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

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

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

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

Alternative, shorter version of consecutive stubbing:

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

11. Stubbing with callbacks

Allows stubbing with generic Answer interface.

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

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

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

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

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

doThrow(Throwable) replaces the stubVoid(Object) method for stubbing voids. The main reason is improved readability and consistency with the family of doAnswer() methods.

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

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

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

Read more about other methods:

doThrow(Throwable)

doAnswer(Answer)

doNothing()

doReturn(Object)

13. Spying on real objects 1111

You can create spies of real objects. When you use the spy then the **real** methods are called (unless a method was stubbed).

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

Spying on real objects can be associated with "partial mocking" concept. **Before the release 1.8**, Mockito spies were not real partial mocks. The reason was we thought partial mock is a code smell. At some point we found legitimate use cases for partial mocks (3rd party interfaces, interim refactoring of legacy code, the full article is here)

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

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

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

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

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

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