

JMock report



By: Ahmed kamel taha

# What is JMock?

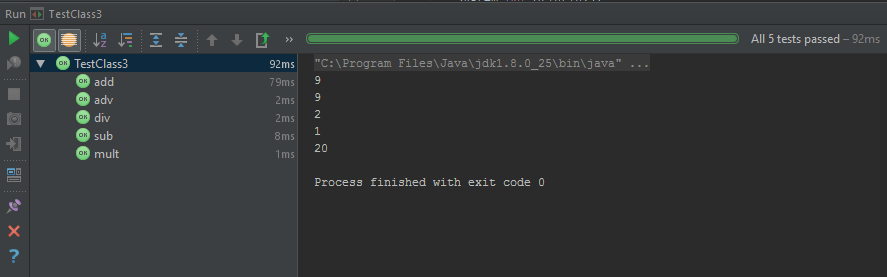
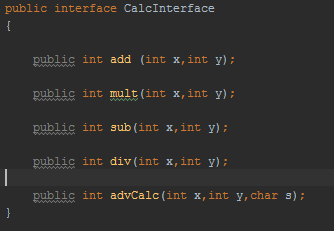
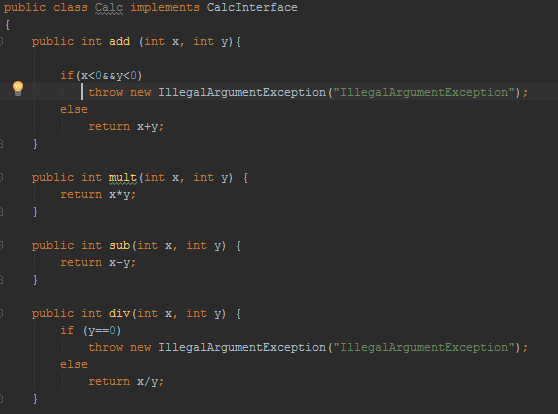
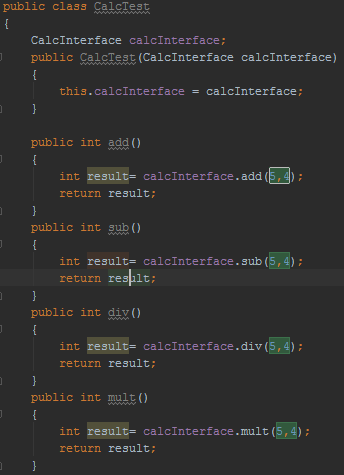
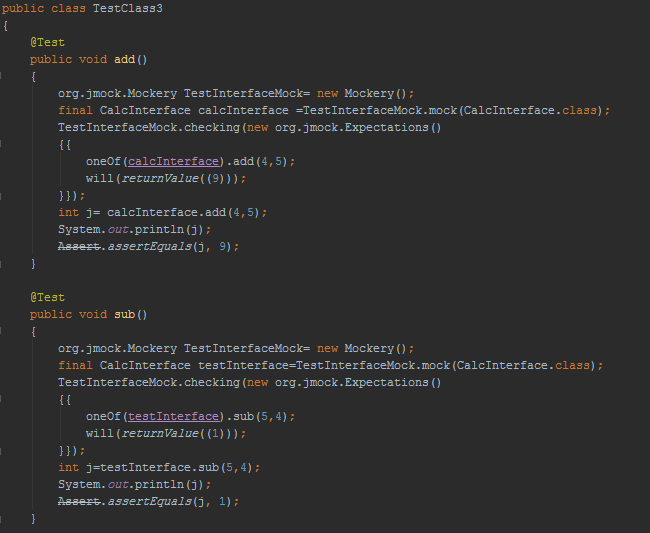
JMock is a library that supports test-driven development of Java code with mock objects.

Mock objects help you design and test the interactions between the objects in your programs.

# Features:

* makes it quick and easy to define mock objects, so you don't break the rhythm of programming.
* lets you precisely specify the interactions between your objects, reducing the brittleness of your tests.
* works well with the auto completion and refactoring features of your IDE
* plugs into your favorite test framework
* is easy to extend.

# Screenshots from the project:



# jMock 2 Cheat Sheet

## Test Fixture Class

Show for: [JUnit 3](javascript:selectTestFramework('JUnit3'))[JUnit 4](javascript:selectTestFramework('JUnit4'))[Other](javascript:selectTestFramework('Raw'))

import org.jmock.Expectations;

import org.jmock.Mockery;

import org.jmock.integration.junit4.JMock;

import org.jmock.integration.junit4.JUnit4Mockery;

@RunWith(JMock.class)

public class AJMock2TestCase {

Mockery context = new JUnit4Mockery();

...

}

A Mockery represents the context of the object under test: the objects that it communicates with. A Mockery creates mock objects and checks expectations that are set upon those mock objects. By convention the Mockery is stored in an instance variable named context.

Tests written with JUnit 4 do not need to extend a specific base class but must specify that they use jMock with the @RunWith attribute, create a JUnit4Mockery that reports expectation failures as JUnit 4 test failures, and store the Mockery in an instance variable.

## Creating Mock Objects

Show for: [JUnit 3](javascript:selectTestFramework('JUnit3'))[JUnit 4](javascript:selectTestFramework('JUnit4'))[Other](javascript:selectTestFramework('Raw'))

Turtle turtle = context.mock(Turtle.class);

Turtle turtle2 = context.mock(Turtle.class, "turtle2");

The examples above assume that the mock object is stored in an instance variable. If a mock object is stored in a local variable, the variable must be declared as final so that it can be referred to from within expectation blocks (see below).

## Tests with Expectations

Show for: [JUnit 3](javascript:selectTestFramework('JUnit3'))[JUnit 4](javascript:selectTestFramework('JUnit4'))[Other](javascript:selectTestFramework('Raw'))

... create mock objects ...

public void testSomeAction() {

... set up ...

context.checking(new Expectations() {{

... expectations go here ...

}});

... call code being tested ...

... assertions ...

}

The JUnit 3 and JUnit 4 integration layers automatically assert that all expectations have been satisfied.

An expectations block can contain any number of expectations. A test can contain multiple expectation blocks. Expectations in later blocks are appended to those in earlier blocks. Expectations can be interspersed with calls to the code under test.

## Expectations

Expectations have the following structure:

invocation-count (mock-object).method(argument-constraints);

inSequence(sequence-name);

when(state-machine.is(state-name));

will(action);

then(state-machine.is(new-state-name));

Except for the invocation count and the mock object, all clauses are optional. You can give an expectation as many inSequence, when, will and then clauses as you wish.

Some examples:

oneOf (turtle).turn(45); // The turtle will be told to turn 45 degrees once only

allowing (turtle).flashLEDs(); // The turtle can be told to flash its LEDs any number of types or not at all

ignoring (turtle2); // Turtle 2 can be told to do anything. This test ignores it.

allowing (turtle).queryPen(); // The turtle can be asked about its pen any number of times and will always

will(returnValue(PEN\_DOWN)); // return PEN\_DOWN

atLeast(1).of (turtle).stop(); // The turtle will be told to stop at least once.

### **Invocation Count**

|  |  |
| --- | --- |
| oneOf | The invocation is expected once and once only. |
| exactly(n).of | The invocation is expected exactly n times. Note: one is a convenient shorthand for exactly(1). |
| atLeast(n).of | The invocation is expected at least n. |
| atMost(n).of | The invocation is expected at most n times. |
| between(min, max).of | The invocation is expected at least min times and at most max times. |
| allowing | The invocation is allowed any number of times but does not have to happen. |
| ignoring | The same as allowing. Allowing or ignoring should be chosen to make the test code clearly express intent. |
| never | The invocation is not expected at all. This is used to make tests more explicit and so easier to understand. |

### **Methods And Expectations**

Expected methods are specified by a literal call to the method within an expectation block.

Arguments passed to the method in an expectation will be compared for equality.

oneOf (calculator).add(1, 1); will(returnValue(2));

oneOf (calculator).add(2, 2); will(returnValue(5));

To define looser constraints, specify all arguments as matchers within with clauses:

allowing (calculator).add(with(any(int.class)), with(any(int.class)));

### **Argument Matchers**

The most commonly used matchers are defined in the [Expectations](http://www.jmock.org/javadoc/2.0.0/org/jmock/Expectations.html) class:

|  |  |
| --- | --- |
| equal(n) | The argument is equal to n. |
| same(o) | The argument is the same object as o. |
| any(Class<T> type) | The argument is any value. The type argument is required to force Java to type-check the argument at compile time. |
| a(Class<T> type) an(Class<T> type) | The argument is an instance of type or a subclass of type. |
| aNull(Class<T> type) | The argument is null. The type argument is required to force Java to type-check the argument at compile time. |
| aNonNull(Class<T> type) | The argument is not null. The type argument is required to force Java to type-check the argument at compile time. |
| not(m) | The argument does not match the Matcher m. |

More matchers are defined as static factory methods of the [Hamcrest Matchers class](http://www.jmock.org/javadoc/2.0.0/org/hamcrest/Matchers.html), which can be statically imported at the top of the test code. For example,

|  |  |
| --- | --- |
| anyOf(m1, m2, ..., mn) | The argument matches one of the Matchers m1 to mn. |
| allOf(m1, m2, ..., mn) | The argument matches all of the Matchers m1 to mn. |

### **Actions**

|  |  |
| --- | --- |
| will(returnValue(v)) | Return v to the caller. |
| will(returnIterator(c)) | Return a new iterator over collection c on each invocation. |
| will(returnIterator(v1, v2, ..., vn)) | Return a new iterator over elements v1 to vn on each invocation. |
| will(throwException(e)) | Throw e to the caller. |
| will(doAll(a1, a2, ..., an)) | Do all actions a1 to an on every invocation. |

### **Sequences**

Invocations that are expected in a sequence must occur in the order in which they appear in the test code. A test can create more than one sequence and an expectation can be part of more than once sequence at a time.

To define a new sequence:

Show for: [JUnit 3](javascript:selectTestFramework('JUnit3'))[JUnit 4](javascript:selectTestFramework('JUnit4'))[Other](javascript:selectTestFramework('Raw'))

final Sequence sequence-name = context.sequence("sequence-name");

To expect a sequence of invocations, write the expectations in order and add the inSequence(sequence) clause to each one. For example:

oneOf (turtle).forward(10); inSequence(drawing);

oneOf (turtle).turn(45); inSequence(drawing);

oneOf (turtle).forward(10); inSequence(drawing);

Expectations in a sequence can have any invocation count.

## States

States are used to constrain invocations to occur only when a condition is true. Conditions are represented as states of state machines. A test can create multiple state machines and each state machine can have multiple states. An invocation can be constrained to occur during a state of one more more state machines.

To define a new state machine:

Show for: [JUnit 3](javascript:selectTestFramework('JUnit3'))[JUnit 4](javascript:selectTestFramework('JUnit4'))[Other](javascript:selectTestFramework('Raw'))

final States state-machine-name = context.states("state-machine-name").startsAs("initial-state");

The intial state is optional. If not specified, the state machine starts in an unnamed initial state.

The following clauses constrain invocations to occur within specific states and define how an invocation will change the current state of a state machine.

|  |  |
| --- | --- |
| when(state-machine.is("state-name")); | Constrains the last expectation to occur only when the state machine is in the named state. |
| then(state-machine.is("state-name")); | Changes the state of state-machine to the named state when the invocation occurs. |

For example:

final States pen = context.states("pen").startsAs("up");

oneOf (turtle).penDown(); then(pen.is("down"));

oneOf (turtle).forward(10); when(pen.is("down"));

oneOf (turtle).turn(90); when(pen.is("down"));

oneOf (turtle).forward(10); when(pen.is("down"));

oneOf (turtle).penUp(); then(pen.is("up"));