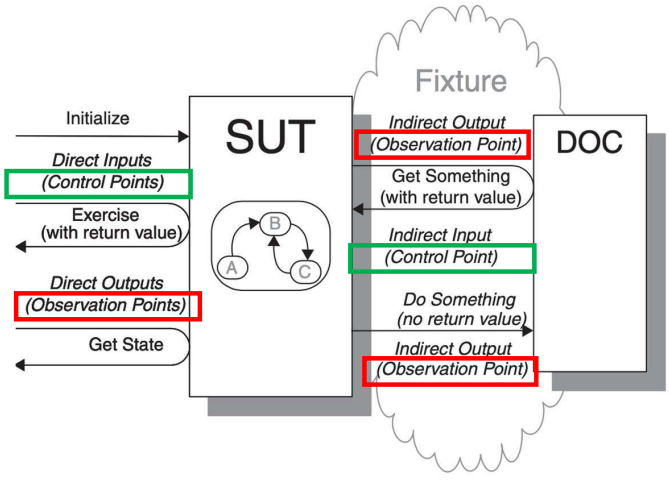
Kotlin + mockito

Mocking frameworks for unit testing in Kotlin

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.



Why is it better than a well-known Mockito library for Kotlin?

- Final by default(Classes and methods in kotlin)
- Object mocking
- Extension functions
- Chained mocking
- Mocking static methods is not easy in mockito but using mockK java static methods can easily be mocked using `mockStatic`

The Phases of Mocking

- Stubbing

```
every { passwordEncoder.encode("1") } returns "a"
```

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

- Verification

```
verify { passwordEncoder.encode("1") }
```

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

Microsoft C# Peoples

- ***Fake*** - A fake is a generic term which can be used to describe *either a stub or a mock* object. Whether it is a stub or a mock depends on the context in which it's used. So in other words, a fake can be a stub or a mock.
- ***Mock*** - A mock object is a fake object in the system that decides whether or not a unit test has passed or failed. A mock starts out as a Fake until it is asserted against.
- ***Stub*** - A stub is a controllable replacement for an existing dependency (or collaborator) in the system. By using a stub, you can test your code without dealing with the dependency directly. By default, a fake starts out as a stub.

Is this a Mock?

```
var mockOrder = new MockOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(purchase.CanBeShipped);
```

A little Better!

```
var stubOrder = new FakeOrder();
var purchase = new Purchase(stubOrder);

purchase.ValidateOrders();

Assert.True(purchase.CanBeShipped);
```


To use it as a Mock …

```
var mockOrder = new FakeOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(mockOrder.Validated);
```

- In this case, you are checking a property on the Fake (asserting against it), so in the above code snippet, the `mockOrder` is a Mock.

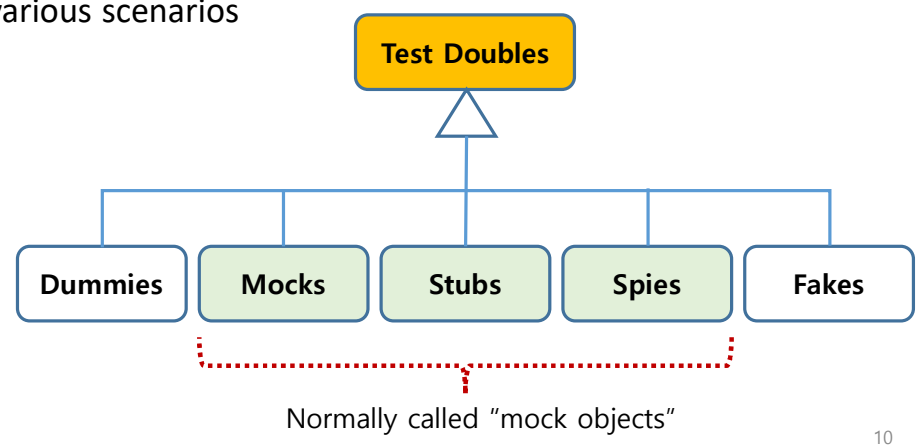
> ⓘ **Important**
>
> It's important to get this terminology correct. If you call your stubs "mocks", other developers are going to make false assumptions about your intent.

The main thing to remember about mocks versus stubs is that mocks are just like stubs, but

- you assert against the mock object, whereas
- you do not assert against a stub.

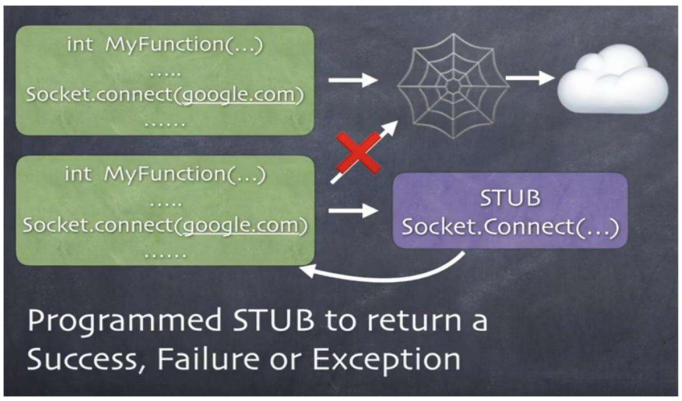
Test Doubles

- Used in lieu of external dependencies
  - DB, Web, API, Library, Network etc.
  - Easy to simulate various scenarios



Stubs

- Generates predefined outputs
- Does not provide validation of how the class uses the dependency
- Used when data is required by the class but the process used to obtain it isn't relevant to what's being tested
- Usually created using a mock framework

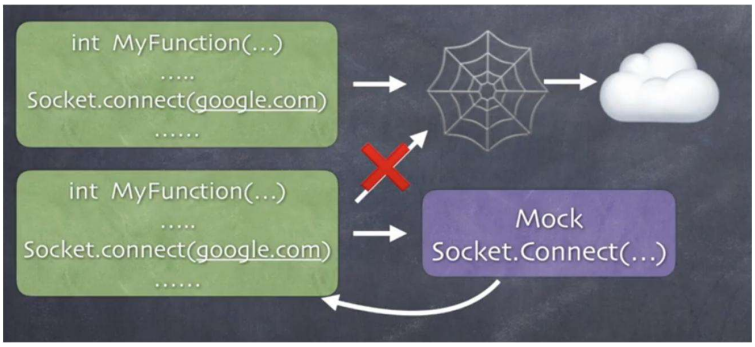


- Returns success, failure or exceptions (as coded)

Checks the behavior of code under test in case of these return values

Mocks

- Mocks replaces external interface
- Mechanism for validating how a dependency is used by the class
- Can provide data required by the class (by stubbing)
- Created by a mocking framework

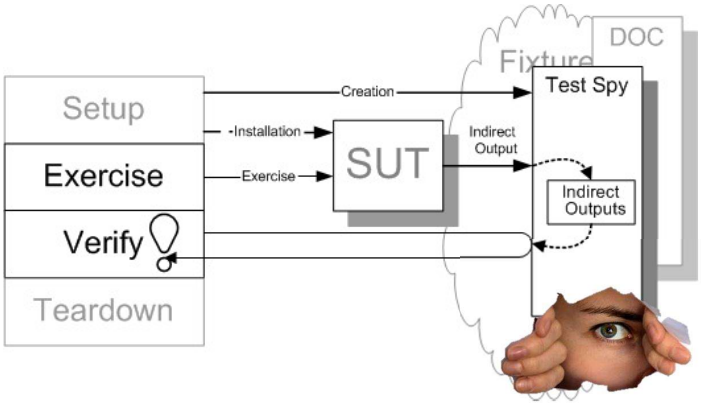


- Mocked function called or NOT?
- How many times it gets called?
- What parameters are passes when it was called?

Right call, Right # of times with Right setup parameters

Spy

- A stand-in for DOC used by SUT
- Creating a spy requires a real object to spy on
- Might be useful for testing legacy code (“partial mock”)
- Consider using mocks instead of spies whenever possible.
- Usually created using a mock framework

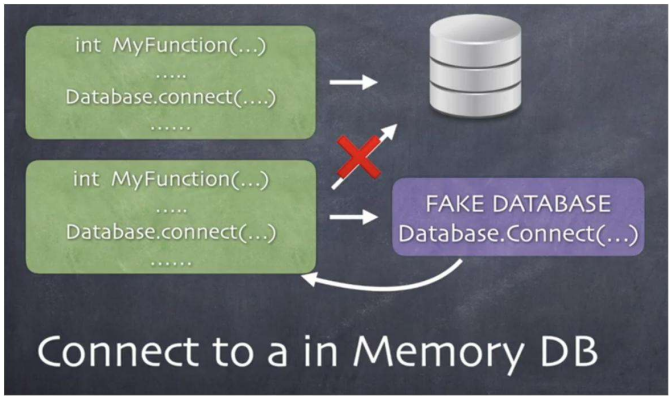


- By default, a spy delegates all method calls to the real object and records what method was called and with what parameters.
- Can selectively stub methods

Like Mocks and Stubs, normally used for **behavior verification** of SUT.

Fakes

- Almost working simplified implementation
- Usually coded directly, without the use of a framework
- Does not provide direct validation of how the class uses dependency
- Used when the class being tested requires a specific logic in the dependency



- Instead of actually going to the internet, it connects to a local (limited) implementation
- Created specifically for this test

Check the behavior with respect to actual (potentially lots of) data it receives.

MockK and Mockito are …

- Open source frameworks that let you create and configure mocked objects, and using them to *verify the expected behavior of the system* being tested.

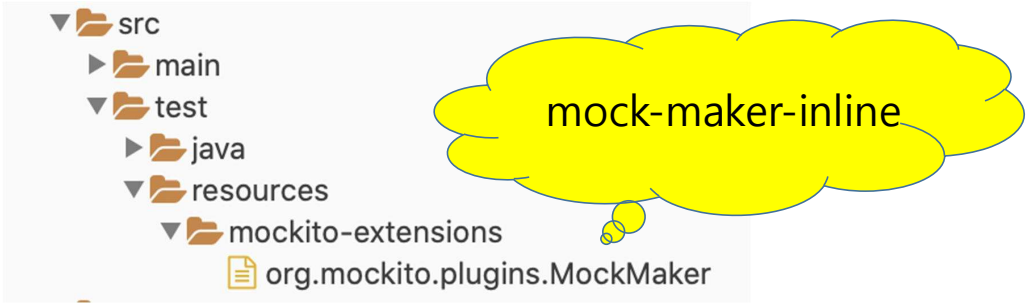
```
// MockK
testImplementation "io.mockk:mock:$mockk_version"
testImplementation "io.mockk:mockk-android:$mockk_version"
// Mockito
def mockito_version = "3.5.5"
testImplementation "org.mockito:mockito-core:$mockito_version"
testImplementation "org.mockito:mockito-android:$mockito_version"
androidTestImplementation "com.linkedin.dexmaker:dexmaker-mockito:$dexmaker_version"
// Mockito-kotlin
testImplementation "org.mockito.kotlin:mockito-kotlin:$mockito_kotlin_version"
```


Mockito can or cannot mock …

Can mock	Cannot mock
• Interfaces	• Final classes
• Abstract classes	• Final methods
• Concrete non-final classes	• Static methods

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)

```
import io.mockk.mockk

class MockCreationTest {

    lateinit var mockedList: List<String>

    @Test
    fun should_create_mock() {
        // mock creation
        mockedList = mockk()
    }

}
```

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

public class MockCreationTest {

    List<String> mockedList;

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


Mock Object Creation and Enabling
(with @MockK/@Mock Annotation)

```
import io.mockk.MockKAnnotations
import io.mockk.impl.annotations.MockK

class MockCreationTest {

    @MockK
    lateinit var mockedList: MutableList<String>

    @Test
    fun should_create_mock() {
        /* Mock object already created */

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

        // verification
        verify { mockedList.add("one") }
    }

    ...
```

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

public class MockCreationTest {

    @Mock
    List<String> mockedList;

    @Test
    public void should_create_mock() {

        /* Mock object already created */

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

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

Mock Object Creation and Enabling
(with @MockK/@Mock Annotation)

- Enable annotations programmatically.

```
@Before
fun init() {
    MockKAnnotations.init(this)
}
```

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


Default Return Values (MockK)

- Reasonable default values depending on types.
- MockK demands a **strict mock** by default.

```
@RelaxedMockK
lateinit var mockCar: Car

@Test
fun should_create_mock() {
    mockCar = mockk(relaxed = true)
}

@Before
fun init() {
    // turn relaxed and relaxUnitFun on for all mocks
    MockKAnnotations.init(this, relaxed = true) // relaxUnitFun = true
}
```

Default Return Values (Mockito)

```
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());
```


Strict Mocks

```
interface Reader {
    fun readText(): String
}
```

```
@Test
fun `MockK - strict mock - violation`() {
    val mockReader = mockk<Reader>()

    val text = mockReader.readText()

    assertThat(text).isEmpty()
    verify { mockReader.readText() }
}
```

```
@Test
fun `Mockito - mock`() {
    val mockReader = mock<Reader>()

    val text = mockReader.readText()

    assertThat(text).isNull()
    verify(mockReader).readText()
}
```

Test Failed

io.mockk.MockKException: no answer found for:
Reader(#5).readText()

MockK: Strick to Lenient Mock

```
@Test
fun `MockK - strict mock`() {
    val mockReader = mockk<Reader>(relaxed = true)

    val text = mockReader.readText()

    assertThat(text).isEmpty()
    verify { mockReader.readText() }
}
```

```
@Test
fun `MockK – strict mock`() {
    val mockReader = mockk<Reader> {
        every { readText() } returns ""
    }

    val text = mockReader.readText()

    assertThat(text).isEmpty()
    verify { mockReader.readText() }
}
```


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."

```
every { passwordEncoder.encode("1")} returns "a"
```

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

Stubbing Methods - Method 2 (only Mockito)

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

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

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


Returning Values (Mockito)

- `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")
```

Returning Values (MockK)

- `returns` is used to specify a value to be returned.
- Can also specify ***multiple*** return values for consecutive method calls. The last value will be used as a result for all further method calls.

```
every {
      passwordEncoder.encode("1")
} returns "a" andThen "b" andThen "c"
```

```
every {
      passwordEncoder.encode("1")
} returnsMany listOf("a", "b", "c")
```


Throwing Exceptions (Mockito)

- `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");
```

Throwing Exceptions (MockK)

- `throws` configures a mocked method to throw an exception.

```
every { passwordEncoder.encode("1") }
      throws IllegalArgumentException("exception")
```


Verifying Behavior

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

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

```
Verify { passwordEncoder.encode("a") }

verify(exactly=2) { mock.add("hello") }

verify(atLeast=1) { mock.clear() }

Verify { mock wasNot Called }

verifyAll { mock1.called(), mock2.called() }
```

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

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

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

verifyNoInteractions(mocked);

verifyNoMoreInteractions(mock);
```

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());
```

- mockK uses `any()` or `ofType<T>()` for most cases.

```
every { passwordEncoder.encode(any()) } returns "exact"
```


Argument Matchers (2) (Mockito)

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

```
when(mock.call("a", anyInt())).thenReturn(false); // 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")));
```

Partial argument matching (MockK)

- In MockK, **matchers** and **exact value** can be mixed.

```
every { mock.call("a", any()) } returns false // No Problem! - OK
```

```
every { mock.call("a", 42) } returns true
every { mock.call(any(), any()) } returns false
verify { mock).call("b", any()) }
verify { passwordEncoder.encode(or("a", match { it.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.

Mocking void Methods with MockK

- If the function is returning `Unit` you can use the `justRun` construct:

```
justRun { obj.sum(any(), 3) }
every { obj.sum(any(), 3) } just Runs
every { obj.sum(any(), 3) } returns Unit
every { obj.sum(any(), 3) } answers { Unit }
```


doAnswer/thenAnswer (Mockito)

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

answers (MockK)

```
every { service.getResponse(any(), ofType<Callback>)) }.
    answer { call ->
        callback = call.invocation.args[1];
        callback.reply(response);
        Unit
    }
```


ArgumentCaptor (Mockito)

```
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());
```

CapturingSlot or MutableList (MockK)

```
val passwordSlot = slot<String>()

passwordEncoder.encode("password");

verify { passwordEncoder.encode(capture(passwordSlot)) }
assertEquals("password", passwordSlot.captured)
```

```
@Captor
val list = mutableListOf<String>()


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

verify(exactly=2) { passwordEncoder.encode(capture(list)) }
assertEquals(listOf("password1", "password2"), list);
```


MockK Only

Verification order

- verifyAll
  - verifies that all calls happened without checking their order.
- verifySequence
  - verifies that the calls happened in a specified sequence.
- verifyOrder
  - verifies that calls happened in a specific order.
- wasNot Called
  - verifies that the mock or the list of mocks was not called at all.

Annotations

```
class CarTest {
  @MockK
  lateinit var car1: Car

  @RelaxedMockK
  lateinit var car2: Car

  @MockK(relaxUnitFun = true)
  lateinit var car3: Car

  @SpyK
  var car4 = Car()

  @InjectMockKs
  var trafficSystem = TrafficSystem()

  @Before
  fun setUp() = MockKAnnotations.init(this, relaxUnitFun = true)
```

Object mocks

- Use `mockkObject`.
- To revert back, use `unmockkAll` or `unmockkObject`:

```
object MockObj {
 fun add(a: Int, b: Int) = a + b
}

mockkObject(MockObj) // applies mocking to an Object
assertEquals(3, MockObj.add(1, 2))
every { MockObj.add(1, 2) } returns 55
assertEquals(55, MockObj.add(1, 2))
unmockObject(MockObj)
```

Despite the Kotlin language limits, you can create new instances of objects if required by testing logic:

```
val newObjectMock = mockk<MockObj>()
```

```
@Before
fun beforeTests() {
    mockkObject(MockObj)
    every { MockObj.add(1,2) } returns 55
}

@Test
fun willUseMockBehaviour() {
    assertEquals(55, MockObj.add(1,2))
}

@After
fun afterTests() {
    unmockkAll()
    // or unmockkObject(MockObj)
}
```


Coroutines

testImplementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:x.x"

- Then you can use `coEvery`, `coVerify`, `coMatch`, `coAssert`, `coRun`, `coAnswers` or `coInvoke` to mock suspend functions.