The Mockito library enables mock 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. It allows access to documentation straight from the IDE even if you work offline. It motivates Mockito developers to keep documentation up-to-date with the code that they write, every day, with every commit.

Contents

0. Migrating to 2.0

In order to continue improving Mockito and further improve the unit testing experience, we want you to upgrade to 2.0. Mockito follows semantic versioning and contains breaking changes only on major version upgrades. In the lifecycle of a library, breaking changes are necessary to roll out a set of brand new features that alter the existing behavior or even change the API. We hope that you enjoy Mockito 2.0!

List of breaking changes:

- Mockito is decoupled from Hamcrest and custom matchers API has changed, see `ArgumentMatcher` for rationale and migration guide
.
- Stubbing API has been tweaked to avoid unavoidable compilation warnings that appeared on JDK7+ platform. This will only affect binary compatibility, compilation compatibility remains unaffected.

The following examples mock a List, because most people are familiar with the interface (such as the `add()`, `get()`, `clear()` methods).
In reality, please don't mock the List class. Use a real instance instead.

1. Let's verify some behaviour! 验证一些行为!

```
//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, a mock will remember all interactions. Then you can selectively verify whatever interactions you are interested in.

2. How about some stubbing? 行为桩定义

```
//You can mock concrete classes, not just 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 even before 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 a value, a mock will return either null, a a primitive/primitive wrapper value, or an empty collection, as appropriate. For example 0 for an int/Integer and false for a 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 a 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 参数匹配器

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 custom matcher (let's say isValid() returns your own matcher implementation):
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 can also be written as Java 8 Lambdas
verify(mockedList).add(someString -> someString.length() > 5);
```

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.

The following 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.
```

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 to static type safety imposed by the 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

```
//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.

6. Verification in order 校验顺序

```
// 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 an InOrder object passing only the 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 模拟创造的简写形式（@Mock 注解）

- 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.

最小化重复的 mock 创建代码
使测试类更具可读性
使验证错误更容易阅读

```
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` or a rule: `MockitoRule`.

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 simply stubbing with `thenReturn()` or `thenThrow()`, which should be enough to test/test-drive any clean & simple code. However, if you do have a need to stub with the generic Answer interface, here is an example:

```
when(mock.someMethod(anyString())).thenAnswer(new Answer() {
```

```
    Object answer(InvocationOnMock invocation) {
        Object[] args = invocation.getArguments();
        Object mock = invocation.getMock();
        return "called with arguments: " + args;
    }
});

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

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

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

Use doThrow() when you want to stub a void method with an exception:

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

//following throws RuntimeException:
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 监视真实的对象

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.

部分 mock

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!

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);
```

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.
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 an 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. 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 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 also be used in combination with the @`Spy` annotation, it means that Mockito will inject mocks into the partial mock under test. This complexity is another good reason why you should only use partial mocks as a last resort. See 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` or rule: `MockitoRule`.

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.

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 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();
```

23. 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)`, `MockitoJUnitRunner` or `MockitoRule`.

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

```
//instead:
@Spy BeerDrinker drinker = new BeerDrinker();
//you can write:
@Spy BeerDrinker drinker;

//same applies to @InjectMocks annotation:
@InjectMocks LocalPub;
```

24. 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:

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

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

25. 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:

```
verify(mock).foo();
verify(mockTwo).bar();

//ignores all stubbed methods:
verifyNoMoreInteractions(ignoreStubs(mock, mockTwo));

//creates InOrder that will ignore stubbed
InOrder inOrder = inOrder(ignoreStubs(mock, mockTwo));
inOrder.verify(mock).foo();
inOrder.verify(mockTwo).bar();
inOrder.verifyNoMoreInteractions();
```

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

26. Mocking details (Since 1.9.5)

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

```
Mockito.mockingDetails(someObject).isMock();
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. 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. Since Mockito 1.10.11, the delegate may or may not be of the same type as the mock. If the type is different, a matching method needs to be found on delegate type otherwise an exception is thrown. Possible use cases for this feature:

- 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. `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`.

29. (new) BDD style verification (Since 1.10.0)

Enables Behavior Driven Development (BDD) style verification by starting verification with the BDD **then** keyword.

```
given(dog.bark()).willReturn(2);

// when
...

then(person).should(times(2)).ride(bike);
```

For more information and an example see `BDDMockito.then(Object)`}

30. (new) Spying or mocking abstract classes (Since 1.10.12)

It is now possible to conveniently spy on abstract classes. Note that overusing spies hints at code design smells (see `spy(Object)`).

Previously, spying was only possible on instances of objects. New API makes it possible to use constructor when creating an instance of the mock. This is particularly useful for mocking abstract classes because the user is no longer required to provide an instance of the abstract class. At the moment, only parameter-less constructor is supported, let us know if it is not enough.

```
//convenience API, new overloaded spy() method:
SomeAbstract spy = spy(SomeAbstract.class);

//Robust API, via settings builder:
OtherAbstract spy = mock(OtherAbstract.class, withSettings()
   .useConstructor().defaultAnswer(CALLS_REAL_METHODS));

//Mocking a non-static inner abstract class:
InnerAbstract spy = mock(InnerAbstract.class, withSettings()
   .useConstructor().outerInstance(outerInstance).defaultAnswer(CALLS_REAL_METHODS));
```

For more information please see `MockSettings.useConstructor()`.

31. (new) Mockito mocks can be *serialized / deserialized* across classloaders (Since 1.10.0)

Mockito introduces serialization across classloader. Like with any other form of serialization, all types in the mock hierarchy have to serializable, inclusing answers. As this serialization mode require considerably more work, this is an opt-in setting.

```
// use regular serialization
mock(Book.class, withSettings().serializable());

// use serialization across classloaders
mock(Book.class, withSettings().serializable(ACROSS_CLASSLOADERS));
```

For more details see `MockSettings.serializable(SerializableMode)`.

32. (new) Better generic support with deep stubs (Since 1.10.0)

Deep stubbing has been improved to find generic information if available in the class. That means that classes like this can be used without having to mock the behavior.

```
class Lines extends List<Line> {
    // ...
}

lines = mock(Lines.class, RETURNS_DEEP_STUBS);

// Now Mockito understand this is not an Object but a Line
Line line = lines.iterator().next();
```

Please note that in most scenarios a mock returning a mock is wrong.

33. (new) Mockito JUnit rule (Since 1.10.17)

Mockito now offers a JUnit rule. Until now in JUnit there was two wasy to initialize fields annotated by Mockito annotations such as `@Mock`, `@Spy`, `@InjectMocks`, etc.

- Annotating the JUnit test class with a `@RunWith(MockitoJUnitRunner.class)`
- Invoking `MockitoAnnotations.initMocks(Object)` in the `@Before` method

Now you can choose to use a rule :

```
@RunWith(YetAnotherRunner.class)
public class TheTest {
    @Rule public MockitoRule mockito = MockitoJUnit.rule();
    // ...
}
```

For more information see `MockitoJUnit.rule()`.

34. (new) Switch *on* or *off* plugins (Since 1.10.15)

An incubating feature made it's way in mockito that will allow to toggle a mockito-plugin. More information here `PluginSwitch`.

35. Custom verification failure message (Since 2.0.0)

Allows specifying a custom message to be printed if verification fails.

Examples:

```
// will print a custom message on verification failure
verify(mock, description("This will print on failure")).someMethod();
```

```
// will work with any verification mode
verify(mock, times(2).description("someMethod should be called twice")).someMethod();
```

36. Java 8 Lambda Matcher Support (Since 2.0.0)

You can use Java 8 lambda expressions with `ArgumentMatcher` to reduce the dependency on `ArgumentCaptor`. If you need to verify that the input to a function call on a mock was correct, then you would normally use the `ArgumentCaptor` to find the operands used and then do subsequent assertions on them. While for complex examples this can be useful, it's also long-winded.

Writing a lambda to express the match is quite easy. The argument to your function, when used in conjunction with argThat, will be passed to the ArgumentMatcher as a strongly typed object, so it is possible to do anything with it.

Examples:

```
// verify a list only had strings of a certain length added to it
// note - this will only compile under Java 8
verify(list, times(2)).add(argThat(string -> string.length() < 5));

// Java 7 equivalent - not as neat
verify(list, times(2)).add(argThat(new ArgumentMatcher(){
    public boolean matches(String arg) {
        return arg.length() < 5;
    }
}));

// more complex Java 8 example - where you can specify complex verification behaviour functionally
verify(target, times(1)).receiveComplexObject(argThat(obj -> obj.getSubObject().get(0).equals("expected")));

// this can also be used when defining the behaviour of a mock under different inputs
// in this case if the input list was fewer than 3 items the mock returns null
when(mock.someMethod(argThat(list -> list.size()<3))).willReturn(null);
```

37. Java 8 Custom Answer Support (Since 2.0.0)

As the `Answer` interface has just one method it is already possible to implement it in Java 8 using a lambda expression for very simple situations. The more you need to use the parameters of the method call, the more you need to typecast the arguments from `InvocationOnMock`.

Examples:

```
// answer by returning 12 every time
doAnswer(invocation -> 12).when(mock).doSomething();

// answer by using one of the parameters - converting into the right
// type as your go - in this case, returning the length of the second string parameter
// as the answer. This gets long-winded quickly, with casting of parameters.
doAnswer(invocation -> ((String)invocation.getArgument(1)).length())
    .when(mock).doSomething(anyString(), anyString(), anyString());
```

For convenience it is possible to write custom answers/actions, which use the parameters to the method call, as Java 8 lambdas. Even in Java 7 and lower these custom answers based on a typed interface can reduce boilerplate. In particular, this approach will make it easier to test functions which use callbacks. The functions answer and answerVoid can be found in `AdditionalAnswers` to create the answer object using the interfaces in `AnswerFunctionalInterfaces` support is provided for functions with up to 5 parameters

Examples:

```
// Example interface to be mocked has a function like:
void execute(String operand, Callback callback);

// the example callback has a function and the class under test
// will depend on the callback being invoked
void receive(String item);

// Java 8 - style 1
doAnswer(AdditionalAnswers.answerVoid((operand, callback) -> callback.receive("dummy"))
```

```
    .when(mock).execute(anyString(), any(Callback.class));

// Java 8 - style 2 - assuming static import of AdditionalAnswers
doAnswer(answerVoid((String operand, Callback callback) -> callback.receive("dummy"))
    .when(mock).execute(anyString(), any(Callback.class));

// Java 8 - style 3 - where mocking function to is a static member of test class
private static void dummyCallbackImpl(String operation, Callback callback) {
    callback.receive("dummy");
}

doAnswer(answerVoid(TestClass::dummyCallbackImpl)
    .when(mock).execute(anyString(), any(Callback.class));

// Java 7
doAnswer(answerVoid(new AnswerFunctionalInterfaces.VoidAnswer2() {
    public void answer(String operation, Callback callback) {
        callback.receive("dummy");
    }})).when(mock).execute(anyString(), any(Callback.class));

// returning a value is possible with the answer() function
// and the non-void version of the functional interfaces
// so if the mock interface had a method like
boolean isSameString(String input1, String input2);

// this could be mocked
// Java 8
doAnswer(AdditionalAnswers.answer((input1, input2) -> input1.equals(input2)))
    .when(mock).execute(anyString(), anyString());

// Java 7
doAnswer(answer(new AnswerFunctionalInterfaces.Answer2() {
    public String answer(String input1, String input2) {
        return input1 + input2;
    }})).when(mock).execute(anyString(), anyString());
```

TODO rework the documentation, write about hamcrest.