Assert Statements

assertThat(actual\_value).isEqualTo(expected\_value);

**Assertions in JUnit 4 and JUnit 5**

**Note 🡪**Assertions are utility methods to support asserting conditions in tests; these methods are accessible through the Assert class, in JUnit 4, and the Assertions one, in JUnit 5.

In order to increase the readability of the test and of the assertions itself, it's always recommended to import statically the respective class. In this way, we can refer directly to the assertion method itself without the representing class as a prefix.

**Assertions in JUnit 4**

In this version of the library, assertions are available for all primitive types, Objects, and arrays (either of primitives or Objects).

The parameters order, within the assertion, is the expected value followed by the actual value; optionally the first parameter can be a String message that represents the message output of the evaluated condition.

There's only one slightly different in how is defined the assertThat assertions, but we'll cover it later on.

Let's start with the assertEquals one.

**assertEquals**

The assertEquals assertion verifies that the expected and the actual values are equal:

@Test

public void whenAssertingEquality\_thenEqual() {

String expected = "Baeldung";

String actual = "Baeldung";

assertEquals(expected, actual);

}

It's also possible to specify a message to display when the assertion fails:

assertEquals("failure - strings are not equal", expected, actual);

**assertArrayEquals**

If we want to assert that two arrays are equals, we can use the assertArrayEquals:

@Test

public void whenAssertingArraysEquality\_thenEqual() {

char[] expected = {'J','u','n','i','t'};

char[] actual = "Junit".toCharArray();

assertArrayEquals(expected, actual);

}

If both arrays are null, the assertion will consider them equal:

@Test

public void givenNullArrays\_whenAssertingArraysEquality\_thenEqual() {

int[] expected = null;

int[] actual = null;

assertArrayEquals(expected, actual);

}

**assertNotNull and assertNull**

When we want to test if an object is null we can use the assertNull assertion:

@Test

public void whenAssertingNull\_thenTrue() {

Object car = null;

assertNull("The car should be null", car);

}

In the opposite way, if we want to assert that an object should not be null we can use the assertNotNull assertion.

**assertNotSame and assertSame**

With assertNotSame, it's possible to verify if two variables don't refer to the same object:

@Test

public void whenAssertingNotSameObject\_thenDifferent() {

Object cat = new Object();

Object dog = new Object();

assertNotSame(cat, dog);

}

Otherwise, when we want to verify that two variables refer to the same object, we can use the assertSame assertion.

**assertTrue and assertFalse**

In case we want to verify that a certain condition is true or false, we can respectively use the assertTrue assertion or the assertFalse one:

@Test

public void whenAssertingConditions\_thenVerified() {

assertTrue("5 is greater then 4", 5 > 4);

assertFalse("5 is not greater then 6", 5 > 6);

}

**fail**

The fail assertion fails a test throwing an AssertionFailedError. It can be used to verify that an actual exception is thrown or when we want to make a test failing during its development.

Let's see how we can use it in the first scenario:

@Test

public void whenCheckingExceptionMessage\_thenEqual() {

try {

methodThatShouldThrowException();

fail("Exception not thrown");

} catch (UnsupportedOperationException e) {

assertEquals("Operation Not Supported", e.getMessage());

}

}

**assertThat**

The assertThat assertion is the only one in JUnit 4 that has a reverse order of the parameters compared to the other assertions.

In this case, the assertion has an optional failure message, the actual value, and a Matcher object.

Let's see how we can use this assertion to check if an array contains particular values:

@Test

public void testAssertThatHasItems() {

assertThat(

Arrays.asList("Java", "Kotlin", "Scala"),

hasItems("Java", "Kotlin"));

}

Additional information, on the powerful use of the assertThat assertion with Matcher object, is available at Testing with Hamcrest**.**

**JUnit 5 Assertions**

JUnit 5 kept many of the assertion methods of JUnit 4 while adding few new ones that take advantage of the Java 8 support.

Also in this version of the library, assertions are available for all primitive types, Objects, and arrays (either of primitives or Objects).

The order of the parameters of the assertions changed, moving the output message parameter as the last parameter. Thanks to the support of Java 8, the output message can be a Supplier, allowing lazy evaluation of it.

Let's start reviewing the assertions available also in JUnit 4.

**assertArrayEquals**

The assertArrayEquals assertion verifies that the expected and the actual arrays are equals:

@Test

public void whenAssertingArraysEquality\_thenEqual() {

char[] expected = { 'J', 'u', 'p', 'i', 't', 'e', 'r' };

char[] actual = "Jupiter".toCharArray();

assertArrayEquals(expected, actual, "Arrays should be equal");

}

If the arrays aren't equal, the message “Arrays should be equal” will be displayed as output.

**assertEquals**

In case we want to assert that two floats are equals, we can use the simple assertEquals assertion:

@Test

public void whenAssertingEquality\_thenEqual() {

float square = 2 \* 2;

float rectangle = 2 \* 2;

assertEquals(square, rectangle);

}

However, if we want to assert that the actual value differs by a predefined delta from the expected value, we can still use the assertEquals but we have to pass the delta value as the third parameter:

@Test

public void whenAssertingEqualityWithDelta\_thenEqual() {

float square = 2 \* 2;

float rectangle = 3 \* 2;

float delta = 2;

assertEquals(square, rectangle, delta);

}

**assertTrue and assertFalse**

With the assertTrue assertion, it's possible to verify the supplied conditions are true:

@Test

public void whenAssertingConditions\_thenVerified() {

assertTrue(5 > 4, "5 is greater the 4");

assertTrue(null == null, "null is equal to null");

}

Thanks to the support of the lambda expression, it's possible to supply a BooleanSupplier to the assertion instead of a boolean condition.

Let's see how we can assert the correctness of a BooleanSupplier using the assertFalse assertion:

@Test

public void givenBooleanSupplier\_whenAssertingCondition\_thenVerified() {

BooleanSupplier condition = () -> 5 > 6;

assertFalse(condition, "5 is not greater then 6");

}

**assertNull and assertNotNull**

When we want to assert that an object is not null we can use the assertNotNull assertion:

@Test

public void whenAssertingNotNull\_thenTrue() {

Object dog = new Object();

assertNotNull(dog, () -> "The dog should not be null");

}

In the opposite way, we can use the assertNull assertion to check if the actual is null:

@Test

public void whenAssertingNull\_thenTrue() {

Object cat = null;

assertNull(cat, () -> "The cat should be null");

}

In both cases, the failure message will be retrieved in a lazy way since it's a Supplier.

**assertSame and assertNotSame**

When we want to assert that the expected and the actual refer to the same Object, we must use the assertSame assertion:

@Test

public void whenAssertingSameObject\_thenSuccessfull() {

String language = "Java";

Optional<String> optional = Optional.of(language);

assertSame(language, optional.get());

}

In the opposite way, we can use the assertNotSame one.

**fail**

The fail assertion fails a test with the provided failure message as well as the underlying cause. This can be useful to mark a test when it's development it's not completed:

@Test

public void whenFailingATest\_thenFailed() {

// Test not completed

fail("FAIL - test not completed");

}

**assertAll**

One of the new assertion introduced in JUnit 5 is assertAll.

This assertion allows the creation of grouped assertions, where all the assertions are executed and their failures are reported together. In details, this assertion accepts a heading, that will be included in the message string for the MultipleFailureError, and a Stream of Executable.

Let's define a grouped assertion:

@Test

public void givenMultipleAssertion\_whenAssertingAll\_thenOK() {

assertAll(

"heading",

() -> assertEquals(4, 2 \* 2, "4 is 2 times 2"),

() -> assertEquals("java", "JAVA".toLowerCase()),

() -> assertEquals(null, null, "null is equal to null")

);

}

The execution of a grouped assertion is interrupted only when one of the executables throws a blacklisted exception (OutOfMemoryError for example).

**assertIterableEquals**

The assertIterableEquals asserts that the expected and the actual iterables are deeply equal.

In order to be equal, both iterable must return equal elements in the same order and it isn't required that the two iterables are of the same type in order to be equal.

With this consideration, let's see how we can assert that two lists of different types (LinkedList and ArrayList for example) are equal:

@Test

public void givenTwoLists\_whenAssertingIterables\_thenEquals() {

Iterable<String> al = new ArrayList<>(asList("Java", "Junit", "Test"));

Iterable<String> ll = new LinkedList<>(asList("Java", "Junit", "Test"));

assertIterableEquals(al, ll);

}

In the same way of the assertArrayEquals, if both iterables are null, they are considered equal.

**assertLinesMatch**

The assertLinesMatch asserts that the expected list of String matches the actual list.

This method differs from the assertEquals and assertIterableEquals since, for each pair of expected and actual lines, it performs this algorithm:

check if the expected line is equal to the actual one. If yes it continues with the next pair

treat the expected line as a regular expression and performs a check with the String.matches() method. If yes it continues with the next pair

check if the expected line is a fast-forward marker. If yes apply fast-forward and repeat the algorithm from the step 1

Let's see how we can use this assertion to assert that two lists of String have matching lines:

@Test

public void whenAssertingEqualityListOfStrings\_thenEqual() {

List<String> expected = asList("Java", "\\d+", "JUnit");

List<String> actual = asList("Java", "11", "JUnit");

assertLinesMatch(expected, actual);

}

**assertNotEquals**

Complementary to the assertEquals, the assertNotEquals assertion asserts that the expected and the actual values aren't equal:

@Test

public void whenAssertingEquality\_thenNotEqual() {

Integer value = 5; // result of an algorithm

assertNotEquals(0, value, "The result cannot be 0");

}

If both are null, the assertion fails.

**assertThrows**

In order to increase simplicity and readability, the new assertThrows assertion allows us a clear and a simple way to assert if an executable throws the specified exception type.

Let's see how we can assert a thrown exception:

@Test

void whenAssertingException\_thenThrown() {

Throwable exception = assertThrows(

IllegalArgumentException.class,

() -> {

throw new IllegalArgumentException("Exception message");

}

);

assertEquals("Exception message", exception.getMessage());

}

The assertion will fail if no exception is thrown, or if an exception of a different type is thrown.

**assertTimeout and assertTimeoutPreemptively**

In case we want to assert that the execution of a supplied Executable ends before a given Timeout, we can use the assertTimeout assertion:

@Test

public void whenAssertingTimeout\_thenNotExceeded() {

assertTimeout(

ofSeconds(2),

() -> {

// code that requires less then 2 minutes to execute

Thread.sleep(1000);

}

);

}

However, with the assertTimeout assertion, the supplied executable will be executed in the same thread of the calling code. Consequently, execution of the supplier won't be preemptively aborted if the timeout is exceeded.

In case we want to be sure that execution of the executable will be aborted once it exceeds the timeout, we can use the assertTimeoutPreemptively assertion.

Both assertions can accept, instead of an Executable, a ThrowingSupplier, representing any generic block of code that returns an object and that can potentially throw a Throwable.

**Refer the link IMP 🡪 https://www.baeldung.com/junit-assertions**

**https://www.baeldung.com/junit-assert-exception**

**Assert an Exception is Thrown in JUnit 4 and 5:**

**JUnit 5**

JUnit 5 Jupiter assertions API introduces the assertThrows method for asserting exceptions.

This takes the type of the expected exception and an Executable functional interface where we can pass the code under test through a lambda expression:

@Test

public void whenExceptionThrown\_thenAssertionSucceeds() {

Exception exception = assertThrows(NumberFormatException.class, () -> {

Integer.parseInt("1a");

});

String expectedMessage = "For input string";

String actualMessage = exception.getMessage();

assertTrue(actualMessage.contains(expectedMessage));

}

If the expected exception is thrown, assertThrows returns the exception, which enables us to also assert on the message.

Furthermore, it's important to note that this assertion is satisfied when the enclosed code throws an exception of type NumberFormatException or any of its derived types.

This means that if we pass Exception as the expected exception type, any exception thrown will make the assertion succeed since Exception is the super-type for all exceptions.

If we change the test above to expect a RuntimeException, this will also pass:

@Test

public void whenDerivedExceptionThrown\_thenAssertionSucceds() {

Exception exception = assertThrows(RuntimeException.class, () -> {

Integer.parseInt("1a");

});

String expectedMessage = "For input string";

String actualMessage = exception.getMessage();

assertTrue(actualMessage.contains(expectedMessage));

}

The assertThrows() method enables more fine-grained control for exception assertion logic because we can use it around specific parts of the code.