Mockito library enables mocks creation, verification and stubbing.
This javadoc content is also available on the http://mockito.org web page. All documentation is kept in javadocs because it guarantees consistency between what's on the web and what's in the source code. Also, it makes possible to access documentation straight from the IDE even if you work offline.

Contents

Following examples mock a List, because everyone knows its interface (methods like add(), get(), clear() will be used).
You probably wouldn't mock List class 'in real'.

1. Let's verify some behaviour!

```
//Let's import Mockito statically so that the code looks clearer
 import static org.mockito.Mockito.*;

 //mock creation
 List mockedList = mock(List.class);

 //using mock object
 mockedList.add("one");
 mockedList.clear();

 //verification
 verify(mockedList).add("one");
 verify(mockedList).clear();
```

Once created, mock will remember all interactions. Then you can selectively verify whatever interaction you are interested in.

2. How about some stubbing?

```
//You can mock concrete classes, not only interfaces
 LinkedList mockedList = mock(LinkedList.class);

 //stubbing
 when(mockedList.get(0)).thenReturn("first");
 when(mockedList.get(1)).thenThrow(new RuntimeException());

 //following prints "first"
 System.out.println(mockedList.get(0));

 //following throws runtime exception
 System.out.println(mockedList.get(1));

 //following prints "null" because get(999) was not stubbed
 System.out.println(mockedList.get(999));

 //Although it is possible to verify a stubbed invocation, usually it's just
redundant
 //If your code cares what get(0) returns then something else breaks (often before
even verify() gets executed).
 //If your code doesn't care what get(0) returns then it should not be stubbed. Not
convinced? See here.
 verify(mockedList).get(0);
```

- By default, for all methods that return value, mock returns null, an empty collection or appropriate primitive/primitive wrapper value (e.g: 0, false, ... for int/Integer, boolean/Boolean, ...).
- Stubbing can be overridden: for example common stubbing can go to fixture setup but the test methods can override it. Please note that overridding stubbing is a potential code smell that points out too much stubbing

- Once stubbed, the method will always return stubbed value regardless of how many times it is called.
- Last stubbing is more important - when you stubbed the same method with the same arguments many times.

3. Argument matchers

Mockito verifies argument values in natural java style: by using an equals() method. Sometimes, when extra flexibility is required then you might use argument matchers:

```
//stubbing using built-in anyInt() argument matcher
 when(mockedList.get(anyInt())).thenReturn("element");

 //stubbing using hamcrest (let's say isValid() returns your own hamcrest matcher):
 when(mockedList.contains(argThat(isValid()))).thenReturn("element");

 //following prints "element"
 System.out.println(mockedList.get(999));

 //you can also verify using an argument matcher
 verify(mockedList).get(anyInt());
```

Argument matchers allow flexible verification or stubbing. `Click here to see` more built-in matchers and examples of **custom argument matchers / hamcrest matchers**.
For information solely on **custom argument matchers** check out javadoc for `ArgumentMatcher` class.
Be reasonable with using complicated argument matching. The natural matching style using equals() with occasional anyX() matchers tend to give clean & simple tests. Sometimes it's just better to refactor the code to allow equals() matching or even implement equals() method to help out with testing.
Also, read section 15 or javadoc for `ArgumentCaptor` class. `ArgumentCaptor` is a special implementation of an argument matcher that captures argument values for further assertions.
Warning on argument matchers:
If you are using argument matchers, **all arguments** have to be provided by matchers.
E.g: (example shows verification but the same applies to stubbing):

```
  verify(mock).someMethod(anyInt(), anyString(), eq("third argument"));
   //above is correct - eq() is also an argument matcher

   verify(mock).someMethod(anyInt(), anyString(), "third argument");
   //above is incorrect - exception will be thrown because third argument is given
without an argument matcher.
```

4. Verifying exact number of invocations / at least x / never

```
//using mock
 mockedList.add("once");

 mockedList.add("twice");
 mockedList.add("twice");
```

```
mockedList.add("three times");
mockedList.add("three times");
mockedList.add("three times");

//following two verifications work exactly the same - times(1) is used by default
verify(mockedList).add("once");
verify(mockedList, times(1)).add("once");

//exact number of invocations verification
verify(mockedList, times(2)).add("twice");
verify(mockedList, times(3)).add("three times");

//verification using never(). never() is an alias to times(0)
verify(mockedList, never()).add("never happened");

//verification using atLeast()/atMost()
verify(mockedList, atLeastOnce()).add("three times");
verify(mockedList, atLeast(2)).add("five times");
verify(mockedList, atMost(5)).add("three times");
```

times(1) is the default. Therefore using times(1) explicitly can be omitted.

5. Stubbing void methods with exceptions

```
doThrow(new RuntimeException()).when(mockedList).clear();

//following throws RuntimeException:
mockedList.clear();
```

Read more about doThrow|doAnswer family of methods in paragraph 12.
Initially, `stubVoid(Object)` was used for stubbing voids. Currently stubVoid() is deprecated in favor of `doThrow(Throwable)`. This is because of improved readability and consistency with the family of `doAnswer(Answer)` methods.

6. Verification in order

```
List firstMock = mock(List.class);
List secondMock = mock(List.class);

//using mocks
firstMock.add("was called first");
secondMock.add("was called second");

//create inOrder object passing any mocks that need to be verified in order
InOrder inOrder = inOrder(firstMock, secondMock);

//following will make sure that firstMock was called before secondMock
```

```
 inOrder.verify(firstMock).add("was called first");
 inOrder.verify(secondMock).add("was called second");
```

Verification in order is flexible - **you don't have to verify all interactions** one-by-one but only those that you are interested in testing in order.
Also, you can create InOrder object passing only mocks that are relevant for in-order verification.

7. Making sure interaction(s) never happened on mock

```
//using mocks - only mockOne is interacted
 mockOne.add("one");

 //ordinary verification
 verify(mockOne).add("one");

 //verify that method was never called on a mock
 verify(mockOne, never()).add("two");

 //verify that other mocks were not interacted
 verifyZeroInteractions(mockTwo, mockThree);
```

8. Finding redundant invocations

```
//using mocks
 mockedList.add("one");
 mockedList.add("two");

 verify(mockedList).add("one");

 //following verification will fail
 verifyNoMoreInteractions(mockedList);
```

A word of **warning**: Some users who did a lot of classic, expect-run-verify mocking tend to use verifyNoMoreInteractions() very often, even in every test method. verifyNoMoreInteractions() is not recommended to use in every test method. verifyNoMoreInteractions() is a handy assertion from the interaction testing toolkit. Use it only when it's relevant. Abusing it leads to overspecified, less maintainable tests. You can find further reading here.
See also `never()` - it is more explicit and communicates the intent well.

9. Shorthand for mocks creation - @Mock annotation

- Minimizes repetitive mock creation code.
- Makes the test class more readable.
- Makes the verification error easier to read because the **field name** is used to identify the mock.

```
 public class ArticleManagerTest {
```

```
    @Mock private ArticleCalculator calculator;
    @Mock private ArticleDatabase database;
    @Mock private UserProvider userProvider;

    private ArticleManager manager;
```

Important! This needs to be somewhere in the base class or a test runner:
`MockitoAnnotations.initMocks(testClass);`

You can use built-in runner: `MockitoJUnitRunner`.
Read more here: `MockitoAnnotations`

10. Stubbing consecutive calls (iterator-style stubbing)

Sometimes we need to stub with different return value/exception for the same method call. Typical use case could be mocking iterators. Original version of Mockito did not have this feature to promote simple mocking. For example, instead of iterators one could use `Iterable` or simply collections. Those offer natural ways of stubbing (e.g. using real collections). In rare scenarios stubbing consecutive calls could be useful, though:

```
when(mock.someMethod("some arg"))
   .thenThrow(new RuntimeException())
   .thenReturn("foo");

 //First call: throws runtime exception:
 mock.someMethod("some arg");

 //Second call: prints "foo"
 System.out.println(mock.someMethod("some arg"));

 //Any consecutive call: prints "foo" as well (last stubbing wins).
 System.out.println(mock.someMethod("some arg"));
```

Alternative, shorter version of consecutive stubbing:

```
when(mock.someMethod("some arg"))
   .thenReturn("one", "two", "three");
```

11. Stubbing with callbacks

Allows stubbing with generic `Answer` interface.
Yet another controversial feature which was not included in Mockito originally. We recommend using simple stubbing with thenReturn() or thenThrow() only. Those two should be **just enough** to test/test-drive any clean & simple code.

```
when(mock.someMethod(anyString())).thenAnswer(new Answer() {
     Object answer(InvocationOnMock invocation) {
         Object[] args = invocation.getArguments();
         Object mock = invocation.getMock();
         return "called with arguments: " + args;
```

```
    }
});

//Following prints "called with arguments: foo"
System.out.println(mock.someMethod("foo"));
```

12. doThrow()|doAnswer()|doNothing()|doReturn() family of methods for stubbing voids (mostly)

Stubbing voids requires different approach from `when(Object)` because the compiler does not like void methods inside brackets...
`doThrow(Throwable)` replaces the `stubVoid(Object)` method for stubbing voids. The main reason is improved readability and consistency with the family of doAnswer() methods.
Use doThrow() when you want to stub a void method with an exception:

```
  doThrow(new RuntimeException()).when(mockedList).clear();

   //following throws RuntimeException:
   mockedList.clear();
```

Read more about other methods:
`doThrow(Throwable)`
`doAnswer(Answer)`
`doNothing()`
`doReturn(Object)`

13. Spying on real objects

You can create spies of real objects. When you use the spy then the **real** methods are called (unless a method was stubbed).
Real spies should be used **carefully and occasionally**, for example when dealing with legacy code.
Spying on real objects can be associated with "partial mocking" concept. **Before the release 1.8**, Mockito spies were not real partial mocks. The reason was we thought partial mock is a code smell. At some point we found legitimate use cases for partial mocks (3rd party interfaces, interim refactoring of legacy code, the full article is here)

```
  List list = new LinkedList();
   List spy = spy(list);

   //optionally, you can stub out some methods:
   when(spy.size()).thenReturn(100);

   //using the spy calls real methods
   spy.add("one");
   spy.add("two");

   //prints "one" - the first element of a list
   System.out.println(spy.get(0));
```

```
   //size() method was stubbed - 100 is printed
   System.out.println(spy.size());

   //optionally, you can verify
   verify(spy).add("one");
   verify(spy).add("two");
```

Important gotcha on spying real objects!

1. Sometimes it's impossible to use `when(Object)` for stubbing spies. Example:

```
  List list = new LinkedList();
   List spy = spy(list);

   //Impossible: real method is called so spy.get(0) throws IndexOutOfBoundsException
(the list is yet empty)
   when(spy.get(0)).thenReturn("foo");

   //You have to use doReturn() for stubbing
   doReturn("foo").when(spy).get(0);
```

2. Watch out for final methods. Mockito doesn't mock final methods so the bottom line is: when you spy on real objects + you try to stub a final method = trouble. What will happen is the real method will be called *on mock* but *not on the real instance* you passed to the spy() method. Typically you may get a NullPointerException because mock instances don't have fields initiated.

14. Changing default return values of unstubbed invocations (Since 1.7)

You can create a mock with specified strategy for its return values. It's quite advanced feature and typically you don't need it to write decent tests. However, it can be helpful for working with **legacy systems**.

It is the default answer so it will be used **only when you don't** stub the method call.

```
  Foo mock = mock(Foo.class, Mockito.RETURNS_SMART_NULLS);
   Foo mockTwo = mock(Foo.class, new YourOwnAnswer());
```

Read more about this interesting implementation of *Answer*: `RETURNS_SMART_NULLS`

15. Capturing arguments for further assertions (Since 1.8.0)

Mockito verifies argument values in natural java style: by using an equals() method. This is also the recommended way of matching arguments because it makes tests clean & simple. In some situations though, it is helpful to assert on certain arguments after the actual verification. For example:

```
  ArgumentCaptor<Person> argument = ArgumentCaptor.forClass(Person.class);
   verify(mock).doSomething(argument.capture());
   assertEquals("John", argument.getValue().getName());
```

Warning: it is recommended to use ArgumentCaptor with verification **but not** with stubbing. Using ArgumentCaptor with stubbing may decrease test readability because captor is created outside of assert

(aka verify or 'then') block. Also it may reduce defect localization because if stubbed method was not called then no argument is captured.
In a way ArgumentCaptor is related to custom argument matchers (see javadoc for `ArgumentMatcher` class). Both techniques can be used for making sure certain arguments where passed to mocks. However, ArgumentCaptor may be a better fit if:

- custom argument matcher is not likely to be reused
- you just need it to assert on argument values to complete verification

Custom argument matchers via `ArgumentMatcher` are usually better for stubbing.

16. Real partial mocks (Since 1.8.0)

Finally, after many internal debates & discussions on the mailing list, partial mock support was added to Mockito. Previously we considered partial mocks as code smells. However, we found a legitimate use case for partial mocks - more reading: here
Before release 1.8 spy() was not producing real partial mocks and it was confusing for some users. Read more about spying: here or in javadoc for `spy(Object)` method.

```
   //you can create partial mock with spy() method:
    List list = spy(new LinkedList());

    //you can enable partial mock capabilities selectively on mocks:
    Foo mock = mock(Foo.class);
    //Be sure the real implementation is 'safe'.
    //If real implementation throws exceptions or depends on specific state of the
object then you're in trouble.
    when(mock.someMethod()).thenCallRealMethod();
```

As usual you are going to read **the partial mock warning**: Object oriented programming is more less tackling complexity by dividing the complexity into separate, specific, SRPy objects. How does partial mock fit into this paradigm? Well, it just doesn't... Partial mock usually means that the complexity has been moved to a different method on the same object. In most cases, this is not the way you want to design your application.
However, there are rare cases when partial mocks come handy: dealing with code you cannot change easily (3rd party interfaces, interim refactoring of legacy code etc.) However, I wouldn't use partial mocks for new, test-driven & well-designed code.

17. Resetting mocks (Since 1.8.0)

Smart Mockito users hardly use this feature because they know it could be a sign of poor tests. Normally, you don't need to reset your mocks, just create new mocks for each test method.
Instead of reset() please consider writing simple, small and focused test methods over lengthy, over-specified tests. **First potential code smell is reset() in the middle of the test method.** This probably means you're testing too much. Follow the whisper of your test methods: "Please keep us small & focused on single behavior". There are several threads about it on mockito mailing list.
The only reason we added reset() method is to make it possible to work with container-injected mocks. See issue 55 (here) or FAQ (here).
Don't harm yourself. reset() in the middle of the test method is a code smell (you're probably testing too much).

```
  List mock = mock(List.class);
```

```
  when(mock.size()).thenReturn(10);
  mock.add(1);

  reset(mock);
  //at this point the mock forgot any interactions & stubbing
```

18. Troubleshooting & validating framework usage (Since 1.8.0)

First of all, in case of any trouble, I encourage you to read the Mockito FAQ: http://code.google.com/p/mockito/wiki/FAQ
In case of questions you may also post to mockito mailing list: http://groups.google.com/group/mockito
Next, you should know that Mockito validates if you use it correctly **all the time**. However, there's a gotcha so please read the javadoc for `validateMockitoUsage()`

19. Aliases for behavior driven development (Since 1.8.0)

Behavior Driven Development style of writing tests uses **//given //when //then** comments as fundamental parts of your test methods. This is exactly how we write our tests and we warmly encourage you to do so! Start learning about BDD here: http://en.wikipedia.org/wiki/Behavior_Driven_Development
The problem is that current stubbing api with canonical role of **when** word does not integrate nicely with **//given //when //then** comments. It's because stubbing belongs to **given** component of the test and not to the **when**component of the test. Hence `BDDMockito` class introduces an alias so that you stub method calls with `BDDMockito.given(Object)` method. Now it really nicely integrates with the **given** component of a BDD style test!
Here is how the test might look like:

```
import static org.mockito.BDDMockito.*;

 Seller seller = mock(Seller.class);
 Shop shop = new Shop(seller);

 public void shouldBuyBread() throws Exception {
   //given
   given(seller.askForBread()).willReturn(new Bread());

   //when
   Goods goods = shop.buyBread();

   //then
   assertThat(goods, containBread());
 }
```

20. (**New**) Serializable mocks (Since 1.8.1)

Mocks can be made serializable. With this feature you can use a mock in a place that requires dependencies to be serializable.
WARNING: This should be rarely used in unit testing.

The behaviour was implemented for a specific use case of a BDD spec that had an unreliable external dependency. This was in a web environment and the objects from the external dependency were being serialized to pass between layers.
To create serializable mock use `MockSettings.serializable()`:

```
  List serializableMock = mock(List.class, withSettings().serializable());
```

The mock can be serialized assuming all the normal serialization requirements are met by the class.
Making a real object spy serializable is a bit more effort as the spy(...) method does not have an overloaded version which accepts MockSettings. No worries, you will hardly ever use it.

```
List<Object> list = new ArrayList<Object>();
 List<Object> spy = mock(ArrayList.class, withSettings()
                 .spiedInstance(list)
                 .defaultAnswer(CALLS_REAL_METHODS)
                 .serializable());
```

21. (**New**) New annotations: @Captor, @Spy, @InjectMocks (Since 1.8.3)

Release 1.8.3 brings new annotations that may be helpful on occasion:

- @`Captor` simplifies creation of `ArgumentCaptor` - useful when the argument to capture is a nasty generic class and you want to avoid compiler warnings
- @`Spy` - you can use it instead `spy(Object)`.
- @`InjectMocks` - injects mocks into tested object automatically.

All new annotations are *only* processed on `MockitoAnnotations.initMocks(Object)`

22. (**New**) Verification with timeout (Since 1.8.5)

Allows verifying with timeout. May be useful for testing in concurrent conditions.
It feels this feature should be used rarely - figure out a better way of testing your multi-threaded system.
Not yet implemented to work with InOrder verification.
Examples:

```
  //passes when someMethod() is called within given time span
   verify(mock, timeout(100)).someMethod();
   //above is an alias to:
   verify(mock, timeout(100).times(1)).someMethod();

   //passes when someMethod() is called *exactly* 2 times within given time span
   verify(mock, timeout(100).times(2)).someMethod();

   //passes when someMethod() is called *at lest* 2 times within given time span
   verify(mock, timeout(100).atLeast(2)).someMethod();

   //verifies someMethod() within given time span using given verification mode
   //useful only if you have your own custom verification modes.
   verify(mock, new Timeout(100, yourOwnVerificationMode)).someMethod();
```