# Testing

jim stafford

Fall 2024 v2020-04-22: Built: 2024-11-19 21:33 EST

# Table of Contents

# Chapter 1. Introduction

## 1.1. Why Do We Test?

- demonstrate capability?
- verify/validate correctness?
- find bugs?
- aid design?
- more ...?

There are many great reasons to incorporate software testing into the application lifecycle. There is no time too early to start.

## 1.2. What are Test Levels?

- Unit Testing - verifies a specific area of code
- Integration Testing - any type of testing focusing on interface between components
- System Testing — tests involving the complete system
- Acceptance Testing — normally conducted as part of a contract sign-off

It would be easy to say that our focus in this lesson will be on unit and integration testing. However, there are some aspects of system and acceptance testing that are applicable as well.

## 1.3. What are some Approaches to Testing?

- Static Analysis — code reviews, syntax checkers
- Dynamic Analysis — takes place while code is running
- White-box Testing — makes use of an internal perspective
- Black-box Testing — makes use of only what the item is required to do
- Many more ...

In this lesson we will focus on dynamic analysis testing using both black-box interface contract testing and white-box implementation and collaboration testing.

## 1.4. Goals

The student will learn:

- to understand the testing frameworks bundled within Spring Boot Test Starter
- to leverage test cases and test methods to automate tests performed
- to leverage assertions to verify correctness

- to integrate mocks into test cases
- to implement unit integration tests within Spring Boot
- to express tests using Behavior-Driven Development (BDD) acceptance test keywords
- to automate the execution of tests using Maven
- to augment and/or replace components used in a unit integration test

## 1.5. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. write a test case and assertions using "Vintage" JUnit 4 constructs
2. write a test case and assertions using JUnit 5 "Jupiter" constructs
3. leverage alternate (JUnit, Hamcrest, AssertJ, etc.) assertion libraries
4. implement a mock (using Mockito) into a JUnit unit test
    a. define custom behavior for a mock
    b. capture and inspect calls made to mocks by subjects under test
5. implement BDD acceptance test keywords into Mockito & AssertJ-based tests
6. implement unit integration tests using a Spring context
7. implement (Mockito) mocks in Spring context for unit integration tests
8. augment and/or override Spring context components using `@TestConfiguration`
9. execute tests using Maven Surefire plugin

# Chapter 2. Test Constructs

At the heart of testing, we want to

- establish a subject under test
- establish a context in which to test that subject
- perform actions on the subject
- evaluate the results

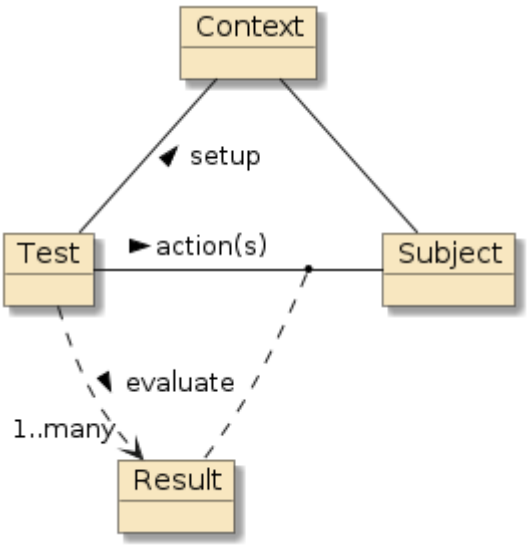


*Figure 1. Basic Test Concepts*

Subjects can vary in scope depending on the type of our test. Unit testing will have class and method-level subjects. Integration tests can span multiple classes/components — whether vertically (e.g., front-end request to database) or horizontally (e.g., peers).

## 2.1. Automated Test Terminology

Unfortunately, you will see the terms "unit" and "integration" used differently as we go through the testing topics and span tooling. There is a conceptual way of thinking of testing and a technical way of how to manage testing to be concerned with when seeing these terms used:

**Conceptual** - At a conceptual level, we simply think of unit tests dealing with one subject at a time and involve varying levels of simulation around them in order to test that subject. We conceptually think of integration tests at the point where multiple real components are brought together to form the overall set of subjects — whether that be vertical (e.g., to the database and back) or horizontal (e.g., peer interactions) in nature.

**Test Management** - At a test management level, we have to worry about what it takes to spin up and shutdown resources to conduct our testing. Build systems like Maven refer to unit tests as anything that can be performed within a single JVM and integration tests as tests that require managing external resources (e.g., start/stop web server). Maven runs these tests in different phases — executing unit tests first with the Surefire plugin and integration tests last with the Failsafe plugin.

## 2.2. Maven Test Types

Maven runs these tests in different phases — executing unit tests first with the Surefire plugin and integration tests last with the Failsafe plugin. By default, Surefire will locate unit tests starting with

"Test" or ending with "Test", "Tests", or "TestCase". Failsafe will locate integration tests starting with "IT" or ending with "IT" or "ITCase".

## 2.3. Test Naming Conventions

Neither tools like JUnit or the IDEs care how are classes are named. However, since our goal is to eventually check these tests in with our source code and run them in an automated manner — we will have to pay early attention to Maven Surefire and Failsafe naming rules while we also address the conceptual aspects of testing.

## 2.4. Lecture Test Naming Conventions

I will try to use the following terms to mean the following:

- Unit Test - conceptual unit test focused on a limited subject and will use the suffix "Test". These will generally be run without a Spring context and will be picked up by Maven Surefire.
- Unit Integration Test - conceptual integration test (vertical or horizontal) runnable within a single JVM and will use the suffix "NTest". This will be picked up by Maven Surefire and will likely involve a Spring context.
- External Integration Test - conceptual integration test (vertical or horizontal) requiring external resource management and will use the suffix "IT". This will be picked up by Maven Failsafe. These will always have Spring context(s) running in one or more JVMs. These will sometimes be termed as "Maven Integration Tests" or "Failsafe Integration Tests".

That means to not be surprised to see a conceptual integration test bringing multiple real components together to be executed during the Maven Surefire test phase if we can perform this testing without the resource management of external processes.

# Chapter 3. Spring Boot Starter Test Frameworks

We want to automate tests as much as possible and can do that with many of the Spring Boot testing options made available using the `spring-boot-starter-test` dependency. This single dependency defines transitive dependencies on several powerful, state of the art as well as legacy, testing frameworks. These dependencies are only used during builds and not in production — so we assign a scope of `test` to this dependency.

*pom.xml spring-boot-test-starter Dependency*

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope> ①
        </dependency>
```

① dependency scope is `test` since these dependencies are not required to run outside of build environment

## 3.1. Spring Boot Starter Transitive Dependencies

If we take a look at the transitive dependencies brought in by `spring-boot-test-starter`, we see a wide array of choices pre-integrated.

*spring-boot-starter-test Transitive Dependencies (reorganized)*

```
[INFO] +- org.springframework.boot:spring-boot-starter-test:jar:3.3.2:test
[INFO] |  +- org.springframework.boot:spring-boot-test:jar:3.3.2:test
[INFO] |  +- org.springframework.boot:spring-boot-test-autoconfigure:jar:3.3.2:test
[INFO] |  +- org.springframework:spring-test:jar:6.1.11:test

[INFO] |  +- org.junit.jupiter:junit-jupiter:jar:5.10.3:test
[INFO] |  |  +- org.junit.jupiter:junit-jupiter-api:jar:5.10.3:test
[INFO] |  |  +- org.junit.jupiter:junit-jupiter-params:jar:5.10.3:test
[INFO] |  |  \- org.junit.jupiter:junit-jupiter-engine:jar:5.10.3:test

[INFO] |  +- org.assertj:assertj-core:jar:3.25.3:test
[INFO] |  |  \- net.bytebuddy:byte-buddy:jar:1.14.18:test
[INFO] |  +- org.hamcrest:hamcrest:jar:2.2:test
[INFO] +- org.exparity:hamcrest-date:jar:2.0.8:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:2.2:test

[INFO] |  +- org.mockito:mockito-core:jar:5.11.0:test
[INFO] |  |  +- net.bytebuddy:byte-buddy-agent:jar:1.14.18:test
[INFO] |  |  \- org.objenesis:objenesis:jar:3.3:test
[INFO] |  +- org.mockito:mockito-junit-jupiter:jar:5.11.0:test
```

```
[INFO] |  +- com.jayway.jsonpath:json-path:jar:2.9.0:test
[INFO] |  |  \- org.slf4j:slf4j-api:jar:2.0.13:compile

[INFO] |  +- org.skyscreamer:jsonassert:jar:1.5.3:test
[INFO] |  \- org.xmlunit:xmlunit-core:jar:2.9.1:test

[INFO] \- org.junit.vintage:junit-vintage-engine:jar:5.10.3:test
[INFO]    +- org.junit.platform:junit-platform-engine:jar:1.10.3:test
[INFO]    +- junit:junit:jar:4.13.2:test
...
```

## 3.2. Transitive Dependency Test Tools

At a high level:

- `spring-boot-test-autoconfigure` - contains many auto-configuration classes that detect test conditions and configure common resources for use in a test mode
- `junit` - required to run the JUnit tests
- `hamcrest` - required to implement Hamcrest test assertions
- `assertj` - required to implement AssertJ test assertions
- `mockito` - required to implement Mockito mocks
- `jsonassert` - required to write flexible assertions for JSON data
- `jsonpath` - used to express paths within JSON structures
- `xmlunit` - required to write flexible assertions for XML data

In the rest of this lesson, I will be describing how JUnit, the assertion libraries, Mockito and Spring Boot play a significant role in unit and integration testing.

# Chapter 4. JUnit Background

JUnit is a test framework that has been around for many years (I found first commit in git from Dec 3, 2000). The test framework was originated by Kent Beck and Erich Gamma during a plane ride they shared in 1997. Its basic structure is centered around:

- **tests** that perform actions on the subjects within a given context and assert proper results
- **test cases** that group tests and wrap in a set of common setup and teardown steps
- **test suites** that provide a way of grouping certain tests

Test Suites are not as pervasive as test cases and tests

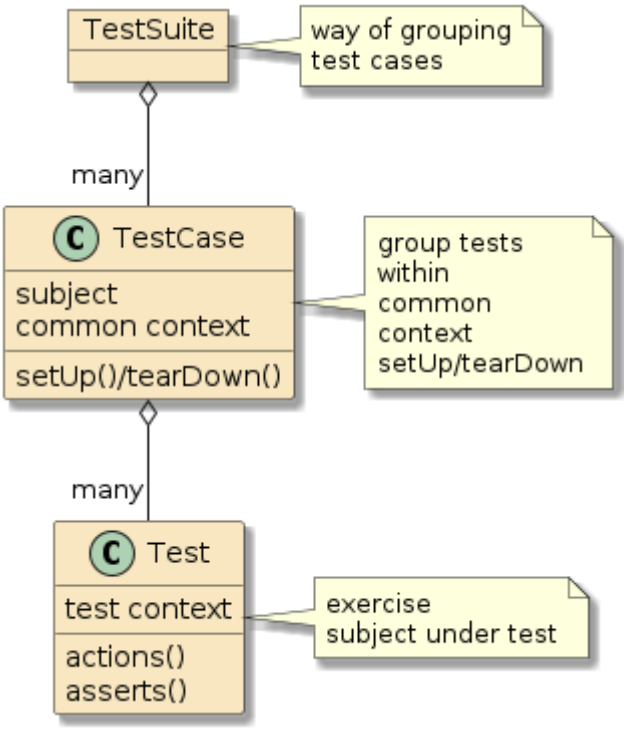


*Figure 2. Basic JUnit Test Framework Constructs*

These constructs have gone through evolutionary changes in Java — to include annotations in Java 5 and lamda functions in Java 8 — which have provided substantial API changes in Java frameworks.

- annotations added in Java 5 permitted frameworks to move away from inheritance-based approaches — with specifically named methods (JUnit 3.8) and to leverage annotations added to classes and methods (JUnit 4)

*JUnit 3.8 Test Case — Inheritance-based*

```
public class MathTest extends TestCase {
    protected void setUp() { } ①
    protected void tearDown() { } ①
    public void testAdd() { ②
       assertEquals(4, 2+2);
    }
}
```

① `setUp()` and `tearDown()` are method overrides of base class `TestCase`

② all test methods were required to start with word `test`

*JUnit 4 Test Case — Annotation-based*

```
public class MathTest {
    @Before
    public void setup() { } ①
    @After
    public void teardown() { } ①
    @Test
    public void add() { ①
       assertEquals(4, 2+2);
    }
}
```

① public methods found by annotation — no naming requirement

- lamda functions (JUnit 5/Jupiter) added in Java 8 permit the flexible expression of code blocks that can extend the behavior of provided functionality without requiring verbose subclassing

*JUnit 4/Vintage Assertions*

```
assertEquals(5,2+2);//fails here
assertEquals(3,2+2);//not eval ①
assertEquals(String.format("try%d",2),
4,2+2); ②
```

① evaluation will stop at first failure

② descriptions were first parameter and always evaluated

*JUnit 5/Jupiter Lambda Assertions*

```
assertAll(//all get eval and reported ①
    () -> assertEquals(5,2+2),
    () -> assertEquals(3,2+2),
    () -> assertEquals(4,2+2, ② ③
        ()->String.format("try%d",2))
);
```

① all assertions can be evaluated

② descriptions are now last argument

③ only evaluated if fail with lambdas

# 4.1. JUnit 5 Evolution

The success and simplicity of JUnit 4 made it hard to incorporate new features. JUnit 4 was a single module/JAR and everything that used JUnit leveraged that single jar.

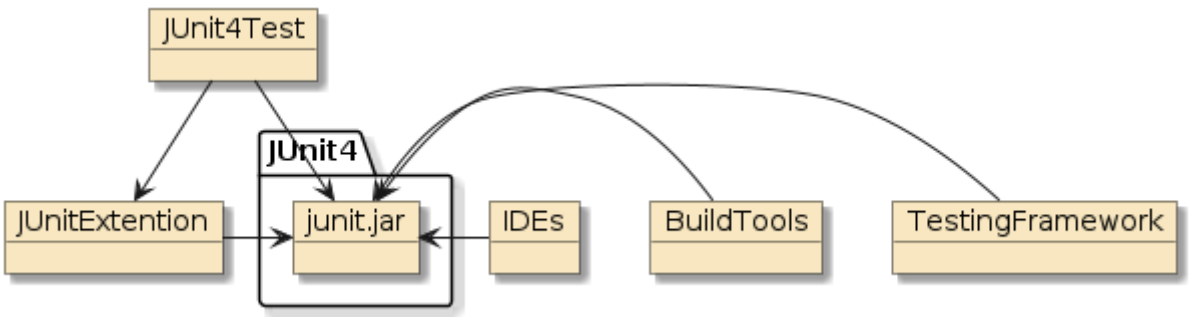


*Figure 3. Everyone uses junit.jar*

## 4.2. JUnit 5 Areas

The next iteration of JUnit involved a total rewrite — that separated the overall project into three (3) modules.

- JUnit Platform
  - foundation for launching tests on a JVM — from anything
  - defines `TestEngine` API for JUnit and 3rd party TestEngines and Extensions to use
- JUnit Jupiter ("new stuff")
  - evolution from legacy
  - provides `TestEngine` for running Jupiter-based tests
- JUnit Vintage ("legacy stuff")
  - provides `TestEngine` for running JUnit 3 and JUnit 4-based tests

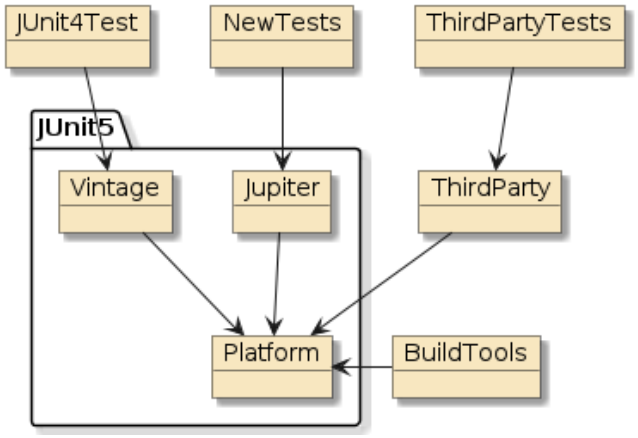


*Figure 4. JUnit 5 Modularization*

The name Jupiter was selected because it is the 5th planet from the Sun

## 4.3. JUnit 5 Module JARs

The JUnit 5 modules have several JARs within them that separate interface from implementation — ultimately decoupling the test code from the core engine.

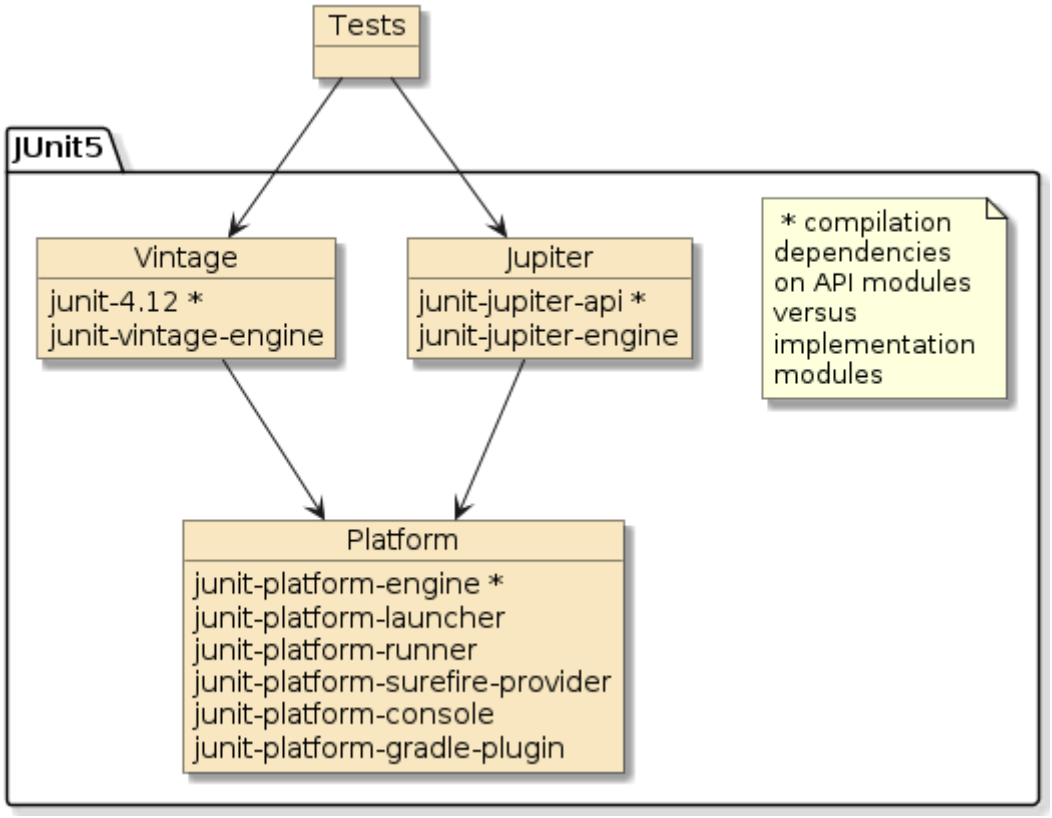


*Figure 5. JUnit 5 Module Contents*

# Chapter 5. Syntax Basics

Before getting too deep into testing, I think it is a good idea to make a very shallow pass at the technical stack we will be leveraging.

- JUnit
- Mockito
- Spring Boot

Each of the example tests that follow can be run within the IDE at the method, class, and parent java package level. The specifics of each IDE will not be addressed here but I will cover some Maven details once we have a few tests defined.

# Chapter 6. JUnit Vintage Basics

It is highly likely that projects will have JUnit 4-based tests around for a significant amount of time without good reason to update them — because we do not have to. There is full backwards-compatibility support within JUnit 5 and the specific libraries to enable that are automatically included by `spring-boot-starter-test`. The following example shows a basic JUnit example using the Vintage syntax.

## 6.1. JUnit Vintage Example Lifecycle Methods

*Basic JUnit Vintage Example Lifecycle Methods*

```
package info.ejava.examples.app.testing.testbasics.vintage;

import lombok.extern.slf4j.Slf4j;
import org.junit.*;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

@Slf4j
public class ExampleUnit4Test {
    @BeforeClass
    public static void setUpClass() {
        log.info("setUpClass");
    }
    @Before
    public void setUp() {
        log.info("setUp");
    }
    @After
    public void tearDown() {
        log.info("tearDown");
    }
    @AfterClass
    public static void tearDownClass() {
        log.info("tearDownClass");
    }
```

- annotations come from the `org.junit.*` Java package
- lifecycle annotations are
  - @BeforeClass — a public static method run before the first @Before method call and all tests within the class
  - @Before - a public instance method run before each test in the class
  - @After - a public instance method run after each test in the class
  - @AfterClass - a public static method run after all tests within the class and the last @After

method called

## 6.2. JUnit Vintage Example Test Methods

*Basic JUnit Vintage Example Test Methods*

```
@Test(expected = IllegalArgumentException.class)
public void two_plus_two() {
    log.info("2+2=4");
    assertEquals(4,2+2);
    throw new IllegalArgumentException("just demonstrating expected exception");
}
@Test
public void one_and_one() {
    log.info("1+1=2");
    assertTrue("problem with 1+1", 1+1==2);
    assertEquals(String.format("problem with %d+%d",1,1), 2, 1+1);
}
```

- @Test - a public instance method where subjects are invoked and result assertions are made
- exceptions can be asserted at overall method level — but not at a specific point in the method and exception itself cannot be inspected without switching to a manual try/catch technique
- asserts can be augmented with a String message in the first position
  - the expense of building String message is always paid whether needed or not

    ```
    assertEquals(String.format("problem with %d+%d",1,1), 2, 1+1);
    ```

Vintage requires the class and methods have public access.

## 6.3. JUnit Vintage Basic Syntax Example Output

The following example output shows the lifecycle of the setup and teardown methods combined with two test methods. Note that:

- the static @BeforeClass and @AfterClass methods are run once
- the instance @Before and @After methods are run for each test

*Basic JUnit Vintage Example Output*

```
16:35:42.293 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - setUpClass ①
16:35:42.297 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - setUp ②
16:35:42.297 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - 2+2=4
16:35:42.297 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - tearDown ②
16:35:42.299 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - setUp ②
16:35:42.300 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - 1+1=2
16:35:42.300 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - tearDown ②
16:35:42.300 INFO ...testing.testbasics.vintage.ExampleJUnit4Test - tearDownClass ①
```

① @BeforeClass and @AfterClass called once per test class

② @Before and @After executed for each @Test

Not demonstrated — a new instance of the test class is instantiated for each test. No object state is retained from test to test without the manual use of static variables.

JUnit Vintage provides no construct to dictate repeatable ordering of test methods within a class — thus making it hard to use test cases to depict lengthy, deterministically ordered scenarios.

# Chapter 7. JUnit Jupiter Basics

To simply change-over from Vintage to Jupiter syntax, there are a few minor changes.

- annotations and assertions have changed packages from `org.junit` to `org.junit.jupiter.api`
- lifecycle annotations have changed names
- assertions have changed the order of optional arguments
- exceptions can now be explicitly tested and inspected within the test method body

Vintage no longer requires classes or methods to be public. Anything non-private should work.

## 7.1. JUnit Jupiter Example Lifecycle Methods

The following example shows a basic JUnit example using the Jupiter syntax.

*Basic JUnit Jupiter Example Lifecycle Methods*

```
package info.ejava.examples.app.testing.testbasics.jupiter;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.*;

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

@Slf4j
class ExampleJUnit5Test {
    @BeforeAll
    static void setUpClass() {
        log.info("setUpClass");
    }
    @BeforeEach
    void setUp() {
        log.info("setUp");
    }
    @AfterEach
    void tearDown() {
        log.info("tearDown");
    }
    @AfterAll
    static void tearDownClass() {
        log.info("tearDownClass");
    }
```

- annotations come from the `org.junit.jupiter.*` Java package
- lifecycle annotations are

- @BeforeAll — a static method run before the first @BeforeEach method call and all tests within the class
- @BeforeEach - an instance method run before each test in the class
- @AfterEach - an instance method run after each test in the class
- @AfterAll - a static method run after all tests within the class and the last @AfterEach method called

## 7.2. JUnit Jupiter Example Test Methods

*Basic JUnit Jupiter Example Test Methods*

```
@Test
void two_plus_two() {
    log.info("2+2=4");
    assertEquals(4,2+2);
    Exception ex=assertThrows(IllegalArgumentException.class, () ->{
        throw new IllegalArgumentException("just demonstrating expected exception");
    });
    assertTrue(ex.getMessage().startsWith("just demo"));
}
@Test
void one_and_one() {
    log.info("1+1=2");
    assertTrue(1+1==2, "problem with 1+1");
    assertEquals(2, 1+1, ()->String.format("problem with %d+%d",1,1));
}
```

- @Test - a instance method where assertions are made
- exceptions can now be explicitly tested at a specific point in the test method — permitting details of the exception to also be inspected
- asserts can be augmented with a String message in the last position
  - this is a breaking change from Vintage syntax
  - the expense of building complex String messages can be deferred to a lambda function

    ```
    assertEquals(2, 1+1, ()→String.format("problem with %d+%d",1,1));
    ```

## 7.3. JUnit Jupiter Basic Syntax Example Output

The following example output shows the lifecycle of the setup/teardown methods combined with two test methods. The default logger formatting added the new lines in between tests.

*Basic JUnit Jupiter Example Output*

```
16:53:44.852 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - setUpClass ①
③
16:53:44.866 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - setUp ②
```

```
16:53:44.869 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - 2+2=4
16:53:44.874 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - tearDown ②
③

16:53:44.879 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - setUp ②
16:53:44.880 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - 1+1=2
16:53:44.881 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - tearDown ②
③
16:53:44.883 INFO ...testing.testbasics.jupiter.ExampleJUnit5Test - tearDownClass ①
```

① @BeforeAll and @AfterAll called once per test class

② @Before and @After executed for each @Test

③ The default IDE logger formatting added the new lines in between tests

Not demonstrated — we have the default option to have a new instance per test like Vintage or same instance for all tests and a defined test method order — which allows for lengthy scenario tests to be broken into increments. See @TestInstance annotation and TestInstance.Lifecycle enum for details.

# Chapter 8. JUnit Jupiter Test Case Adjustments

## 8.1. Test Instance

State used by tests can be expensive to create or outside the scope of individual tests. JUnit allows this state to be initialized and shared between test methods using one of two test instance techniques using the `@TestInstance` annotation.

### 8.1.1. Shared Static State - PER_METHOD

The default test instance is `PER_METHOD`. With this option, the instance of the class is torn down and re-instantiated between each test. We must declare any shared state as `static` to have it live during the lifecycle of all instance methods. The `@BeforeAll` and `@AfterAll` methods that initialize and tear down this data must be declared static when using `PER_METHOD`.

*TestInstance.PER_METHOD Shared State Example*

```
@TestInstance(TestInstance.Lifecycle.PER_METHOD) //the default ①
class StaticShared {
    private static int staticState; ②
    @BeforeAll
    static void init() { ③
        log.info("state={}", staticState++);
    }
    @Test
    void testA() { log.info("state={}", staticState); }  ④
    @Test
    void testB() { log.info("state={}", staticState); }
```

① test case class is instantiated per method

② any shared state must be declared private

③ `@BeforeAll` and `@AfterAll` methods must be declared static

④ `@Test` methods are normal instance methods with access to the static state

## 8.2. Shared Instance State - PER_CLASS

There are often times during an integration test where shared state (e.g., injected components) is only available once the test case is instantiated. We can make instance state sharable by using the `PER_CLASS` option. This makes the test case injectable by the container.

*TestInstance.PER_CLASS Shared State Example*

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS) ①
class InstanceShared {
    private int instanceState; ②
```

```
    @BeforeAll
    void init() { ③
        log.info("state={}", instanceState++);
    }
    @Test
    void testA() { log.info("state={}", instanceState); }
    @Test
    void testB() { log.info("state={}", instanceState); }
```

① one instance is created for all tests

② any shared state must be declared private

③ `@BeforeAll` and `@AfterAll` methods must be declared **non-static**

Use of `@ParameterizedTest` and `@MethodSource` (later topics), where the method source uses components injected into the test case instance is one example of when one needs to use `PER_CLASS`.

### 8.2.1. Test Ordering

Although it is a "best practice" to make tests independent and be executed in any order — there can be times when one wants a specified order. There are a few options: [1]

- Random Order
- Specified Order
- by Method Name
- by Display Name
- (custom order)

*Method Ordering Options*

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

@TestMethodOrder(
//        MethodOrderer.OrderAnnotation.class
//        MethodOrderer.MethodName.class
//        MethodOrderer.DisplayName.class
          MethodOrderer.Random.class
)
class ExampleJUnit5Test {
    @Test
    @Order(1)
    void two_plus_two() {
        ...
    @Test
    @Order(2)
    void one_and_one() {
```

*Explicit Method Ordering is the Exception*

It is best practice to make test cases and tests within test cases modular and independent of one another. To require a specific order violates that practice — but sometimes there are reasons to do so. One example violation is when the overall test case is broken down into test methods that addresses a multi-step scenario. In older versions of JUnit — that would have been required to be a single `@Test` calling out to helper methods.

[1] *"The Order of Tests in JUnit"*, Baeldung, May 2022

# Chapter 9. Assertion Basics

The setup methods (`@BeforeAll` and `@BeforeEach`) of the test case and early parts of the test method (`@Test`) allow for us to define a given test context and scenario for the subject of the test. Assertions are added to the evaluation portion of the test method to determine whether the subject performed correctly. The result of the assertions determine the pass/fail of the test.

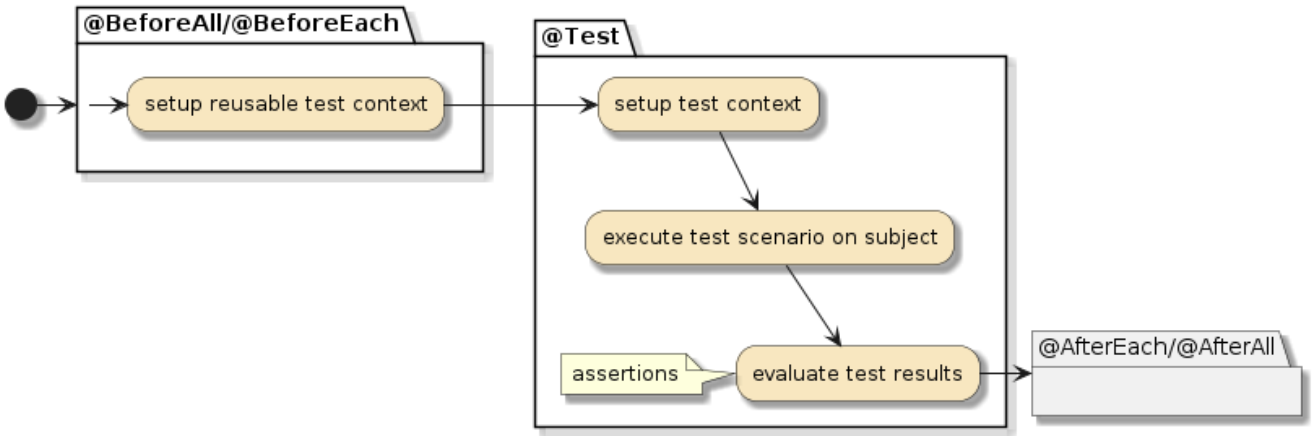


*Figure 6. Assertions are Key Point in Tests*

## 9.1. Assertion Libraries

There are three to four primary general purpose assertion libraries available for us to use within the `spring-boot-starter-test` suite before we start considering data format assertions for XML and JSON or add custom libraries of our own:

- JUnit - has built-in, basic assertions like True, False, Equals, NotEquals, etc.
  - Vintage - original assertions
  - Jupiter - same basic assertions with some new options and parameters swapped
- Hamcrest - uses natural-language expressions for matches
- AssertJ - an improvement to natural-language assertion expressions using type-based builders

The built-in JUnit assertions are functional enough to get any job done. The value in using the other libraries is their ability to express the assertion using natural-language terms without using a lot of extra, generic Java code.

### 9.1.1. JUnit Assertions

The assertions built into JUnit are basic and easy to understand — but limited in their expression. They have the basic form of taking subject argument(s) and the name of the static method is the assertion made about the arguments.

*Example JUnit Assertion*

```
import static org.junit.jupiter.api.Assertions.*;
...
```

```
        assertEquals(expected, lhs+rhs); ①
```

① JUnit static method assertions express assertion of one to two arguments

We are limited by the number of static assertion methods present and have to extend them by using code to manipulate the arguments (e.g., to be equal or true/false). However, once we get to that point — we can easily bring in robust assertion libraries. In fact, that is exactly what JUnit describes for us to do in the JUnit User Guide.

### 9.1.2. Hamcrest Assertions

Hamcrest has a common pattern of taking a subject argument and a `Matcher` argument.

*Example Hamcrest Assertion*

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.equalTo;

...
        assertThat(beaver.getFirstName(), equalTo("Jerry")); ①
```

① LHS argument is value being tested, RHS `equalTo` returns an object implementing `Matcher` interface

The `Matcher` interface can be implemented by an unlimited number of expressions to implement the details of the assertion.

### 9.1.3. AssertJ Assertions

AssertJ uses a builder pattern that starts with the subject and then offers a nested number of assertion builders that are based on the previous node type.

*Example AssertJ Assertion*

```
import static org.assertj.core.api.Assertions.assertThat;

...
        assertThat(beaver.getFirstName()).isEqualTo("Jerry"); ①
```

① `assertThat` is a builder of assertion factories and `isEqual` executes an assertion in chain

#### Custom AssertJ Assertions

Custom extensions are accomplished by creating a new builder factory at the start of the call tree. See the following link for a small example.

AssertJ also provides an Assertion Generator that generates assertion source code based on specific POJO classes and templates we can override using a maven or gradle plugin. This allows us to express assertions about a `Person` class using the following syntax.

*AssertJ Custom Assertion for Person*

```
import static info.ejava.examples.app.testing.testbasics.Assertions.*;
...
        assertThat(beaver).hasFirstName("Jerry");
```

IDEs have an easier time suggesting assertion builders with AssertJ because everything is a method call on the previous type. IDEs have a harder time suggesting Hamcrest matchers because there is very little to base the context on.

**AssertJ Generator and Jakarta**

Even though the Assertj core library has kept up to date, the assertions generator plugin has not. Current default execution of the plugin results in classes annotated with a `javax.annotation.Generated` annotation that has since been changed to `jakarta`.

I won't go into the details here, but the class example in gitlab shows where I downloaded the source templates from the plugin source repository and edited for use with Spring Boot 3 and Jakarta-based libraries.

A reply to one of the Assertj support tickets indicates they are working on it as a part of a Java 17 upgrade.

## 9.2. Example Library Assertions

The following example shows a small peek at the syntax for each of the four assertion libraries used within a JUnit Jupiter test case. They are shown without an `import static` declaration to better see where each comes from.

*Example Assertions*

```
package info.ejava.examples.app.testing.testbasics.jupiter;

import lombok.extern.slf4j.Slf4j;
import org.hamcrest.MatcherAssert;
import org.hamcrest.Matchers;
import org.junit.Assert;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

@Slf4j
class AssertionsTest {
    int lhs=1;
    int rhs=1;
    int expected=2;

    @Test
    void one_and_one() {
```

```
        //junit 4/Vintage assertion
        Assert.assertEquals(expected, lhs+rhs); ①
        //Jupiter assertion
        Assertions.assertEquals(expected, lhs+rhs); ①
        //hamcrest assertion
        MatcherAssert.assertThat(lhs+rhs, Matchers.is(expected)); ②
        //AssertJ assertion
        org.assertj.core.api.Assertions.assertThat(lhs+rhs).isEqualTo(expected); ③
    }
}
```

① JUnit assertions are expressed using a static method and one or more subject arguments

② Hamcrest asserts that the subject matches a `Matcher` that can be infinitely extended

③ AssertJ's extensible subject assertion provides type-specific assertion builders

## 9.3. Assertion Failures

Assertions will report a generic message when they fail. If we change the expected result of the example from 2 to 3, the following error message will be reported. It contains a generic message of the assertion failure (location not shown) without context other than the test case and test method it was generated from (not shown).

*Example Default Assert Failure Message*

```
java.lang.AssertionError: expected:<3> but was:<2> ①
```

① we are not told what 3 and 2 are within a test except that 3 was expected and they are not equal

### 9.3.1. Adding Assertion Context

However, there are times when some additional text can help to provide more context about the problem. The following example shows the previous test augmented with an optional message. Note that JUnit Jupiter assertions permit the lazy instantiation of complex message strings using a lamda. AssertJ provides for lazy instantiation using `String.format` built into the `as()` method.

*Example Assertions with Message Supplied*

```
@Test
void one_and_one_description() {
    //junit 4/Vintage assertion
    Assert.assertEquals("math error", expected, lhs+rhs); ①
    //Jupiter assertions
    Assertions.assertEquals(expected, lhs+rhs, "math error"); ②
    Assertions.assertEquals(expected, lhs+rhs,
            ()->String.format("math error %d+%d!=%d",lhs,rhs,expected)); ③
    //hamcrest assertion
    MatcherAssert.assertThat("math error",lhs+rhs, Matchers.is(expected)); ④
    //AssertJ assertion
    org.assertj.core.api.Assertions.assertThat(lhs+rhs)
```

```
            .as("math error") ⑤
            .isEqualTo(expected);
    org.assertj.core.api.Assertions.assertThat(lhs+rhs)
            .as("math error %d+%d!=%d",lhs,rhs,expected) ⑥
            .isEqualTo(expected);
}
```

① JUnit Vintage syntax places optional message as first parameter

② JUnit Jupiter moves the optional message to the last parameter

③ JUnit Jupiter also allows optional message to be expressed thru a lambda function

④ Hamcrest passes message in first position like JUnit Vintage syntax

⑤ AspectJ uses an `as()` builder method to supply a message

⑥ AspectJ also supports `String.format` and args when expressing message

*Example Assert Failure with Supplied Message*

```
java.lang.AssertionError: math error expected:<3> but was:<2> ①
```

① an extra "math error" was added to the reported error to help provide context

Although AssertJ supports multiple asserts in a single call chain, your description (`as("description")`) must come before the first failing assertion.

Because AssertJ uses chaining

- there are fewer imports required
- IDEs are able to more easily suggest a matcher based on the type returned from the end of the chain. There is always a context specific to the next step.

## 9.4. Testing Multiple Assertions

The above examples showed several ways to assert the same thing with different libraries. However, evaluation would have stopped at the first failure in each test method. There are many times when we want to know the results of several assertions. For example, take the case where we are testing different fields in a returned object (e.g., person.getFirstName(), person.getLastName()). We may want to see all the results to give us better insight for the entire problem.

JUnit Jupiter and AssertJ support testing multiple assertions prior to failing a specific test and then go on to report the results of each failed assertion.

### 9.4.1. JUnit Jupiter Multiple Assertion Support

JUnit Jupiter uses a variable argument list of Java 8 lambda functions in order to provide support for testing multiple assertions prior to failing a test. The following example will execute both assertions and report the result of both when they fail.

*JUnit Jupiter Multiple Assertion Support*

```
@Test
void junit_all() {
    Assertions.assertAll("all assertions",
        () -> Assertions.assertEquals(expected, lhs+rhs, "jupiter assertion"), ①
        () -> Assertions.assertEquals(expected, lhs+rhs,
            ()->String.format("jupiter format %d+%d!=%d",lhs,rhs,expected))
    );
}
```

① JUnit Jupiter uses Java 8 lambda functions to execute and report results for multiple assertions

### 9.4.2. AssertJ Multiple Assertion Support

AssertJ uses a special factory class (`SoftAssertions`) to build assertions from to support that capability. Notice also that we have the chance to inspect the state of the assertions before failing the test. That can give us the chance to gather additional information to place into the log. We also have the option of not technically failing the test under certain conditions.

*AssertJ Multiple Assertion Support*

```
import org.assertj.core.api.SoftAssertions;
...
    @Test
    public void all() {
        Person p = beaver; //change to eddie to cause failures
        SoftAssertions softly = new SoftAssertions(); ①
        softly.assertThat(p.getFirstName()).isEqualTo("Jerry");
        softly.assertThat(p.getLastName()).isEqualTo("Mathers");
        softly.assertThat(p.getDob()).isAfter(wally.getDob());

        log.info("error count={}", softly.errorsCollected().size()); ②
        softly.assertAll(); ③
    }
```

① a special `SoftAssertions` builder is used to construct assertions

② we are able to inspect the status of the assertions before failure thrown

③ assertion failure thrown during later `assertAll()` call

## 9.5. Asserting Exceptions

JUnit Jupiter and AssertJ provide direct support for inspecting Exceptions within the body of the test method. Surprisingly, Hamcrest offers no built-in matchers to directly inspect Exceptions.

### 9.5.1. JUnit Jupiter Exception Handling Support

JUnit Jupiter allows for an explicit testing for Exceptions at specific points within the test method.

The type of Exception is checked and made available to follow-on assertions to inspect. From that point forward JUnit assertions do not provide any direct support to inspect the Exception.

*JUnit Jupiter Exception Handling Support*

```
import org.junit.jupiter.api.Assertions;
...
    @Test
    public void exceptions() {
        RuntimeException ex1 = Assertions.assertThrows(RuntimeException.class, ①
            () -> {
                throw new IllegalArgumentException("example exception");
            });
    }
```

① JUnit Jupiter provides means to assert an Exception thrown and provide it for inspection

### 9.5.2. AssertJ Exception Handling Support

AssertJ has an Exception testing capability that is similar to JUnit Jupiter — where an explicit check for the Exception to be thrown is performed and the thrown Exception is made available for inspection. The big difference here is that AssertJ provides Exception assertions that can directly inspect the properties of Exceptions using natural-language calls.

*AssertJ Exception Handling and Inspection Support*

```
Throwable ex1 = catchThrowable( ①
        ()->{ throw new IllegalArgumentException("example exception"); });
assertThat(ex1).hasMessage("example exception"); ②

RuntimeException ex2 = catchThrowableOfType( ①
        ()->{ throw new IllegalArgumentException("example exception"); },
        RuntimeException.class);
assertThat(ex1).hasMessage("example exception"); ②
```

① AssertJ provides means to assert an Exception thrown and provide it for inspection

② AssertJ provides assertions to directly inspect Exceptions

AssertJ goes one step further by providing an assertion that not only is the exception thrown, but can also tack on assertion builders to make on-the-spot assertions about the exception thrown. This has the same end functionality as the previous example — except:

- previous method returned the exception thrown that can be subject to independent inspection
- this technique returns an assertion builder with the capability to build further assertions against the exception

*AssertJ Integrated Exception Handling Support*

```
assertThatThrownBy( ①
        () -> {
```

```
            throw new IllegalArgumentException("example exception");
        }).hasMessage("example exception");

assertThatExceptionOfType(RuntimeException.class).isThrownBy( ①
        () -> {
            throw new IllegalArgumentException("example exception");
        }).withMessage("example exception");
```

① AssertJ provides means to use the caught Exception as an assertion factory to directly inspect the Exception in a single chained call

## 9.6. Asserting Dates

AssertJ has built-in support for date assertions. We have to add a separate library to gain date matchers for Hamcrest.

### 9.6.1. AssertJ Date Handling Support

The following shows an example of AssertJ's built-in, natural-language support for Dates.

*AssertJ Exception Handling Support*

```
import static org.assertj.core.api.Assertions.*;
...
    @Test
    public void dateTypes() {
        assertThat(beaver.getDob()).isAfter(wally.getDob());
        assertThat(beaver.getDob())
             .as("beaver NOT younger than wally")
             .isAfter(wally.getDob()); ①
    }
```

① AssertJ builds date assertions that directly inspect dates using natural-language

### 9.6.2. Hamcrest Date Handling Support

Hamcrest can be extended to support date matches by adding an external `hamcrest-date` library.

*Hamcrest Date Support Dependency*

```
<!-- for hamcrest date comparisons -->
<dependency>
    <groupId>org.exparity</groupId>
    <artifactId>hamcrest-date</artifactId>
    <version>2.0.7</version>
    <scope>test</scope>
</dependency>
```

That dependency adds at least a `DateMatchers` class with date matchers that can be used to express

date assertions using natural-language expression.

*Hamcrest Date Handling Support*

```
import org.exparity.hamcrest.date.DateMatchers;
import static org.hamcrest.MatcherAssert.assertThat;
...
    @Test
    public void dateTypes() {
        //requires additional org.exparity:hamcrest-date library
        assertThat(beaver.getDob(), DateMatchers.after(wally.getDob()));
        assertThat("beaver NOT younger than wally", beaver.getDob(),
                   DateMatchers.after(wally.getDob())); ①
    }
```

① `hamcrest-date` adds matchers that can directly inspect dates

# Chapter 10. Mockito Basics

Without much question — we will have more complex software to test than what we have briefly shown so far in this lesson. The software will inevitably be structured into layered dependencies where one layer cannot be tested without the lower layers it calls. To implement unit tests, we have a few choices:

1. use the real lower-level components (i.e., "all the way to the DB and back", remember — I am calling that choice "Unit Integration Tests" if it can be technically implemented/managed within a single JVM)
2. create a stand-in for the lower-level components (aka "test double")

We will likely take the first approach during integration testing but the lower-level components may bring in too many dependencies to realistically test during a separate unit's own detailed testing.

## 10.1. Test Doubles

The second approach ("test double") has a few options:

- fake - using a scaled down version of the real component (e.g., in-memory SQL database)
- stub - simulation of the real component by using pre-cached test data
- mock - defining responses to calls and the ability to inspect the actual incoming calls made

## 10.2. Mock Support

`spring-boot-starter-test` brings in a pre-integrated, mature open source mocking framework called Mockito. See the example below for an example unit test augmented with mocks using Mockito. It uses a simple Java `Map<String, String>` to demonstrate some simulation and inspection concepts. In a real unit test, the Java Map interface would stand for:

- an interface we are designing (i.e., testing the interface contract we are designing from the client-side)
- a test double we want to inject into a component under test that will answer with pre-configured answers and be able to inspect how called (e.g., testing collaborations within a white box test)

## 10.3. Mockito Learning Example Declarations

*Basic Mockito Example Declarations*

```
package info.ejava.examples.app.testing.testbasics.mockito;

import org.junit.jupiter.api.*;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.ArgumentCaptor;
```

```
import org.mockito.Captor;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.Map;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
public class ExampleMockitoTest {
    @Mock //creating a mock to configure for use in each test
    private Map<String, String> mapMock;
    @Captor
    private ArgumentCaptor<String> stringArgCaptor;
```

- `@ExtendWith` bootstraps Mockito behavior into test case
- `@Mock` can be used to inject a mock of the defined type
  - "nice mock" is immediately available - will react in potentially useful manner by default
- `@Captor` can be used to capture input parameters passed to the mock calls

@InjectMocks will be demonstrated in later white box testing — where the defined mocks get injected into component under test.

## 10.4. Mockito Learning Example Test

The following learning example provides a demonstration of Mock capability — using only the Mock, without the component under test. This is not something anyone would do because it is only demonstrating the Mock capability versus testing the component under test using the assembled Mock. The calls to the Mock during the "conduct test" are calls we would anticipate the component under test would make while being tested.

*Basic Mockito Learning Example Test*

```
@Test
public void listMap() {
    //define behavior of mock during test
    when(mapMock.get(stringArgCaptor.capture()))
            .thenReturn("springboot", "testing"); ①

    //conduct test
    int size = mapMock.size();
    String secret1 = mapMock.get("happiness");
    String secret2 = mapMock.get("joy");

    //evaluate results
```

```
    verify(mapMock).size(); //verify called once  ③
    verify(mapMock, times(2)).get(anyString()); //verify called twice
    //verify what was given to mock
    assertThat(stringArgCaptor.getAllValues().get(0)).isEqualTo("happiness"); ②
    assertThat(stringArgCaptor.getAllValues().get(1)).isEqualTo("joy");
    //verify what was returned by mock
    assertThat(size).as("unexpected size").isEqualTo(0);
    assertThat(secret1).as("unexpected first result").isEqualTo("springboot");
    assertThat(secret2).as("unexpected second result").isEqualTo("testing");
}
```

① when()/then() define custom conditions and responses for mock within scope of test

② getValue()/getAllValues() can be called on the captor to obtain value(s) passed to the mock

③ verify() can be called to verify what was called of the mock

`mapMock.size()` returned 0 while `mapMock.get()` returned values. We defined behavior for `mapMock.get()` but left other interface methods in their default, "nice mock" state.

# Chapter 11. BDD Acceptance Test Terminology

Behavior-Driven Development (BDD) can be part of an agile development process and adds the use of natural-language constructs to express behaviors and outcomes. The BDD behavior specifications are stories with a certain structure that contain an acceptance criteria that follows a "given", "when", "then" structure:

- **given** - initial context
- **when** - event triggering scenario under test
- **then** - expected outcome

## 11.1. Alternate BDD Syntax Support

There is also a strong push to express acceptance criteria in code that can be executed versus a document. Although far from a perfect solution, JUnit, AssertJ, and Mockito do provide some syntax support for BDD-based testing:

- **JUnit Jupiter** allows the assignment of meaningful natural-language phrases for test case and test method names. Nested classes can also be employed to provide additional expression.
- **Mockito** defines alternate method names to better map to the given/when/then language of BDD
- **AssertJ** defines alternate assertion factory names using `then()` and `and.then()` wording

## 11.2. Example BDD Syntax Support

The following shows an example use of the BDD syntax.

*Example BDD Syntax Support*

```
import org.junit.jupiter.api.*;
import static org.assertj.core.api.BDDAssertions.and;
import static org.mockito.BDDMockito.given;
import static org.mockito.BDDMockito.then;

@ExtendWith(MockitoExtension.class)
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class) ①
@DisplayName("map") ②
public class ExampleMockitoTest {
...
    @Nested ③
    public class when_has_key { ①
        @Test
        public void returns_values() {
            //given
            given(mapMock.get(stringArgCaptor.capture()))
```

```
                    .willReturn("springboot", "testing"); ④
//alt syntax
//        doReturn("springboot", "testing")
//                .when(mapMock).get(stringArgCaptor.capture());
            //when
            int size = mapMock.size();
            String secret1 = mapMock.get("happiness");
            String secret2 = mapMock.get("joy");

            //then - can use static import for BDDMockito or BDDAssertions, not both
            then(mapMock).should().size(); //verify called once ⑤
            then(mapMock).should(times(2)).get(anyString()); //verify called twice
⑦ ⑥
            and.then(stringArgCaptor.getAllValues().get(0)).isEqualTo("happiness");
            and.then(stringArgCaptor.getAllValues().get(1)).isEqualTo("joy");
            and.then(size).as("unexpected size").isEqualTo(0);
            and.then(secret1).as("unexpected first result").isEqualTo("springboot");
            and.then(secret2).as("unexpected second result").isEqualTo("testing");
        }
    }
```

① JUnit `DisplayNameGenorator.ReplaceUnderscores` will form a natural-language display name by replacing underscores with spaces

② JUnit `DisplayName` sets the display name to a specific value

③ JUnit `Nested` classes can be used to better express test context

④ Mockito `when/then` syntax replaced by `given/will` syntax expresses the definition of the mock

⑤ Mockito `verify/then` syntax replaced by `then/should` syntax expresses assertions made on the mock

⑥ AssertJ `then` syntax expresses assertions made to supported object types

⑦ AssertJ `and` field provides a natural-language way to access both AssertJ `then` and Mockito `then` in the same class/method

AssertJ provides a static final `and` field to allow its static `then()` and Mockito's static `then()` to be accesses in the same class/test

## 11.3. Example BDD Syntax Output

When we run our test — the following natural-language text is displayed.

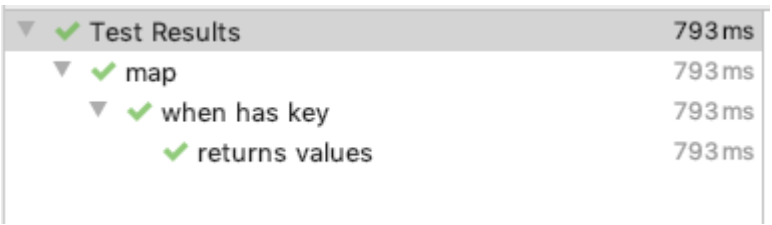


*Figure 7. Example BDD Syntax Output*

## 11.4. JUnit Options Expressed in Properties

We can define a global setting for the display name generator using junit-platform.properties

*test-classes/junit-platform.properties*

```
junit.jupiter.displayname.generator.default = \
  org.junit.jupiter.api.DisplayNameGenerator$ReplaceUnderscores
```

This can also be used to express:

- method order
- class order
- test instance lifecycle
- @Parameterized test naming
- parallel execution

# Chapter 12. Tipping Example

To go much further describing testing — we need to assemble a small set of interfaces and classes to test. I am going to use a common problem when several people go out for a meal together and need to split the check after factoring in the tip.

- **TipCalculator** - returns the amount of tip required when given a certain bill total and rating of service. We could have multiple evaluators for tips and have defined an interface for clients to depend upon.
- **BillCalculator** - provides the ability to calculate the share of an equally split bill given a total, service quality, and number of people.

The following class diagram shows the relationship between the interfaces/classes. They will be the subject of the following Unit Integration Tests involving the Spring context.

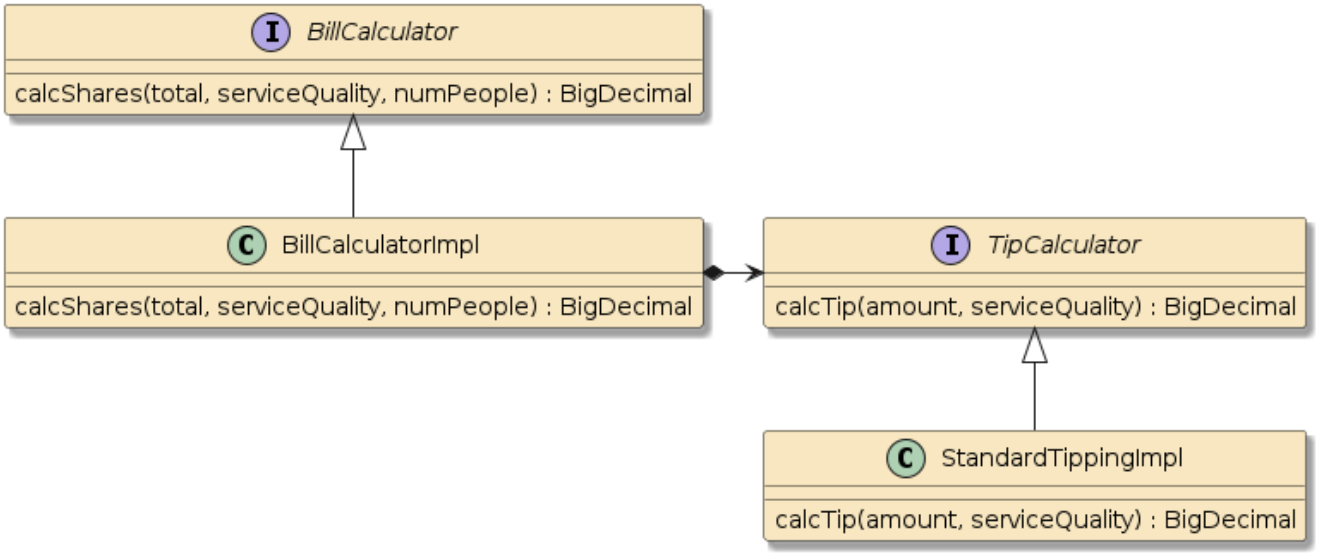


*Figure 8. Tipping Example Class Model*

# Chapter 13. Review: Unit Test Basics

In previous chapters we have looked at pure unit test constructs with an eye on JUnit, assertion libraries, and a little of Mockito. In preparation for the unit integration topic and adding the Spring context in the following chapter — I want to review the simple test constructs in terms of the Tipping example.

## 13.1. Review: POJO Unit Test Setup

*Review: POJO Unit Test Setup*

```
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class) ①
@DisplayName("Standard Tipping Calculator")
public class StandardTippingCalculatorImplTest {
    //subject under test
    private TipCalculator tipCalculator; ②

    @BeforeEach ③
    void setup() { //simulating a complex initialization
        tipCalculator=new StandardTippingImpl();
    }
```

① DisplayName is part of BDD naming and optional for all tests

② there will be one or more objects under test. These will be POJOs.

③ `@BeforeEach` plays the role of a the container — wiring up objects under test

## 13.2. Review: POJO Unit Test

The unit test is being expressed in terms of BDD conventions. It is broken up into "given", "when", and "then" blocks and highlighted with use of BDD syntax where provided (JUnit and AssertJ in this case).

*Review: POJO Unit Test*

```
@Test
public void given_fair_service() { ①
    //given - a $100 bill with FAIR service ②
    BigDecimal billTotal = new BigDecimal(100);
    ServiceQuality serviceQuality = ServiceQuality.FAIR;

    //when - calculating tip ②
    BigDecimal resultTip = tipCalculator.calcTip(billTotal, serviceQuality);

    //then - expect a result that is 15% of the $100 total ②
    BigDecimal expectedTip = billTotal.multiply(BigDecimal.valueOf(0.15));
    then(resultTip).isEqualTo(expectedTip); ③
}
```

① using JUnit snake_case natural language expression for test name

② BDD convention of given, when, then blocks. Helps to be short and focused

③ using AssertJ assertions with BDD syntax

## 13.3. Review: Mocked Unit Test Setup

The following example moves up a level in the hierarchy and forces us to test a class that had a dependency. A pure unit test would mock out all dependencies — which we are doing for `TipCalculator`.

*Review: Mocked Unit Test Setup*

```
@ExtendWith(MockitoExtension.class) ①
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
@DisplayName("Bill CalculatorImpl Mocked Unit Test")
public class BillCalculatorMockedTest {
    //subject under test
    private BillCalculator billCalculator;

    @Mock ②
    private TipCalculator tipCalculatorMock;

    @BeforeEach
    void init() { ③
        billCalculator = new BillCalculatorImpl(tipCalculatorMock);
    }
```

① Add Mockito extension to JUnit

② Identify which interfaces to Mock

③ In this example, we are manually wiring up the subject under test

## 13.4. Review: Mocked Unit Test

The following shows the TipCalculator mock being instructed on what to return based on input criteria and making call activity available to the test.

*Review: Mocked Unit Test*

```
@Test
public void calc_shares_for_people_including_tip() {
    //given - we have a bill for 4 people and tip calculator that returns tip amount
    BigDecimal billTotal = new BigDecimal(100.0);
    ServiceQuality service = ServiceQuality.GOOD;
    BigDecimal tip = billTotal.multiply(new BigDecimal(0.18));
    int numPeople = 4;
    //configure mock
    given(tipCalculatorMock.calcTip(billTotal, service)).willReturn(tip); ①
```

```
    //when - call method under test
    BigDecimal shareResult = billCalculator.calcShares(billTotal, service, numPeople);

    //then - tip calculator should be called once to get result
    then(tipCalculatorMock).should(times(1)).calcTip(billTotal,service); ②

    //verify correct result
    BigDecimal expectedShare = billTotal.add(tip).divide(new BigDecimal(numPeople));
    and.then(shareResult).isEqualTo(expectedShare);
}
```

① configuring response behavior of Mock

② optionally inspecting subject calls made

## 13.5. @InjectMocks

The final unit test example shows how we can leverage Mockito to instantiate our subject(s) under test and inject them with mocks. That takes over at least one job the `@BeforeEach` was performing.

*Alternative Mocked Unit Test*

```
@ExtendWith(MockitoExtension.class)
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
@DisplayName("Bill CalculatorImpl")
public class BillCalculatorImplTest {
    @Mock
    TipCalculator tipCalculatorMock;
    /*
    Mockito is instantiating this implementation class for us an injecting Mocks
     */
    @InjectMocks ①
    BillCalculatorImpl billCalculator;
```

① instantiates and injects out subject under test

# Chapter 14. Spring Boot Unit Integration Test Basics

Pure unit testing can be efficiently executed without a Spring context, but there will eventually be a time to either:

- integrate peer components with one another (horizontal integration)
- integrate layered components to test the stack (vertical integration)

These goals are not easily accomplished without a Spring context and whatever is created outside of a Spring context will be different from production. Spring Boot and the Spring context can be brought into the test picture to more seamlessly integrate with other components and component infrastructure present in the end application. Although valuable, it will come at a performance cost and potentially add external resource dependencies — so don't look for it to replace the lightweight pure unit testing alternatives covered earlier.

## 14.1. Adding Spring Boot to Testing

There are two primary things that will change with our Spring Boot integration test:

1. define a Spring context for our test to operate using `@SpringBootTest`
2. inject components we wish to use/test from the Spring context into our tests using `@Autowire`

I found the following article: Integration Tests with @SpringBootTest, by Tom Hombergs and his "Testing with Spring Boot" series to be quite helpful in clarifying my thoughts and originally preparing these lecture notes. The Spring Boot Testing reference web page provides detailed coverage of the test constructs that go well beyond what I am covering at this point in the course. We will pick up more of that material as we get into web and data tier topics.

## 14.2. @SpringBootTest

To obtain a Spring context and leverage the auto-configuration capabilities of Spring Boot, we can take the easy way out and annotate our test with `@SpringBootTest`. This will instantiate a default Spring context based on the configuration defined or can be found.

*@SpringBootTest Defines Spring Context for Test*

```
package info.ejava.examples.app.testing.testbasics.tips;
...
import org.springframework.boot.test.context.SpringBootTest;
...
@SpringBootTest ①
public class BillCalculatorNTest {
```

① using the default configuration search rules

## 14.3. Default @SpringBootConfiguration Class

By default, Spring Boot will look for a class annotated with `@SpringBootConfiguration` that is present at or above the Java package containing the test. Since we have a class in a parent directory that represents our `@SpringBootApplication` and that annotation wraps `@SpringBootConfiguration`, that class will be used to define the Spring context for our test.

*Example @SpringBootConfiguration Class*

```
package info.ejava.examples.app.testing.testbasics;
...
@SpringBootApplication
// wraps => @SpringBootConfiguration
public class TestBasicsApp {
    public static void main(String...args) {
        SpringApplication.run(TestBasicsApp.class,args);
    }
}
```

## 14.4. Conditional Components

When using the @SpringBootApplication, all components normally a part of the application will be part of the test. Be sure to define auto-configuration exclusions for any production components that would need to be turned off during testing.

```
@Configuration
@ConditionalOnProperty(prefix="hello", name="enable", matchIfMissing=
"true")
public Hello quietHello() {
...
@SpringBootTest(properties = { "hello.enable=false" }) ①
```

① test setting property to trigger disable of certain component(s)

## 14.5. Explicit Reference to @SpringBootConfiguration

Alternatively, we could have made an explicit reference as to which class to use if it was not in a standard relative directory or we wanted to use a custom version of the application for testing.

*Explicit Reference to @SpringBootConfiguration Class*

```
import info.ejava.examples.app.testing.testbasics.TestBasicsApp;
...
@SpringBootTest(classes = TestBasicsApp.class)
public class BillCalculatorNTest {
```

## 14.6. Explicit Reference to Components

Assuming the components required for test is known and a manageable number…

*Components Under Test*

```
@Component
@RequiredArgsConstructor
public class BillCalculatorImpl implements BillCalculator {
    private final TipCalculator tipCalculator;
...

@Component
public class StandardTippingImpl implements TipCalculator {
   ...
```

We can explicitly reference component classes needed to be in the Spring context.

*Explicitly Referencing Components Under Test*

```
@SpringBootTest(classes = {BillCalculatorImpl.class, StandardTippingImpl.class})
public class BillCalculatorNTest {
    @Autowired
    BillCalculator billCalculator;
```

## 14.7. Active Profiles

Prior to adding the Spring context, Spring Boot configuration and logging conventions were not being enacted. However, now that we are bringing in a Spring context — we can designate special profiles to be activated for our context. This can allow us to define properties that are more relevant to our tests (e.g., expressive log context, increased log verbosity).

*Example @ActiveProfiles Declaration*

```
package info.ejava.examples.app.testing.testbasics.tips;

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;

@SpringBootTest
@ActiveProfiles("test") ①
public class BillCalculatorNTest {
```

① activating the "test" profile for this test

*Example application-test.properties*

```
# application-test.properties ①
```

```
logging.level.info.ejava.examples.app.testing.testbasics=DEBUG
```

① "test" profile setting loggers for package under test to `DEBUG` severity threshold

## 14.8. Example @SpringBootTest Unit Integration Test

Putting the pieces together, we have

- a complete Spring context
- `BillCalculator` injected into the test from the Spring context
- `TipCalculator` injected into billCalculator instance from Spring context
- a BDD natural-language, unit integration test that verifies result of bill calculator and tip calculator working together

```
@SpringBootTest
@ActiveProfiles("test")
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
@DisplayName("bill calculator")
public class BillCalculatorNTest {
    @Autowired
    BillCalculator billCalculator;

    @Test
    public void calc_shares_for_bill_total() {
        //given
        BigDecimal billTotal = BigDecimal.valueOf(100.0);
        ServiceQuality service = ServiceQuality.GOOD;
        BigDecimal tip = billTotal.multiply(BigDecimal.valueOf(0.18));
        int numPeople = 4;

        //when - call method under test
        BigDecimal shareResult=billCalculator.calcShares(billTotal,service,numPeople);

        //then - verify correct result
        BigDecimal expectedShare = billTotal.add(tip).divide(BigDecimal.valueOf(4));
        then(shareResult).isEqualTo(expectedShare);
    }
}
```

## 14.9. Example @SpringBootTest NTest Output

When we run our test we get the following console information printed. Note that

- the `DEBUG` messages are from the `BillCalculatorImpl`
- `DEBUG` is being printed because the "test" profile is active and the "test" profile set the severity threshold for that package to be `DEBUG`

- method and line number information is also displayed because the test profile defines an expressive log event pattern

*Example @SpringBootTest Unit Integration Test Output*

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v3.3.2)

14:17:15.427 INFO  BillCalculatorNTest#logStarting:55  - Starting BillCalculatorNTest
14:17:15.429 DEBUG BillCalculatorNTest#logStarting:56  - Running with Spring Boot
v2.2.6.RELEASE, Spring v5.2.5.RELEASE
14:17:15.430 INFO  BillCalculatorNTest#logStartupProfileInfo:655 - The following
profiles are active: test
14:17:16.135 INFO  BillCalculatorNTest#logStarted:61   - Started BillCalculatorNTest
in 6.155 seconds (JVM running for 8.085)
14:17:16.138 DEBUG BillCalculatorImpl#calcShares:24    - tip=$9.00, for $50.00 and
GOOD service
14:17:16.142 DEBUG BillCalculatorImpl#calcShares:33    - share=$14.75 for $50.00, 4
people and GOOD service
14:17:16.143 INFO  BillHandler#run:24 - bill total $50.00, share=$14.75 for 4 people,
after adding tip for GOOD service

14:17:16.679 DEBUG BillCalculatorImpl#calcShares:24    - tip=$18.00, for $100.00 and
GOOD service
14:17:16.679 DEBUG BillCalculatorImpl#calcShares:33    - share=$29.50 for $100.00, 4
people and GOOD service
```

## 14.10. Alternative Test Slices

The `@SpringBootTest` annotation is a general purpose test annotation that likely will work in many generic cases. However, there are other cases where we may need a specific database or other technologies available. Spring Boot pre-defines a set of Test Slices that can establish more specialized test environments. The following are a few examples:

- @DataJpaTest - JPA/RDBMS testing for the data tier
- @DataMongoTest - MongoDB testing for the data tier
- @JsonTest - JSON data validation for marshalled data
- @RestClientTest - executing tests that perform actual HTTP calls for the web tier

We will revisit these topics as we move through the course and construct tests relative additional domains and technologies.

# Chapter 15. Mocking Spring Boot Unit Integration Tests

In the previous `@SpringBootTest` example I showed you how to instantiate a complete Spring context to inject and execute test(s) against an integrated set of real components. However, in some cases we may need the Spring context — but do not need or want the interfacing components. In this example I am going to mock out the `TipCalculator` to produce whatever the test requires.

*Example @SpringBoot/Mockito Definition*

```
import org.springframework.boot.test.mock.mockito.MockBean;

import static org.assertj.core.api.BDDAssertions.and;
import static org.mockito.BDDMockito.given;
import static org.mockito.BDDMockito.then;
import static org.mockito.Mockito.times;

@SpringBootTest(classes={BillCalculatorImpl.class})//defines custom Spring context ①
@ActiveProfiles("test")
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
@DisplayName("Bill CalculatorImpl Mocked Integration")
public class BillCalculatorMockedNTest {
    @Autowired //subject under test ②
    private BillCalculator billCalculator;

    @MockBean //will satisfy Autowired injection point within BillCalculatorImpl ③
    private TipCalculator tipCalculatorMock;
```

① defining a custom context that excludes `TipCalculator` component(s)

② injecting `BillCalculator` bean under test from Spring context

③ defining a mock to be injected into `BillCalculatorImpl` in Spring context

## 15.1. Example @SpringBoot/Mockito Test

The actual test is similar to the earlier example when we injected a real `TipCalculator` from the Spring context. However, since we have a mock in this case we must define its behavior and then optionally determine if it was called.

*Example @SpringBoot/Mockito Test*

```
@Test
public void calc_shares_for_people_including_tip() {
    //given - we have a bill for 4 people and tip calculator that returns tip amount
    BigDecimal billTotal = BigDecimal.valueOf(100.0);
    ServiceQuality service = ServiceQuality.GOOD;
    BigDecimal tip = billTotal.multiply(BigDecimal.valueOf(0.18));
    int numPeople = 4;
```

```
    //configure mock
    given(tipCalculatorMock.calcTip(billTotal, service)).willReturn(tip); ①

    //when - call method under test ②
    BigDecimal shareResult = billCalculator.calcShares(billTotal, service, numPeople);

    //then - tip calculator should be called once to get result
    then(tipCalculatorMock).should(times(1)).calcTip(billTotal,service); ③

    //verify correct result
    BigDecimal expectedShare = billTotal.add(tip).divide(BigDecimal.valueOf(numPeople
));
    and.then(shareResult).isEqualTo(expectedShare); ④
}
```

① instruct the Mockito mock to return a tip result

② call method on subject under test

③ verify mock was invoked N times with the value of the bill and service

④ verify with AssertJ that the resulting share value was the expected share value

# Chapter 16. Maven Unit Testing Basics

At this point we have some technical basics for how tests are syntactically expressed. Now lets take a look at how they fit into a module and how we can execute them as part of the Maven build.

You learned in earlier lessons that production artifacts that are part of our deployed artifact are placed in `src/main` (`java` and `resources`). Our test artifacts are placed in `src/test` (`java` and `resources`). The following example shows the layout of the module we are currently working with.

*Example Module Test Source Tree*

```
|-- pom.xml
`-- src
    `-- test
        |-- java
        |   `-- info
        |       `-- ejava
        |           `-- examples
        |               `-- app
        |                   `-- testing
        |                       `-- testbasics
        |                           |-- PeopleFactory.java
        |                           |-- jupiter
        |                           |   |-- AspectJAssertionsTest.java
        |                           |   |-- AssertionsTest.java
        |                           |   |-- ExampleJUnit5Test.java
        |                           |   `-- HamcrestAssertionsTest.java
        |                           |-- mockito
        |                           |   `-- ExampleMockitoTest.java
        |                           |-- tips
        |                           |   |-- BillCalculatorContractTest.java
        |                           |   |-- BillCalculatorImplTest.java
        |                           |   |-- BillCalculatorMockedNTest.java
        |                           |   |-- BillCalculatorNTest.java
        |                           |   `-- StandardTippingCalculatorImplTest.java
        |                           `-- vintage
        |                               `-- ExampleJUnit4Test.java
        `-- resources
            |-- application-test.properties
```

## 16.1. Maven Surefire Plugin

The Maven Surefire plugin looks for classes that have been compiled from the `src/test/java` source tree that have a prefix of "Test" or suffix of "Test", "Tests", or "TestCase" by default. Surefire starts up the JUnit context(s) and provides test results to the console and target/surefire-reports directory.

Surefire is part of the standard "jar" profile we use for normal Java projects and will run automatically. The following shows the final output after running all the unit tests for the module.

*Example Surefire Execution of All Example Unit Tests*

```
$ mvn clean test
...
[INFO] Results:
[INFO]
[INFO] Tests run: 24, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  14.280 s
```

Consult online documentation on how Maven Surefire can be configured. However, I will demonstrate at least one feature that allows us to filter tests executed.

## 16.2. Filtering Tests

One new JUnit Jupiter feature is the ability to categorize tests using `@Tag` annotations. The following example shows a unit integration test annotated with two tags: "springboot" and "tips". The "springboot" tag was added to all tests that launch the Spring context. The "tips" tag was added to all tests that are part of the tips example set of components.

*Example @Tag*

```
import org.junit.jupiter.api.*;
...
@SpringBootTest(classes = {BillCalculatorImpl.class}) //defining custom Spring context
@Tag("springboot") @Tag("tips") ①
...
public class BillCalculatorMockedNTest {
```

① test case has been tagged with JUnit "springboot" and "tips" tag values

## 16.3. Filtering Tests Executed

We can use the tag names as a "groups" property specification to Maven Surefire to only run matching tests. The following example requests all tests tagged with "tips" but not tagged with "springboot" are to be run. Notice we have fewer tests executed and a much faster completion time.

*Filtering Tests Executed using Surefire Groups*

```
$ mvn clean test -Dgroups='tips & !springboot' -Pbdd ① ②
...
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running Bill Calculator Contract
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.41 s - in
```

```
Bill Calculator Contract
[INFO] Running Bill CalculatorImpl
15:43:47.605 [main] DEBUG
info.ejava.examples.app.testing.testbasics.tips.BillCalculatorImpl - tip=$50.00, for
$100.00 and GOOD service
15:43:47.608 [main] DEBUG
info.ejava.examples.app.testing.testbasics.tips.BillCalculatorImpl - share=$37.50 for
$100.00, 4 people and GOOD service
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.165 s - in
Bill CalculatorImpl
[INFO] Running Standard Tipping Calculator
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 s - in
Standard Tipping Calculator
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.537 s
```

① execute tests with tag "tips" and without tag "springboot"

② activating "bdd" profile that configures Surefire reports within the example Maven environment setup to understand display names

## 16.4. Maven Failsafe Plugin

The Maven Failsafe plugin looks for classes compiled from the `src/test/java` tree that have a prefix of "IT" or suffix of "IT", or "ITCase" by default. Like Surefire, Failsafe is part of the standard Maven "jar" profile and runs later in the build process. However, unlike Surefire that runs within one Maven phase (`test`), Failsafe runs within the scope of four Maven phases: `pre-integration-test`, `integration-test`, `post-integration-test`, and `verify`

- **pre-integration-test** - when external resources get started (e.g., web server)
- **integration-test** - when tests are executed
- **post-integration-test** - when external resources are stopped/cleaned up (e.g., shutdown web server)
- **verify** - when results of tests are evaluated and build potentially fails

## 16.5. Failsafe Overhead

Aside from the integration tests, all other processes are normally started and stopped through the use of Maven plugins. Multiple phases are required for IT tests so that:

- all resources are ready to test once the tests begin

- all resources can be shutdown prior to failing the build for a failed test

With the robust capability to stand up a Spring context within a single JVM, we really have limited use for Failsafe for testing Spring Boot applications. The exception for that is when we truly need to interface with something external — like stand up a real database or host endpoints in Docker images. I will wait until we get to topics like that before showing examples. Just know that when Maven references "integration tests", they come with extra hooks and overhead that may not be technically needed for integration tests — like the ones we have demonstrated in this lesson — that can be executed within a single JVM.

# Chapter 17. @TestConfiguration

Tests often require additional components that are not part of the Spring context under test — or need to override one or more of those components. SpringBoot supplies a `@TestConfiguration` annotation that:

- allows the class to be skipped by standard component scan
- is loaded into a `@SpringBootTest` to add or replace components

## 17.1. Example Spring Context

In our example Spring context, we will have a `TipCalculator` component located using a component scan. It will have the name "standardTippingImpl" if we do not supply an override in the `@Component` annotation.

*Example standardTippingImpl Bean*

```
@Primary ①
@Component
public class StandardTippingImpl implements TipCalculator {
```

① declaring type as primary to make example more significant

That bean gets injected into `BillCalculatorImpl.tipCalculator` because it implements the required type.

*Example Injection Target for Bean*

```
@Component
@RequiredArgsConstructor
public class BillCalculatorImpl implements BillCalculator {
    private final TipCalculator tipCalculator;
```

## 17.2. Test TippingCalculator

Our intent here is to manually write a stub and have it replace the `TipCalculator` from the application's Spring context.

```
import org.springframework.boot.test.context.TestConfiguration;
...

@TestConfiguration(proxyBeanMethods = false) //skipped in component scan -- manually
included ①
public class MyTestConfiguration {
    @Bean
    public TipCalculator standardTippingImpl() { ②
        return new TipCalculator() {
```

```
            @Override
            public BigDecimal calcTip(BigDecimal amount, ServiceQuality
serviceQuality) {
                return BigDecimal.ZERO; ③
            }
        };
    }
}
```

① `@TestConfiguration` annotation prevents class from being picked up in normal component scan

② `standardTippingImpl` name matches existing component

③ test-specific custom response

## 17.3. Enable Component Replacement

Since we are going to replace an existing component, we need to enable bean overrides using the following property definition.

*Enable Bean Override*

```
@SpringBootTest(
        properties = "spring.main.allow-bean-definition-overriding=true"
)
public class TestConfigurationNTest {
```

Otherwise, we end up with the following error when we make our follow-on changes.

*Bean Override Error Message*

```
***************************
APPLICATION FAILED TO START
***************************

Description:

The bean 'standardTippingImpl', defined in class path resource
[.../testconfiguration/MyTestConfiguration.class], could not be registered.
A bean with that name has already been defined in file
[.../tips/StandardTippingImpl.class] and overriding is disabled.

Action:

Consider renaming one of the beans or enabling overriding by setting
spring.main.allow-bean-definition-overriding=true
```

## 17.4. Embedded TestConfiguration

We can have the `@TestConfiguration` class automatically found using an embedded **static** class.

*Embedded TestConfiguration*

```
@SpringBootTest(properties={"..."})
public class TestConfigurationNTest {
    @Autowired
    BillCalculator billCalculator; ①

    @TestConfiguration(proxyBeanMethods = false)
    static class MyEmbeddedTestConfiguration { ②
        @Bean
        public TipCalculator standardTippingImpl() { ... }
    }
```

① injected `billCaculator` will be injected with `@Bean` from @TestConfiguration

② embedded static class used automatically

## 17.5. External TestConfiguration

Alternatively, we can place the configuration in a separate/stand-alone class.

```
@TestConfiguration(proxyBeanMethods = false)
public class MyTestConfiguration {
    @Bean
    public TipCalculator tipCalculator() {
        return new TipCalculator() {
            @Override
            public BigDecimal calcTip(BigDecimal amount, ServiceQuality
serviceQuality) {
                return BigDecimal.ZERO;
            }
        };
    }
}
```

## 17.6. Using External Configuration

The external `@TestConfiguration` will only be used if specifically named in either:

- @SpringBootTest.classes
- @ContextConfiguration.classes
- @Import.value

Pick one way.

*Imported TestConfiguration*

```
@SpringBootTest(
        classes=MyTestConfiguration.class, //way1 ①
        properties = "spring.main.allow-bean-definition-overriding=true"
)
@ContextConfiguration(classes=MyTestConfiguration.class) //way2 ②
@Import(MyTestConfiguration.class) //way3 ③
public class TestConfigurationNTest {
```

① way1 leverages the `@SpringBootTest configuration

② way2 pre-dates `@SpringBootTest`

③ way3 pre-dates `@SpringBootTest` and is a standard way to import a configuration definition from one class to another

## 17.7. TestConfiguration Result

Running the following test results in:

- a single `TipCalculator` registered in the list because each considered have the same name and overriding is enabled
- the `TipCalculator` used is one of the `@TestConfiguration`-supplied components

*TipCalculator Replaced by @TestConfiguration-supplied Component*

```
@SpringBootTest(
        classes=MyTestConfiguration.class,
        properties = "spring.main.allow-bean-definition-overriding=true")
public class TestConfigurationNTest {
    @Autowired
    BillCalculator billCalculator;
    @Autowired
    List<TipCalculator> tipCalculators;

    @Test
    void calc_has_been_replaced() {
        //then
        then(tipCalculators).as("too many topCalculators").hasSize(1);
        then(tipCalculators.get(0).getClass()).hasAnnotation(TestConfiguration.class);
①
    }
```

① `@Primary TipCalculator` bean replaced by our `@TestConfiguration`-supplied bean

# Chapter 18. Summary

In this module we:

- learned the importance of testing
- introduced some of the testing capabilities of libraries integrated into `spring-boot-starter-test`
- went thru an overview of JUnit Vintage and Jupiter test constructs
- stressed the significance of using assertions in testing and the value in making them based on natural-language to make them easy to understand
- introduced how to inject a mock into a subject under test
- demonstrated how to define a mock for testing a particular scenario
- demonstrated how to inspect calls made to the mock during testing of a subject
- discovered how to switch default Mockito and AssertJ methods to match Business-Driven Development (BDD) acceptance test keywords
- implemented unit integration tests with Spring context using `@SpringBootTest`
- implemented mocks into the Spring context of a unit integration test
- ran tests using Maven Surefire
- implemented a `@TestConfiguration` with component override