Nolix

Nolix tests

2020-06-05

Table of contents

1 Introduction

1.1 What Nolix tests are

The Nolix Test class is a base class for custom tests. There is used the following terminology.

test unit	An object whose creation or methods are tested.
test case / test method	A method that tests the creation or a method of a test unit.
test class	A class whose purpose is, and only is, to contain test cases. The test units of the test cases of a test class should be all of the same type.
test	An instance of a test class.
unit test	A test where the dependencies of the test units are injected by the test.
integration test / integration test class	A test where the dependencies of the test units can, but do not need to, be injected by the test.

1.2 Why to use Nolix tests

- Nolix tests provide **many** methods to validate test units.
- Validations in Nolix tests can be written in **very legible** code.
- The given TestPools can comfortably **bundle** Nolix Test classes.

1.3 Where the Nolix tests are

The Nolix tests are defined in the Nolix library. To use the Nolix tests, import the Nolix library into your project.

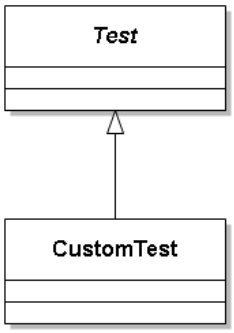
2 Tests

2.1 Import Test

```
import ch.nolix.common.test.test;
```

The Test can be found in the package ‘ch.nolix.common.test’.

2.2 Define a custom Test




```
public class CustomTest extends Test {
  …
}
```

A custom Test can be defined by inheriting from the Test class. A Test is empty at the beginning.

2.3 Add test cases to a Test

```
import ch.nolix.common.baseTest.TestCase;

public class CustomTest extends Test {

  @TestCase
  public void testCase_length() {

    //setup
    var testUnit = “Hello!”;

    //execution
    var length = testUnit.length();

    //verification
    expect(length).isEqualTo(6);
  }
}
```

To a Test, an arbitrary number of test cases can be added. To the shown test a test case is added, that validates the ‘length’ method of a String. A method becomes a test case when it has the TestCase annotation. The TestCase annotation is in the package ‘ch.nolix.common.baseTest’.

The output of the run of the shown test is:

```
   PASSED: testLength (0ms)
   Summary CustomTest: 1/1 test cases passed (0ms)
```

2.4 Create and run a Test

```
var Test = new CustomTest();

test.run();
```

To run a Test, create an instance of it and call the method ‘run’ on it. The run method runs all test cases of a Test for 1 time. The test cases of a Test are run in alphabetic order.

This can also be done in 1 line:

```
new CustomTest().run();
```

2.5 Output of a Test

```
   Started MatrixTest
   PASSED: loopTest_createIdendityMatrix (0ms)
   PASSED: loopTest_getRank (0ms)
-->FAILED: loopTest_getTrace: (0ms)
   1) 1.0 was expected, but 0.0 was received.
   (ch.nolix.commonTest.mathematicsTest.MatrixTest.java:69)
   2) 2.0 was expected, but 0.0 was received.
   (ch.nolix.commonTest.mathematicsTest.MatrixTest.java:69)
   3) 3.0 was expected, but 0.0 was received.
   (ch.nolix.commonTest.mathematicsTest.MatrixTest.java:69)
   4) 4.0 was expected, but 0.0 was received.
   (ch.nolix.commonTest.mathematicsTest.MatrixTest.java:69)
   5) 5.0 was expected, but 0.0 was received.
   (ch.nolix.commonTest.mathematicsTest.MatrixTest.java:69)
   PASSED: test_add (0ms)
   PASSED: test_appendAtRight (0ms)
   PASSED: test_getInverse (0ms)
   PASSED: test_getInverse_2 (0ms)
   PASSED: test_getInverse_3 (0ms)
   PASSED: test_getProduct (0ms)
   PASSED: test_getSolutionAsExtendedMatrix (0ms)
   PASSED: test_getSolutionAsExtendedMatrix_2 (0ms)
   PASSED: test_getTransposed (0ms)
   PASSED: test_toString (16ms)
   PASSED: test_toString_2 (0ms)
   PASSED: test_toString_3 (0ms)
   PASSED: test_toString_4 (0ms)
   PASSED: test_toString_5 (0ms)
   PASSED: test_toString_6 (0ms)
   PASSED: test_toString_7 (0ms)
   PASSED: test_toString_8 (0ms)
   Summary MatrixTest: 19/20 test cases passed (16ms)
```

When a Test is run, the following output is produced:

- All errors of all failed test cases.
- The running duration of all test cases.
- The count of the passed test cases.
- The total running duration of the test.

The running duration is always shown in milliseconds (ms).

The shown output is from a Test for a Matrix class. One test case of the Test failed.

2.6 Validate that a boolean is true

```
boolean b;
…
expect(b);
```

If the given boolean is false, the Test will fail. The error message will be:

”True was expected, but false was received.”

2.7 Validate that a boolean is false

```
boolean b;
…
expectNot(b);
```

If the given boolean is true, the Test will fail. The error message will be:

"False was expected, but true was received."

2.8 Validate that a number is positive

```
int  value;
…
expect(value).isPositive();
```

If the given value is e.g. -10, the Test will fail. The error message will be:

”A positive value was expected, but ‘-10’ was received.”

2.9 Validate that a number is bigger than a given limit

```
int value;
…
expect(value).isBiggerThan(100);
```

If the given value is e.g. 50, the Test will fail. The error message will be:

”A value, that is bigger than 100, was expected, but ‘50’ was received.”

2.10 Validate that a number is in a given range

```
int value;
…
expect(value).isBetween(-50, 50);
```

If the given value is e.g. 100, the Test will fail. The error message will be:

”A value, that is in [-50, 50], was expected, but ‘100’ was received.”

2.11 Validate floating point numbers with deviations

Floating point numbers can have rounding errors, especially when they are calculated by a complex algorithm. For example, when the ideal result is 2.0, but 1.999999 or 2.000001 comes out.

Tests provide functionalities to Test floating point numbers with deviations. A value is regarded as correct, when it does not deviate more than the default deviation or a given maximum deviation. The default deviation is 10^{-9}.

```
var result = getSquareRoot(2.0);

expect(result).withMaxDeviation(0.001).isEqualTo(1.4142);
```

If the given result does not equal 1.4142 with a maximum deviation of 0.001, the Test will fail. The error message will be:

'A value that equals 1.4142 with a max deviation of 0.001 was expected, but 2 was received.'

2.12 Validate that a String is not empty

```
String string;

…

expect(string).isNotEmpty();
```

If the given string is empty, the Test will fail. The error message will be:

”A String, that is not empty, was expected, but an empty String was received.”

2.13 Validate that a String is not longer than a given length

```
String string;
…
expect(string).isNotLongerThan(10);
```

If the given string is e.g. “Hello World!”, the Test will fail. The error message will be:

”A String, that is not longer than 10, was expected, but a String with the length 12 was received.”

2.14 Validate that an action does not throw an Exception

```
expect(() -> 10 / 2).doesNotThrowException();
```

If the given action throws an exception, the Test will fail. The error message will be:

”An action, that does not throw an Exception, was expected, but an action, that throws an Exception, was received.”

2.15 Validate that an action throws an Exception

```
String string = null;

expect(() -> System.out.println(string.length())
.throwsException();
```

If the given action does not throw an exception, the Test will fail. The error message will be:

"An action, that throws an Exception was expected, but an action, that does not throw an Exception, was received."

2.16 Validate that an action throws an Exception of a specific type

```
expect(() -> 10 / 0)
.throwException()
.ofType(ArithmeticException.class);
```

If the given action does not throw an exception, the Test will fail. The error message will be:

”An action, that throws a ArithmeticException, was expected, but an action, that does not throw an Exception, was received.”

2.17 Validate that an action throws an exception with a specific message

```
Lecture lecture;

expect(() -> lecture.registerStudent(null))
.throwsException()
.withMessage("The given student is null.");
```

If the given action does not throw an Exception, the Test will fail. The error message will be:

"An action, that throws an Exception was expected, but an action, that does not throw an Exception, was received."

If the given action throws an Exception with an unexpected message e.g. 'blah blah', the Test will fail. The error message will be:

"An action, that throws an Exception with the message 'The given student is null.' was expected, but an action, that throws an Exception with the message 'blah blah', was received."

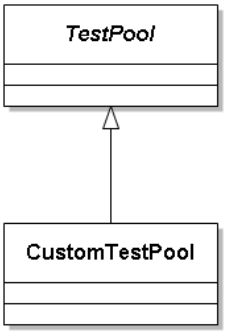
3 Test pools

3.1 Import TestPool

```
import ch.nolix.common.test.TestPool;
```

The TestPool is in the package ‘ch.nolix.common.testoid’.

3.2 Define a custom TestPool




```
public class CustomTestPoolextends TestPool {
  …
}
```

A custom TestPool can be defined by inheriting from the TestPool class. A TestPool is empty at the beginning.

3.3 Add Test classes to a TestPool

```
public class CustomTestPoolextends TestPool{

  public CustomTestPool() {
    addTestClass(
      CustomTest.class,
      MatrixTest.class
    );
  }
}
```

To a TestPool, an arbitrary number of Tests can be added. The Tests have to be added in the constructor of the TestPool. To the shown TestPool, 2 Test classes are added.

The method ‘addTest’ can take an arbitrary number of Tests.

3.4 Create and run a TestPool

```
var testPool = new CustomTest();

testPool.run();
```

To run a TestPool, create an instance of it and call the method ‘run’ on it. The run method creates and run a Test from all of its Test classes. The Tests of a TestPool are run in the same order how the Test classes were added to the TestPool. The Tests are run the same way as when they were created and run separately one after one.

This can also be done in 1 line:

```
new CustomTestPool().run();
```

3.5 Nest TestPools

```
public class CustomTestPoolextends TestPool{

  public CustomTestPool() {

    addTestPool(
      new CustomSubTestPool1(),
      new CustomSubTestPool2()
    );

    addTestClass(
      CustomTest.class,
      MatrixTest.class
    );
  }
}
```

To a TestPool, an arbitrary number of other TestPools can be added. The other TestPools have to be added in the constructor of the TestPool. To the shown TestPool, 2 other TestPools and 2 Test classes are added.

A TestPoolcan contain both, Test classes and other TestPools. The method ‘addTestPool’ can take an arbitrary number of TestPools. A TestPool must not contain itself recursively. If there is tried to add a TestPool recursively to itself, the method ‘addTestPool’ will throw an InvalidArgumentException.