org.mockito
Class Mockito

```
java.lang.Object
  └org.mockito.Matchers
      └org.mockito.Mockito
```

Direct Known Subclasses:
BDDMockito

```
public class Mockito
extends Matchers
```

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

1. Let's verify some behaviour!
2. How about some stubbing?
3. Argument matchers
4. Verifying exact number of invocations / at least once / never
5. Stubbing void methods with exceptions
6. Verification in order
7. Making sure interaction(s) never happened on mock
8. Finding redundant invocations
9. Shorthand for mocks creation - `@Mock` annotation
10. Stubbing consecutive calls (iterator-style stubbing)
11. Stubbing with callbacks
12. `doReturn()`|`doThrow()`|`doAnswer()`|`doNothing()`|`doCallRealMethod()` family of methods
13. Spying on real objects
14. Changing default return values of unstubbed invocations (Since 1.7)
15. Capturing arguments for further assertions (Since 1.8.0)
16. Real partial mocks (Since 1.8.0)
17. Resetting mocks (Since 1.8.0)
18. Troubleshooting & validating framework usage (Since 1.8.0)
19. Aliases for behavior driven development (Since 1.8.0)
20. Serializable mocks (Since 1.8.1)
21. New annotations: `@Captor`, `@Spy`, `@InjectMocks` (Since 1.8.3)
22. Verification with timeout (Since 1.8.5)

23. (New) Automatic instantiation of `@Spies`, `@InjectMocks` and constructor injection goodness (Since 74.125.31.82
24. (New) One-liner stubs (Since 1.9.0)
25. (New) Verification ignoring stubs (Since 1.9.0)
26. (**New**) Mocking details (Since 1.9.5)
27. (**New**) Delegate calls to real instance (Since 1.9.5)
28. (**New**) MockMaker API (Since 1.9.5)

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!

```
01 //Let's import Mockito statically so that the code looks clearer
02 import static org.mockito.Mockito.*;
03
04 //mock creation
05 List mockedList = mock(List.class);
06
07 //using mock object
08 mockedList.add("one");
09 mockedList.clear();
10
11 //verification
12 verify(mockedList).add("one");
13 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?

```
01 //You can mock concrete classes, not only interfaces
02 LinkedList mockedList = mock(LinkedList.class);
03
04 //stubbing
05 when(mockedList.get(0)).thenReturn("first");
06 when(mockedList.get(1)).thenThrow(new RuntimeException());
07
08 //following prints "first"
09 System.out.println(mockedList.get(0));
10
11 //following throws runtime exception
12 System.out.println(mockedList.get(1));
13
14 //following prints "null" because get(999) was not stubbed
15 System.out.println(mockedList.get(999));
16
17 //Although it is possible to verify a stubbed invocation, usually it's just
   redundant
18 //If your code cares what get(0) returns then something else breaks (often
   before even verify() gets executed).
19 //If your code doesn't care what get(0) returns then it should not be
   stubbed. Not convinced? See here.
20 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. Other words: **the order of stubbing matters** but it is only meaningful rarely, e.g. when stubbing exactly the same method calls or sometimes when argument matchers are used, etc.

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:

```
01 //stubbing using built-in anyInt() argument matcher
02 when(mockedList.get(anyInt())).thenReturn("element");
03
04 //stubbing using hamcrest (let's say isValid() returns your own hamcrest
   matcher):
05 when(mockedList.contains(argThat(isValid()))).thenReturn("element");
06
07 //following prints "element"
08 System.out.println(mockedList.get(999));
09
10 //you can also verify using an argument matcher
11 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):

```
1 verify(mock).someMethod(anyInt(), anyString(), eq("third argument"));
2 //above is correct - eq() is also an argument matcher
3
4 verify(mock).someMethod(anyInt(), anyString(), "third argument");
5 //above is incorrect - exception will be thrown because third argument is
  given without an argument matcher.
```

Matcher methods like `anyObject()`, `eq()` **do not** return matchers. Internally, they record a matcher on a stack and return a dummy value (usually null). This implementation is due static type safety imposed by java compiler. The consequence is that you cannot use `anyObject()`, `eq()` methods outside of verified/stubbed method.

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

```
01 //using mock
02 mockedList.add("once");
03
04 mockedList.add("twice");
05 mockedList.add("twice");
06
07 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

```
// A. Single mock whose methods must be invoked in a particular order
List singleMock = mock(List.class);

//using a single mock
singleMock.add("was added first");
singleMock.add("was added second");

//create an inOrder verifier for a single mock
InOrder inOrder = inOrder(singleMock);

//following will make sure that add is first called with "was added first, then with "was added second"
inOrder.verify(singleMock).add("was added first");
inOrder.verify(singleMock).add("was added second");

// B. Multiple mocks that must be used in a particular 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");


// Oh, and A + B can be mixed together at will
```

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. `doReturn()`|`doThrow()`|`doAnswer()`|`doNothing()`|`doCallRealMethod()` family of methods

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:
```

```
4 mockedList.clear();
```

You can use `doThrow()`, `doAnswer()`, `doNothing()`, `doReturn()` and `doCallRealMethod()` in place of the corresponding call with `when()`, for any method. It is necessary when you

- stub void methods
- stub methods on spy objects (see below)
- stub the same method more than once, to change the behaviour of a mock in the middle of a test.

but you may prefer to use these methods in place of the alternative with `when()`, for all of your stubbing calls.

Read more about these methods:

`doReturn(Object)`

`doThrow(Throwable)`

`doThrow(Class)`

`doAnswer(Answer)`

`doNothing()`

`doCallRealMethod()`

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)

```
01 List list = new LinkedList();
02 List spy = spy(list);
03
04 //optionally, you can stub out some methods:
05 when(spy.size()).thenReturn(100);
06
07 //using the spy calls *real* methods
08 spy.add("one");
09 spy.add("two");
10
11 //prints "one" - the first element of a list
12 System.out.println(spy.get(0));
13
14 //size() method was stubbed - 100 is printed
15 System.out.println(spy.size());
16
17 //optionally, you can verify
18 verify(spy).add("one");
19 verify(spy).add("two");
```

Important gotcha on spying real objects!

1. Sometimes it's impossible or impractical to use `when(Object)` for stubbing spies. Therefore when using spies please consider `doReturn|Answer|Throw()` family of methods for stubbing. 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. Mockito ***does not*** delegate calls to the passed real instance, instead it actually creates a copy of it. So if you keep the real instance and interact with it, don't expect the spied to be aware of those interaction and their effect on real instance state. The corollary is that when an ***unstubbed*** method is called ***on the spy*** but ***not on the real instance***, you won't see any effects on the real instance.
3. 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. Also you won't be able to verify those method as well.

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:

```
01 import static org.mockito.BDDMockito.*;
02
03 Seller seller = mock(Seller.class);
04 Shop shop = new Shop(seller);
05
06 public void shouldBuyBread() throws Exception {
07   //given
08   given(seller.askForBread()).willReturn(new Bread());
09
10   //when
11   Goods goods = shop.buyBread();
12
13   //then
14   assertThat(goods, containBread());
15 }
```

20. 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()`:

```
1 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.

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

21. 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 mock or spy fields into tested object automatically.

Note that @`InjectMocks` can only be used in combination with the @`Spy` annotation, it means that Mockito will inject mocks in a partial mock under testing. As a remainder, please read point 16 about partial mocks.

All new annotations are ***only*** processed on `MockitoAnnotations.initMocks(Object)`. Just like for `@Mock` annotation you can use the built-in runner: `MockitoJUnitRunner`.

22. Verification with timeout (Since 1.8.5)

Allows verifying with timeout. It causes a verify to wait for a specified period of time for a desired interaction rather than fails immediately if had not already happened. 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:

```
01 //passes when someMethod() is called within given time span
02 verify(mock, timeout(100)).someMethod();
03 //above is an alias to:
04 verify(mock, timeout(100).times(1)).someMethod();
05
06 //passes when someMethod() is called *exactly* 2 times within given time
   span
07 verify(mock, timeout(100).times(2)).someMethod();
08
09 //passes when someMethod() is called *at least* 2 times within given time
   span
10 verify(mock, timeout(100).atLeast(2)).someMethod();
11
12 //verifies someMethod() within given time span using given verification
   mode
13 //useful only if you have your own custom verification modes.
14 verify(mock, new Timeout(100, yourOwnVerificationMode)).someMethod();
```

23. (New) Automatic instantiation of `@Spies`, `@InjectMocks` and constructor injection goodness (Since 1.9.0)

Mockito will now try to instantiate `@Spy` and will instantiate `@InjectMocks` fields using **constructor** injection, **setter** injection, or **field** injection.

To take advantage of this feature you need to use `MockitoAnnotations.initMocks(Object)` or `MockitoJUnitRunner`.

Read more about available tricks and the rules of injection in the javadoc for `InjectMocks`

```
1 //instead:
2 @Spy BeerDrinker drinker = new BeerDrinker();
3 //you can write:
4 @Spy BeerDrinker drinker;
5
6 //same applies to @InjectMocks annotation:
7 @InjectMocks LocalPub;
```

24. (New) One-liner stubs (Since 1.9.0)

Mockito will now allow you to create mocks when stubbing. Basically, it allows to create a stub in one line of code. This can be helpful to keep test code clean. For example, some boring stub can be created & stubbed at field initialization in a test:

```
1 public class CarTest {
2   Car boringStubbedCar =
  when(mock(Car.class).shiftGear()).thenThrow(EngineNotStarted.class).getMock()
3
```

```
4 |  @Test public void should... {}
```

25. (New) Verification ignoring stubs (Since 1.9.0)

Mockito will now allow to ignore stubbing for the sake of verification. Sometimes useful when coupled with `verifyNoMoreInteractions()` or verification `inOrder()`. Helps avoiding redundant verification of stubbed calls - typically we're not interested in verifying stubs.

Warning, `ignoreStubs()` might lead to overuse of verifyNoMoreInteractions(ignoreStubs(...)); Bear in mind that Mockito does not recommend bombarding every test with `verifyNoMoreInteractions()` for the reasons outlined in javadoc for `verifyNoMoreInteractions(Object...)`

Some examples:

```
01 verify(mock).foo();
02 verify(mockTwo).bar();
03
04 //ignores all stubbed methods:
05 verifyNoMoreInvocations(ignoreStubs(mock, mockTwo));
06
07 //creates InOrder that will ignore stubbed
08 InOrder inOrder = inOrder(ignoreStubs(mock, mockTwo));
09 inOrder.verify(mock).foo();
10 inOrder.verify(mockTwo).bar();
11 inOrder.verifyNoMoreInteractions();
```

Advanced examples and more details can be found in javadoc for `ignoreStubs(Object...)`

26. (**New**) Mocking details (Since 1.9.5)

To identify whether a particular object is a mock or a spy:

```
1 Mockito.mockingDetails(someObject).isMock();
2 Mockito.mockingDetails(someObject).isSpy();
```

Both the `MockingDetails.isMock()` and `MockingDetails.isSpy()` methods return `boolean`. As a spy is just a different kind of mock, `isMock()` returns true if the object is a spy. In future Mockito versions MockingDetails may grow and provide other useful information about the mock, e.g. invocations, stubbing info, etc.

27. (**New**) Delegate calls to real instance (Since 1.9.5)

Useful for spies or partial mocks of objects **that are difficult to mock or spy** using the usual spy API. Possible use cases:

- Final classes but with an interface
- Already custom proxied object
- Special objects with a finalize method, i.e. to avoid executing it 2 times

The difference with the regular spy:

- The regular spy (`spy(Object)`) contains **all** state from the spied instance and the methods are invoked on the spy. The spied instance is only used at mock creation to copy the state from. If you call a method on a regular spy and it internally calls other methods on this spy, those calls are remembered for verifications, and they can be effectively stubbed.
- The mock that delegates simply delegates all methods to the delegate. The delegate is used all the time as methods are delegated onto it. If you call a method on a mock that delegates and it internally calls other methods on this mock, those calls are **not** remembered for verifications, stubbing does not have effect on them, too. Mock that delegates is less powerful than the regular spy but it is useful when the regular spy cannot be created.

See more information in docs for `AdditionalAnswers.delegatesTo(Object)`.

28. (**New**) `MockMaker` API (Since 1.9.5)

Driven by requirements and patches from Google Android guys Mockito now offers an extension point that allows replacing the proxy generation engine. By default, Mockito uses cglib to create dynamic proxies.

The extension point is for advanced users that want to extend Mockito. For example, it is now possible to use Mockito for Android testing with a help of dexmaker.

For more details, motivations and examples please refer to the docs for `MockMaker`.

Field Summary

static Answer<java.lang.Object>	**CALLS_REAL_METHODS** Optional `Answer` to be used with `mock(Class, Answer)`
static Answer<java.lang.Object>	**RETURNS_DEEP_STUBS** Optional `Answer` to be used with `mock(Class, Answer)`.
static Answer<java.lang.Object>	**RETURNS_DEFAULTS** The default `Answer` of every mock **if** the mock was not stubbed.
static Answer<java.lang.Object>	**RETURNS_MOCKS** Optional `Answer` to be used with `mock(Class, Answer)`
static Answer<java.lang.Object>	**RETURNS_SMART_NULLS** Optional `Answer` to be used with `mock(Class, Answer)`.

Constructor Summary

Mockito()

Method Summary

static VerificationMode	**atLeast**(int minNumberOfInvocations) Allows at-least-x verification.
static VerificationMode	**atLeastOnce**() Allows at-least-once verification.
static VerificationMode	**atMost**(int maxNumberOfInvocations) Allows at-most-x verification.
static VerificationMode	**calls**(int wantedNumberOfInvocations) Allows non-greedy verification in order.
static Stubber	**doAnswer**(Answer answer) Use `doAnswer()` when you want to stub a void method with generic `Answer`.
static Stubber	**doCallRealMethod**() Use `doCallRealMethod()` when you want to call the real implementation of a method.
static Stubber	**doNothing**()

	Use `doNothing()` for setting void methods to do nothing.
static Stubber	**doReturn**(java.lang.Object toBeReturned) Use `doReturn()` in those rare occasions when you cannot use `when(Object)`.
static Stubber	**doThrow**(java.lang.Class<? extends java.lang.Throwable> toBeThrown) Use `doThrow()` when you want to stub the void method to throw exception of specified class.
static Stubber	**doThrow**(java.lang.Throwable toBeThrown) Use `doThrow()` when you want to stub the void method with an exception.
static java.lang.Object[]	**ignoreStubs**(java.lang.Object... mocks) Ignores stubbed methods of given mocks for the sake of verification.
static InOrder	**inOrder**(java.lang.Object... mocks) Creates `InOrder` object that allows verifying mocks in order.
static <T> T	**mock**(java.lang.Class<T> classToMock) Creates mock object of given class or interface.
static <T> T	**mock**(java.lang.Class<T> classToMock, Answer defaultAnswer) Creates mock with a specified strategy for its answers to interactions.
static <T> T	**mock**(java.lang.Class<T> classToMock, MockSettings mockSettings) Creates a mock with some non-standard settings.
static <T> T	**mock**(java.lang.Class<T> classToMock, ReturnValues returnValues) **Deprecated.** ***Please use mock(Foo.class, defaultAnswer);***
static <T> T	**mock**(java.lang.Class<T> classToMock, java.lang.String name) Specifies mock name.
static MockingDetails	**mockingDetails**(java.lang.Object toInspect) Returns a MockingDetails instance that enables inspecting a particular object for Mockito related information.
static VerificationMode	**never**() Alias to `times(0)`, see `times(int)`
static VerificationMode	**only**() Allows checking if given method was the only one invoked.
static <T> void	**reset**(T... mocks) Smart Mockito users hardly use this feature because they know it could be a sign of poor tests.
static <T> T	**spy**(T object) Creates a spy of the real object.
static <T> DeprecatedOngoingStubbing<T>	**stub**(T methodCall) Stubs a method call with return value or an exception.
static <T> VoidMethodStubbable<T>	**stubVoid**(T mock) **Deprecated.** *Use `doThrow(Throwable)` method for stubbing voids*
static VerificationWithTimeout	**timeout**(int millis) Allows verifying with timeout.

static VerificationMode	**times**(int wantedNumberOfInvocations) Allows verifying exact number of invocations.
static void	**validateMockitoUsage**() First of all, in case of any trouble, I encourage you to read the Mockito FAQ: http://code.google.com/p/mockito/wiki/FAQ
static <T> T	**verify**(T mock) Verifies certain behavior **happened once**.
static <T> T	**verify**(T mock, VerificationMode mode) Verifies certain behavior happened at least once / exact number of times / never.
static void	**verifyNoMoreInteractions**(java.lang.Object... mocks) Checks if any of given mocks has any unverified interaction.
static void	**verifyZeroInteractions**(java.lang.Object... mocks) Verifies that no interactions happened on given mocks.
static <T> OngoingStubbing<T>	**when**(T methodCall) Enables stubbing methods.
static MockSettings	**withSettings**() Allows mock creation with additional mock settings.

Methods inherited from class org.mockito.Matchers

any, any, anyBoolean, anyByte, anyChar, anyCollection, anyCollectionOf, anyDouble, anyFloat, anyInt, anyList, anyListOf, anyLong, anyMap, anyMapOf, anyObject, anySet, anySetOf, anyShort, anyString, anyVararg, argThat, booleanThat, byteThat, charThat, contains, doubleThat, endsWith, eq, eq, eq, eq, eq, eq, eq, eq, eq, floatThat, intThat, isA, isNotNull, isNotNull, isNull, isNull, longThat, matches, notNull, notNull, refEq, same, shortThat, startsWith

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

RETURNS_DEFAULTS

```
public static final Answer<java.lang.Object> RETURNS_DEFAULTS
```

The default Answer of every mock **if** the mock was not stubbed. Typically it just returns some empty value.

Answer can be used to define the return values of unstubbed invocations.

This implementation first tries the global configuration. If there is no global configuration then it uses ReturnsEmptyValues (returns zeros, empty collections, nulls, etc.)

RETURNS_SMART_NULLS

```
public static final Answer<java.lang.Object> RETURNS_SMART_NULLS
```

Optional Answer to be used with mock(Class, Answer).

`Answer` can be used to define the return values of unstubbed invocations.

This implementation can be helpful when working with legacy code. Unstubbed methods often return null. If your code uses the object returned by an unstubbed call you get a NullPointerException. This implementation of Answer **returns SmartNull instead of null**. `SmartNull` gives nicer exception message than NPE because it points out the line where unstubbed method was called. You just click on the stack trace.

`ReturnsSmartNulls` first tries to return ordinary return values (see `ReturnsMoreEmptyValues`) then it tries to return SmartNull. If the return type is final then plain null is returned.

`ReturnsSmartNulls` will be probably the default return values strategy in Mockito 2.0.

Example:

```
Foo mock = (Foo.class, RETURNS_SMART_NULLS);

//calling unstubbed method here:
Stuff stuff = mock.getStuff();

//using object returned by unstubbed call:
stuff.doSomething();

//Above doesn't yield NullPointerException this time!
//Instead, SmartNullPointerException is thrown.
//Exception's cause links to unstubbed mock.getStuff() - just click on
the stack trace.
```

RETURNS_MOCKS

```
public static final Answer<java.lang.Object> RETURNS_MOCKS
```

Optional `Answer` to be used with `mock(Class, Answer)`

`Answer` can be used to define the return values of unstubbed invocations.

This implementation can be helpful when working with legacy code.

ReturnsMocks first tries to return ordinary return values (see `ReturnsMoreEmptyValues`) then it tries to return mocks. If the return type cannot be mocked (e.g. is final) then plain null is returned.

RETURNS_DEEP_STUBS

```
public static final Answer<java.lang.Object> RETURNS_DEEP_STUBS
```

Optional `Answer` to be used with `mock(Class, Answer)`.

Example that shows how deep stub works:

```
Foo mock = mock(Foo.class, RETURNS_DEEP_STUBS);

// note that we're stubbing a chain of methods here: getBar().getName()
when(mock.getBar().getName()).thenReturn("deep");


// note that we're chaining method calls: getBar().getName()
assertEquals("deep", mock.getBar().getName());
```

WARNING: This feature should rarely be required for regular clean code! Leave it for legacy code. Mocking a mock to return a mock, to return a mock, (...), to return something meaningful hints at violation of Law of Demeter or mocking a value object (a well known anti-pattern).

Good quote I've seen one day on the web: **every time a mock returns a mock a fairy dies**.

Please note that this answer will return existing mocks that matches the stub. This behavior is ok with deep stubs and allows verification to work on the last mock of the chain.

```
1  when(mock.getBar(anyString()).getThingy().getName()).thenReturn("deep");
2
3  mock.getBar("candy bar").getThingy().getName();
4
5  assertSame(mock.getBar(anyString()).getThingy().getName(),
   mock.getBar(anyString()).getThingy().getName());
6  verify(mock.getBar("candy bar").getThingy()).getName();
7  verify(mock.getBar(anyString()).getThingy()).getName();
```

Verification only works with the last mock in the chain. You can use verification modes.

```
01 when(person.getAddress(anyString()).getStreet().getName()).thenReturn("dee
02 when(person.getAddress(anyString()).getStreet(Locale.ITALIAN).getName()).t
03 when(person.getAddress(anyString()).getStreet(Locale.CHINESE).getName()).t
04
05 person.getAddress("the docks").getStreet().getName();
06 person.getAddress("the docks").getStreet().getLongName();
07 person.getAddress("the docks").getStreet(Locale.ITALIAN).getName();
08 person.getAddress("the docks").getStreet(Locale.CHINESE).getName();
09
10 // note that we are actually referring to the very last mock in the
   stubbing chain.
11 InOrder inOrder = inOrder(
12     person.getAddress("the docks").getStreet(),
13     person.getAddress("the docks").getStreet(Locale.CHINESE),
14     person.getAddress("the docks").getStreet(Locale.ITALIAN)
15 );
16 inOrder.verify(person.getAddress("the docks").getStreet(),
   times(1)).getName();
17 inOrder.verify(person.getAddress("the
   docks").getStreet()).getLongName();
18 inOrder.verify(person.getAddress("the
   docks").getStreet(Locale.ITALIAN), atLeast(1)).getName();
19 inOrder.verify(person.getAddress("the
   docks").getStreet(Locale.CHINESE)).getName();
```

How deep stub work internally?

```
1  //this:
2  Foo mock = mock(Foo.class, RETURNS_DEEP_STUBS);
3  when(mock.getBar().getName(), "deep");
4
5  //is equivalent of
6  Foo foo = mock(Foo.class);
7  Bar bar = mock(Bar.class);
8  when(foo.getBar()).thenReturn(bar);
9  when(bar.getName()).thenReturn("deep");
```

This feature will not work when any return type of methods included in the chain cannot be mocked (for example: is a primitive or a final class). This is because of java type system.

CALLS_REAL_METHODS

```
public static final Answer<java.lang.Object> CALLS_REAL_METHODS
```

Optional `Answer` to be used with `mock(Class, Answer)`

`Answer` can be used to define the return values of unstubbed invocations.

This implementation can be helpful when working with legacy code. When this implementation is used, unstubbed methods will delegate to the real implementation. This is a way to create a partial mock object that calls real methods by default.

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.

Example:

```
Foo mock = mock(Foo.class, CALLS_REAL_METHODS);

// this calls the real implementation of Foo.getSomething()
value = mock.getSomething();

when(mock.getSomething()).thenReturn(fakeValue);

// now fakeValue is returned
value = mock.getSomething();
```

Constructor Detail

Mockito

```
public Mockito()
```

Method Detail

mock

```
public static <T> T mock(java.lang.Class<T> classToMock)
```

Creates mock object of given class or interface.

See examples in javadoc for `Mockito` class

Parameters:
`classToMock` - class or interface to mock
Returns:
mock object

---

mock

```
public static <T> T mock(java.lang.Class<T> classToMock,
                         java.lang.String name)
```

Specifies mock name. Naming mocks can be helpful for debugging - the name is used in all verification errors.

Beware that naming mocks is not a solution for complex code which uses too many mocks or collaborators. **If you have too many mocks then refactor the code** so that it's easy to test/debug without necessity of naming mocks.

If you use `@Mock` annotation then you've got naming mocks for free! `@Mock` uses field name as mock

name. Read more.

See examples in javadoc for Mockito class

Parameters:
classToMock - class or interface to mock
name - of the mock
Returns:
mock object

---

mockingDetails

```
@Incubating
public static MockingDetails mockingDetails(java.lang.Object toInspect)
```

Returns a MockingDetails instance that enables inspecting a particular object for Mockito related information. Can be used to find out if given object is a Mockito mock or to find out if a given mock is a spy or mock.

In future Mockito versions MockingDetails may grow and provide other useful information about the mock, e.g. invocations, stubbing info, etc.

Parameters:
toInspect - - object to inspect
Returns:
A MockingDetails instance.
Since:
1.9.5

---

mock

```
@Deprecated
public static <T> T mock(java.lang.Class<T> classToMock,
                         ReturnValues returnValues)
```

Deprecated. *Please use mock(Foo.class, defaultAnswer);*

Deprecated : Please use mock(Foo.class, defaultAnswer);

See mock(Class, Answer)

Why it is deprecated? ReturnValues is being replaced by Answer for better consistency & interoperability of the framework. Answer interface has been in Mockito for a while and it has the same responsibility as ReturnValues. There's no point in mainting exactly the same interfaces.

Creates mock with a 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 when working with legacy systems.

Obviously return values are used only when you don't stub the method call.

```
1 Foo mock = mock(Foo.class, Mockito.RETURNS_SMART_NULLS);
2 Foo mockTwo = mock(Foo.class, new YourOwnReturnValues());
```

See examples in javadoc for Mockito class

Parameters:
classToMock - class or interface to mock
returnValues - default return values for unstubbed methods
Returns:

mock object

---

mock

```
public static <T> T mock(java.lang.Class<T> classToMock,
                         Answer defaultAnswer)
```

Creates mock with a specified strategy for its answers to interactions. It's quite advanced feature and typically you don't need it to write decent tests. However it can be helpful when working with legacy systems.

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

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

See examples in javadoc for `Mockito` class

Parameters:

`classToMock` - class or interface to mock

`defaultAnswer` - default answer for unstubbed methods

Returns:

mock object

---

mock

```
public static <T> T mock(java.lang.Class<T> classToMock,
                         MockSettings mockSettings)
```

Creates a mock with some non-standard settings.

The number of configuration points for a mock grows so we need a fluent way to introduce new configuration without adding more and more overloaded Mockito.mock() methods. Hence `MockSettings`.

```
1 Listener mock = mock(Listener.class, withSettings()
2   .name("firstListner").defaultBehavior(RETURNS_SMART_NULLS));
3 );
```

Use it carefully and occasionally. What might be reason your test needs non-standard mocks? Is the code under test so complicated that it requires non-standard mocks? Wouldn't you prefer to refactor the code under test so it is testable in a simple way?

See also `withSettings()`

See examples in javadoc for `Mockito` class

Parameters:

`classToMock` - class or interface to mock

`mockSettings` - additional mock settings

Returns:

mock object

---

spy

```
public static <T> T spy(T object)
```

Creates a spy of the real object. The spy calls **real** methods unless they are stubbed.

Real spies should be used **carefully and occasionally**, for example when dealing with legacy code.

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.

Example:

```
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 or impractical to use `when(Object)` for stubbing spies. Therefore for spies it is recommended to always use `doReturn|Answer|Throw()|CallRealMethod` family of methods for stubbing. 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. Mockito ***does not*** delegate calls to the passed real instance, instead it actually creates a copy of it. So if you keep the real instance and interact with it, don't expect the spied to be aware of those interaction and their effect on real instance state. The corollary is that when an ***unstubbed*** method is called ***on the spy*** but ***not on the real instance***, you won't see any effects on the real instance.
3. 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. Also you won't be able to verify those method as well.

See examples in javadoc for `Mockito` class

Parameters:

`object` - to spy on

Returns:

a spy of the real object

---

stub

```
public static <T> DeprecatedOngoingStubbing<T> stub(T methodCall)
```

Stubs a method call with return value or an exception. E.g:

```
stub(mock.someMethod()).toReturn(10);

//you can use flexible argument matchers, e.g:
stub(mock.someMethod(anyString())).toReturn(10);

//setting exception to be thrown:
stub(mock.someMethod("some arg")).toThrow(new RuntimeException());

//you can stub with different behavior for consecutive method calls.
//Last stubbing (e.g: toReturn("foo")) determines the behavior for
further consecutive calls.
stub(mock.someMethod("some arg"))
 .toThrow(new RuntimeException())
 .toReturn("foo");
```

Some users find stub() confusing therefore `when(Object)` is recommended over stub()

```
//Instead of:
stub(mock.count()).toReturn(10);

//You can do:
when(mock.count()).thenReturn(10);
```

For stubbing void methods with throwables see: `doThrow(Throwable)`

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.

Although it is possible to verify a stubbed invocation, usually **it's just redundant**. Let's say you've stubbed foo.bar(). If your code cares what foo.bar() 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.

Parameters:

`methodCall` - method call

Returns:

DeprecatedOngoingStubbing object to set stubbed value/exception

---

when

```
public static <T> OngoingStubbing<T> when(T methodCall)
```

Enables stubbing methods. Use it when you want the mock to return particular value when particular method is called.

Simply put: "**When** the x method is called **then** return y".

when() is a successor of deprecated `stub(Object)`

Examples:

```
when(mock.someMethod()).thenReturn(10);

//you can use flexible argument matchers, e.g:
when(mock.someMethod(anyString())).thenReturn(10);

//setting exception to be thrown:
when(mock.someMethod("some arg")).thenThrow(new RuntimeException());

//you can set different behavior for consecutive method calls.
//Last stubbing (e.g: thenReturn("foo")) determines the behavior of
further consecutive calls.
when(mock.someMethod("some arg"))
 .thenThrow(new RuntimeException())
 .thenReturn("foo");

//Alternative, shorter version for consecutive stubbing:
when(mock.someMethod("some arg"))
 .thenReturn("one", "two");
//is the same as:
when(mock.someMethod("some arg"))
 .thenReturn("one")
 .thenReturn("two");

//shorter version for consecutive method calls throwing exceptions:
when(mock.someMethod("some arg"))
 .thenThrow(new RuntimeException(), new NullPointerException();

```

For stubbing void methods with throwables see: `doThrow(Throwable)`

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.

Although it is possible to verify a stubbed invocation, usually **it's just redundant**. Let's say you've stubbed `foo.bar()`. If your code cares what `foo.bar()` 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.

See examples in javadoc for `Mockito` class

Parameters:

`methodCall` - method to be stubbed

Returns:

OngoingStubbing object used to stub fluently. **Do not** create a reference to this returned object.

---

verify

`public static <T> T verify(T mock)`

Verifies certain behavior **happened once**.

Alias to `verify(mock, times(1))` E.g:

```
verify(mock).someMethod("some arg");
```

Above is equivalent to:

```
verify(mock, times(1)).someMethod("some arg");
```

Arguments passed are compared using `equals()` method. Read about `ArgumentCaptor` or `ArgumentMatcher` to find out other ways of matching / asserting arguments passed.

Although it is possible to verify a stubbed invocation, usually **it's just redundant**. Let's say you've stubbed `foo.bar()`. If your code cares what `foo.bar()` 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.

See examples in javadoc for `Mockito` class

Parameters:
`mock` - to be verified
Returns:
mock object itself

---

verify

```
public static <T> T verify(T mock,
                           VerificationMode mode)
```

Verifies certain behavior happened at least once / exact number of times / never. E.g:

```
verify(mock, times(5)).someMethod("was called five times");

verify(mock, atLeast(2)).someMethod("was called at least two times");

//you can use flexible argument matchers, e.g:
verify(mock, atLeastOnce()).someMethod(anyString());
```

times(1) is the default and can be omitted

Arguments passed are compared using `equals()` method. Read about `ArgumentCaptor` or `ArgumentMatcher` to find out other ways of matching / asserting arguments passed.

Parameters:
`mock` - to be verified
`mode` - times(x), atLeastOnce() or never()
Returns:
mock object itself

---

reset

```
public static <T> void reset(T... mocks)
```

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
```

Type Parameters:

`T` - The Type of the mocks

Parameters:

`mocks` - to be reset

---

verifyNoMoreInteractions

```
public static void verifyNoMoreInteractions(java.lang.Object... mocks)
```

Checks if any of given mocks has any unverified interaction.

You can use this method after you verified your mocks - to make sure that nothing else was invoked on your mocks.

See also `never()` - it is more explicit and communicates the intent well.

Stubbed invocations (if called) are also treated as interactions.

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.

This method will also detect unverified invocations that occurred before the test method, for example: in `setUp()`, `@Before` method or in constructor. Consider writing nice code that makes interactions only in test methods.

Example:

```
//interactions
mock.doSomething();
mock.doSomethingUnexpected();

//verification
verify(mock).doSomething();

//following will fail because 'doSomethingUnexpected()' is unexpected
verifyNoMoreInteractions(mock);
```

See examples in javadoc for `Mockito` class

Parameters:

`mocks` - to be verified

---

verifyZeroInteractions

```
public static void verifyZeroInteractions(java.lang.Object... mocks)
```

Verifies that no interactions happened on given mocks.

```
verifyZeroInteractions(mockOne, mockTwo);
```

This method will also detect invocations that occurred before the test method, for example: in `setUp()`, `@Before` method or in constructor. Consider writing nice code that makes interactions only in test methods.

See also `never()` - it is more explicit and communicates the intent well.

See examples in javadoc for `Mockito` class

Parameters:
`mocks` - to be verified

---

stubVoid

`public static <T> VoidMethodStubbable<T> stubVoid(T mock)`

Deprecated. *Use `doThrow(Throwable)` method for stubbing voids*

```
//Instead of:
stubVoid(mock).toThrow(e).on().someVoidMethod();

//Please do:
doThrow(e).when(mock).someVoidMethod();
```

doThrow() replaces stubVoid() because of improved readability and consistency with the family of doAnswer() methods.

Originally, `stubVoid()` was used for stubbing void methods with exceptions. E.g:

```
stubVoid(mock).toThrow(new RuntimeException()).on().someMethod();

//you can stub with different behavior for consecutive calls.
//Last stubbing (e.g. toReturn()) determines the behavior for further consecutive calls.
stubVoid(mock)
  .toThrow(new RuntimeException())
  .toReturn()
  .on().someMethod();
```

See examples in javadoc for `Mockito` class

Parameters:
`mock` - to stub
Returns:
stubbable object that allows stubbing with throwable

---

doThrow

`public static Stubber doThrow(java.lang.Throwable toBeThrown)`

Use `doThrow()` when you want to stub the void method with an exception.

Stubbing voids requires different approach from `when(Object)` because the compiler does not like void methods inside brackets...

Example:

```
doThrow(new RuntimeException()).when(mock).someVoidMethod();
```

Parameters:
`toBeThrown` - to be thrown when the stubbed method is called
Returns:
stubber - to select a method for stubbing

---

doThrow

```
public static Stubber doThrow(java.lang.Class<? extends java.lang.Throwable> toBeThrown)
```

Use `doThrow()` when you want to stub the void method to throw exception of specified class.

A new exception instance will be created for each method invocation.

Stubbing voids requires different approach from `when(Object)` because the compiler does not like void methods inside brackets...

Example:

```
1 doThrow(RuntimeException.class).when(mock).someVoidMethod();
```

Parameters:
`toBeThrown` - to be thrown when the stubbed method is called
Returns:
stubber - to select a method for stubbing
Since:
1.9.0

---

doCallRealMethod

```
public static Stubber doCallRealMethod()
```

Use `doCallRealMethod()` when you want to call the real implementation of a method.

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.

See also javadoc `spy(Object)` to find out more about partial mocks. **Mockito.spy() is a recommended way of creating partial mocks.** The reason is it guarantees real methods are called against correctly constructed object because you're responsible for constructing the object passed to spy() method.

Example:

```
1 Foo mock = mock(Foo.class);
2 doCallRealMethod().when(mock).someVoidMethod();
3
4 // this will call the real implementation of Foo.someVoidMethod()
5 mock.someVoidMethod();
```

See examples in javadoc for `Mockito` class

Returns:
stubber - to select a method for stubbing

Since:
1.9.5

---

doAnswer

```
public static Stubber doAnswer(Answer answer)
```

Use `doAnswer()` when you want to stub a void method with generic `Answer`.

Stubbing voids requires different approach from `when(Object)` because the compiler does not like void methods inside brackets...

Example:

```
doAnswer(new Answer() {
    public Object answer(InvocationOnMock invocation) {
        Object[] args = invocation.getArguments();
        Mock mock = invocation.getMock();
        return null;
    }})
.when(mock).someMethod();
```

See examples in javadoc for `Mockito` class

Parameters:
`answer` - to answer when the stubbed method is called
Returns:
stubber - to select a method for stubbing

---

doNothing

```
public static Stubber doNothing()
```

Use `doNothing()` for setting void methods to do nothing. **Beware that void methods on mocks do nothing by default!** However, there are rare situations when doNothing() comes handy:

1. Stubbing consecutive calls on a void method:

```
doNothing().
doThrow(new RuntimeException())
.when(mock).someVoidMethod();

//does nothing the first time:
mock.someVoidMethod();

//throws RuntimeException the next time:
mock.someVoidMethod();
```

2. When you spy real objects and you want the void method to do nothing:

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

//let's make clear() do nothing
doNothing().when(spy).clear();

spy.add("one");

//clear() does nothing, so the list still contains "one"
spy.clear();
```

See examples in javadoc for `Mockito` class

Returns:
stubber - to select a method for stubbing

---

doReturn

```
public static Stubber doReturn(java.lang.Object toBeReturned)
```

Use `doReturn()` in those rare occasions when you cannot use `when(Object)`.

Beware that `when(Object)` is always recommended for stubbing because it is argument type-safe and more readable (especially when stubbing consecutive calls).

Here are those rare occasions when doReturn() comes handy:

1. When spying real objects and calling real methods on a spy brings side effects

```
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. Overriding a previous exception-stubbing:

```
when(mock.foo()).thenThrow(new RuntimeException());

//Impossible: the exception-stubbed foo() method is called so
RuntimeException is thrown.
when(mock.foo()).thenReturn("bar");

//You have to use doReturn() for stubbing:
doReturn("bar").when(mock).foo();
```

Above scenarios shows a tradeoff of Mockito's elegant syntax. Note that the scenarios are very rare, though. Spying should be sporadic and overriding exception-stubbing is very rare. Not to mention that in general overridding stubbing is a potential code smell that points out too much stubbing.

See examples in javadoc for `Mockito` class

Parameters:
`toBeReturned` - to be returned when the stubbed method is called
Returns:
stubber - to select a method for stubbing

---

inOrder

```
public static InOrder inOrder(java.lang.Object... mocks)
```

Creates `InOrder` object that allows verifying mocks in order.

```
InOrder inOrder = inOrder(firstMock, 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.

`InOrder` verification is 'greedy'. You will hardly every notice it but if you want to find out more search for 'greedy' on the Mockito wiki pages.

As of Mockito 1.8.4 you can verifyNoMoreInvocations() in order-sensitive way. Read more: `InOrder.verifyNoMoreInteractions()`

See examples in javadoc for `Mockito` class

Parameters:
`mocks` - to be verified in order
Returns:
InOrder object to be used to verify in order

---

ignoreStubs

`public static java.lang.Object[] ignoreStubs(java.lang.Object... mocks)`

Ignores stubbed methods of given mocks for the sake of verification. Sometimes useful when coupled with `verifyNoMoreInteractions()` or verification `inOrder()`. Helps avoiding redundant verification of stubbed calls - typically we're not interested in verifying stubs.

Warning, `ignoreStubs()` might lead to overuse of `verifyNoMoreInteractions(ignoreStubs(...));` Bear in mind that Mockito does not recommend bombarding every test with `verifyNoMoreInteractions()` for the reasons outlined in javadoc for `verifyNoMoreInteractions(Object...)` Other words: all ***stubbed*** methods of given mocks are marked ***verified*** so that they don't get in a way during verifyNoMoreInteractions().

This method **changes the input mocks**! This method returns input mocks just for convenience.

Ignored stubs will also be ignored for verification inOrder, including `InOrder.verifyNoMoreInteractions()`. See the second example.

Example:

```
01 //mocking lists for the sake of the example (if you mock List in real
   you will burn in hell)
02 List mock1 = mock(List.class), mock2 = mock(List.class);
03
04 //stubbing mocks:
05 when(mock1.get(0)).thenReturn(10);
06 when(mock2.get(0)).thenReturn(20);
07
08 //using mocks by calling stubbed get(0) methods:
09 System.out.println(mock1.get(0)); //prints 10
10 System.out.println(mock2.get(0)); //prints 20
11
12 //using mocks by calling clear() methods:
13 mock1.clear();
14 mock2.clear();
15
16 //verification:
17 verify(mock1).clear();
18 verify(mock2).clear();
19
20 //verifyNoMoreInteractions() fails because get() methods were not
   accounted for.
```

```
try { verifyNoMoreInteractions(mock1, mock2); } catch
(NoInteractionsWanted e);

//However, if we ignore stubbed methods then we can
verifyNoMoreInteractions()
verifyNoMoreInteractions(ignoreStubs(mock1, mock2));

//Remember that ignoreStubs() *changes* the input mocks and returns
them for convenience.
```

Ignoring stubs can be used with **verification in order**:

```
List list = mock(List.class);
when(mock.get(0)).thenReturn("foo");

list.add(0);
System.out.println(list.get(0)); //we don't want to verify this
list.clear();

InOrder inOrder = inOrder(ignoreStubs(list));
inOrder.verify(list).add(0);
inOrder.verify(list).clear();
inOrder.verifyNoMoreInteractions();
```

Parameters:

`mocks` - input mocks that will be changed

Returns:

the same mocks that were passed in as parameters

Since:

1.9.0

---

times

```
public static VerificationMode times(int wantedNumberOfInvocations)
```

Allows verifying exact number of invocations. E.g:

```
verify(mock, times(2)).someMethod("some arg");
```

See examples in javadoc for `Mockito` class

Parameters:

`wantedNumberOfInvocations` - wanted number of invocations

Returns:

verification mode

---

never

```
public static VerificationMode never()
```

Alias to `times(0)`, see `times(int)`

Verifies that interaction did not happen. E.g:

```
verify(mock, never()).someMethod();
```

If you want to verify there were NO interactions with the mock check out `verifyZeroInteractions(Object...)` or `verifyNoMoreInteractions(Object...)`

See examples in javadoc for `Mockito` class

Returns:
verification mode

---

atLeastOnce

```
public static VerificationMode atLeastOnce()
```

Allows at-least-once verification. E.g:

```
verify(mock, atLeastOnce()).someMethod("some arg");
```

Alias to `atLeast(1)`.

See examples in javadoc for `Mockito` class

Returns:
verification mode

---

atLeast

```
public static VerificationMode atLeast(int minNumberOfInvocations)
```

Allows at-least-x verification. E.g:

```
verify(mock, atLeast(3)).someMethod("some arg");
```

See examples in javadoc for `Mockito` class

Parameters:
`minNumberOfInvocations` - minimum number of invocations
Returns:
verification mode

---

atMost

```
public static VerificationMode atMost(int maxNumberOfInvocations)
```

Allows at-most-x verification. E.g:

```
verify(mock, atMost(3)).someMethod("some arg");
```

See examples in javadoc for `Mockito` class

Parameters:
`maxNumberOfInvocations` - max number of invocations
Returns:
verification mode

---

calls

```
public static VerificationMode calls(int wantedNumberOfInvocations)
```

Allows non-greedy verification in order. For example

```
inOrder.verify( mock, calls( 2 )).someMethod( "some arg" );
```

- will not fail if the method is called 3 times, unlike times(2)

- will not mark the third invocation as verified, unlike atLeast(2)

This verification mode can only be used with in order verification.

Parameters:
`wantedNumberOfInvocations` - number of invocations to verify
Returns:
verification mode

---

only

```
public static VerificationMode only()
```

Allows checking if given method was the only one invoked. E.g:

```
verify(mock, only()).someMethod();
//above is a shorthand for following 2 lines of code:
verify(mock).someMethod();
verifyNoMoreInvocations(mock);
```

See also `verifyNoMoreInteractions(Object...)`

See examples in javadoc for `Mockito` class

Returns:
verification mode

---

timeout

```
public static VerificationWithTimeout timeout(int millis)
```

Allows verifying with timeout. It causes a verify to wait for a specified period of time for a desired interaction rather than fails immediately if had not already happened. 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.

```
//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 least* 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();
```

See examples in javadoc for `Mockito` class

Parameters:
`millis` - - time span in millisecond

Returns:
verification mode

---

validateMockitoUsage

`public static void validateMockitoUsage()`

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

`validateMockitoUsage()` **explicitly validates** the framework state to detect invalid use of Mockito. However, this feature is optional **because Mockito validates the usage all the time...** but there is a gotcha so read on.

Examples of incorrect use:

```
1 //Oups, someone forgot thenReturn() part:
2 when(mock.get());
3
4 //Oups, someone put the verified method call inside verify() where it
  should be outside:
5 verify(mock.execute());
6
7 //Oups, someone has used EasyMock for too long and forgot to specify
  the method to verify:
8 verify(mock);
```

Mockito throws exceptions if you misuse it so that you know if your tests are written correctly. The gotcha is that Mockito does the validation **next time** you use the framework (e.g. next time you verify, stub, call mock etc.). But even though the exception might be thrown in the next test, the exception **message contains a navigable stack trace element** with location of the defect. Hence you can click and find the place where Mockito was misused.

Sometimes though, you might want to validate the framework usage explicitly. For example, one of the users wanted to put `validateMockitoUsage()` in his `@After` method so that he knows immediately when he misused Mockito. Without it, he would have known about it not sooner than **next time** he used the framework. One more benefit of having `validateMockitoUsage()` in `@After` is that jUnit runner will always fail in the test method with defect whereas ordinary 'next-time' validation might fail the **next** test method. But even though JUnit might report next test as red, don't worry about it and just click at navigable stack trace element in the exception message to instantly locate the place where you misused mockito.

Built-in runner: `MockitoJUnitRunner` does validateMockitoUsage() after each test method.

Bear in mind that **usually you don't have to** `validateMockitoUsage()` and framework validation triggered on next-time basis should be just enough, mainly because of enhanced exception message with clickable location of defect. However, I would recommend validateMockitoUsage() if you already have sufficient test infrastructure (like your own runner or base class for all tests) because adding a special action to `@After` has zero cost.

See examples in javadoc for `Mockito` class

---

withSettings

`public static MockSettings withSettings()`

Allows mock creation with additional mock settings.

Don't use it too often. Consider writing simple tests that use simple mocks. Repeat after me: simple tests push simple, KISSy, readable & maintainable code. If you cannot write a test in a simple way - refactor the code under test.

Examples of mock settings:

```
01 //Creates mock with different default answer & name
02 Foo mock = mock(Foo.class, withSettings()
03     .defaultAnswer(RETURNS_SMART_NULLS)
04     .name("cool mockie"));
05
06 //Creates mock with different default answer, descriptive name and
   extra interfaces
07 Foo mock = mock(Foo.class, withSettings()
08     .defaultAnswer(RETURNS_SMART_NULLS)
09     .name("cool mockie")
10     .extraInterfaces(Bar.class));
```

`MockSettings` has been introduced for two reasons. Firstly, to make it easy to add another mock settings when the demand comes. Secondly, to enable combining different mock settings without introducing zillions of overloaded mock() methods.

See javadoc for `MockSettings` to learn about possible mock settings.

Returns:

mock settings instance with defaults.
