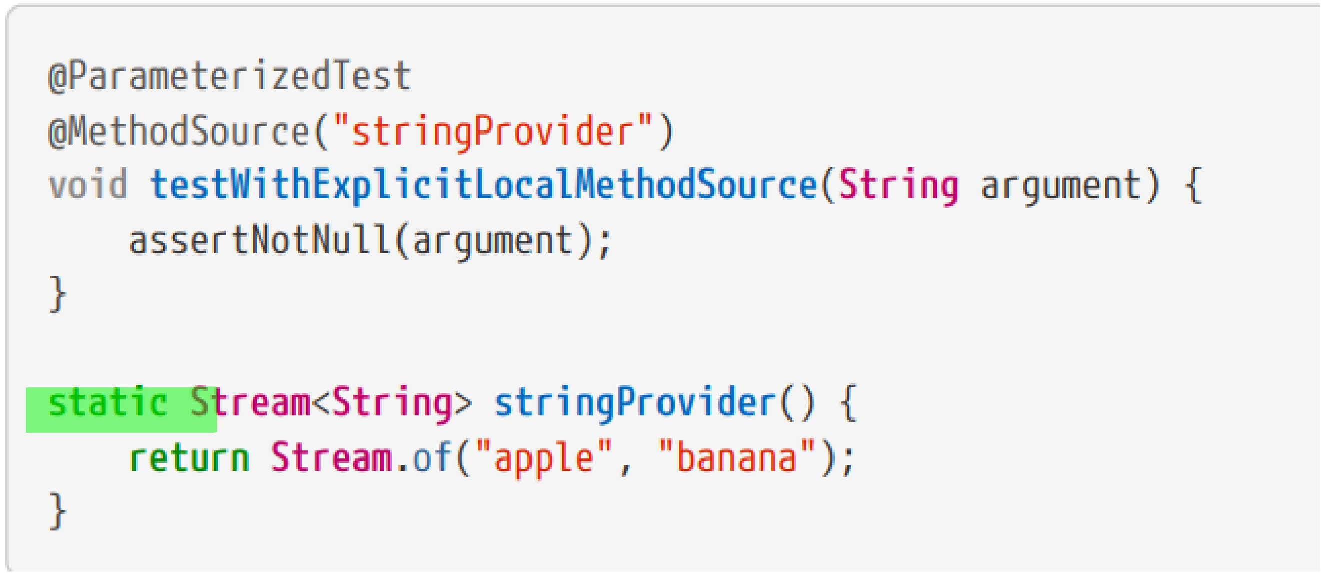
```java
@ParameterizedTest
@EnumSource(mode = EXCLUDE, names = { "ERAS", "FOREVER" })
void testWithEnumSourceExclude(ChronoUnit unit) {
    assertFalse(EnumSet.of(ChronoUnit.ERAS, ChronoUnit.FOREVER).contains(unit));
}
```

Exclude these

```java
@ParameterizedTest
@EnumSource(mode = MATCH_ALL, names = "^.*DAYS$")
void testWithEnumSourceRegex(ChronoUnit unit) {
    assertTrue(unit.name().endsWith("DAYS"));
}
```

regular expressions

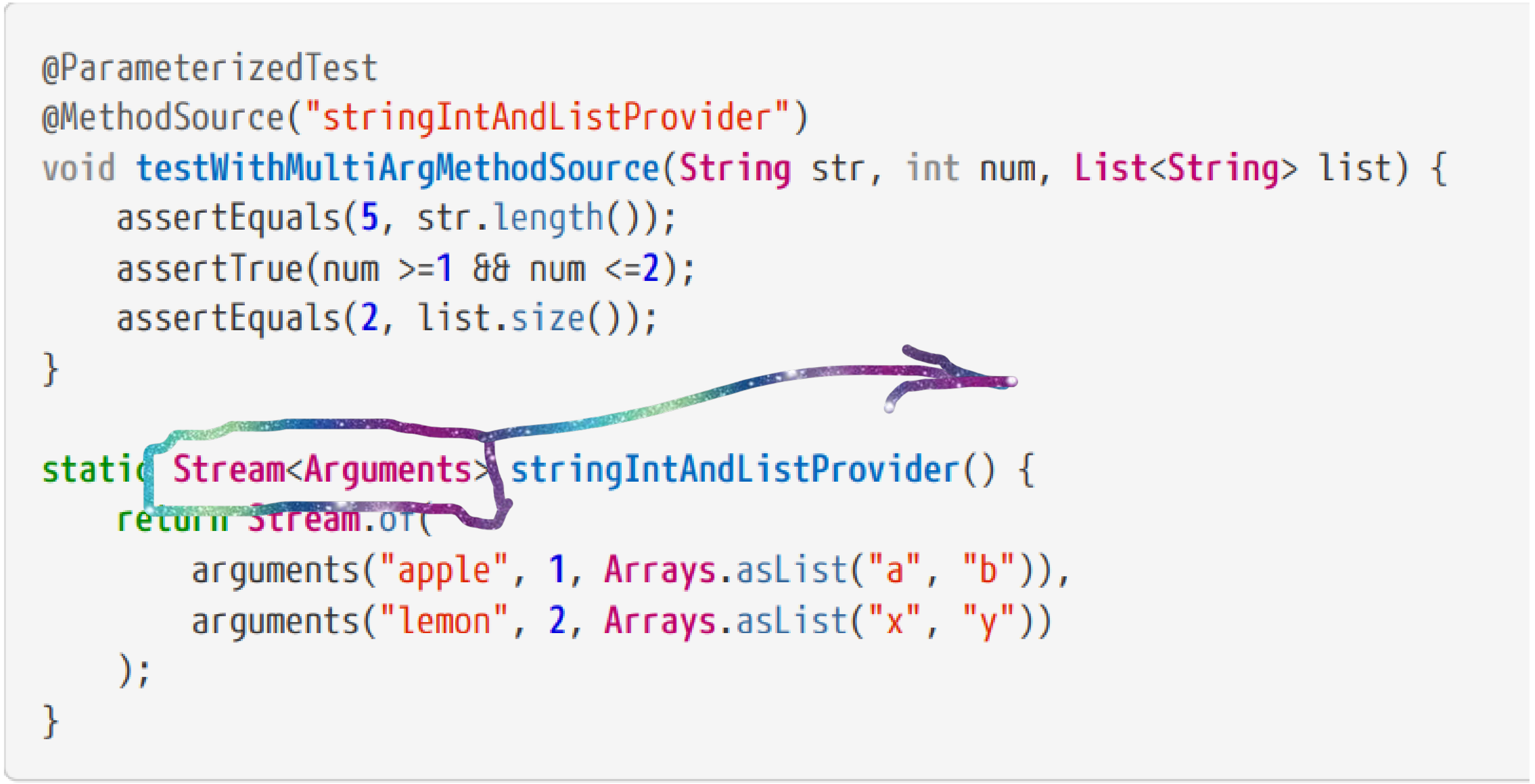
@MethodSource:: factory methods



```
@ParameterizedTest
@MethodSource("stringProvider")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> stringProvider() {
    return Stream.of("apple", "banana");
}
```

- Each factory method must generate a stream of arguments, and each set of arguments within the stream will be provided as the physical arguments for individual invocations of the annotated
- Streams for primitive types (DoubleStream, IntStream, and LongStream) are also supported

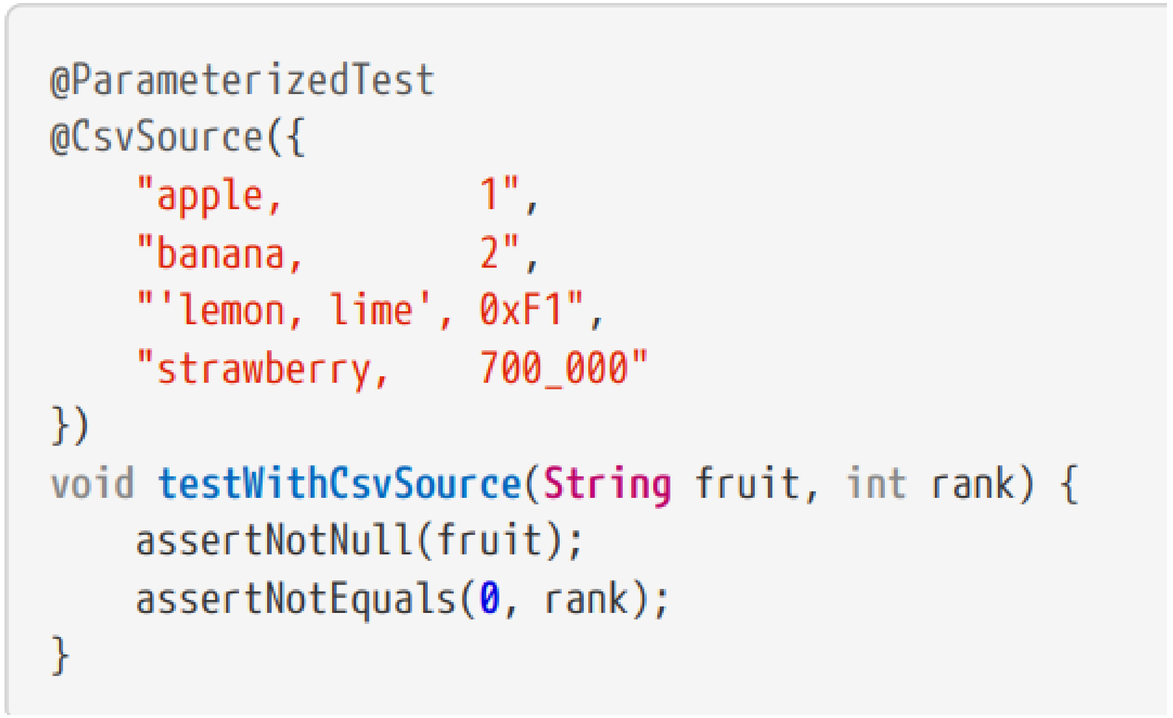
@MethodSource:: Multiple Parameters



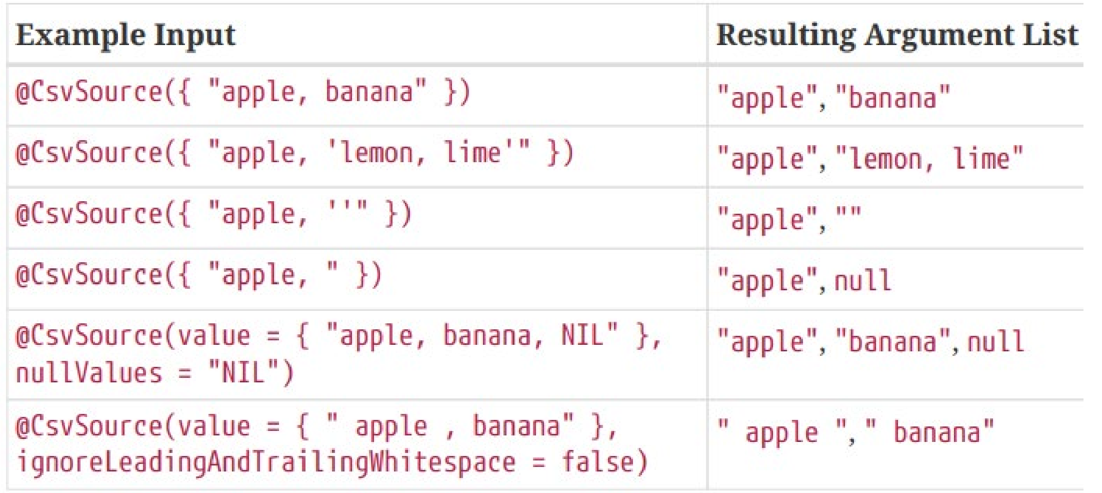
```java
@ParameterizedTest
@MethodSource("stringIntAndListProvider")
void testWithMultiArgMethodSource(String str, int num, List<String> list) {
    assertEquals(5, str.length());
    assertTrue(num >=1 && num <=2);
    assertEquals(2, list.size());
}

static Stream<Arguments> stringIntAndListProvider() {
    return Stream.of(
        arguments("apple", 1, Arrays.asList("a", "b")),
        arguments("lemon", 2, Arrays.asList("x", "y"))
    );
}
```

- Returns Collection stream or array of arguments

@CsvSource



```java
@ParameterizedTest
@CsvSource({
    "apple,         1",
    "banana,        2",
    "'lemon, lime', 0xF1",
    "strawberry,    700_000"
})
void testWithCsvSource(String fruit, int rank) {
    assertNotNull(fruit);
    assertNotEquals(0, rank);
}
```



Example Input	Resulting Argument List
`@CsvSource({ "apple, banana" })`	`"apple", "banana"`
`@CsvSource({ "apple, 'lemon, lime'" })`	`"apple", "lemon, lime"`
`@CsvSource({ "apple, ''" })`	`"apple", ""`
`@CsvSource({ "apple, " })`	`"apple", null`
`@CsvSource(value = { "apple, banana, NIL" }, nullValues = "NIL")`	`"apple", "banana", null`
`@CsvSource(value = { " apple , banana" }, ignoreLeadingAndTrailingWhitespace = false)`	`" apple ", " banana"`

@CsvFileSource

```java
@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 reference) {
    assertNotNull(country);
    assertNotEquals(0, reference);
}

@ParameterizedTest(name = "[{index}] {arguments}")
@CsvFileSource(resources = "/two-column.csv", useHeadersInDisplayName = true)
void testWithCsvFileSourceAndHeaders(String country, int reference) {
    assertNotNull(country);
    assertNotEquals(0, reference);
}
```

- @CsvFileSource lets you use comma-separated value (CSV) files from the classpath or the local file system. Each record from a CSV file results in one invocation of the parameterized test. The first record may optionally be used to supply CSV headers

Lab: in class exercise

- The aim is to practice TDD, designing test cases with Junit 5 features.
- Use case: validate the password policy
  - Password length
  - Password with special characters
  - Password with no digits
  - Password with no characters
  - at least 8 characters, max of 12
  - at least one uppercase
  - at least one lowercase
  - at least one number
  - at least one symbol @#$%=:?

Reference

- https://junit.org/junit5/docs/current/user-guide/
- https://github.com/jashburn8020/junit-5-user-guide