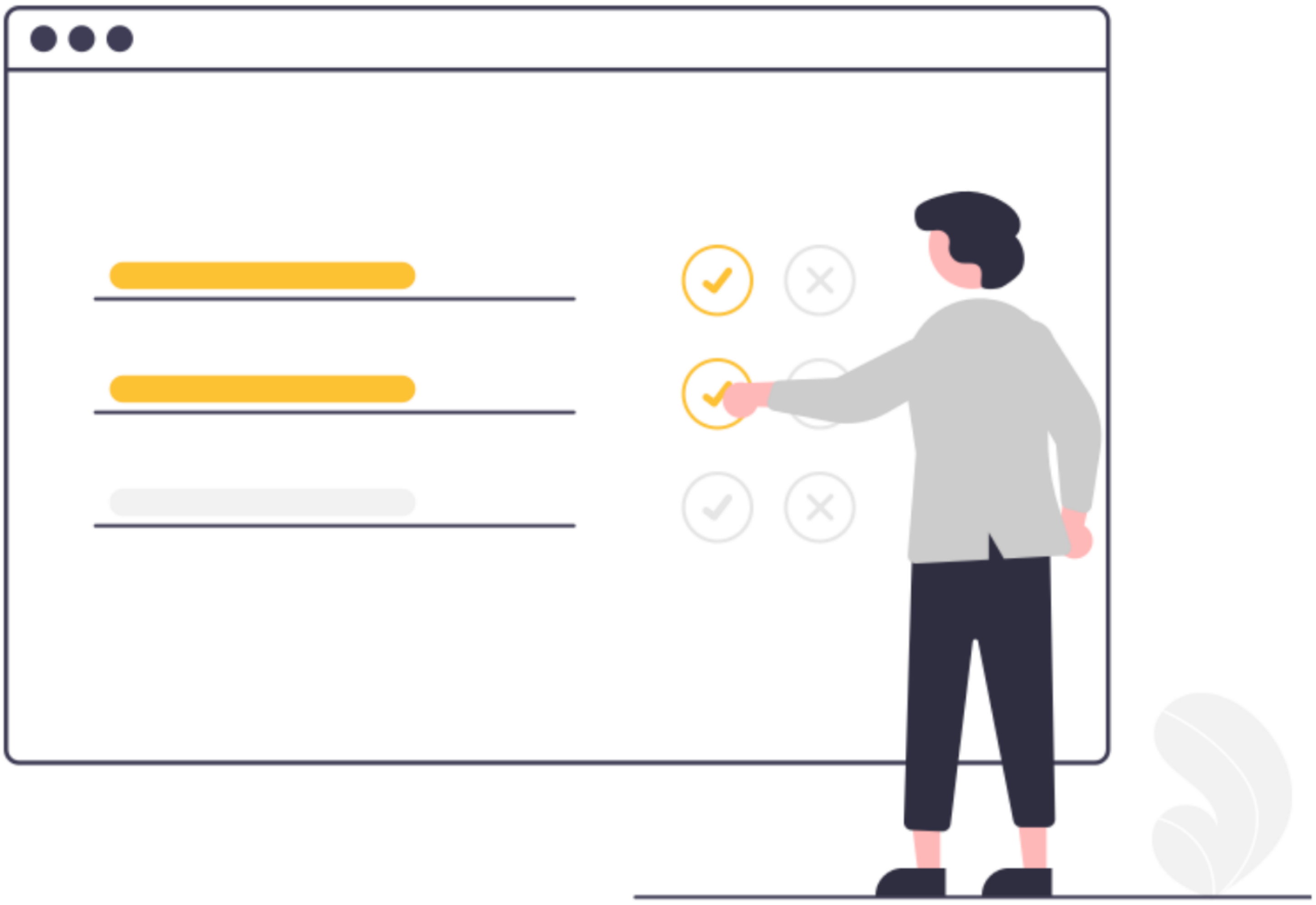


JOYFUL TESTING JAVA APPLICATIONS

JUNIT 5 AND MOCKITO CHEAT SHEET

ANSWERING 24 QUESTIONS FOR THE TWO MOST ESSENTIAL JAVA TESTING LIBRARIES

JUnit 5 and Mockito Cheat Sheet

Philip Riecks (rieckpil)
Version: 1.0

Table of Contents

Introduction to this Cheat Sheet

This Cheat Sheet follows a question & answer style to provide quickstart solutions for JUnit 5 and Mockito.

The following code examples can be copied and pasted to your project as-is. You'll find instructions for both Maven & Gradle.

While this document does not cover all your questions around JUnit 5 and Mockito, it follows the Pareto principle and targets the 20% of the features that you use 80% of the time.

Looking for more content and deep dive into Testing Java Applications?

- Sign up for the Testing Spring Boot Application Masterclass
- Visit my YouTube channel
- Find more Testing articles & guides on my blog
- Join the FREE 14 Days Testing Java Applications Course

Let's get started!

Quickstart solutions for JUnit 5

What is the difference between JUnit 4 and JUnit 5

Before we start with the basics, let’s have a short look at the history of JUnit.

For a long time JUnit 4.12 was the main framework version. In 2017 JUnit 5 was launched and is now composed of several modules:

> JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

- JUnit Platform: Foundation to launch testing frameworks & it defines the `TestEngine` API for developing testing frameworks
- JUnit Jupiter: New programming model and extension model and provides the `JupiterTestEngine` that implements the `TestEngine` inteface to run tests on the JUnit Platform
- JUnit Vintage: Provides a `TestEngine` to run both JUnit 3 and JUnit 4 tests

The JUnit team invested a lot in this refactoring to now have a more **platform-based** approach with a **comprehensive extension model**.

Nevertheless, migrating from JUnit 4.X to 5.X requires effort. All annotations, like `@Test`, now reside in the package `org.junit.jupiter.api` and some annotations were renamed or dropped and have to be replaced.

More about the migration from JUnit 4 to JUnit 5 in one of the next sections.

How can I include JUnit 5 to my project?

> Throughout this cheat sheet I’ll use the term JUnit 5 to refer to JUnit Jupiter (which is formally incorrect).

As a short recap, JUnit 5 is composed of several modules: JUnit Platform + JUnit Jupiter + JUnit Vintage.

If you only use JUnit Jupiter for your tests and have no JUnit 3 or JUnit 4 *left-overs*, your Maven project requires the following dependency:

```xml
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.7.0</version>
  <scope>test</scope>
</dependency>
```

For Gradle:

```
dependencies {
  testImplementation('org.junit.jupiter:junit-jupiter:5.7.0')
}
```

For those of you that still have old tests written in JUnit 4, make sure to include the following dependencies:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.7.0</version>
  <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
</dependency>
```

For Gradle

```
dependencies {
  testImplementation('org.junit.jupiter:junit-jupiter:5.7.0')
  testImplementation('org.junit.vintage:junit-vintage-engine:5.7.0')
}
```

If your project uses Spring Boot, the Spring Boot Starter Test dependency includes everything you need. You only have to decided whether or not you want to include the Vintage Engine.

Including the Vintage Engine:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

Excluding the Vintage Engine:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>org.junit.vintage</groupId>
      <artifactId>junit-vintage-engine</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Take a look at the different JUnit 5 project setup examples on GitHub.

How to migrate from JUnit 4 to JUnit 5?

Migration from JUnit 4 to JUnit 5 is not *just* a dependency version bump. The lifecycle annotations have be renmaned, the extension model now superseeds JUnit 4's rules, etc.

A short overview of the differences between both framework versions is the following:

- Assertions reside in `org.junit.jupiter.api.Assertions`
- Assumptions reside in `org.junit.jupiter.api.Assumptions`
- `@Before` and `@After` no longer exist; use `@BeforeEach` and `@AfterEach` instead.
- `@BeforeClass` and `@AfterClass` no longer exist; use `@BeforeAll` and `@AfterAll` instead.
- `@Ignore` no longer exists: use `@Disabled` or one of the other built-in execution conditions instead
- `@Category` no longer exists; use `@Tag` instead
- `@RunWith` no longer exists; superseded by `@ExtendWith`
- `@Rule` and `@ClassRule` no longer exist; superseded by `@ExtendWith` and `@RegisterExtension`

If your codebase is using JUnit 4, changing the annotations to the JUnit 5 ones is the first step. The most effort is required for migrating custom JUnit 4 rules to JUnit 5 extensions.

You can automate the trivial parts of the migration, with e.g. Sam Brannen's script. The JUnit documentation also includes an own section with hints & pitfalls for the migration

How to write your first test?

Writing your first test with JUnit 5, is as simple as the following:

```java
import org.junit.jupiter.api.Test;

class FirstTest {

  @Test // ①
  void firstTest() {
    System.out.println("Running my first test with JUnit 5");
  }

}
```

① Here starts your testing journey

All you need is to annotate a non-private method (the Java class can be package-private) with `@Test`.

Your test then usually reside inside `src/test/java` to follow default conventions.

Running `mvn test` or `gradle test` will then execute your test:

```
[INFO] --- maven-surefire-plugin:2.22.2:test (default-test) @ junit-5-and-mockito-cheat-sheet ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running de.rieckpil.products.junit5.FirstTest
Running my first test with JUnit 5
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.02 s - in
de.rieckpil.products.junit5.FirstTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.723 s
[INFO] Finished at: 2020-10-18T11:20:00+02:00
[INFO] ------------------------------------------------------------------------
```

How can I intercept the lifecycle of my test?

JUnit 5 offers several interception points to execute code within the test's lifecycle.

```
class LifecycleTest {

  public LifecycleTest() {
    System.out.println("-- Constructor called");
  }

  @BeforeAll
  static void beforeAllTests() {
    System.out.println("- Before all tests");
  }

  @BeforeEach
  void beforeEachTest() {
    System.out.println("--- Before each test");
  }

  @AfterAll
  static void afterAllTests() {
    System.out.println("- After all tests");
  }

  @AfterEach
  void afterEachTest() {
    System.out.println("--- After each test");
  }

  @Test
  void shouldAddIntegers() {
    System.out.println("Test: Adding integers with Java");
    assertEquals(4, 2 + 2);
  }

  @Test
  void shouldSubtractIntegers() {
    System.out.println("Test: Subtracting integers with Java");
    assertEquals(4, 6 - 2);
  }
}
```

```
- Before all tests
-- Constructor called

--- Before each test
Test: Subtracting integers with Java
--- After each test

-- Constructor called

--- Before each test
Test: Adding integers with Java
--- After each test

- After all tests
```

How can I change the name and structure of my test?

As the name of your test should be explicit and reflect the tested scenario, the Java method name might grow. To make it more readable, you can use the camel case notation `shouldThrowExceptionWhenDividingByZero` or underscores `_`.

However, if you want to write a more readable sentence, you can use `@DisplayName` for your test:

```
class DisplayNameTest {

  @Test
  @DisplayName("This test verifies that Java can add integers")
  void shouldAddIntegers() {
    System.out.println("Test: Adding integers with Java");
    assertEquals(4, 2 + 2);
  }

  @Test
  @DisplayName("This test verifies that Java can subtract integers")
  void shouldSubtractIntegers() {
    System.out.println("Test: Subtracting integers with Java");
    assertEquals(4, 6 - 2);
  }
}
```

If you project uses Kotlin, you can achieve the same with:

```
@Test
fun `should throw an exception when dividing by zero`() {
  //
}
```

Once your test suite grows, you might want to structure them. You can place tests that verify a similar outcome together.

If you want to structure tests within a test class, you can use the `@Nested` annotation:

```
class NestedTest {

  @Nested
  class Addition {

    @Test
    void shouldAddIntegers() {
      System.out.println("Test: Adding integers with Java");
      assertEquals(4, 2 + 2);
    }
  }

  @Nested
  class Subtraction {

    @Test
    void shouldSubtractIntegers() {
      System.out.println("Test: Subtracting integers with Java");
      assertEquals(4, 6 - 2);
    }
  }
}
```

or you can tag them (either the whole test class or specific tests) with `@Tag`:

```
class TaggingTest {

  @Test
  @Tag("slow")
  void shouldAddIntegers() {
    System.out.println("Test: Adding integers with Java");
    assertEquals(4, 2 + 2);
  }

  @Test
  @Tag("fast")
  void shouldSubtractIntegers() {
    System.out.println("Test: Subtracting integers with Java");
    assertEquals(4, 6 - 2);
  }
}
```

How to provide parameterized inputs?

Sometimes you want to execute the same test for different input parameters. To avoid duplicating the same test multiple times, you can make use of parameterized tests.

JUnit 5 allows the following sources to feed input for a parameterized test:

- inline values
- method sources
- enums
- CSV files

```
class ParameterizedExampleTest {

  @ParameterizedTest
  @ValueSource(ints = {2, 4, 6, 8, 10})
  void shouldAddIntegersValueSource(Integer input) {
    assertEquals(input * 2, input + input);
  }

  @ParameterizedTest
  @MethodSource("integerProvider")
  void shouldAddIntegersMethodSource(Integer input) {
    assertEquals(input * 2, input + input);
  }

  static Stream<Integer> integerProvider() {
    return Stream.of(2, 4, 6, 8, 10);
  }

  @ParameterizedTest
  @CsvFileSource(resources = "/integers.csv")
  void shouldAddIntegersCsvSource(Integer firstOperand,
                                  Integer secondOperand,
                                  Integer expectedResult) {
    assertEquals(expectedResult, firstOperand + secondOperand);
  }
}
```

How to write assertions with JUnit 5?

JUnit 5 also ships with assertions. There is no need to include AssertJ or Hamcrest except you favour their assertion style over JUnit 5's.

```
import org.junit.jupiter.api.Test;

import static org.junit.Assert.*;

class AssertionExampleTest {

  @Test
  void demoJUnit5Assertions() {

    assertEquals(4, 2 + 2);
    assertNotEquals(4, 2 + 1);

    assertFalse("Optional message", 2 == 3);
    assertTrue("Optional message", 2 == 1 + 1);

    assertArrayEquals(new int[]{1, 2, 3}, new int[]{1, 2, 3});

    String message = null;
    assertNull(message);
  }
}
```

How can I verify that my code throws an exception?

```
class AssertThrowsExampleTest {

  @Test
  void invocationShouldThrowException() {
    assertThrows(IllegalArgumentException.class, () -> {
      divide(5, 0);
    });
  }

  private Integer divide(int first, int second) {
    if (second == 0) {
      throw new IllegalArgumentException("Can't divide by zero");
    }

    return first / second;
  }
}
```

How can I parallelize my test?

While you might have configured this in the past with the corresponding Maven or Gradle plugin, you can now configure this as an experimental feature with JUnit (since version 5.3).

This allows more fine-grain control on how to parallelize the tests.

A basic configuration can look like the following:

```
junit.jupiter.execution.parallel.enabled = true
junit.jupiter.execution.parallel.mode.default = concurrent
```

This enables parallel execution for all your tests and set the execution mode to concurrent. Compared to `same_thread`, `concurrent` does not enforce to execute the test in the same thread of the parent. For a per test class or method mode configuration, you can use the @Execution annotation.

There are multiple ways to set these configuration values, one is to use the Maven Surefire plugin for it:

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.2</version>
  <configuration>
    <properties>
      <configurationParameters>
        junit.jupiter.execution.parallel.enabled = true
        junit.jupiter.execution.parallel.mode.default = concurrent
      </configurationParameters>
    </properties>
  </configuration>
</plugin>
```

or a `junit-platform.properties` file inside `src/test/resources` with the configuration values as content.

You should benefit the most when using this feature for unit tests. Enabling parallelization for integration tests might not be possible or easy to achieve depending on your setup.

Therefore, I recommend executing your unit tests with the Maven Surefire Plugin and configure parallelization for them. All your integration tests can then be executed with the Maven Failsafe Plugin where you don't specify these JUnit 5 configuration parameters.

For a more fine-grain parallelism configuration, take a look at the official JUnit 5 documentation.

How can I write my own extension?

The extension model of JUnit 5 allows you to create your own cross-cutting concerns and write callbacks, parameter resolver etc.

If you have used Spring Boot or Mockito in the past, you probably already saw `@ExtendWith` on top of your test class, e.g. `@ExtendWith(SpringExtension.class)`.

This is the declarative approach to register an extension. You also use the programmatic approach using `@RegisterExtension` or use Java's `ServiceLoader` to automatically register them.

You can write extensions for the following use cases:

- BeforeAllCallback
- BeforeEachCallback
- BeforeTestExecutionCallback
- AfterTestExecutionCallback
- AfterEachCallback
- AfterAllCallback
- ParameterResolver
- TestExecutionExceptionHandler
- InvocationInterceptor
- etc.

As an example, injecting Spring Beans (using `@Autowired`) during your test is achieved with a `ParameterResolver`.

Next, whenever you use the `@Testcontainers` annotation for your test, Testcontainers registers an extension in the background to start and stop your `@Container` accordingly.

Let's write our first extension that resolves a random `UUID`.

```
public class RandomUUIDParameterResolver implements ParameterResolver {

  @Retention(RetentionPolicy.RUNTIME)
  @Target(ElementType.PARAMETER)
  public @interface RandomUUID {
  }

  @Override
  public boolean supportsParameter(ParameterContext parameterContext, ExtensionContext extensionContext) {
    return parameterContext.isAnnotated(RandomUUID.class);
  }

  @Override
  public Object resolveParameter(ParameterContext parameterContext, ExtensionContext extensionContext) {
    return UUID.randomUUID().toString();
  }
}
```

This extension class includes a custom annotation `@RandomUUID`. Whenever we now use this annotation next to a `UUID` and `RandomUUIDParameterResolver` extension is registered for this test, we'll get a random `UUID`.

Let's use this parameter resolver and register it for one of our tests:

```
import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.extension.ExtendWith;

@ExtendWith(RandomUUIDParameterResolver.class)
public class ParameterInjectionTest {

  @RepeatedTest(5)
  public void testUUIDInjection(@RandomUUIDParameterResolver.RandomUUID String uuid) {
    System.out.println("Random UUID: " + uuid);
  }
}
```

What other resources do you recommend regarding JUnit 5?

Find further resources on JUnit 5 and unit testing in general here:

- Excellent User Guide
- Testing Spring Boot Applications Fundamentals
- Java Unit Testing with JUnit 5: Test Driven Development with JUnit 5
- Good Test, Bad Tests
- Modern Best Practices for Testing in Java
- Starting to Unit Test: Not as Hard as You Think

Quickstart solutions for Mockito

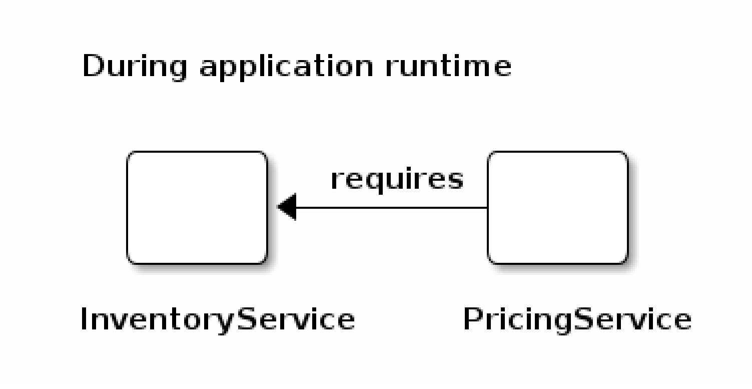
What do I need Mockito for?

Mockito is a …

> Tasty mocking framework for unit tests in Java

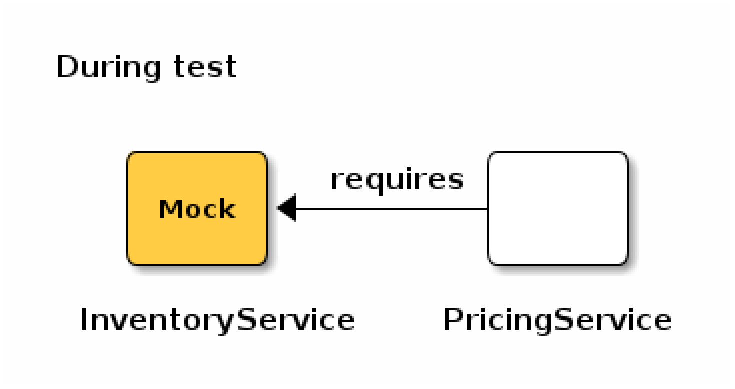
The main reason to use Mockito is to **stub methods calls** and **verify interaction** of objects. The first is important if you write unit tests, and your test class requires other objects to work (so called collaborators). As your unit test should focus on just testing your class under test, you mock the behaviour of the dependent objects of this class.

A visual overview might explain this even better. Let's take the following example. Our application has a `PricingService` that makes use of public methods of the `InventoryService`.



When we now want to write a **unit test** for the `PricingService` we don't want to interact with the *real* `InventoryService`, but rather a mock of it.

This way we can control the behaviour of the collaborator and test different scenarios:



Code-wise this example looks like the following:

```java
public class PricingService {

  private final InventoryService inventoryService;

  public PricingService(InventoryService inventoryService) {
    this.inventoryService = inventoryService;
  }

  public BigDecimal calculatePrice(String productName) {
    if (inventoryService.isCurrentlyInStockOfCompetitor(productName)) {
      return new BigDecimal("99.99");
    }

    return new BigDecimal("149.99");
  }
}
```

```java
public class InventoryService {
  public boolean isCurrentlyInStockOfCompetitor(String productName) {
    // Determine if product is in stock of competitor, e.g. HTTP call
    return false;
  }

  public boolean isAvailable(String productName) {
    // Determine if product is available, e.g. database access
    return true;
  }
}
```

We'll use this example for most of the following sections.

How to include Mockito to my project?

If your project uses Spring Boot, the Spring Boot Starter Test dependency already includes Mockito.

Otherwise, you need to include the following dependencies (for Maven):

```xml
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>3.5.13</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-junit-jupiter</artifactId>
  <version>3.5.13</version>
  <scope>test</scope>
</dependency>
```

The corresponding Gradle import looks like the following:

```groovy
dependencies {
  testImplementation('org.mockito:mockito-core:3.5.13')
  testImplementation('org.mockito:mockito-junit-jupiter:3.5.13')
}
```

How can I create mocks?

You can't create mocks from final classes and methods with Mockito. Everything else is *mockable*. Be aware of this if you are using Kotlin, as you have to declare your classes as open.

If we now want to create a mock for the `InventoryService` while unit testing the `PricingService`, we can do the following:

```java
import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.mock;

class PricingServiceTest {

  private InventoryService inventoryService = mock(InventoryService.class); // ①
  private PricingService pricingService = new PricingService(inventorySerivce);

  @Test
  void shouldReturnHigherPriceIfProductIsInStockOfCompetitor() {
    // Test comes here
  }
}
```

① Using Mockito's static `mock()` method to create a mocked instance of the `inventoryService`

or …

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

class PricingServiceTest {

  @Mock
  private InventoryService inventoryService;

  private PricingService pricingService;

  @BeforeEach
  void setup() {
    MockitoAnnotations.openMocks(this); // ①
    this.pricingService = new PricingService(inventoryService);
  }

  @Test
  void shouldReturnHigherPriceIfProductIsInStockOfCompetitor() {
    // Test comes here
  }
}
```

① Initializing all `@Mock` annotated fields within this test class

There is a third way to create mocks that I prefer for my tests. More about this approach on the next page.

How can I create mocks with the JUnit 5 extension?

The `mockito-junit-jupiter` comes with a JUnit 5 extension that we can use to initialize our mocks and stubs `MockitoExtension`:

```java
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class) // ①
class PricingServiceTest {

  @Mock // ②
  private InventoryService inventoryService;

  @InjectMocks // ③
  private PricingService pricingService;

  @Test
  void shouldReturnHigherPriceIfProductIsInStockOfCompetitor() {
    // Test comes here
  }
}
```

① Register the `MockitoExtension` for this test

② Mark the `inventoryService` to be mocked

③ Instruct Mockito to inject all mocks while instantiating the `PricingService`

How can I mock the response of a mocked class?

You can stub the response of a mock using Mockito's stubbing setup `when().thenReturn()`:

```java
@Test
void shouldReturnHigherPriceIfProductIsInStockOfCompetitor() {
  when(inventoryService.isCurrentlyInStockOfCompetitor("MacBook")).thenReturn(false);

  BigDecimal result = pricingService.calculatePrice("MacBook");

  assertEquals(new BigDecimal("149.99"), result);
}
```

For this stubbing to work **your setup has to match the exact method arguments** your stub is called with during test execution.

This might be sometimes hard to achieve, and you can use the following workaround:

```
@Test
void mockWithAnyInputParameter() {
  when(inventoryService.isCurrentlyInStockOfCompetitor(anyString())).thenReturn(true);

  BigDecimal result = pricingService.calculatePrice("MacBook");

  assertEquals(new BigDecimal("99.99"), result);
}
```

Or use `any(JavaClass.class)` to match all passed objects of a Java class.

```
@Test
void mockWithAnyClassInputParameter() {
  when(inventoryService.isCurrentlyInStockOfCompetitor(any(String.class))).thenReturn(true);

  BigDecimal result = pricingService.calculatePrice("MacBook");

  assertEquals(new BigDecimal("99.99"), result);
}
```

Apart from returning a value, you can also instruct your mock to throw an exception:

```
@Test
void shouldThrowExceptionWhenCheckingForAvailability() {
  when(inventoryService.isCurrentlyInStockOfCompetitor("MacBook"))
    .thenThrow(new RuntimeException("Network error"));

  assertThrows(RuntimeException.class, () -> pricingService.calculatePrice("MacBook"));
}
```

If you want to return a value based on the method argument of your mock, `thenAnswer` fits perfectly:

```
@Test
void mockWithThenAnswer() {
  when(inventoryService.isCurrentlyInStockOfCompetitor(any(String.class))).thenAnswer(invocation -> {
    String productName = invocation.getArgument(0);
    return productName.contains("Mac");
  });

  BigDecimal result = pricingService.calculatePrice("MacBook");

  assertEquals(new BigDecimal("99.99"), result);
}
```

I've a recorded a video that explains where `thenAnswer` can reduce your boilerplate setup.

What happens when I don’t stub a method invocation?

Whenever you don’t stub the method invocation, your mock will return the default values for primitive types (e.g. `false` for `boolean`).

If the method of your mocked class does not return a primitive type (e.g. `int`, `boolean`, etc.) it returns `null`, e.g. `List<User>`, `Order`, `String`, etc.

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

  @Mock
  private InventoryService inventoryService;

  @InjectMocks
  private PricingService pricingService;

  @Test
  void shouldReturnHigherPriceIfProductIsInStockOfCompetitor() {
    // the inventoryService will return false
   BigDecimal result = pricingService.calculatePrice("MacBook");
  }
}
```

How can I verify the mock was invoked during test execution?

Sometimes it’s helpful to verify that a mock was called. This might be the case if you have a collaborator that return `void` but you want to ensure the interaction with this mock.

On the other side, you might also want to test that no interaction happened with a mock.

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

  @Mock
  private InventoryService inventoryService;

  @InjectMocks
  private PricingService pricingService;

  @Test
  void shouldReturnHigherPriceIfProductIsInStockOfCompetitor() {
    when(inventoryService.isCurrentlyInStockOfCompetitor(anyString())).thenReturn(false);

    BigDecimal result = pricingService.calculatePrice("MacBook");

    verify(inventoryService).isCurrentlyInStockOfCompetitor("MacBook");
    // verifyNoInteractions(otherMock);

    assertEquals(new BigDecimal("149.99"), result);
  }

}
```

How can I capture the argument my mock was called with?

With the help of an `ArgumentCaptor` you can capture the method arguments of your mocked instance.

```java
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import java.math.BigDecimal;

import static org.junit.Assert.assertEquals;
import static org.mockito.ArgumentMatchers.anyString;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

@ExtendWith(MockitoExtension.class)
class PricingServiceArgumentCaptureTest {

  @Mock
  private InventoryService inventoryService;

  @InjectMocks
  private PricingService pricingService;

  @Captor
  private ArgumentCaptor<String> stringArgumentCaptor;

  @Test
  void shouldReturnHigherPriceIfProductIsInStockOfCompetitor() {
    when(inventoryService.isCurrentlyInStockOfCompetitor(anyString())).thenReturn(false);

    BigDecimal result = pricingService.calculatePrice("MacBook");

    verify(inventoryService).isCurrentlyInStockOfCompetitor(stringArgumentCaptor.capture());

    assertEquals(new BigDecimal("149.99"), result);
    assertEquals("MacBook", stringArgumentCaptor.getValue());
  }

}
```

After capturing the arguments, you can then inspect them and write assertions for them.

How can I change the Mockito settings?

Recently Mockito became more strict when it comes to stubbing methods that are not used.

If you stub a method during your test setup that is not used during your test execution, Mockito will fail the test:

```
org.mockito.exceptions.misusing.UnnecessaryStubbingException:
Unnecessary stubbings detected.
Clean & maintainable test code requires zero unnecessary code.
Following stubbings are unnecessary (click to navigate to relevant line of code):
1. -> at
de.rieckpil.products.mockito.PricingServiceMockitoSettingsTest.shouldReturnHigherPriceIfProductIsInStockOfCo
mpetitor(PricingServiceMockitoSettingsTest.java:27)
Please remove unnecessary stubbings or use 'lenient' strictness. More info: javadoc for
UnnecessaryStubbingException class.
```

In this example the stubbing setup `when(inventoryService.isAvailable("MacBook")).thenReturn(false);` is not used while calculating the price and hence unnecessary.

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import org.mockito.junit.jupiter.MockitoSettings;
import org.mockito.quality.Strictness;

import static org.mockito.Mockito.when;

@ExtendWith(MockitoExtension.class)
@MockitoSettings(strictness = Strictness.WARN)
class PricingServiceMockitoSettingsTest {

  @Mock
  private InventoryService inventoryService;

  @InjectMocks
  private PricingService pricingService;

  @Test
  void shouldReturnHigherPriceIfProductIsInStockOfCompetitor() {
    when(inventoryService.isAvailable("MacBook")).thenReturn(false);
    when(inventoryService.isCurrentlyInStockOfCompetitor("MacBook")).thenReturn(false);

    pricingService.calculatePrice("MacBook");
  }
}
```

You can override this strictness setting to be either lenient, produce warn logs or fail on unnecessary stubs (default):

- `Strictness.LENIENT`
- `Strictness.WARN`
- `Strictness.STRICT_STUBS`

How can I mock static methods calls?

For this to work you have to replace/add `mockito-core` with `mockito-inline`. More information in one of my blog post.

Let's take a look at the following `OrderService` that makes use of two static methods `UUID.randomUUID()` and `LocalDateTime.now()`:

```java
public class OrderService {

  public Order createOrder(String productName, Long amount, String parentOrderId) {
    Order order = new Order();

    order.setId(parentOrderId == null ? UUID.randomUUID().toString() : parentOrderId);
    order.setCreationDate(LocalDateTime.now());
    order.setAmount(amount);
    order.setProductName(productName);

    return order;
  }
}
```

Starting with version 3.4.0 can now mock these static calls with Mockito

```java
import org.junit.jupiter.api.Test;
import org.mockito.MockedStatic;
import org.mockito.Mockito;

import java.time.LocalDateTime;
import java.util.UUID;

import static org.junit.jupiter.api.Assertions.assertEquals;

class OrderServiceTest {

  private OrderService cut = new OrderService();
  private UUID defaultUuid = UUID.fromString("8d8b30e3-de52-4f1c-a71c-9905a8043dac");
  private LocalDateTime defaultLocalDateTime = LocalDateTime.of(2020, 1, 1, 12, 0);

  @Test
  void shouldIncludeRandomOrderIdWhenNoParentOrderExists() {
    try (MockedStatic<UUID> mockedUuid = Mockito.mockStatic(UUID.class)) {
      mockedUuid.when(UUID::randomUUID).thenReturn(defaultUuid);

      Order result = cut.createOrder("MacBook Pro", 2L, null);

      assertEquals("8d8b30e3-de52-4f1c-a71c-9905a8043dac", result.getId());
    }
  }

  @Test
  void shouldIncludeCurrentTimeWhenCreatingANewOrder() {
    try (MockedStatic<LocalDateTime> mockedLocalDateTime = Mockito.mockStatic(LocalDateTime.class)) {
      mockedLocalDateTime.when(LocalDateTime::now).thenReturn(defaultLocalDateTime);

      Order result = cut.createOrder("MacBook Pro", 2L, "42");

      assertEquals(defaultLocalDateTime, result.getCreationDate());
    }
  }
}
```

What is the difference between @Mock and @MockBean?

While we use `@Mock` to define a mock for our unit tests, `@MockBean` is used to replace a bean with a mock inside the Spring Context.

Whenever you launch a sliced Spring Context (e.g. `@WebMvcTest` or `@DataJpaTest`) for your test, you might want to mock collaborators of your class under test.

A good is example is testing your web layer. Therefore, you can use `@WebMvcTest` and `MockMvc` to start a mocked Servlet environment to access your controller endpoints.

However, you don't want any interaction with your *service layer* and hence mock any collaborator of your controller with `@MockBean`.

What other resources do you recommend for Mockito?

Find further resources on Mockito and testing in general here:

- Project Wiki of Mockito
- Practical Unit Testing with JUnit and Mockito
- Mockito Tips & Tricks on YouTube
- Testing Spring Boot Applications Fundamentals
- PS: I'm currently in the progress to create a dedicated course for Mockito. If you are subscriber of my Newsletter you'll be informed.