MockK, the idiomatic mocking framework for Kotlin

Java and Mockito

Java and Mockito

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {
    @Mock
    private UserRepository userRepository;

    private UserService userService;

    @Before
    public void setUp() {
         userService = new UserService(userRepository);
    }

    @Test
    public void user_service_when_valid_id_specified_should_return_user() {
        given(this.userRepository.findUser(1L)).willReturn(new User(1L,
"John"));
        final User user = this.userService.lookupUser(1L);

        assertThat(user.getId()).isEqualTo(1L);
        assertThat(user.getName()).isEqualTo("John");
    }
}
```

Kotlin and Mockito?

Adding the dependency

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
```

Writing some cool classes

```
class Generator {
     fun generate(): String = "Foo"
}
class Dao {
fun insert(record: String) = println("""Inserting "$record"""")
}
class Service(private val generator: Generator, private val dao:
Dao) {
    fun calculate() {
         val record = generator.generate()
         dao.insert(record)
   }
}
```

Adding our unit test

```
val generator = Mockito.mock(Generator::class.java)
val dao = Mockito.mock(Dao::class.java)
val service = Service(generator, dao)

@Test fun `calculate should generate and insert a record`() {
        val mockedRecord = "mocked String"
        Mockito.`when`(generator.generate()).thenReturn(mockedRecord)

        service.calculate()

        Mockito.verify(generator).generate()
        Mockito.verify(dao).insert(mockedRecord)
        Mockito.verifyNoMoreInteractions(generator, dao)
}
```

Taking it for a spin

booooom!!!!

org.mockito.exceptions.base.MockitoException:
Cannot mock/spy class
be.yannickdeturck.HelloTest$Generator
Mockito cannot mock/spy because :
- final class
- anonymous classes
- primitive types

What? 

So… now what?

Classes and functions are final by default in Kotlin

This doesn’t bode too well with how Mockito creates its mocks as it relies on subclassing

What are our options then?

Add open to everything?!

Define an interface for everything?

Enable incubating, opt-in feature

Mock the unmockable: opt-in mocking of final classes/methods

For a long time our users suffered a disbelief when Mockito refused to mock a final class. Mocking of final methods was even more problematic, causing surprising behavior of the framework and generating angry troubleshooting. The lack of mocking finals was a chief limitation of Mockito since its inception in 2007. The root cause was the lack of sufficient support in mock creation / bytecode generation. Until Rafael Winterhalter decided to fix the problem and provide opt-in implementation in Mockito 2.1.0. In the releases, Mockito team will make mocking the unmockable completely seamless, greatly improving developer experience.

Mocking of final classes and methods is an **incubating**, opt-in feature. It uses a combination of Java agent instrumentation and subclassing in order to enable mockability of these types. As this works differently to our current mechanism and this one has different limitations and as we want to gather experience and user feedback, this feature had to be explicitly activated to be available ; it can be done via the mockito extension mechanism by creating the file `src/test/resources/mockito-extensions/org.mockito.plugins.MockMaker` containing a single line:

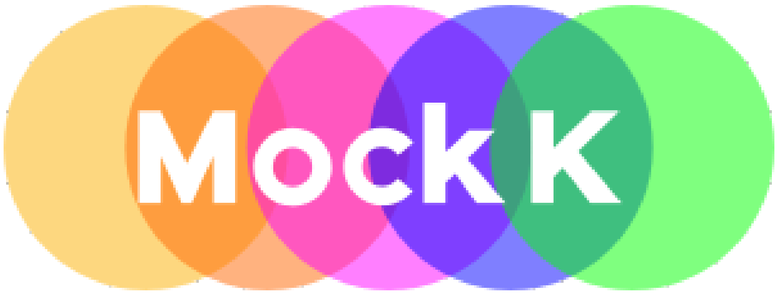
```
mock-maker-inline
```

Enable incubating, opt-in feature

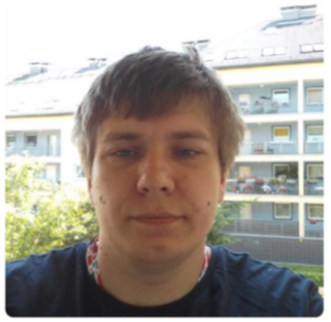
"Add a certain file with a certain content in a certain location for it to work"

A programmatic way of using this feature is foreseen later on

Enter MockK



Enter MockK



Oleksiy Pylypenko
oleksiyp

Follow

Block or report user

mockk author, Platform Engineering team tech-lead

UBS
Wroclaw
oleksiy.pylypenko@gmail.com
http://mockk.io

Overview | Repositories 90 | Projects 0 | Stars 84 | Followers 48 | Following 4

Pinned

mockk/mockk
mocking library for Kotlin
Kotlin ★ 1.7k 71

equal5
engine and GUI application to build plots and plot timelapses of mostly any indirect functional equation f(x,y)=0
Java ★ 1

byte-lib
Library to work with byte strings and streams
Java

computer-vision-youtube-switcher
OpenCV based script to switch youtube videos (cartoons for disabled kid)
Jupyter Notebook

raytracer-mit6.172-java
Rewrite of MIT 6.172 raytracer from C to Java
Java

textspace
Text space is a unbounded html5 console. Just for fun.
Java ★ 1

619 contributions in the last year

Apr May Jun Jul Aug Sep Oct Nov Dec Jan Feb Mar Apr
Mon
Wed
Fri

Learn how we count contributions.
Less More

2019
2018
2017
2016

Matcher expressions

By default simple arguments are matched using *eq*()

Lots of matchers available!

MockK's main philosophy

- First-class support for Kotlin features
- Offer a pure Kotlin mocking DSL for writing clean and idiomatic code
- Mock support for final classes and functions

Matcher expressions

Matcher	Description
`any()`	Matches any argument
`isNull()`	Checks if values is null
`isNull(inverse=true)`	Checks if value is not null
`oftype(type)`	Checks if value belong to the type
`match { it.startsWith("string") }`	Matches via passed predicate
`matchNullable { it?.startsWith("string") }`	Matches nullable value via passed predicate
`eq(value)`	Matches if value is equal to the provided via deepEquals function
`neq(value)`	Matches if value is not equal to the provided via deepEquals function
`refEq(value)`	Matches if value is equal to the provided via reference comparation
`nrefEq(value)`	Matches if value is not equal to the provided via reference comparation

Mocking final classes and functions

- MockK builds proxies for mocked classes
- Code of existing methods is transformed the way it includes calls to a mocking library
- Small trade-off to performance

Adding the dependency

```
<dependency>
    <groupId>io.mockk</groupId>
    <artifactId>mockk</artifactId>
    <version>${mockk.version}</version>
    <scope>test</scope>
</dependency>
```

Matcher expressions

Matcher	Description
`cmpEq(value)`	Matches if value is equal to the provided via compare to function
`less(value)`	Matches if value is less to the provided via compare to function
`more(value)`	Matches if value is more to the provided via compare to function
`less(value, andEquals=false)`	Matches if value is less or equals to the provided via compare to function
`more(value, andEquals=false)`	Matches if value is more or equals to the provided via compare to function
`range(from, to, fromInclusive=true, toInclusive=true)`	Matches if value is in range via compare to function
`and(left, right)`	Combines two matches via logical and
`or(left, right)`	Combines two matches via logical or
`not(matcher)`	negates the matcher

Meet the Car class

```
class Car {
    fun drive(direction: Direction,
              speed: Long = 30): String {
        return "Driving $direction at $speed km/h"
    }
}
```

Combine them!

Combine them as you see fit

```
every {
    car.drive(
        direction = or(Direction.EAST, Direction.WEST),
        speed = range(10L, 50L, fromInclusive = true, toInclusive = true))
} returns "Driving!"
```

Syntax

```
// Create mock/spy
val car = mockk<Car>()

// Stub calls
every { car.drive(Direction.NORTH) } returns
"Driving north!”

// Execute code to test
car.drive(Direction.NORTH)

// Verify
verify { car.drive(Direction.NORTH) }
```

Capturing arguments

Single value captured

```
val car = mockk<Car>()
val slot = slot<Long>()
every { car.drive(any(), capture(slot)) } returns "Driving!"
car.drive(Direction.EAST, 55)
assert(slot.captured).isEqualTo(55L)
```

Syntax

```
@Test
fun `car should be able to drive in a direction`() {
    every { car.drive(Direction.NORTH) } returns
"Driving north!"


car.drive(Direction.NORTH)

verify { car.drive(Direction.NORTH) }
}
```

Capturing arguments

Multiple values captured

```
val car = mockk<Car>()
val list = mutableListOf<Long>()
every { car.drive(any(), capture(list)) } returns "Driving..."
car.drive(Direction.EAST, 55)
car.drive(Direction.EAST, 80)
assert(list).containsAll(55L, 80L
```

Strict mocking

By default mocks are strict, so you need to define some behavior

```
// every { car.drive(Direction.NORTH) } returns "Driving north!”
car.drive(Direction.NORTH)
```

```
io.mockk.MockKException: no answer found for: Car(#2).drive(north)
```

Relaxed mock

```
val car = mockk<Car>(relaxed = true)
```

MockK will provide a dummy stub

Tip: Stick to strict mocking for nice and clear tests

Constructor mocks

Mock newly created objects

```
mockkConstructor(Car::class)
every { anyConstructed<Car>().drive(any()) } returns
"Driving!"
Car().drive(Direction.EAST)
verify { anyConstructed<Car>().drive(Direction.EAST) }
```

Object mocks

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

mockkObject(Math)

assertEquals(3, Math.add(1, 2))

every { Math.add(1, 2)) returns 4

assertEquals(4, Math.add(1, 2))
```

Private function mocking

```
class Car {
    fun drive(direction: Direction, speed: Long = 30): String {
        return accelerate()
    }
    private fun accelerate() = "going fast"
}
```

Hierarchical mocking

Mix mocks and real objects

```
interface AddressBook {
    val contacts: List<Contact>
}

interface Contact {
    val name: String
    val telephone: String
    val address: Address
}
interface Address {
    val city: String
    val zip: String
}
```

Private function mocking

```
val car = spyk<Car>(recordPrivateCalls = true)
every { car["accelerate"]() } returns "going not so fast”
car.drive(Direction.EAST)
verifySequence {
    car.drive(Direction.EAST)
    car["accelerate"]()
}
```

Hierarchical mocking

```
val addressBook = mockk<AddressBook> {
    every { contacts } returns listOf(
            mockk {
                every { name } returns "John"
                every { telephone } returns "123-456-789"
                every { address.city } returns "New-York"
                every { address.zip } returns "123-45"
            },
            mockk {
                every { name } returns "Alex"
                every { telephone } returns "789-456-123"
                every { address } returns mockk {
                        every { city } returns "Wroclaw"
                        every { Zip } returns "543-21"
                }
            }
        )
}
```

Private function mocking

```
every { car getProperty "speed" } returns 33
every { car setProperty "acceleration" value less(5) } just Runs
every { car invokeNoArgs "accelerate" } returns ”going slowly”
every { car invoke "accelerate" withArguments listOf("foo",
        "bar") } returns "going slowly"
```

Spy

Mix mocks and real objects

```
val car = spyk(Car())

car.drive(Direction.NORTH)

verify { car.drive(Direction.NORTH) }
```

Coroutines mocking

Requires an additional dependency

```
<dependency>
    <groupId>org.jetbrains.kotlinx</groupId>
    <artifactId>kotlinx-coroutines-core</artifactId>
    <version>${kotlinx-coroutines-core.version}</version>
    <scope>test</scope>
</dependency>
```

Annotations

```
@MockK
lateinit var car1: Car

@MockK
lateinit var car2: Car

@MockK (relaxUnitFun = true)
lateinit var car2: Car

@SpyK
var car4 = Car()

@InjectMockKs
var trafficSystem = TrafficSystem()

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

Verify

Verify whether call has happened

```
verify { car.drive(any()) }

verify ( atLeast = 1) { car.drive(any()) }

verify ( atMost = 3) { car.drive(any()) }

verify ( exactly = 2) { car.drive(any()) }

verify ( timeout = 1000L) { car.drive(any()) }
```

Coroutines mocking

Usual functions with prefox “co“

```
val car = mockk<Car>()
coEvery { car.drive(Direction.NORTH) } returns "Driving...”
car.drive(Direction.NORTH)
coVerify { car.drive(Direction.NORTH) }
```

Group verifications

```
car.drive(Direction.NORTH)
car.drive(Direction.EAST)

verify {

    car.drive(Direction.NORTH)
    car.drive(Direction.EAST)
}
```

Clear, informative error messages

```
val car = mockk<Car>()
car.drive(Direction.EAST)
verify { car.drive(Direction.EAST) }
```

```
io.mockk.MockKException: no answer found for: Car(#3).drive(east, 30)
```

Group verifications

```
car.drive(Direction.NORTH)
car.drive(Direction.EAST)
car.drive(Direction.SOUTH)

verify {

    car.drive(Direction.SOUTH)
    car.drive(Direction.NORTH)
}
```

Clear, informative error messages

java.lang.AssertionError: Verification failed: number of calls happened not matching exact number of verification sequence

Matchers:

Car(#2).drive(eq(north), eq(30)))

Car(#2).drive(eq(south), eq(30)))

Calls:

1) Car(#2).drive(north, 30)

2) Car(#2).drive(east, 30)

3) Car(#2).drive(south, 30)

Group verifications

```
car.drive(Direction.NORTH)
car.drive(Direction.EAST)
car.drive(Direction.SOUTH)

verify {

    car.drive(Direction.NORTH)
    car.drive(Direction.SOUTH)
}
```

Clear, informative error messages

java.lang.AssertionError: Verification failed: call 1 of 1: Car(#2).drive(any(), eq(30))). 3 matching calls found, but needs at least 2 and at most 2 calls

Calls:

1) Car(#2).drive(north, 30)

2) Car(#2).drive(east, 30)

3) Car(#2).drive(south, 30)

Group verifications

```
car.drive(Direction.NORTH)
car.drive(Direction.EAST)
car.drive(Direction.SOUTH)

verify {

    car.drive(Direction.NORTH)
    car.drive(Direction.EAST)
    car.drive(Direction.SOUTH)
}
```

Comparing both

```
val generator =
Mockito.mock(Generator:: class. java )
val dao = Mockito.mock(Dao:: class. java )
val service = Service( generator, dao )

    @Test
    fun `my test`() {
    val mockedRecord = "mocked String"
    Mockito.`when`( generator.generate())
    .thenReturn(mockedRecord)

    service. calculate()

    Mockito.verify(generator).generate()
    Mockito.verify(dao).insert(mockedRecord)
    Mockito.verifyNoMoreInteractions(generator, dao)

    }
```

Group verifications

```
car.drive(Direction.NORTH)
car.drive(Direction.EAST)
car.drive(Direction.SOUTH)

verify {

    car.drive(Direction.NORTH)
    car.drive(Direction.SOUTH)
}
```

My own experiences

- A very promising library
- Very pleasant to use
- Suits the Kotlin language
- Author is very approachable on Gitter and GitHub

Confirming verifications

Check whether all calls were covered by verify statements

```
val cat = mockk<Cat>()

every { cat.makeSound() } returns ”Meow"

cat.makeSound()

// verify { cat.makeSound() }

confirmVerified(cat)
```

Similar to Mockito’s verifyNoMoreInteractions

So… who’s going to try out MockK?

Confirming verifications

Option to exclude less significant calls

```
excludeRecords { cat.makeSound() }
```

Extra resources

mockk.io

Mocking is not Rocket Science: Part 1 Part 2 Part 3 Advance

Mocking in Kotlin with Mockk - Yannick De Turck

KotlinConf 2018 - Best Practices for Unit Testing in Kotlin by Philipp Hauer

springmockk

Thank you