JUnit 5 and Mockito 3

Modern Java Testing

Contact Info

Ken Kousen

Kousen IT, Inc.

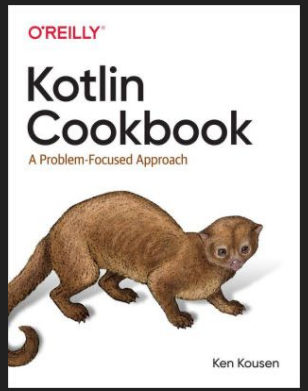
ken.kousen@kousenit.com

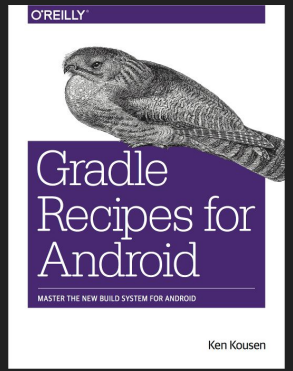
http://www.kousenit.com

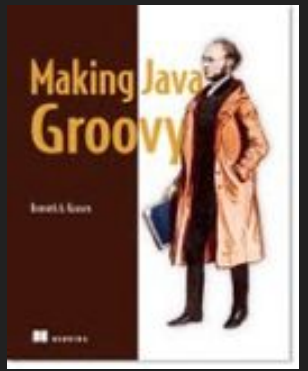
http://kousenit.org (blog)

@kenkousen (twitter)

https://kenkousen.substack.com (newsletter)







GitHub repositories

JUnit 5:

https://github.com/kousen/junit5_workshop

Mockito (and the Hamcrest Matchers):

https://github.com/kousen/mockito-hamcrest

JUnit 5 Links

- Home page: https://junit.org/junit5/

- User manual: https://junit.org/junit5/docs/current/user-guide/
- API: https://junit.org/junit5/docs/current/api/overview-summary.html
- GitHub: https://github.com/junit-team/junit5/
- Stack Overflow: http://stackoverflow.com/questions/tagged/junit5
- Gitter: https://gitter.im/junit-team/junit5

JUnit 5

JUnit 5 = JUnit **Platform** + JUnit ***Jupiter*** + JUnit **Vintage**

Platform: Foundation

Jupiter: Programming and Extension Model

Vintage: Test Engine for JUnit 3 and 4 tests

Why Jupiter?

Starts with JU

5th planet from sun

Setup

- Use Gradle or Maven
  - Standalone console launcher available (see docs)
  - Ant support available

Gradle

Use Gradle 4.6+

```
test {
    useJUnitPlatform()
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.3.1'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.3.1'

    testImplementation 'org.junit.jupiter:junit-jupiter:5.4.0'
}
```

Old and busted

New Hotness

Gradle

For older JUnit 3 and 4 support, add:

```
dependencies {
    testCompileOnly 'junit:junit:4.12'
    testRuntimeOnly 'org.junit.vintage:junit-vintage-engine:5.4.0'
}
```

Can run **both** current and *vintage* in same project

Gradle

`useJUnitPlatform` block can use config options:

```
test {
    useJUnitPlatform {
        includeEngines 'junit-vintage'
        // excludeEngines 'junit-jupiter'
    }
}
```

Default is to use all available engines

Gradle

Starter project: junit5-jupiter-starter-gradle

Maven

Starter project: junit5-jupiter-starter-maven

For details, see the Maven support section of the User Guide

@Test

New package: `org.junit.jupiter.api`

Annotate test methods

No attributes -- use extensions instead

Exceptions example discussed below

Neither test methods or classes need to be public

Other test annotations (discussed later)

- @RepeatedTest
- @ParameterizedTest
- @TestFactory

Lifecycle Annotations

Each test gets `@Test`

`@BeforeEach`, `@AfterEach`

Before and after each test

`@BeforeAll`, `@AfterAll`

Before any test runs; After all tests have run

Must be static methods (by default)

Disabled tests

`@Disabled` → skip a particular test or tests

- Method level or class level
- Optional parameter to give a reason

Replaces `@Ignored` in JUnit 4

Test names

Use `@DisplayName` on class or methods

Supports Unicode and even emojis

From User Manual:

```
@Test
@DisplayName("╯°□°)╯")
void testWithDisplayNameContainingSpecialCharacters() {}

@Test
@DisplayName("😱")
void testWithDisplayNameContainingEmoji() {}
```

Assertions

Static methods in `org.junit.jupiter.api.Assertions`

Most similar to previous versions:

- assertTrue, assertFalse
- assertEquals (lots of overloads)
- assertNull, assertNotNull
- assertSame, assertNotSame
- fail

Assertions

New methods in JUnit 5

- assertAll
- assertThrows, assertDoesNotThrow
- assertTimeout
- assertTimeoutPreemptively

Handful of others not as common

Assertions

- Parameters "expected", "actual"
  - Optional string message now comes last
  - Can use `Supplier<String>` for message for lazy eval

Assertions

`assertAll` → group assertions; first failure does not skip rest

`assertThrows` → returns the exception

`assertTimeout` → Run in same thread, so can't interrupt

`assertTimeoutPreemptively` → Run in separate thread, so can interrupt

- Timeout tests show by how much time was exceeded
- Also return values that can be tested

Assertions

Note arg to assertion methods take an `Executable`

Just like Runnable, but throws Throwable

Assumptions

Let you test pre-conditions

Static methods in `org.junit.jupiter.api.Assumptions`

Failure aborts test

Methods like:

- assumeTrue(boolean)
- assumeFalse(boolean)
- assumingThat(boolean, Executable)

Conditional Execution

Can make tests or test classes conditional, based on:

- Operating system
- Java version
- Environment variables
- System properties

Conditional Execution

`@EnabledOnOs, @DisabledOnOs`

Takes constant from `org.junit.api.condition.OS` enum:

- AIX
- MAC
- LINUX
- WINDOWS
- SOLARIS
- OTHER

Conditional Execution

`@EnabledOnJre, @DisabledOnJre`

Takes constant from `org.junit.api.condition.JRE` enum:

- JAVA_8
- JAVA_9
- JAVA_10
- JAVA_11
- JAVA_12
- JAVA_13
- ...
- OTHER

Conditional Execution

- `@EnabledIfEnvironmentVariable`, `@DisabledIfEnvironmentVariable`
- `@EnabledIfSystemProperty`, `@DisabledIfSystemProperty`
  - All take strings as arguments
  - String arg can be a regular expression
    - Uses String.matches(regex) for matching

Tagging and Filtering

```
@Tag("string")
```

Can label tests or test classes

Filter using settings in Gradle, Maven, Ant

```
test {
    useJUnitPlatform {
        includeTags 'fast', 'smoke & feature-a'
        // excludeTags 'slow', 'ci'
    }
}
```

Test Instance Lifecycle

Default: new instance of test class for each test

Re-initialize attributes in between tests

```
@TestInstance(TestInstance.Lifecycle.PER_METHOD)
```

Test Instance Lifecycle

Can use one instance per class

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
```

May need to reset vars in `@BeforeEach`

`@BeforeAll`, `@AfterAll` no longer static

Can also use @BeforeAll, @AfterAll in @Nested test classes

Nested Test Classes

Use `@Nested` on non-static inner classes

Nesting can be as deep as you want

If you use @BeforeAll / @AfterAll on nested classes,

Use PER_CLASS lifecycle

Nice Stack example in User Guide

Constructor and Method Parameters

In JUnit 4, no parameters in constructors or test methods

Now, parameters can be injected automatically

- `TestInfo`
- `TestReporter`
- `RepetitionInfo`
- Other custom ParameterResolvers supplied by extensions

Constructor and Method Parameters

`TestInfo`

- getDisplayName()
- getTags()
- getTestClass() → Optional<Class<?>>
- getTestMethod() → Optional<Method>

Constructor and Method Parameters

`TestReporter` functional interface

- publishEntry(String key, String value) (default)
- publishEntry(Map<String,String> map) (single abstract method)

Constructor and Method Parameters

`RepetitionInfo`

Used in `@RepeatedTest(int)` tests

- getCurrentRepetition()
- getTotalRepetitions()

RepeatedTest

Speaking of `@RepeatedTest(int)`,

Placeholders in "name" attribute:

- currentRepetition
- totalRepetitions
- displayName

Parameterized Tests

Arguably more useful than just repeated

Run a test multiple times with different arguments

`@ParameterizedTest`

Need at least one source of parameters

NOTE: Experimental even in 5.6, but not expected to change

Add the `junit-jupiter-params` dependency
(now included by default in junit-jupiter)

Parameterized Tests

Sources:

- @ValueSource
- @EnumSource
- @MethodSource
- @CsvSource
- @CsvFileSource
- @ArgumentsSource

Parameterized Tests

`@ValueSource`

Single array providing single values

All 8 primitive types, plus String and Class

Parameterized Tests

`@EnumSource`

Provides all values of an Enum

Can filter using "mode = INCLUDE" and "mode = EXCLUDE"

Parameterized Tests

`@MethodSource`

Use a static factory method (with no args) to provide single values

- Must return Stream, Iterable, Iterator, or array

To provide multiple values, use `Arguments` interface

- to return Stream<Arguments>

Parameterized Tests

`@CsvSource`

Provide arguments as a comma-separated list of values

Use single quotes if entry contains a comma

Parameterized Tests

`@CsvFileSource`

Use CSV files in the classpath

Each line in the file invokes the test once

Dynamic Tests

Generated at runtime by a factory method

Annotated with `@TestFactory`

Method returns Stream, Collection, Iterable, Iterator of *dynamic nodes*

- `DynamicContainer`
- `DynamicTest`

NOTE: Experimental as of 5.0

Dynamic Tests

Consist of a display name and an Executable

```
static DynamicTest dynamicTest(String displayName,
                               Executable executable)
```

Also static method to generate a stream

No lifecycle callbacks for individually generated tests

See details in the User Guide

Extensions

Add `@ExtendWith` to test class

```
@ExtendWith(SpringExtension.class)
```

Third-Party Extensions

- Spring
- Mockito
- Docker
- JPA
- Selenium/WebDriver
- Kafka
- Jersey
- See (https://github.com/junit-team/junit5/wiki/Third-party-Extensions)

Mockito

Mocks, Stubs, and Spies

Consider an order processor

```
public Confirmation processOrder() {

    // calculate total cost
    // look up customer info
    // process any discounts
    // add required taxes and fees
    // add in shipping costs
    // process credit card

}
```

Order processor

Most of that is local, but what about the credit card processor?

```
public Confirmation processOrder() {

    // …
    cardService.chargeCard(String number);
}
```

Don't want to call the real credit card service while testing

Mocks vs Stubs

What we need for the credit card processor is a

Mock object

… or is that a stub?

Mocks vs Stubs

A stub

stands in for the real object

provides specific responses to method calls

mockCardService.chargeCard("12345") → true

This is called setting expectations

Mocks vs Stubs

A mock

- stands in for the real object
- verifies that methods were called:
  - the right number of times
  - in the right order

You verify a mock

Mocks vs Stubs

```
@Test
public void testProcessOrder() {

    // verify:
    // customer lookup called first
    // total price calculator called next
    // shipping service called after that
    // credit card service called last

    // each called exactly once
}
```

Mocks vs Stubs

The created object (mock or stub) is the same

The difference is how they are used:

stubs provide known outputs for method calls

mocks verify the protocol
the interaction between our test class and the mock

Spies

A spy is a partial mock

method calls pass through to underlying real object

can replace some calls with specific outputs

Mockito makes a copy of the real instance

Mockito discourages their use, but allows it when necessary

Mockito

A mocking (and stubbing) tool

Enables true unit tests in Java

Programmatic stubbing via

- Mockito.when(mock.action()).thenReturn(true)
- BDDMockito.given(mock.action()).willReturn(true)

Documentation

- Home page: http://mockito.org/
- JavaDocs: http://javadoc.io/page/org.mockito/mockito-core/latest/org/mockito/Mockito.html
- Release Notes: https://github.com/mockito/mockito/blob/release/2.x/doc/release-notes/official.md
- FAQ: https://github.com/mockito/mockito/wiki/FAQ
- Mailing list: http://groups.google.com/group/mockito

Mockito

Current version is 3.4+

Mockito

Limitations (some by design):

- Cannot mock static methods (until 3.4.0)
- Cannot mock constructors
- Cannot mock equals(), hashCode()
- Cannot mock private methods

Capabilities:

- Can mock both classes and interfaces

Mockito

Gradle dependency:

```
repositories {
    jcenter()
}

dependencies {
    testImplementation 'org.mockito:mockito-core:3.4.3'
}
```

Using Mockito

Create mocks with:

- static mock() method
- @Mock annotation

Mockito

Programmatic verification via

- Mockito.verify(mock).action()
- BDDMockito.then(mock).should().action()

Annotations available for mocking

- @Mock
- @Spy
- @Captor
- @InjectMocks

Mockito

Provides its own JUnit runner:

org.mockito.MockitoJUnitRunner

Using Annotations

Add @Mock (or @Spy) to attributes

Be sure to call MockitoAnnotations.initMocks(this)

Or use JUnit Rule

```
@Rule
public MockitoRule mockitoRule = MockitoJUnit.rule();
```

Or use Mockito JUnit Runner

```
@RunWith(MockitoJUnitRunner.class)
```

Configuring Mocks

Mocked objects return default values if not specified

- null for object references
- zero for numbers
- false for booleans
- empty collections for collections
- etc.

Configuring Mocks

Setting expectations

After mock(MyClass.class),

when(...).thenReturn(...)

Can chain thenReturn(...) calls

Returns in order, then the final one repeatedly

when(...).thenReturn(...).thenReturn(...)

Configuring Mocks

Configure mock based on specific argument

when(42).thenReturn(true)

Can use ArgumentMatchers

anyInt(), anyBoolean(), anyString(), …

any(), any(Class<T>)

Verifying Invocations

verify(mock).method() checks that method() is called on mock

verify(mock, times(1)).method()

- times(int)
- never()
- atLeastOnce()
- atLeast(int)
- atMost(int)

Configuring Mocks

Can not mock methods that return void the same way

Can not mock methods that throw exceptions the same way

Use the "doReturn" methods instead

```
doReturn(...).when(...).action()
```

```
doThrow(new RuntimeException()).when(...).action()
```

Ordering

Can verify methods invoked in the proper order

Use InOrder class, which takes a mock as arg

Spies

Wraps a real object

Every call is delegated to the object unless specified otherwise

Use the spy() method or @Spy

Verifying Behavior

- Mockito keeps track of all method calls and parameters on real object
- Use verify() on mock
- Distinguishes between true mocks and true stubs
  - Stubs just return specified values
  - Mocks verify the protocol
    - Methods are called the right number of times, in proper order

Verifying Behavior

Use ArgumentMatchers to check argument types

- static methods like eq(), anyInt(), any()

Check multiplicity

- never(), atLeastOnce(), atLeast(num)
- times(...), atMost(...)

ArgumentCaptor

ArgumentCaptor allows access to arguments of method calls

verify(mockedList).addAll(captor.capture())

Then captor.getValue() returns actual value

BDD

Behavior Driven Development

Use BDDMockito class

given(mock.method()).willReturn(value)

then(mock).should(times(1)).method()

Mocking final types

Incubating capability to mock:

- final types
- enums
- final methods

Use Mockito extension mechanism

/mockito-extensions/org.mockito.plugins.MockMaker

containing "mock-maker-inline"

Mocking final types

Alternatively, use "mockito-inline" artifact in build

Still restrictions; see docs for details

JUnit 5 Extension

Use "org.mockito:mockito-junit-jupiter" artifact

```
@ExtendWith(MockitoExtension.class)
```

Summary

Source code:

https://github.com/kousen/junit5_workshop
https://github.com/kousen/mockito-hamcrest

- New packages
- New assertions, with lazy evaluation
- New names for lifecycle methods
- Conditional tests
- Nested tests
- Repeated and parameterized tests
- Dynamic tests
- Extensions