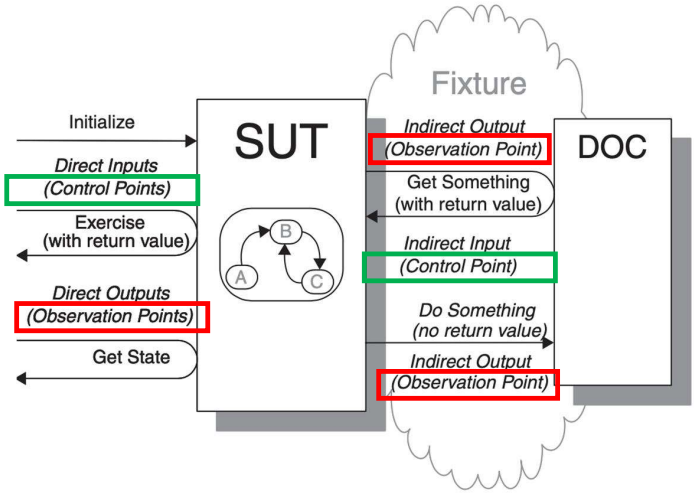
Tasty mocking framework for unit tests in Java

Unit Testing, SUT and its Dependencies

- Unit tests are designed to test the behavior of specific classes or methods without relying on the behavior of *their depended-on objects* (DOCs).
- Don't need to use actual implementations of DOCs.
- Usually, create '**stubs**'– specific implementations of an interface suitable for a given scenario.



The Phases of Mocking

- Stubbing

```
when(passwordEncoder.encode("1")).thenReturn("a");
```

- Verification

```
verify(passwordEncoder).encode("1"));
```


”Mockito is a **mocking framework** that tastes really good. It lets you write beautiful tests with clean & simple API. Mockito doesn’t give you hangover because the tests are very readable and they produce clean verification errors.”

https://site.mockito.org

Mockito is …

- An open source framework that lets you create and configure mocked objects, and using them to *verify the expected behavior of the system* being tested.

```
// Mockito
def mockito_version = "3.5.5"
testImplementation "org.mockito:mockito-core:$mockito_version"
testImplementation "org.mockito:mockito-android:$mockito_version"
androidTestImplementation "com.linkedin.dexmaker:dexmaker-mockito:2.25.1"
```


Mockito mocks …

- Interfaces
- Abstract classes
- Concrete non-final classes

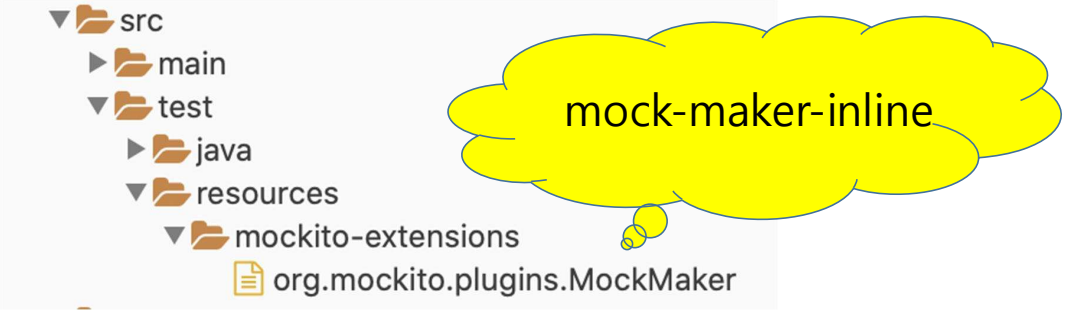
Mockito cannot mock …

- Final classes
- Final methods
- Static methods

Also note that the methods `equals()` and `hashCode()` cannot be mocked.

But if you really need it …

- Mockito 2+ provides the experimental **MockMaker** plugin
- Disabled by default
- Create `/mockito-extensions/org.mackito.plugins.MockMaker`



OR …

- Add the `mockito-inline` instead of the `mockito-core` artifact as follows:

```
testImplementation "org.mockito:mockito-inline:3.5.5"
```

Mock Object Creation (w/o Annotation)

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

public class MockCreationTest {

    List<String> mockedList;

    @Test
    public void should_create_mock() throws Exception {
        // mock creation
        @SuppressWarnings("unchecked")
        mockedList = mock(List.class);

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

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

    }
```

Mock Object Creation and Enabling
(with @Mock Annotation)

```
import static org.mockito.Mockito.*;
import org.mockito.Mock;

public class MockCreationTest {

    @Mock
    List<String> mockedList;

    @Test
    public void should_create_mock() throws Exception {

        /* Mock object already created */

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

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


Mock Object Creation and Enabling
(with @Mock Annotation)

- Enable annotations programmatically.

```
@Before
public void init() {
    MockitoAnnotations.openMocks(this); // initMocks deprecated
}
```

Default Return Values

```
interface Demo {
    int getInt();
    Integer getInteger();
    double getDouble();
    boolean getBoolean();
    String getObject();
    Collection<String> getCollection();
    String[] getArray();
    Stream<?> getStream();
    Optional<?> getOptional();
}
```

```
Demo demo = mock(Demo.class);
assertEquals(0, demo.getInt());
assertEquals(0, demo.getInteger().intValue());
assertEquals(0d, demo.getDouble(), 0d);
assertFalse(demo.getBoolean());
assertNull(demo.getObject());
assertEquals(Collections.emptyList(),demo.getCollection()
);
assertNull(demo.getArray());
assertEquals(0L, demo.getStream().count());
assertFalse(demo.getOptional().isPresent());
```


Stubbing Methods - Method 1

- To configure and define what to do when specific methods of the mock are invoked is called ***stubbing***.

"*when* this method is called, *then* do something."

```
when(passwordEncoder.encode("1")).thenReturn("a");
```

Stubbing Methods - Method 2

"**Do** something **when** this mock's method is called with the following arguments."

```
doReturn("a").when(passwordEncoder).encode("1");
```

```
when(passwordEncoder.encode("1")).thenReturn("a");
```


Returning Values

- `thenReturn()` or `doReturn()` are used to specify a value to be returned upon method invocation.
- Can also specify ***multiple*** return values for consecutive method calls. The last value will be used as a result for all further method calls.

```
when(passwordEncoder.encode("1"))
        .thenReturn("a")
        .thenReturn("b");       // .thenReturn("b", "c")
```

Throwing Exceptions

- `thenThrow()` and `doThrow()` configure a mocked method to throw an exception.

```
when(passwordEncoder.encode("1"))
        .thenThrow(IllegalArgumentException.class);


doThrow(new IllegalArgumentException())
        .when(passwordEncoder).encode("1");
```


Verifying Behavior

- Once a mock or spy has been used, we can verify that specific interactions took place.

"Hey, Mockito, make sure this method was called with these arguments."

```
verify(passwordEncoder).encode("a");

verify(mockedList, times(2)).add("hello");

verify(mockedList, atLeastOnce()).clear();

verifyNoInteractions(mockedList);

verifyNoMoreInteractions(mockedList);
```

Argument Matchers (1)

- If you want to define a reaction for a wider range of argument values, you can use **argument matchers** to match method arguments against.

```
when(passwordEncoder.encode(anyString())).thenReturn("exact");

doReturn("apple").when(myList).get(anyInt());

verify(yourList).get(anyBoolean());
```


Argument Matchers (2)

- Mockito requires you to provide ***all arguments*** either **by matchers** or **by exact value**.

```
when(mock.call("a", anyInt())).thenReturn(false); // oops! – compile error
```

```
when(mock.call("a", 42)).thenReturn(true);

when(mock.call(anyString(), anyInt())).thenReturn(false);

verify(mock).call(eq("b"), anyInt());

verify(passwordEncoder).encode(or(eq("a"), endsWith("b")));
```

Mocking void Methods with Mockito

- Use `doThrow()/doAnswer()/doNothing()/doReturn()` and `doCallRealMethod()` instead of `when()`.

- It is necessary when you
  1. stub `void` methods
  2. stub methods on **Spy** objects
  3. stub the same method more than once, to change the behavior of a mock in the middle of a test.

doAnswer/thenAnswer

```
doAnswer(new Answer() {
    @Override
    public Object answer(InvocationOnMock invocation) throws Throwable {
        Callback callback = invocation.getArgument(1);
        callback.reply(response);
        return null;
    }
}).when(service).getResponse(anyString(), isA(Callback.class));
```

```
doAnswer(invocation -> {
    Callback callback = invocation.getArgument(1);
    callback.reply(response);
    return null;
}).when(service).getResponse(anyString(), isA(Callback.class));
```

ArgumentCaptor

```
ArgumentCaptor<String> passwordCaptor = ArgumentCaptor.forClass(String.class);

passwordEncoder.encode("password");

verify(passwordEncoder).encode(passwordCaptor.capture());
assertEquals("password", passwordCaptor.getValue());
```

```
@Captor
ArgumentCaptor<String> passwordCaptor;

passwordEncoder.encode("password1");
passwordEncoder.encode("password2");

verify(passwordEncoder, times(2)).encode(passwordCaptor.capture());
assertEquals(Arrays.asList("password1", "password2"),
        passwordCaptor.getAllValues());
```


Mockito Verify Cookbook

```
1  public class MyList extends AbstractList<String> {
2
3      @Override
4      public String get(final int index) {
5          return null;
6      }
7      @Override
8      public int size() {
9          return 0;
10     }
11 }
```

verify simple invocation on mock

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
verify(mockedList).size();
```

verify number of interactions with mock

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
verify(mockedList, times(1)).size();
```

verify no interaction with the whole mock occurred

```
List<String> mockedList = mock(MyList.class);
verifyZeroInteractions(mockedList);
```

verify no interaction with a specific method occurred

```
List<String> mockedList = mock(MyList.class);
verify(mockedList, times(0)).size();
```


verify there are no unexpected interactions – this should fail:

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
mockedList.clear();
verify(mockedList).size();
verifyNoMoreInteractions(mockedList);
```

verify order of interactions

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
mockedList.add("a parameter");
mockedList.clear();

InOrder inOrder = Mockito.inOrder(mockedList);
inOrder.verify(mockedList).size();
inOrder.verify(mockedList).add("a parameter");
inOrder.verify(mockedList).clear();
```

verify an interaction has not occurred

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
verify(mockedList, never()).clear();
```

verify an interaction has occurred at least certain number of times

```
List<String> mockedList = mock(MyList.class);
mockedList.clear();
mockedList.clear();
mockedList.clear();

verify(mockedList, atLeast(1)).clear();
verify(mockedList, atMost(10)).clear();
```

verify interaction with exact argument

```
List<String> mockedList = mock(MyList.class);
mockedList.add("test");
verify(mockedList).add("test");
```


verify interaction with flexible/any argument

```
List<String> mockedList = mock(MyList.class);
mockedList.add("test");
verify(mockedList).add(anyString());
```

verify interaction using argument capture

```
List<String> mockedList = mock(MyList.class);
mockedList.addAll(Lists.<String> newArrayList("someElement"));
ArgumentCaptor<List> argumentCaptor = ArgumentCaptor.forClass(List.class);
verify(mockedList).addAll(argumentCaptor.capture());
List<String> capturedArgument = argumentCaptor.<List<String>> getValue();
assertThat(capturedArgument, hasItem("someElement"));
```

configure simple return behavior for mock

```
MyList listMock = Mockito.mock(MyList.class);
when(listMock.add(anyString())).thenReturn(false);

boolean added = listMock.add(randomAlphabetic(6));
assertThat(added, is(false));
```

configure return behavior for mock in an alternative way

```
MyList listMock = Mockito.mock(MyList.class);
doReturn(false).when(listMock).add(anyString());

boolean added = listMock.add(randomAlphabetic(6));
assertThat(added, is(false));
```

configure mock to throw an exception on a method call

```
@Test(expected = IllegalStateException.class)
public void givenMethodIsConfiguredToThrowException_whenCallingMethod_thenExceptionIsThrown(
    MyList listMock = Mockito.mock(MyList.class);
    when(listMock.add(anyString())).thenThrow(IllegalStateException.class);

    listMock.add(randomAlphabetic(6));
}
```

configure the behavior of a method with void return type – to throw an exception

```
MyList listMock = Mockito.mock(MyList.class);
doThrow(NullPointerException.class).when(listMock).clear();

listMock.clear();
```


configure the behavior of multiple calls

```
MyList listMock = Mockito.mock(MyList.class);
when(listMock.add(anyString()))
  .thenReturn(false)
  .thenThrow(IllegalStateException.class);

listMock.add(randomAlphabetic(6));
listMock.add(randomAlphabetic(6)); // will throw the exception
```

configure the behavior of a spy

```
MyList instance = new MyList();
MyList spy = Mockito.spy(instance);

doThrow(NullPointerException.class).when(spy).size();
spy.size(); // will throw the exception
```

configure method to call the real, underlying method on a mock

```
MyList listMock = Mockito.mock(MyList.class);
when(listMock.size()).thenCallRealMethod();

assertThat(listMock.size(), equalTo(1));
```

configure mock method call with custom Answer

```
MyList listMock = Mockito.mock(MyList.class);
doAnswer(invocation -> "Always the same").when(listMock).get(anyInt());

String element = listMock.get(1);
assertThat(element, is(equalTo("Always the same")));
```


BDDMockito – BDD-like consistent Syntax

- Alternative naming convention
  - `given…will` instead of `when…then`
- To follow *given-when-then* test structure

```
@Test
public void shouldReturnGivenValueUsingBDDNotation() {
    // given
    TacticalStation tacticalStationMock = mock(TacticalStation.class);
    given(tacticalStationMock.getNumberOfTubes()).willReturn(TEST_NUMBER_OF_TORPEDO_TUBES);
    // when
    int numberOfTubes = tacticalStationMock.getNumberOfTubes();
    // then
    assertThat(numberOfTubes, is(equalTo(TEST_NUMBER_OF_TORPEDO_TUBES)));
}
```

BDDMockito – new alias for verification

- `then` instead of `verify`

```
@Test
public void shouldVerifyWithSimpleArgumentMatching() {
    // given
    TacticalStation tacticalStationMock = mock(TacticalStation.class);
    // when
    tacticalStationMock.fireTorpedo(5);
    // then
    then(tacticalStationMock).should().fireTorpedo(gt(3));
    // verify(tacticalStationMock).fireTorpedo(gt(3));
}
```

- with its counterparts
  - `then().should(InOrder inOrder)`
  - `then().should(InOrder inOrder, VerificationMode mode)`
  - `then().shouldHaveZeroInteractions()`
  - `then().shouldHaveNoMoreInteractions()`

BDDMockito – adjustment to classic Mockito

- New/renamed equivalent methods
  - `given().willThrow(Class<? Extends Throwable> throwableType)`
  - `given().will(Answer<?> answer)`
  - `given().willReturn(Object value, Object... nextValues)`
  - `given().willDoNothing()`