

# Nolix Tests

## Table of contents

1	Tests.....	2
1.1	What a test is.....	2
1.2	How a test is defined .....	3
1.3	How test methods are added to a test.....	4
1.4	How a test is created and run .....	5
1.5	What the output of a test is .....	6
1.6	How test units are tested .....	7
1.7	How floating point numbers with deviations are tested .....	8
1.8	How methods are tested to exceptions .....	9
2	Test pools .....	10
2.1	What a test pool is.....	10
2.2	How a test pool is defined .....	10
2.3	How tests are added to a test pool .....	11
2.4	How a test pool is created and run .....	12
2.5	How test pools are nested.....	13

## 1 Tests

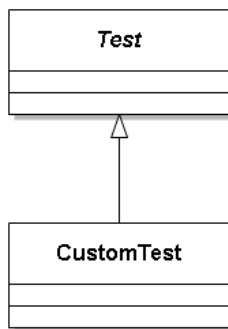
### 1.1 What a test is

The Nolix test classes are base classes for custom unit tests and integration tests.

test unit	An object whose creation or methods are tested.
test method	A method that tests the creation or a method of a test unit.
test / test class	A class whose purpose is, and only is, to contain test methods. The test units of a test should be all of the same type.

## 1.2 How a test is defined

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



```
import ch.nolix.core.test2.test;

public class CustomTest extends Test {

}
```

The test class can be found in the package 'ch.nolix.core.test2'.

## 1.3 How test methods are added to a test

To a test, an arbitrary number of test methods can be added. A test method always must be public. All public methods are test methods.

Lets add a first test method to our custom test. This test method tests a certain case of the length method of a string.

```
import ch.nolix.core.test2.Test;

public class CustomTest extends Test {

    //test method
    public void testLength() {

        //setup
        String testUnit = "Hello!";

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

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

The output of the execution of this test is:

```
PASSED: testLength (0ms)
=CustomTest: 1/1 test methods passed (0ms)
```

## 1.4 How a test is created and run

To run a test, an instance of it must be created and its run method must be called.

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

This can be done in 1 line.

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

The run method runs all test methods of a test for 1 time. The test methods are run in alphabetic order.

## 1.5 What the output of a test is

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

- The running duration of all test methods.
- The number of the passed test methods.
- The total running duration of the test.
- All errors of all failed test methods.

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

The following is the output of a test of a Matrix class. One test method of the test failed.

```
PASSED: loopTest_createIdendityMatrix (0ms)
PASSED: loopTest_getRank (0ms)
-->FAILED: loopTest_getTrace: (0ms)
#1 1.0 was expected, but 0.0 was received.
(ch.nolix.coreTest.mathematicsTest.MatrixTest.java:69)
#2 2.0 was expected, but 0.0 was received.
(ch.nolix.coreTest.mathematicsTest.MatrixTest.java:69)
#3 3.0 was expected, but 0.0 was received.
(ch.nolix.coreTest.mathematicsTest.MatrixTest.java:69)
#4 4.0 was expected, but 0.0 was received.
(ch.nolix.coreTest.mathematicsTest.MatrixTest.java:69)
#5 5.0 was expected, but 0.0 was received.
(ch.nolix.coreTest.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)
=MatrixTest: 19/20 test methods passed (16ms)
```

## 1.6 How test units are tested

A test provides various methods to test the test units.

### Example (boolean)

```
boolean boolean1 = true;
boolean boolean2 = false;

//Expects that 'bool1' is true.
expect(boolean1);

//Expects that 'bool2' is false.
expectNot(boolean2);
```

### Example (int)

```
//Expects that an int is positive.
expect(15).isPositive();

//Expects that an int is bigger than 10.
expect(15).isBiggerThan(10);
```

### Example (string)

```
//Expects that a string is not empty.
expect("Hello World!").isNotEmpty();

//Expects that a string is not longer than 20 characters.
expect("Hello World!").hasMaxLength(20);
```

## 1.7 How floating point numbers with deviations are tested

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  $10^{-9}$ .

```
double result = getSquareRoot(2.0);  
  
//Expects that the result is 1.4142 with a maximum deviation of 0.001.  
expect(result).withMaxDeviation(0.001).equals(1.4142);
```



## 1.8 How methods are tested to exceptions

Tests are able to test if a method throws an exception.

```
//Expects that dividing by 0 throws an arithmetic exception.  
expect(() -> 10 / 0).throwsExceptionOfType(ArithmeticException);  
  
//Expects that dividing by 2 throws no exception.  
expect(() -> 10 / 2).throwsNoException();
```

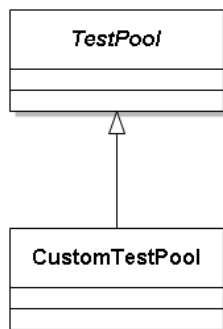
## 2 Test pools

### 2.1 What a test pool is

A test pool can contain tests and can run them all.

### 2.2 How a test pool is defined

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



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

public class CustomTestPool extends TestPool {
}
```

The test pool class can be found in the package 'ch.nolix.core.test'.

## 2.3 How tests are added to a test pool

A test pool can contain several tests. Tests cannot be added to a test pool from outside. So the tests of a test pool need to be added in the constructor.

Lets add 2 tests to our custom test pool.

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

public class CustomTestPool extends TestPool {

    //constructor
    public CustomTestPool() {
        addTest(
            new CustomTest(),
            new MatrixTest()
        );
    }
}
```

The method 'addTest' can take an arbitrary number of tests.

## 2.4 How a test pool is created and run

To run the tests of a test pool, an instance of the test pool must be created and its run method must be called.

```
CustomPool testPool = new CustomTestPool();  
testPool.run();
```

This can be done in 1 line.

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

The tests of the test pool are run the same way as when they were created and run separately one after one. The tests of a test pool are run in the same order how they were added to the test pool.

## 2.5 How test pools are nested

A test pool can also contain other test pools. Test pools cannot be added to another test pool from outside. The internal test pools of a test pool need to be added in the constructor. A test pool can contain both, tests and other test pools.

Lets add two other test pools to our custom test pool.

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

public class CustomTestPool extends TestPool {

    //constructor
    public CustomTestPool() {

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

        addTest(
            new CustomTest(),
            new MatrixTest()
        );
    }
}
```

The method 'addTestPool' can take an arbitrary number of test pools.

A test pool cannot contain itself directly or recursively to avoid circular execution calls. If there is tried to add a test pool that violates this condition, the method 'addTestPool' throws an exception.