# JUnit 4

https://github.com/junit-team/junit4/wiki

To download and install JUnit you currently have the following options.

## Plain-old JAR

Download the following JARs and add them to your test classpath:

* junit.jar
* hamcrest-core.jar

## Maven

Add a dependency to junit:junit in test scope. (Note: 4.12 is the latest stable version as of the latest edit on this page.)

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

## Gradle

See Use-with-Gradle

# Getting started

*Felipe augusto edited this page on 26 Aug 2017 · 21 revisions*

This small example shows you how to write a unit test. You need to have a JDK installed and a text editor. (In general it is recommended to use a build tool for building your software and running the tests.)

## Preparation

Create a new folder junit-example and download the current junit-4.XX.jar from JUnit's release page and Hamcrest to this folder. Change to the folder junit-example. All files are created within this folder and all commands are executed there, too.

## Create the class under test

Create a new file Calculator.java and copy the following code to this file.

public class Calculator {

public int evaluate(String expression) {

int sum = 0;

for (String summand: expression.split("\\+"))

sum += Integer.valueOf(summand);

return sum;

}

}

Now compile this class:

javac Calculator.java

The Java compiler creates a file Calculator.class.

## Create a test

Create the file CalculatorTest.java and copy the following code to this file.

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class CalculatorTest {

@Test

public void evaluatesExpression() {

Calculator calculator = new Calculator();

int sum = calculator.evaluate("1+2+3");

assertEquals(6, sum);

}

}

Compile the test. On Linux or MacOS

javac -cp .:junit-4.XX.jar:hamcrest-core-1.3.jar CalculatorTest.java

and on Windows

javac -cp .;junit-4.XX.jar;hamcrest-core-1.3.jar CalculatorTest.java

The Java compiler creates a file CalculatorTest.class.

## Run the test

Run the test from the command line. On Linux or MacOS

java -cp .:junit-4.XX.jar:hamcrest-core-1.3.jar

org.junit.runner.JUnitCore CalculatorTest

and on Windows

java -cp .;junit-4.XX.jar;hamcrest-core-1.3.jar

org.junit.runner.JUnitCore CalculatorTest

The output is

JUnit version 4.12

.

Time: 0,006

OK (1 test)

The single . means that one test has been run and the OK in the last line tells you that your test is successful.

## Let the test fail

Modify Calculator.java in order to get a failing test. Replace the line

sum += Integer.valueOf(summand);

with

sum -= Integer.valueOf(summand);

and recompile the class.

javac Calculator.java

Run the test again. On Linux or MacOS

java -cp .:junit-4.XX.jar:hamcrest-core-1.3.jar

org.junit.runner.JUnitCore CalculatorTest

and on Windows

java -cp .;junit-4.XX.jar;hamcrest-core-1.3.jar

org.junit.runner.JUnitCore CalculatorTest

Now the test fails and the output is

JUnit version 4.12

.E

Time: 0,007

There was 1 failure:

1) evaluatesExpression(CalculatorTest)

java.lang.AssertionError: expected:<6> but was:<-6>

at org.junit.Assert.fail(Assert.java:88)

...

FAILURES!!!

Tests run: 1, Failures: 1

JUnit tells you which test failed (evaluatesExpression(CalculatorTest)) and what went wrong:

java.lang.AssertionError: expected:<6> but was:<-6>

# JUnit Usage and Idioms

* **Assertions** - your bread and butter for unit testing
* **Test** **Runners** - how tests should be executed
* **Aggregating tests in Suites** - how to combine multiple related tests into a test suite
* **Test Execution Order** - specifying what order to run unit tests
* **Exception Testing** - how to specify expected exceptions in unit tests
* **Matchers and assertThat** - how to use Hamcrest matchers and more descriptive assertions
* **Ignoring Tests** - how to disable test methods or classes
* **Timeout** **for** **Tests** - how to specify maximum execution times for tests
* **Parameterized** **Tests** - writing tests that can be executed multiple times with different parameter values
* **Assumptions** **with** **Assume** - similar to assertions, but without making tests fail
* **Rules** - stop extending abstract test classes and start writing test rules
* **Theories** - write tests that are more like scientific experiments using randomly generated data
* **Test** **Fixtures** - specify set up and clean up methods on a per-method and per-class basis
* **Categories** - group your tests together for easier test filtering
* **Use** **with** **Maven** - how to use JUnit with your Maven build
* **Use** **with** **Gradle** - how to use JUnit with your Gradle build
* **Multithreaded code and Concurrency** - basic ideas behind testing concurrent code
* **Java contract test helpers**
* **Continuous** **Testing**

# Assertions

your bread and butter for unit testing

*Dariusz Andrzej Stefański edited this page on 19 Mar 2016 · 7 revisions*

JUnit provides overloaded assertion methods for all primitive types and Objects and arrays (of primitives or Objects). The parameter order is expected value followed by actual value. Optionally the first parameter can be a String message that is output on failure. There is a slightly different assertion, assertThat that has parameters of the optional failure message, the actual value, and a Matcher object. Note that expected and actual are reversed compared to the other assert methods.

## Examples

A representative of each assert method is shown.

import static org.hamcrest.CoreMatchers.allOf;

import static org.hamcrest.CoreMatchers.anyOf;

import static org.hamcrest.CoreMatchers.both;

import static org.hamcrest.CoreMatchers.containsString;

import static org.hamcrest.CoreMatchers.equalTo;

import static org.hamcrest.CoreMatchers.everyItem;

import static org.hamcrest.CoreMatchers.hasItems;

import static org.hamcrest.CoreMatchers.not;

import static org.hamcrest.CoreMatchers.sameInstance;

import static org.hamcrest.CoreMatchers.startsWith;

import static org.junit.Assert.assertArrayEquals;

import static org.junit.Assert.assertEquals;

import static org.junit.Assert.assertFalse;

import static org.junit.Assert.assertNotNull;

import static org.junit.Assert.assertNotSame;

import static org.junit.Assert.assertNull;

import static org.junit.Assert.assertSame;

import static org.junit.Assert.assertThat;

import static org.junit.Assert.assertTrue;

import java.util.Arrays;

import org.hamcrest.core.CombinableMatcher;

import org.junit.Test;

public class AssertTests {

@Test

public void testAssertArrayEquals() {

byte[] expected = "trial".getBytes();

byte[] actual = "trial".getBytes();

assertArrayEquals("failure - byte arrays not same", expected, actual);

}

@Test

public void testAssertEquals() {

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

}

@Test

public void testAssertFalse() {

assertFalse("failure - should be false", false);

}

@Test

public void testAssertNotNull() {

assertNotNull("should not be null", new Object());

}

@Test

public void testAssertNotSame() {

assertNotSame("should not be same Object", new Object(), new Object());

}

@Test

public void testAssertNull() {

assertNull("should be null", null);

}

@Test

public void testAssertSame() {

Integer aNumber = Integer.valueOf(768);

assertSame("should be same", aNumber, aNumber);

}

// JUnit Matchers assertThat

@Test

public void testAssertThatBothContainsString() {

assertThat("albumen", both(containsString("a")).and(containsString("b")));

}

@Test

public void testAssertThatHasItems() {

assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));

}

@Test

public void testAssertThatEveryItemContainsString() {

assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }),

everyItem(containsString("n")));

}

// Core Hamcrest Matchers with assertThat

@Test

public void testAssertThatHamcrestCoreMatchers() {

assertThat("good", allOf(equalTo("good"), startsWith("good")));

assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));

assertThat("good", anyOf(equalTo("bad"), equalTo("good")));

assertThat(7, not(CombinableMatcher.<Integer> either(equalTo(3)).or(equalTo(4))));

assertThat(new Object(), not(sameInstance(new Object())));

}

@Test

public void testAssertTrue() {

assertTrue("failure - should be true", true);

}

}

# Test runners

how tests should be executed

*cttillman edited this page on 18 Sep 2017 · 31 revisions*

## IDE support - graphical runners

NetBeans, Eclipse and IntelliJ IDEA have native graphical test runners built in.

## Console based Test runner

JUnit provides tools to define the suite to be run and to display its results. To run tests and see the results on the console, run this from a Java program:

org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...);

or this from the command line, with both your test class and junit on the classpath:

java org.junit.runner.JUnitCore TestClass1 [...other test classes...]

This usage is documented further here:

http://junit.org/javadoc/latest/org/junit/runner/JUnitCore.html

## Using older runners with Adapter

JUnit4TestAdapter enables running JUnit-4-style tests using a JUnit-3-style test runner. To use it, add the following to a test class:

public static Test suite() {

return new JUnit4TestAdapter('YourJUnit4TestClass'.class);

}

Caveat: See #1189 for issues with including a JUnit-4-style suite that contains a JUnit-3-style suite.

## @RunWith annotation

When a class is annotated with @RunWith or extends a class annotated with @RunWith, JUnit will invoke the class it references to run the tests in that class instead of the runner built into JUnit.

JavaDoc for @RunWith http://junit.org/javadoc/latest/org/junit/runner/RunWith.html

The default runner is BlockJUnit4ClassRunner which supersedes the older JUnit4ClassRunner

Annotating a class with @RunWith(JUnit4.class) will always invoke the default JUnit 4 runner in the current version of JUnit, this class aliases the current default JUnit 4 class runner.

## Specialized Runners

### Suite

* Suite is a standard runner that allows you to manually build a suite containing tests from many classes.
* More information at Aggregating tests in Suites page.
* http://junit.org/javadoc/latest/org/junit/runners/Suite.html

### Parameterized

* Parameterized is a standard runner that implements parameterized tests. When running a parameterized test class, instances are created for the cross-product of the test methods and the test data elements.
* More information at Parameterized Tests page.
* Javadoc: http://junit.org/javadoc/latest/org/junit/runners/Parameterized.html

### Categories

You can specify groups of tests to be excluded or included by using the Categories runner. Once you have annotated certain methods with @Category(MyCategory.class), you can use the --filter option to restrict which tests will be run:

java org.junit.runner.JunitCore

--filter=org.junit.experimental.categories.IncludeCategories=MyCat1,MyCat2

--filter=org.junit.experimental.categories.ExcludeCategories=MyCat3,MyCat4

You may filter tests according to any instance of FilterFactory. The --filter option takes the general form:

java [Runner] --filter=[FilterFactory]=[Categories,]

* Categories is a standard runner enabling subsets of tests tagged with certain categories to execute/be excluded from a given test run.
* More information at Categories page.

## Experimental Runners

### Enclosed

* Enclosed - If you put tests in inner classes, Ant, for example, won't find them. By running the outer class with Enclosed, the tests in the inner classes will be run. You might put tests in inner classes to group them for convenience or to share constants.
* Javadoc:
* http://junit.org/javadoc/latest/org/junit/experimental/runners/Enclosed.html
* Working Example of use on the 'Enclosed'-test-runner-example page

### Third Party Runners

Some popular third party implementations of runners for use with @RunWith include:

* SpringJUnit4ClassRunner
* MockitoJUnitRunner
* HierarchicalContextRunner
* Avh4's Nested
* NitorCreation's NestedRunner

# Aggregating tests in suites

how to combine multiple related tests into a test suite

*jason edited this page on 27 Oct 2017 · 7 revisions*

Using Suite as a runner allows you to manually build a suite containing tests from many classes. It is the JUnit 4 equivalent of the JUnit 3.8.x static Test suite() method. To use it, annotate a class with @RunWith(Suite.class) and @SuiteClasses(TestClass1.class, ...). When you run this class, it will run all the tests in all the suite classes.

## Example

The class below is a placeholder for the suite annotations, no other implementation is required. Note the @RunWith annotation, which specifies that the JUnit 4 test runner to use is org.junit.runners.Suite for running this particular test class. This works in conjunction with the @Suite.SuiteClasses annotation, which tells the Suite runner which test classes to include in this suite and in which order.

import org.junit.runner.RunWith;

import org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({

TestFeatureLogin.class,

TestFeatureLogout.class,

TestFeatureNavigate.class,

TestFeatureUpdate.class

})

public class FeatureTestSuite {

// the class remains empty,

// used only as a holder for the above annotations

}

# Test execution order

specifying what order to run unit tests

*Robin Hellemans edited this page on 28 Mar · 9 revisions*

## Test execution order

By design, JUnit does not specify the execution order of test method invocations. Until now, the methods were simply invoked in the order returned by the reflection API. However, using the JVM order is unwise since the Java platform does not specify any particular order, and in fact JDK 7 returns a more or less random order. Of course, well-written test code would not assume any order, but some do, and a predictable failure is better than a random failure on certain platforms.

From version 4.11, JUnit will by default use a deterministic, but not predictable, order (MethodSorters.DEFAULT). To change the test execution order simply annotate your test class using @FixMethodOrder and specify one of the available MethodSorters:

@FixMethodOrder(MethodSorters.JVM): Leaves the test methods in the order returned by the JVM. This order may vary from run to run.

@FixMethodOrder(MethodSorters.NAME\_ASCENDING): Sorts the test methods by method name, in lexicographic order.

## Example

import org.junit.FixMethodOrder;

import org.junit.Test;

import org.junit.runners.MethodSorters;

@FixMethodOrder(MethodSorters.NAME\_ASCENDING)

public class TestMethodOrder {

@Test

public void testA() {

System.out.println("first");

}

@Test

public void testB() {

System.out.println("second");

}

@Test

public void testC() {

System.out.println("third");

}

}

Above code will execute the test methods in the order of their names, sorted in ascending order

# Exception testing

how to specify expected exceptions in unit tests

*Marc Philipp edited this page on 11 Jan 2017 · 18 revisions*

## Expected Exceptions

How do you verify that code throws exceptions as expected? Verifying that code completes normally is important, but making sure the code behaves as expected in exceptional situations is vital too. For example:

new ArrayList<Object>().get(0);

This code should throw an IndexOutOfBoundsException. The @Test annotation has an optional parameter "expected" that takes as values subclasses of Throwable. If we wanted to verify that ArrayList throws the correct exception, we would write:

@Test(expected = IndexOutOfBoundsException.class)

public void empty() {

new ArrayList<Object>().get(0);

}

The expected parameter should be used with care. The above test will pass if any code in the method throws IndexOutOfBoundsException. For longer tests, it's recommended to use the ExpectedException rule, which is described below.

## Deeper Testing of the Exception

The above approach is useful for simple cases, but it has its limits. For example, you can't test the value of the message in the exception, or the state of a domain object after the exception has been thrown.

### Try/Catch Idiom

To address this you can use the try/catch idiom which prevailed in JUnit 3.x:

@Test

public void testExceptionMessage() {

try {

new ArrayList<Object>().get(0);

fail("Expected an IndexOutOfBoundsException to be thrown");

} catch (IndexOutOfBoundsException anIndexOutOfBoundsException) {

assertThat(anIndexOutOfBoundsException.getMessage(), is("Index: 0, Size: 0"));

}

}

### ExpectedException Rule

Alternatively, use the ExpectedException rule. This rule lets you indicate not only what exception you are expecting, but also the exception message you are expecting:

@Rule

public ExpectedException thrown = ExpectedException.none();

@Test

public void shouldTestExceptionMessage() throws IndexOutOfBoundsException {

List<Object> list = new ArrayList<Object>();

thrown.expect(IndexOutOfBoundsException.class);

thrown.expectMessage("Index: 0, Size: 0");

list.get(0); // execution will never get past this line

}

The expectMessage also lets you use Matchers, which gives you a bit more flexibility in your tests. An example:

thrown.expectMessage(Matchers.containsString("Size: 0"));

Moreover, you can use Matchers to inspect the Exception, useful if it has embedded state you wish to verify. For example

import static org.hamcrest.Matchers.hasProperty;

import static org.hamcrest.Matchers.is;

import static org.hamcrest.Matchers.startsWith;

import javax.ws.rs.NotFoundException;

import javax.ws.rs.core.Response;

import javax.ws.rs.core.Response.Status;

import org.junit.Rule;

import org.junit.Test;

import org.junit.rules.ExpectedException;

public class TestExy {

@Rule

public ExpectedException thrown = ExpectedException.none();

@Test

public void shouldThrow() {

TestThing testThing = new TestThing();

thrown.expect(NotFoundException.class);

thrown.expectMessage(startsWith("some Message"));

thrown.expect(hasProperty("response", hasProperty("status", is(404))));

testThing.chuck();

}

private class TestThing {

public void chuck() {

Response response = Response.status(Status.NOT\_FOUND).

entity("Resource not found").build();

throw new NotFoundException("some Message", response);

}

}

}

For an expanded discussion of the ExpectedException rule, see this blog post.

# Matchers and assertThat

how to use Hamcrest matchers and more descriptive assertions

*jason edited this page on 27 Oct 2017 · 1 revision*

## assertThat

Joe Walnes built a new assertion mechanism on top of what was then JMock 1. The method name was assertThat, and the syntax looked like this:

assertThat(x, is(3));

assertThat(x, is(not(4)));

assertThat(responseString, either(containsString("color")).or(containsString("colour")));

assertThat(myList, hasItem("3"));

More generally:

assertThat([value], [matcher statement]);

Advantages of this assertion syntax include:

More readable and typeable: this syntax allows you to think in terms of subject, verb, object (assert "x is 3") rather than assertEquals, which uses verb, object, subject (assert "equals 3 x")

Combinations: any matcher statement s can be negated (not(s)), combined (either(s).or(t)), mapped to a collection (each(s)), or used in custom combinations (afterFiveSeconds(s))

Readable failure messages. Compare:

assertTrue(responseString.contains("color") || responseString.contains("colour"));

// ==> failure message:

// java.lang.AssertionError:

assertThat(responseString, anyOf(containsString("color"), containsString("colour")));

// ==> failure message:

// java.lang.AssertionError:

// Expected: (a string containing "color" or a string containing "colour")

// got: "Please choose a font"

Custom Matchers. By implementing the Matcher interface yourself, you can get all of the above benefits for your own custom assertions.

For a more thorough description of these points, see Joe Walnes's original post.

We have decided to include this API directly in JUnit. It's an extensible and readable syntax, and it enables new features, like assumptions and theories.

Some notes:

The old assert methods are never, ever, going away. Developers may continue using the old assertEquals, assertTrue, and so on. The second parameter of an assertThat statement is a Matcher. We include the Matchers we want as static imports, like this:

import static org.hamcrest.CoreMatchers.is;

or:

import static org.hamcrest.CoreMatchers.\*;

Manually importing Matcher methods can be frustrating. Eclipse 3.3 includes the ability to define "Favorite" classes to import static methods from, which makes it easier (Search for "Favorites" in the Preferences dialog). We expect that support for static imports will improve in all Java IDEs in the future.

To allow compatibility with a wide variety of possible matchers, we have decided to include the classes from hamcrest-core, from the Hamcrest project. This is the first time that third-party classes have been included in JUnit.

JUnit currently ships with a few matchers, defined in org.hamcrest.CoreMatchers and

org.junit.matchers.JUnitMatchers.

To use many, many more, consider downloading the full hamcrest package:

* Hamcrest page http://code.google.com/p/hamcrest/wiki/Tutorial
* Hamcrest Java on GitHub: https://github.com/hamcrest/JavaHamcrest

JUnit contains special support for comparing string and array values, giving specific information on how they differ. This is not yet available using the assertThat syntax, but we hope to bring the two assert methods into closer alignment in future releases.

## JUnit Matchers

JUnit includes useful matchers for use with the assertThat method, but they are not currently included in the basic CoreMatchers class from hamcrest.

Javadoc JUnitMatchers http://junit.org/junit4/javadoc/latest/org/junit/matchers/JUnitMatchers.html

## Hamcrest CoreMatchers

Useful Hamcrest CoreMatchers are included in the JUnit distribution

* JavaDoc Hamcrest CoreMatchers

http://junit.org/junit4/javadoc/latest/org/hamcrest/CoreMatchers.html

## Thirdparty Matchers

Other, potentially Matchers out there include

* Excel spreadsheet matchers
* JSON matchers
* XML/XPath matchers

# Ignoring tests

how to disable test methods or classes

*Daan van Berkel edited this page on 30 Jul 2015 · 4 revisions*

## Ignoring a Test

If for some reason, you don't want a test to fail, you just want it ignored, you temporarily disable a test.

To ignore a test in JUnit you can either comment a method, or delete the @Test annotation; but the test runner will not report such a test. Alternatively, you can add the @Ignore annotation in front or after @Test. Test runners will report the number of ignored tests, along with the number of tests that ran and the number of tests that failed.

Note that @Ignore takes an optional parameter (a String) if you want to record a reason why a test is being ignored.

@Ignore("Test is ignored as a demonstration")

@Test

public void testSame() {

assertThat(1, is(1));

}

# Timeout for tests

how to specify maximum execution times for tests

AW edited this page on 23 Mar 2016 · 14 revisions

Tests that 'runaway' or take too long, can be automatically failed. There are two options for implementing this behavior:

## Timeout parameter on @Test Annotation (applies to test method)

You can optionally specify timeout in milliseconds to cause a test method to fail if it takes longer than that number of milliseconds. If the time limit is exceeded, then the failure is triggered by an Exception being thrown:

@Test(timeout=1000)

public void testWithTimeout() {

...

}

This is implemented by running the test method in a separate thread. If the test runs longer than the allotted timeout, the test will fail and JUnit will interrupt the thread running the test. If a test times out while executing an interruptible operation, the thread running the test will exit (if the test is in an infinite loop, the thread running the test will run forever, while other tests continue to execute).

## Timeout Rule (applies to all test cases in the test class)

The Timeout Rule applies the same timeout to all test methods in a class, and will currently execute in addition to any timeout specified by the timeout parameter on an individual Test annotation.:

import org.junit.Rule;

import org.junit.Test;

import org.junit.rules.Timeout;

public class HasGlobalTimeout {

public static String log;

private final CountDownLatch latch = new CountDownLatch(1);

@Rule

public Timeout globalTimeout = Timeout.seconds(10); // 10 seconds max per method tested

@Test

public void testSleepForTooLong() throws Exception {

log += "ran1";

TimeUnit.SECONDS.sleep(100); // sleep for 100 seconds

}

@Test

public void testBlockForever() throws Exception {

log += "ran2";

latch.await(); // will block

}

}

The timeout specified in the Timeout rule applies to the entire test fixture, including any @Before or @After methods. If the test method is in an infinite loop (or is otherwise not responsive to interrupts) then @After methods will not be called.

# Parameterized tests

writing tests that can be executed multiple times with different parameter values

*Jagadeeswara Rao Adireddi edited this page on 20 Mar · 25 revisions*

The custom runner Parameterized implements parameterized tests. When running a parameterized test class, instances are created for the cross-product of the test methods and the test data elements.

For example, to test a Fibonacci function, write:

import static org.junit.Assert.assertEquals;

import java.util.Arrays;

import java.util.Collection;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.junit.runners.Parameterized;

import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)

public class FibonacciTest {

@Parameters

public static Collection<Object[]> data() {

return Arrays.asList(new Object[][] {

{ 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }

});

}

private int fInput;

private int fExpected;

public FibonacciTest(int input, int expected) {

fInput= input;

fExpected= expected;

}

@Test

public void test() {

assertEquals(fExpected, Fibonacci.compute(fInput));

}

}

public class Fibonacci {

public static int compute(int n) {

int result = 0;

if (n <= 1) {

result = n;

} else {

result = compute(n - 1) + compute(n - 2);

}

return result;

}

}

Each instance of FibonacciTest will be constructed using the two-argument constructor and the data values in the @Parameters method.

## Using @Parameter for Field injection instead of Constructor

It is also possible to inject data values directly into fields without needing a constructor using the @Parameter annotation, like so:

import static org.junit.Assert.assertEquals;

import java.util.Arrays;

import java.util.Collection;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.junit.runners.Parameterized;

import org.junit.runners.Parameterized.Parameter;

import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)

public class FibonacciTest {

@Parameters

public static Collection<Object[]> data() {

return Arrays.asList(new Object[][] {

{ 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }

});

}

@Parameter // first data value (0) is default

public /\* NOT private \*/ int fInput;

@Parameter(1)

public /\* NOT private \*/ int fExpected;

@Test

public void test() {

assertEquals(fExpected, Fibonacci.compute(fInput));

}

}

public class Fibonacci {

...

}

This currently only works for public fields (see https://github.com/junit-team/junit/pull/737).

## Tests with single parameter

(Since 4.12-beta-3)

If your test needs a single parameter only, you don't have to wrap it with an array. Instead you can provide an Iterable or an array of objects.

@Parameters

public static Iterable<? extends Object> data() {

return Arrays.asList("first test", "second test");

}

or

@Parameters

public static Object[] data() {

return new Object[] { "first test", "second test" };

}

## Identify Individual test cases

In order to easily identify the individual test cases in a Parameterized test, you may provide a name using the @Parameters annotation. This name is allowed to contain placeholders that are replaced at runtime:

* {index}: the current parameter index
* {0}, {1}, …: the first, second, and so on, parameter value. NOTE: single quotes ' should be escaped as two single quotes ''.

## Example

import static org.junit.Assert.assertEquals;

import java.util.Arrays;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.junit.runners.Parameterized;

import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)

public class FibonacciTest {

@Parameters(name = "{index}: fib({0})={1}")

public static Iterable<Object[]> data() {

return Arrays.asList(new Object[][] {

{ 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }

});

}

private int input;

private int expected;

public FibonacciTest(int input, int expected) {

this.input = input;

this.expected = expected;

}

@Test

public void test() {

assertEquals(expected, Fibonacci.compute(input));

}

}

public class Fibonacci {

...

}

In the example given above, the Parameterized runner creates names like [3: fib(3)=2]. If you don't specify a name, the current parameter index will be used by default.

## IDE Bug (Eclipse)

If using the name annotation param and one of the inputs has a rounded bracket, e.g. @Parameters(name = "test({index})"), then the name gets truncated in Eclipse versions prior to 4.4 (Luna). See https://bugs.eclipse.org/bugs/show\_bug.cgi?id=102512.

Before the Mars M4 release Eclipse wasn't able to run individual test subtrees, such as the ones create by the Parameterized runner. See http://blog.moritz.eysholdt.de/2014/11/new-eclipse-junit-feature-run-subtrees.html and https://bugs.eclipse.org/bugs/show\_bug.cgi?id=443498.

## See also

* As an alternative to parameterized tests you can also use the plugin JUnitParams
* If you want to define the parameters for your tests at the tests' Suite, you can use the ParameterizedSuite runner that is available in a separate library.

# Assumptions with assume

similar to assertions, but without making tests fail

*Daan van Berkel edited this page on 30 Jul 2015 · 4 revisions*

## Assumptions

Ideally, the developer writing a test has control of all of the forces that might cause a test to fail. If this isn't immediately possible, making dependencies explicit can often improve a design. For example, if a test fails when run in a different locale than the developer intended, it can be fixed by explicitly passing a locale to the domain code.

However, sometimes this is not desirable or possible. It's good to be able to run a test against the code as it is currently written, implicit assumptions and all, or to write a test that exposes a known bug. For these situations, JUnit now includes the ability to express "assumptions":

import static org.junit.Assume.\*

@Test public void filenameIncludesUsername() {

assumeThat(File.separatorChar, is('/'));

assertThat(new User("optimus").configFileName(), is("configfiles/optimus.cfg"));

}

@Test public void correctBehaviorWhenFilenameIsNull() {

assumeTrue(bugFixed("13356")); // bugFixed is not included in JUnit

assertThat(parse(null), is(new NullDocument()));

}

The default JUnit runner treats tests with failing assumptions as ignored. Custom runners may behave differently.

We have included assumeTrue for convenience, but thanks to the inclusion of Hamcrest, we do not need to create assumeEquals, assumeSame, and other analogues to the assert\* methods. All of those functionalities are subsumed in assumeThat, with the appropriate matcher.

A failing assumption in a @Before or @BeforeClass method will have the same effect as a failing assumption in each @Test method of the class.