A deep dive into JUnit 5

Ministry of Testing

99 Minute Workshop

21/06/2021

Boni García

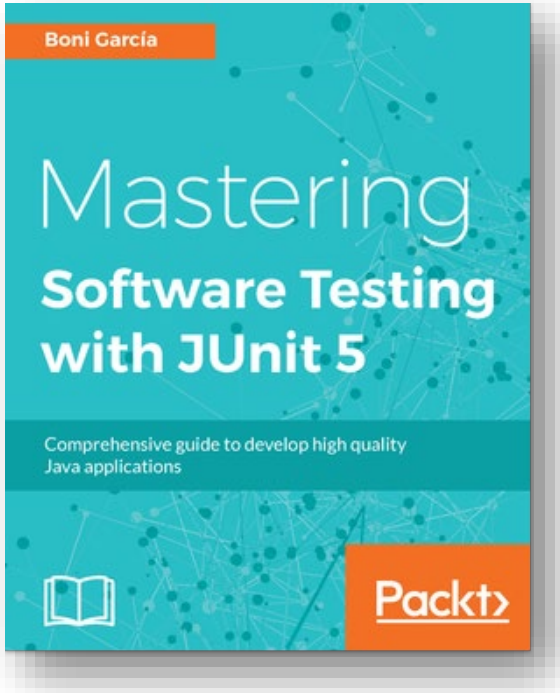
 boni.garcia@uc3m.es  http://bonigarcia.github.io/

 @boni_gg  https://github.com/bonigarcia

Table of contents

1. Introduction - Presentation

- **JUnit** is the most popular testing framework for Java and can be used to implement different types of tests (unit, integration, end-to-end, …)
- **JUnit 5** (first GA released on September 2017) provides a brand-new programming an extension model called **Jupiter**



JUnit 5

https://junit.org/junit5/docs/current/user-guide/

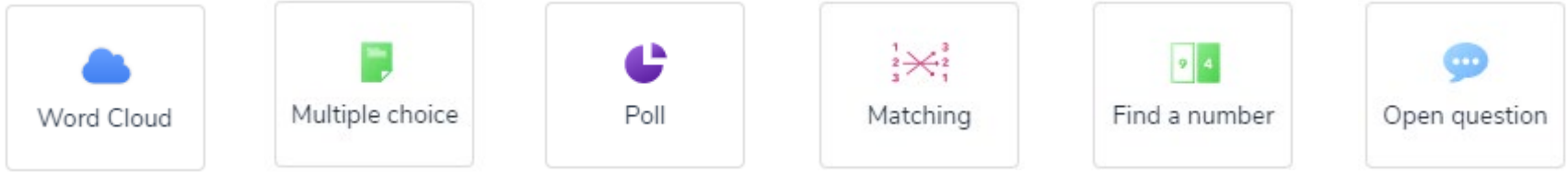
1. Introduction - Objectives

- Learning outcomes:
  1. Describe the modular architecture of JUnit 5
  2. Execute basic test cases with JUnit 5
  3. Develop advanced test cases using Jupiter
- Requirements:
  - Basic knowledge of Java language
  - Software:
    - Java 8+ (JDK)
    - Some IDE (e.g. Eclipse, IntellijIDEA, Visual Studio, or NetBeans)
    - A build tool such as Maven or Gradle (optional)
    - Code examples GitHub repo:

https://github.com/bonigarcia/mastering-junit5

1. Introduction - Activities

- We are going to use **wooclap** to make different activities during the workshop
- Wooclap is a collaborative platform for training sessions providing a rich variety of questions, such as:



wooclap

(activities 1-2)

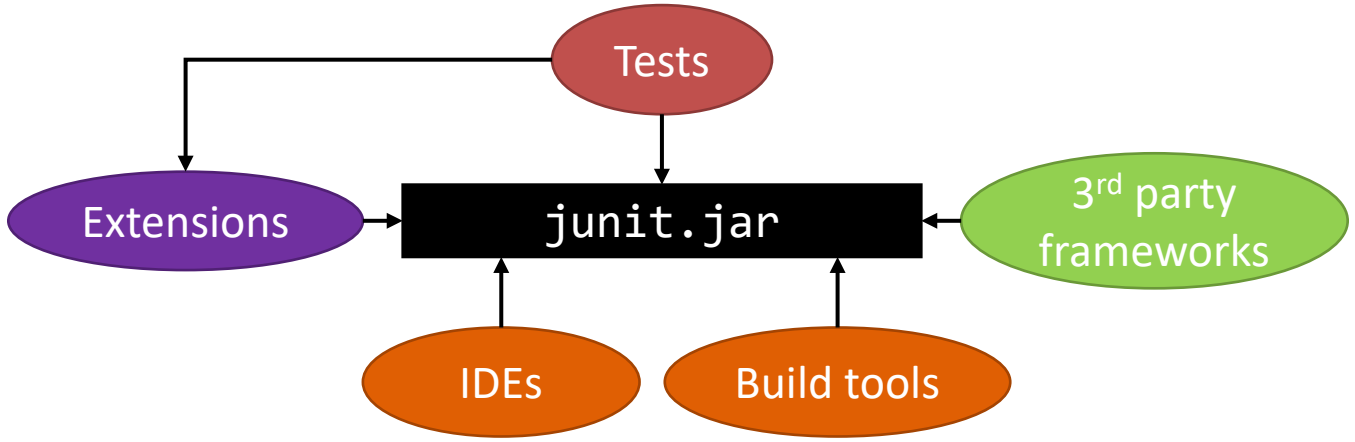
https://app.wooclap.com/XPEEYA

Table of contents

2. JUnit 5 overview - Motivation

- **JUnit 4** is extensively adopted by the Java community
- Nevertheless, it has relevant limitations, namely:

1. It's **monolithic** (single component). Some features (such as test discovery and execution) are highly coupled



JUnit

2. JUnit 5 overview - Motivation

2. Test cases are executed on JUnit 4 using special classes called ***Test Runners***. These runners have a fundamental limitation: are not composable

```java
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

@RunWith(Parameterized.class)
public class MyTest1 {

    @Test
    public void myTest() {
        // my test code
    }

}
```

```java
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
public class MyTest2 {

    @Test
    public void myTest() {
        // my test code
    }

}
```

JUnit

2. JUnit 5 overview - Motivation

3. To improve the management of the test life cycle in JUnit 4, the rules (***Test Rules***) were developed, implemented with the `@Rule` and `@ClassRule` annotations. The downside is that it can be difficult to manage both areas simultaneously (runners and rules)

```java
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ErrorCollector;

public class MyTest3 {

    @Rule
    public ErrorCollector errorCollector = new ErrorCollector();

    @Test
    public void myTest() {
        // my test code
    }

}
```

```java
import org.junit.ClassRule;
import org.junit.Test;
import org.junit.rules.TemporaryFolder;

public class MyTest4 {

    @ClassRule
    public TemporaryFolder temporaryFolder = new TemporaryFolder();

    @Test
    public void myTest() {
        // my test code
    }

}
```

2. JUnit 5 overview - Motivation

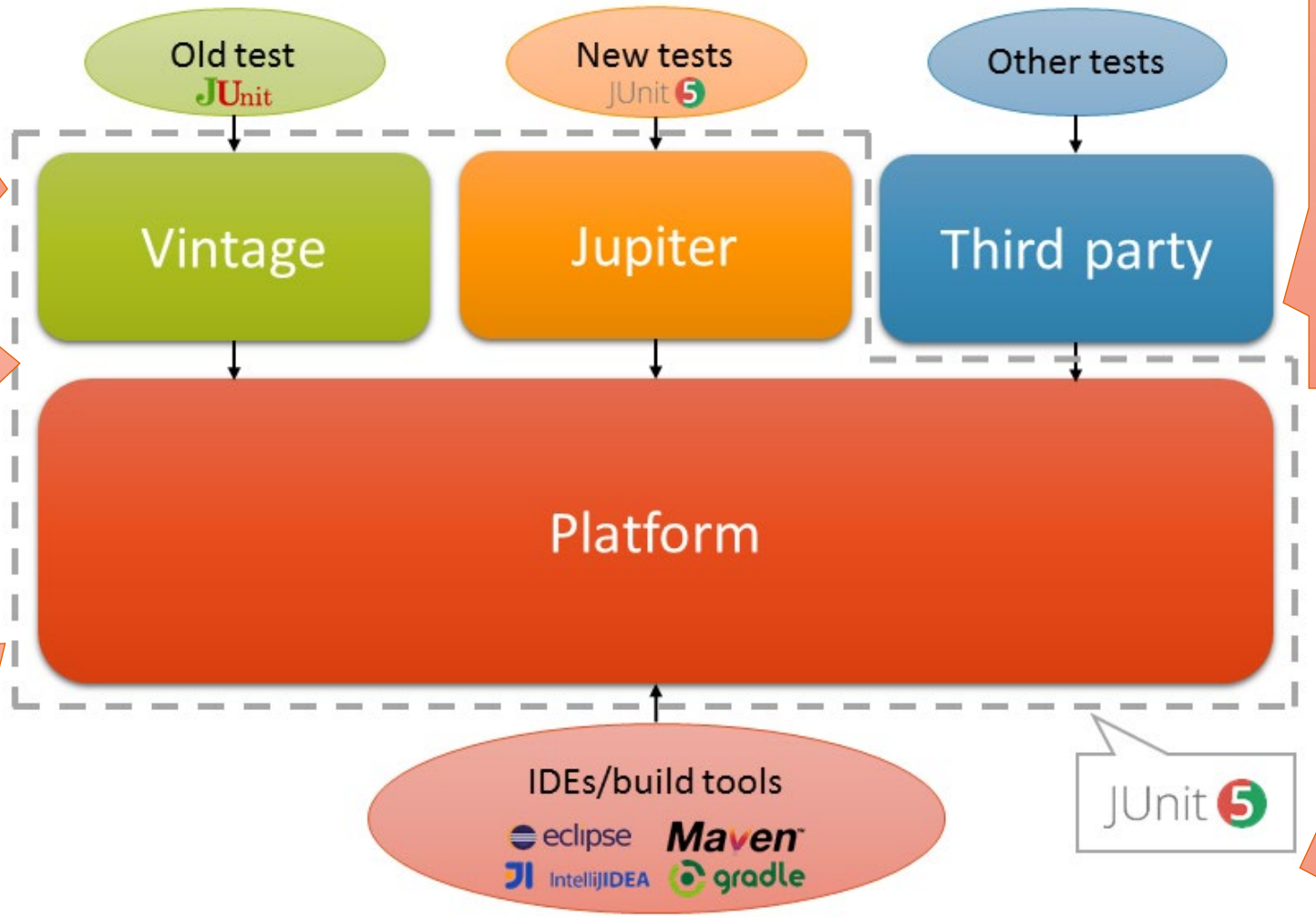
- To try to solve these problems, in July 2015 Johannes Link and Mark Philipp launched a campaign to raise funds and create a new version of JUnit
- This campaign became known as the JUnit Lambda crowdfunding campaign
- Thanks to this campaign, the JUnit 5 team was launched, with members of different companies (Eclipse, Gradle, and IntelliJ among others)

2. JUnit 5 overview - Architecture

Test of previous versions of JUnit (3 and 4) will be executed through the ***Vintage*** component

Jupiter is a component that implements the new programming and extension model in JUnit 5

The JUnit platform (***Platform***) is a component that acts as a generic executor for tests within the JVM



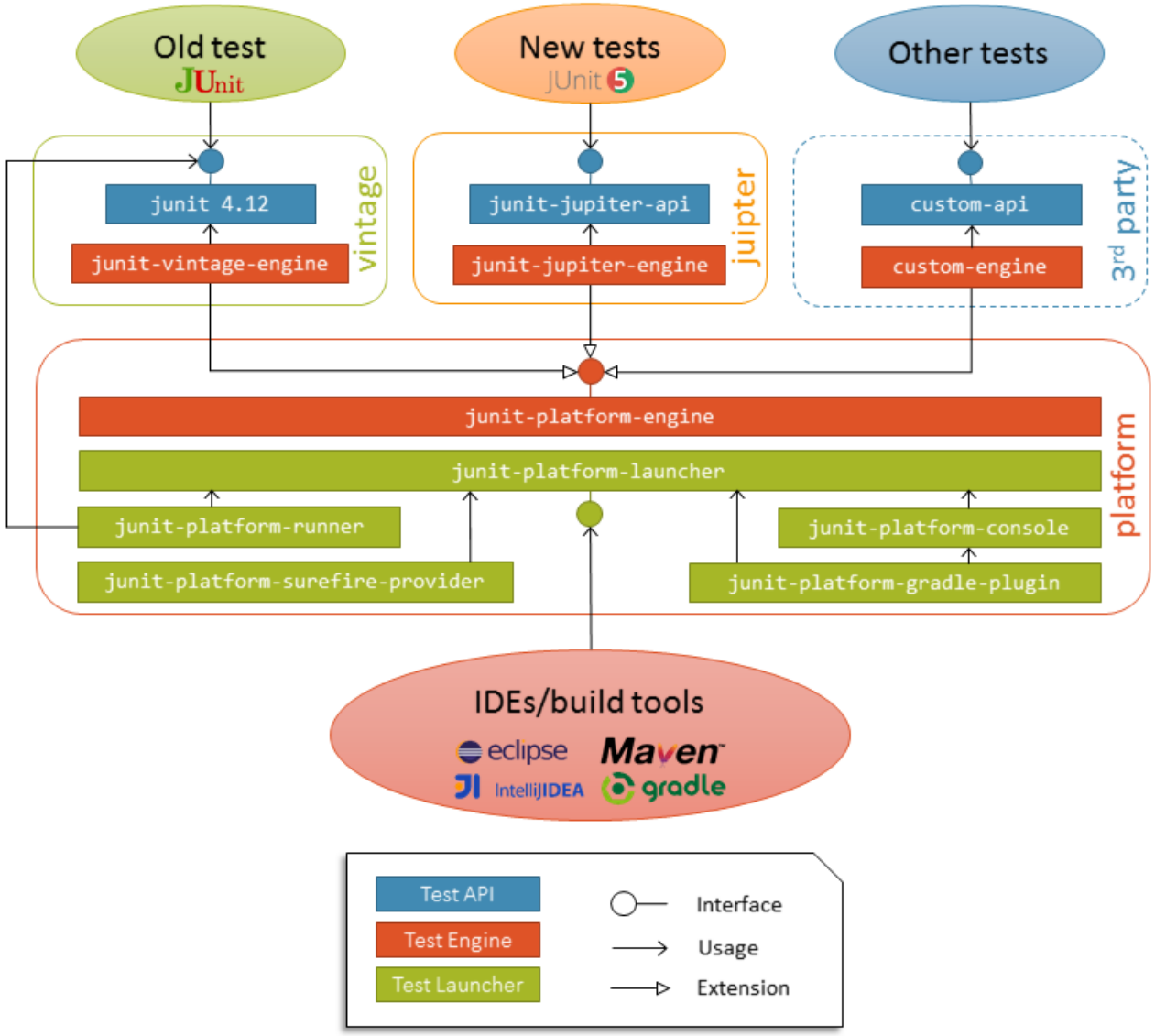
The idea is that other frameworks (e.g. Spock, Cucumber) can run their own test cases by extending the platform

Programmatic clients use the platform for discovery and test execution

2. JUnit 5 overview - Architecture

- There are three types of modules:

1. **Test API**: Modules used by testers to implement test cases
2. **Test Engine SPI**: Extended modules for a Java test framework for the execution of a specific test model
3. **Test Launcher API**: Modules used by programmatic clients for discovery and execution of tests



2. JUnit 5 overview - Support

- JUnit 5 test can be executed in different ways:

1. Using a **build tools**:

2. Using an **IDE**:

3. Using the **console launcher** (standalone JAR provided by the JUnit 5 team):

```
java -jar junit-platform-console-standalone-version.jar <Options>
```

wooclap

(activities 3 - 6)

Table of contents

3. Basic tests - Setup

- To execute Jupiter tests with **Maven** we need to configure `pom.xml`:

```xml
<properties>
    <junit5.version>5.7.2</junit5.version>
    <maven-surefire-plugin.version>2.22.2</maven-surefire-plugin.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>${junit5.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>${maven-surefire-plugin.version}</version>
        </plugin>
    </plugins>
</build>
```

The artifact `junit-jupiter` transitively pulls the following dependencies

```xml
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>${junit5.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>${junit5.version}</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
       <groupId>org.junit.jupiter</groupId>
       <artifactId>junit-jupiter-params</artifactId>
       <version>${junit.jupiter.version}</version>
       <scope>test</scope>
    </dependency>
</dependencies>
```

3. Basic tests - Setup

- To execute Jupiter tests with **Gradle** we need to configure `build.gradle`:

```
apply plugin: 'java'

test {
    useJUnitPlatform()
}

ext {
    junit5  = '5.7.2'
}

dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:${junit5}")
}
```

The artifact `junit-jupiter` transitively pulls the following dependencies

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter-api:${junit5}")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:${junit5}")
    testImplementation("org.junit.jupiter:junit-jupiter-params:${junitJupiterVersion}")
}
```

3. Basic tests - Logging

- Although not mandatory, it is recommended to use **logger objects** to log messages in our applications and tests
- In the examples, we use the following libraries for logging:
  - Simple Logging Facade for Java (SLF4J): Facade for various logging frameworks
  - Logback: Logging framework


```xml
<properties>
    <slf4j.version>1.7.31</slf4j.version>
    <logback.version>1.2.3</logback.version>
</properties>

<dependencies>
   <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>${slf4j.version}</version>
   </dependency>
   <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>${logback.version}</version>
   </dependency>
</dependencies>
```

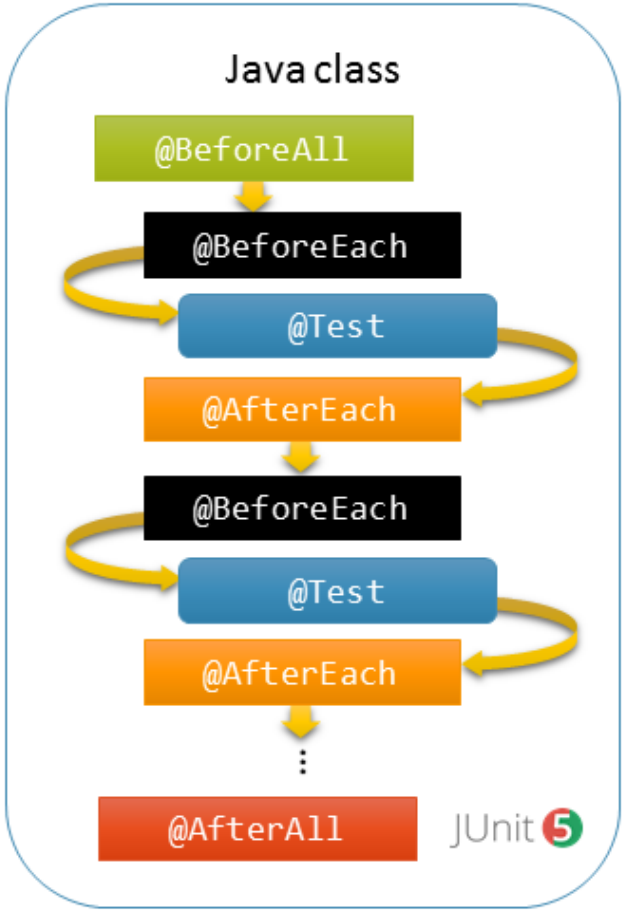
Maven

```
ext {
    slf4jVersion  = '1.7.31'
    logbackVersion  = '1.2.3'
}

dependencies {
    implementation("org.slf4j:slf4j-api:${slf4jVersion}")
    implementation("ch.qos.logback:logback-classic:${logbackVersion}")
}
```

3. Basic tests - Test lifecycle

mastering-junit5/junit5-basic-tests

- **Java annotations** are the main building block to create Jupiter tests:



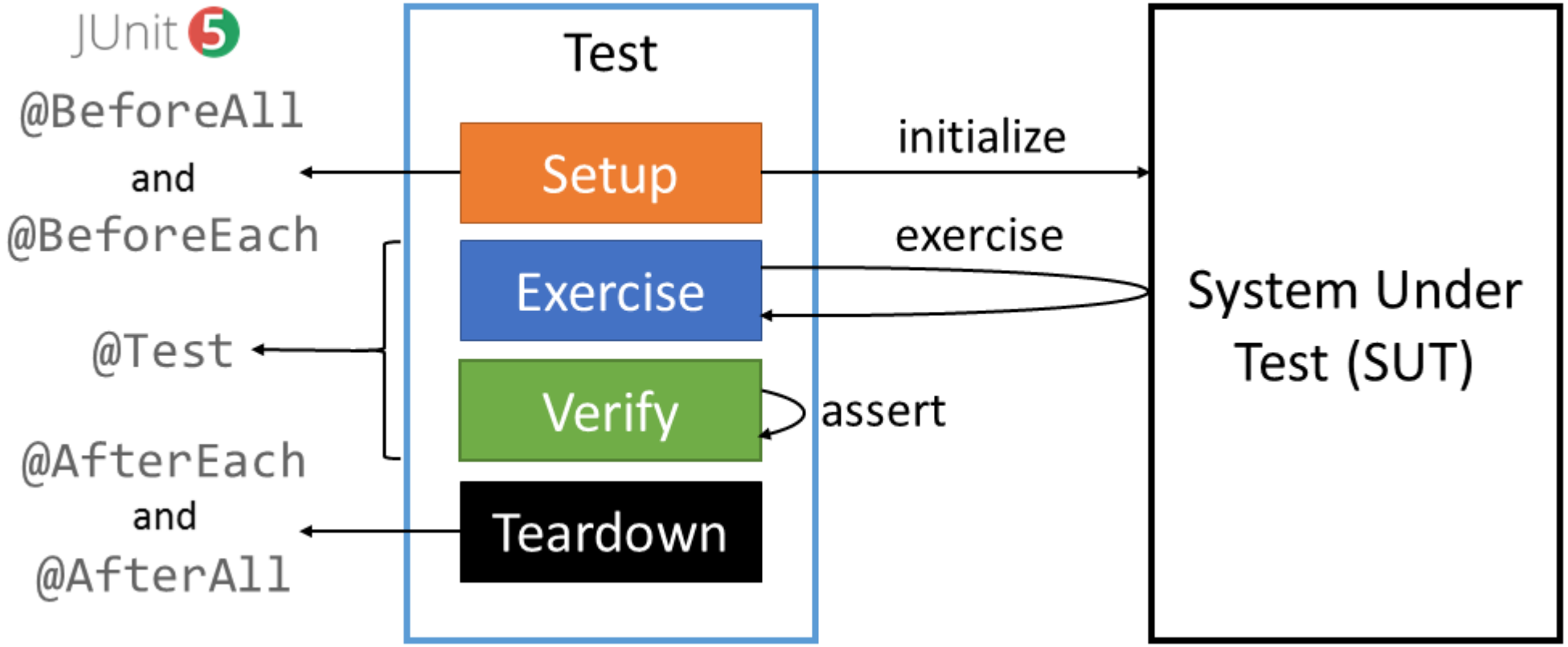

```
class LifecycleTest {

    static final Logger log = getLogger(lookup().lookup

    @BeforeAll
    static void setupAll() {
        log.debug("@BeforeAll");
    }

    @BeforeEach
    void setup() {
        log.debug("@BeforeEach");
    }

    @Test
    void test1() {
        log.debug("@Test [1]");
    }

    @Test
    void test2() {
        log.debug("@Test [2]");
    }

    @AfterEach
    void teardown() {
        log.debug("@AfterEach");
    }

    @AfterAll
    static void teardownAll() {
        log.debug("@AfterAll");
    }

}
```

Fork me on GitHub

Classes and methods are not required to be `public` in Jupiter

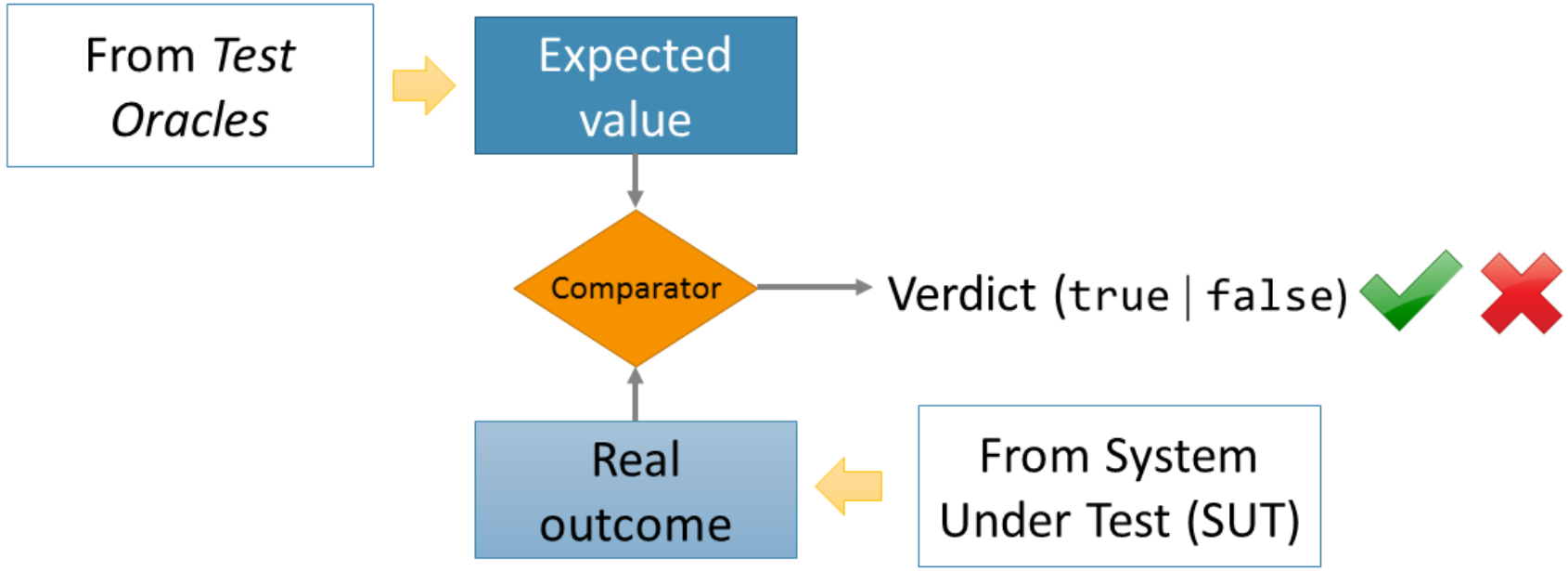
3. Basic tests - Test lifecycle

- The complete **test life cycle** of a JUnit 5 test in context with the System Under Test (SUT) is the following:



3. Basic tests - Assertions

- Conceptually, an **assertion** (or predicate) is made up of:
  - Expected data, obtained from what is known as an oracle (typically the SUT specification)
  - Actual data, obtained from exercising the system under test (SUT)
  - A logical operator that compares both values



3. Basic tests - Assertions

- JUnit 5 provides a rich variety of assertions (static methods of the class `org.junit.jupiter.api.Assertions`):

Assertion	Description
`fail`	Fail a test by providing an error or exception message
`assertTrue`	Evaluate if a condition is true
`assertFalse`	Evaluates if a condition is false
`assertNull`	Evaluates if an object is null
`assertNotNull`	Evaluates if an object is not null
`assertEquals`	Evaluates if one object is equal to another
`assertNotEquals`	Evaluates if one object is not equal to another
`assertArrayEquals`	Evaluates if an array is equal to others
`assertIterableEquals`	Evaluates if two iterable objects are equal
`assertLinesMatch`	Evaluate if two lists of String are equal
`assertSame`	Evaluate if one object is the same as another
`assertNotSame`	Evaluate if one object is not the same as another

3. Basic tests - Assertions

mastering-junit5/junit5-basic-tests


```java
class BasicTest {

    static final Logger log = getLogger(lookup().lookupClass());

    MySUT mySut;

    @BeforeEach
    void setup() {
        mySut = new MySUT("[Basic test]");
        mySut.initId();
    }

    @Test
    void sumTest() {
        log.debug("Testing sum method in {}", mySut.getName());

        // exercise
        int sum = mySut.sum(1, 2, 3);

        // verify
        Assertions.assertTrue(sum == 6);
    }

    @Test
    void concanateTest() {
        log.debug("Testing sum concatenate in {} SUT", mySut.getName());

        // exercise
        String phrase = mySut.concatenate("hello", "world");

        // verify
        Assertions.assertTrue(phrase.equals("hello world"));
    }

    @AfterEach
    void teardown() {
        mySut.releaseId();
        mySut.close();
    }

}
```

3. Basic tests - Test execution

- The very basic commands to run tests using the shell are:

Build tool	Command	Description
Maven	`mvn test`	Run all tests in a Maven project (using the plugin `maven-surefire-plugin`*)
	`mvn test -Dtest=MyTestClass`	Run all tests in a single class
	`mvn test -Dtest=MyTestClass#myTestMethod`	Run a single test in a single class
Gradle	`gradle test`	Run all tests in a Gradle project
	`gradle test --rerun-tasks`	Run all tests in a Gradle project (even if everything is up-to-date)
	`gradle test --tests MyTestClass`	Run all tests in a single class
	`gradle test --tests MyTestClass.MyTestMethod`	Run a single test in a single class

* The Maven build lifecycle also defines "*integration*" tests (executed **after** packaging), to be executed with the plugin (`maven-failsafe-plugin`) and the command `mvn verify`

3. Basic tests - Test execution

- When a test is executed, the result could be:
- `Pass`: A test is executed correctly
- `Skip`: Some test is not executed
- `Fail`: Some assertion in a test fails
- `Error`: Some uncontrolled exception happens during the test execution

Maven™

- When there are failed or errored tests, a build is declared as failed (`BUILD FAILED`)

wooclap

(activities 7-8)

3. Basic tests - Disabled tests

mastering-junit5/junit5-disabled-tests

- Jupiter provides different annotations to disable (`skip`) tests
- The annotation `@Disabled` is used to skip tests (used at method or class level)

```
@Disabled("All tests in this class are skipped")
class DisabledAllTest {

    static final Logger log =
           getLogger(lookup().lookupClass());

    @Test
    void skippedTestOne() {
        log.debug("This test is NOT executed");
    }

    @Test
    void skippedTestTwo() {
        log.debug("This test is NOT executed");
    }

}
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.DisabledAllTest
[INFO] Results:
[WARNING] Tests run: 1, Failures: 0, Errors: 0, Skipped: 1
```



```
class DisabledTest {

    static final Logger log =
           getLogger(lookup().lookupClass());

    @Test
    void test() {
        log.debug("This test is executed");
    }

    @Disabled
    @Test
    void skippedTest() {
        log.debug("This test is NOT executed");
    }

}
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.DisabledTest
[main] DEBUG io.github.bonigarcia.DisabledTest.test(32) --
This test is executed
[INFO] Results:
[INFO]
[WARNING] Tests run: 2, Failures: 0, Errors: 0, Skipped: 1
```

3. Basic tests - Disabled tests

mastering-junit5/junit5-disabled-tests

- The annotation `@DisabledOnOs` is used to skip tests depending on the operating system

```
class DisabledOnOsTest {

    static final Logger log =
           getLogger(lookup().lookupClass());

    @DisabledOnOs(LINUX)
    @Test
    void notLinuxTest() {
        log.debug("Disabled on Linux");
    }

    @DisabledOnOs(WINDOWS)
    @Test
    void notWinTest() {
        log.debug("Disabled on Windows");
    }

    @DisabledOnOs(MAC)
    @Test
    void notMacTest() {
        log.debug("Disabled on Mac");
    }

}
```

3. Basic tests - Disabled tests

mastering-junit5/junit5-disabled-tests

- The annotations `@EnabledOnJre` and `@DisabledOnJre` are used to enable/disable tests depending on the JRE version (or range)

```
class DisabledOnJreRangeTest {

    static final Logger log = getLogger(lookup().lookupClass());

    @Test
    @EnabledOnJre(JAVA_8)
    void onlyOnJava8() {
        log.debug("This test is executed only for JRE 8");
    }

    @Test
    @DisabledOnJre(JAVA_9)
    void notOnJava9() {
        log.debug("This test is executed only for JRE != 9");
    }

    @Test
    @DisabledForJreRange(min = JAVA_9, max = JAVA_11)
    void notFromJava9to11() {
        log.debug("This test is executed only for JRE != 9 to 11");
    }

    // ...

}
```

3. Basic tests - Disabled tests

mastering-junit5/junit5-disabled-tests

- Assumptions are used to skip tests programmatically
- For this, we can use the static methods *`assumeTrue`* and *`assumeFalse`* of the class `Assumptions` (package `org.junit.jupiter.api`)

```
class AssumptionsTest {

    static final Logger log = getLogger(lookup().lookupClass());

    MySUT mySut;

    @BeforeEach
    void setup() {
        mySut = new MySUT("[Test with assumptions]");
    }

    @Test
    void assumeTrueTest() {
        assumeTrue(mySut.getId() != null);
        log.debug("This test is skipped");
    }

    @Test
    void assumeFalseTest() {
        assumeFalse(mySut.getId() != null);
        log.debug("This test is executed");
    }

}
```

wooclap

(activity 9)

Table of contents

4. Advanced tests - High-level and fluent assertions

mastering-junit5/junit5-assertions

- The occurrence of exceptions is implemented by the assertion *assertThrows*:

```
class ExceptionTest {

    static final Logger log = getLogger(lookup().lookupClass());

    MySUT mySut;

    @BeforeEach
    void setup() {
        mySut = new MySUT("[Assertions test]");
    }

    @Test
    void exceptionTesting() {
        log.debug("Testing with assertThrows");
        assertThrows(RuntimeException.class, mySut::releaseId);
        log.debug("End of test");
    }

}
```

In this example the test will pass since we are waiting for `RuntimeException`, and it actually happens the SUT

We can pass a lambda expression or a method reference operator

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.ExceptionTest
[main] INFO  io.github.bonigarcia.MySUT.<init>(37) -- [Assertions test] created
[main] DEBUG io.github.bonigarcia.ExceptionTest.exceptionTesting(40) -- Testing
with assertThrows
[main] DEBUG io.github.bonigarcia.ExceptionTest.exceptionTesting(44) -- End of
test
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.127 s
- in io.github.bonigarcia.ExceptionTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

4. Advanced tests - High-level and fluent assertions

mastering-junit5/junit5-assertions

- We can use `assertTimeout` to evaluate execution time:

This test passes

This test fails

```
import static java.time.Duration.ofMillis;
import static java.time.Duration.ofMinutes;
import static org.junit.jupiter.api.Assertions.assertTimeout;

import org.junit.jupiter.api.Test;

class TimeoutExceededTest {

    @Test
    void timeoutNotExceeded() {
        assertTimeout(ofMinutes(2), () -> {
            // Perform task that takes less than 2 minutes
        });
    }

    @Test
    void timeoutExceeded() {
        assertTimeout(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms
            Thread.sleep(100);
        });
    }

}
```

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.TimeoutExceededTest
Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time
elapsed: 0.18 sec <<< FAILURE! - in
io.github.bonigarcia.TimeoutExceededTest
timeoutExceeded()  Time elapsed: 0.126 sec  <<< FAILURE!
org.opentest4j.AssertionFailedError: execution exceeded
timeout of 10 ms by 90 ms
        at
io.github.bonigarcia.TimeoutExceededTest.timeoutExceeded(
TimeoutExceededTest.java:36)


Results :

Failed tests:
  TimeoutExceededTest.timeoutExceeded:36 execution
exceeded timeout of 10 ms by 90 ms

Tests run: 2, Failures: 1, Errors: 0, Skipped: 0
```

4. Advanced tests - High-level and fluent assertions

mastering-junit5/junit5-assertions

- In addition, there is a number of Java libraries providing fluent APIs for assertions, such as:
  - Hamcrest: http://hamcrest.org/
  - AssertJ: https://assertj.github.io/doc/
  - Truth: https://truth.dev/

```
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.notNullValue;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;

class HamcrestTest {

    @Test
    void assertWithHamcrestMatcher() {
        assertThat(2 + 1, equalTo(3));
        assertThat("Foo", notNullValue());
        assertThat("Hello world", containsString("world"));
    }

}
```

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.HamcrestTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.059 sec - in io.github.bonigarcia.HamcrestTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

wooclap

(activity 10)

4. Advanced tests - Tagging and filtering

mastering-junit5/junit5-tagging-filtering

- Classes and test methods in JUnit 5 can be tagged using the @Tag annotation
- These tags can be used for the discovery and execution of the tests (filtering)

```
@Tag("functional")
class FunctionalTest {

    static final Logger log =
           getLogger(lookup().lookupClass());

    @Test
    void testOne() {
        log.debug("Functional Test 1");
    }

    @Test
    void testTwo() {
        log.debug("Functional Test 2");
    }

}
```

```
@Tag("non-functional")
class NonFunctionalTest {

    static final Logger log = getLogger(lookup().lookupClass());

    @Test
    @Tag("performance")
    @Tag("load")
    void testOne() {
        log.debug("Non-Functional Test 1 (Performance/Load)");
    }

    @Test
    @Tag("performance")
    @Tag("stress")
    void testTwo() {
        log.debug("Non-Functional Test 2 (Performance/Stress)");
    }

    @Test
    @Tag("security")
    void testThree() {
        log.debug("Non-Functional Test 3 (Security)");
    }

    @Test
    @Tag("usability")
    void testFour() {
        log.debug("Non-Functional Test 4 (Usability)");
    }

}
```

4. Advanced tests - Tagging and filtering

mastering-junit5/junit5-tagging-filtering

- We can filter by tags when executing tests:

```
> mvn test -Dgroups=functional

[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.FunctionalTest
2021-04-22 10:56:08 [main] DEBUG
io.github.bonigarcia.FunctionalTest.testOne(33) -- Functional Test 1
2021-04-22 10:56:08 [main] DEBUG
io.github.bonigarcia.FunctionalTest.testTwo(38) -- Functional Test 2
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.116 s - in io.github.bonigarcia.FunctionalTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```


```
> mvn test -DexcludeGroups=functional

[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.NonFunctionalTest
2021-04-22 10:57:49 [main] DEBUG i.g.bonigarcia.NonFunctionalTest.testOne(35)
-- Non-Functional Test 1 (Performance/Load)
2021-04-22 10:57:49 [main] DEBUG i.g.bonigarcia.NonFunctionalTest.testTwo(42)
-- Non-Functional Test 2 (Performance/Stress)
2021-04-22 10:57:49 [main] DEBUG
i.g.bonigarcia.NonFunctionalTest.testThree(48) -- Non-Functional Test 3
(Security)
2021-04-22 10:57:49 [main] DEBUG i.g.bonigarcia.NonFunctionalTest.testFour(54)
-- Non-Functional Test 4 (Usability)
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.12 s
- in io.github.bonigarcia.NonFunctionalTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
```

4. Advanced tests - Tagging and filtering

mastering-junit5/junit5-tagging-filtering

- We can filter by tags when executing tests:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>${maven-surefire-plugin.version}</version>
            <configuration>
                <groups>functional</groups>
                <excludedGroups>non-functional</excludedGroups>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Maven™

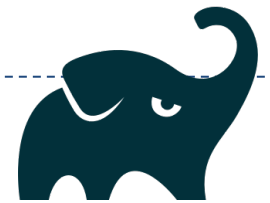
```
> mvn test
...
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.FunctionalTest
[main] DEBUG io.github.bonigarcia.FunctionalTest.testOne(33) --
Functional Test 1
[main] DEBUG io.github.bonigarcia.FunctionalTest.testTwo(38) --
Functional Test 2
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.118 s - in io.github.bonigarcia.FunctionalTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

4. Advanced tests - Tagging and filtering

mastering-junit5/junit5-tagging-filtering

- We can filter by tags when executing tests:

```
test {
    useJUnitPlatform {
        includeTags 'non-functional'
        excludeTags 'functional'
    }

    testLogging {
        events "passed", "skipped", "failed"
        showStandardStreams = true
    }
}

tasks.register("functionalTest", Test) {
    useJUnitPlatform {
        includeTags 'functional'
        excludeTags 'non-functional'
    }

    mustRunAfter check

    testLogging {
        events "passed", "skipped", "failed"
        showStandardStreams = true
    }
}
```



```
> gradle test

Task :junit5-tagging-filtering:test

NonFunctionalTest > testOne() STANDARD_OUT
    2021-06-20 15:32:38 [Test worker] DEBUG
i.g.bonigarcia.NonFunctionalTest.testOne(35) -- Non-Functional Test 1
(Performance/Load)

NonFunctionalTest > testOne() PASSED

NonFunctionalTest > testTwo() STANDARD_OUT
    2021-06-20 15:32:38 [Test worker] DEBUG
i.g.bonigarcia.NonFunctionalTest.testTwo(42) -- Non-Functional Test 2
(Performance/Stress)

NonFunctionalTest > testTwo() PASSED

NonFunctionalTest > testThree() STANDARD_OUT
    2021-06-20 15:32:38 [Test worker] DEBUG
i.g.bonigarcia.NonFunctionalTest.testThree(48) -- Non-Functior

NonFunctionalTest > testThree() PASSED

NonFunctionalTest > testFour() STANDARD_OUT
    2021-06-20 15:32:38 [Test worker] DEBUG
i.g.bonigarcia.NonFunctionalTest.testFour(54) -- Non-Functiona

NonFunctionalTest > testFour() PASSED

BUILD SUCCESSFUL in 3s
```

```
> gradle functionalTest

Task :junit5-tagging-filtering:functionalTest

FunctionalTest > testOne() STANDARD_OUT
    2021-06-20 15:35:20 [Test worker] DEBUG
io.github.bonigarcia.FunctionalTest.testOne(33) --
Functional Test 1

FunctionalTest > testOne() PASSED

FunctionalTest > testTwo() STANDARD_OUT
    2021-06-20 15:35:20 [Test worker] DEBUG
io.github.bonigarcia.FunctionalTest.testTwo(38) --
Functional Test 2

FunctionalTest > testTwo() PASSED

BUILD SUCCESSFUL in 3s
```

4. Advanced tests - Parameterized tests

- Parameterized tests reuse the same logic with different test data
- To implement this type of test, we need to add the `junit-jupiter-params` module in our project

Maven™

```
<dependencies>
   <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-params</artifactId>
      <version>${junit.jupiter.version}</version>
      <scope>test</scope>
   </dependency>
 </dependencies>
```

We do not need to include this dependency if we are already using the artifact `junit-jupiter` (since `junit-jupiter-params` is transitively pulled)

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter-params:${junitJupiterVersion}")
}
```

4. Advanced tests - Parameterized tests

- The steps to implement a parameterized test are:

1. Use the `@ParameterizedTest` annotation in the test declaration
2. Choose an argument provider:

Arguments provider	Description
`@ValueSource`	An array of primitive (e.g. `int`, `long`, `boolean`) values (or `String`)
`@EnumSource`	Enumerated values (`java.lang.Enum`)
`@MethodSource`	A static method of the class that provides a `Stream` of values
`@CsvSource`	Comma-separated values (CSV)
`@CsvFileSource`	Values in CSV format in a file located in the classpath
`@ArgumentsSource`	A class that implements the `org.junit.jupiter.params.provider.ArgumentsProvider` interface

4. Advanced tests - Parameterized tests

mastering-junit5/junit5-parameterized

```
class ValueSourcePrimitiveTypesParameterizedTest {

    static final Logger log = getLogger(lookup().lookupClass());

    @ParameterizedTest
    @ValueSource(ints = { 0, 1 })
    void testWithInts(int argument) {
        log.debug("Parameterized test with (int) argument: {}", argument);
        assertNotNull(argument);
    }

    @ParameterizedTest
    @ValueSource(longs = { 2L, 3L })
    void testWithLongs(long argument) {
        log.debug("Parameterized test with (long) argument: {}", argument);
        assertNotNull(argument);
    }

    @ParameterizedTest
    @ValueSource(doubles = { 4d, 5d })
    void testWithDoubles(double argument) {
        log.debug("Parameterized test with (double) argument: {}", argument);
        assertNotNull(argument);
    }

    @ParameterizedTest
    @ValueSource(strings = { "aa", "bb" })
    void testWithsStrings(String argument) {
        log.debug("Parameterized test with (String) argument: {}", argument);
        assertNotNull(argument);
    }

}
```

@ValueSource

Fork me on GitHub

wooclap

(activities 11 - 12)

4. Advanced tests - Parameterized tests

mastering-junit5/junit5-parameterized


```java
class EnumSourceParameterizedTest {

    static final Logger log = getLogger(lookup().lookupClass());

    @ParameterizedTest
    @EnumSource(TimeUnit.class)
    void testWithEnum(TimeUnit argument) {
        log.debug("Parameterized test with (TimeUnit) argument: {}", argument);
        assertNotNull(argument);
    }
}
```

@EnumSource

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.EnumSourceParameterizedTest
[main] DEBUG Parameterized test with (TimeUnit) argument: NANOSECONDS
[main] DEBUG Parameterized test with (TimeUnit) argument: MICROSECONDS
[main] DEBUG Parameterized test with (TimeUnit) argument: MILLISECONDS
[main] DEBUG Parameterized test with (TimeUnit) argument: SECONDS
[main] DEBUG Parameterized test with (TimeUnit) argument: MINUTES
[main] DEBUG Parameterized test with (TimeUnit) argument: HOURS
[main] DEBUG Parameterized test with (TimeUnit) argument: DAYS
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.177 s - in io.github.bonigarcia.EnumSourceParameterizedTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0
```

4. Advanced tests - Parameterized tests

mastering-junit5/junit5-parameterized


```java
class MethodSourceStringsParameterizedTest {

    static final Logger log = getLogger(lookup().lookupClass());

    static Stream<String> stringProvider() {
        return Stream.of("hello", "world");
    }

    @ParameterizedTest
    @MethodSource("stringProvider")
    void testWithStringProvider(String argument) {
        log.debug("Parameterized test with (String) argument: {}", argument);
        assertNotNull(argument);
    }

}
```

@MethodSource

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running
io.github.bonigarcia.MethodSourceStringsParameterizedTest
[main] DEBUG Parameterized test with (String) argument: hello
[main] DEBUG Parameterized test with (String) argument: world
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.161 s - in io.github.bonigarcia.MethodSourceStringsParameterizedTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

4. Advanced tests - Parameterized tests

mastering-junit5/junit5-parameterized


```java
class CsvSourceParameterizedTest {

    static final Logger log = getLogger(lookup().lookupClass());

    @ParameterizedTest
    @CsvSource({ "hello, 1", "world, 2", "'happy, testing', 3" })
    void testWithCsvSource(String first, int second) {
        log.debug("Parameterized test with (String) {} and (int) {} ", first,
                second);

        assertNotNull(first);
        assertNotEquals(0, second);
    }

}
```

@CsvSource

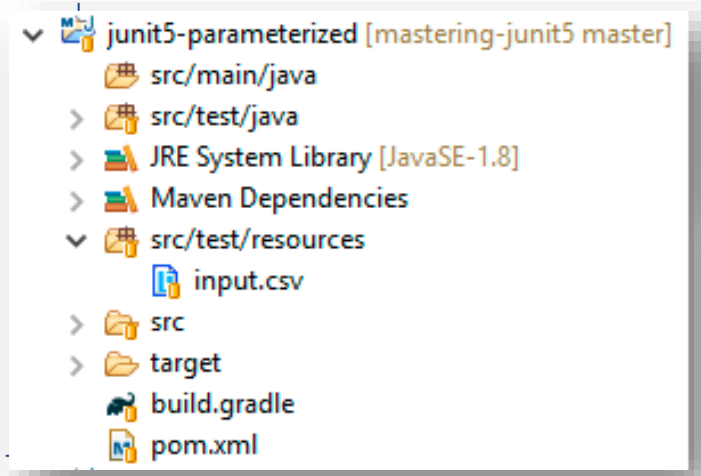
```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.CsvSourceParameterizedTest
[main] DEBUG Parameterized test with (String) hello and (int) 1
[main] DEBUG Parameterized test with (String) world and (int) 2
[main] DEBUG Parameterized test with (String) happy, testing and (int) 3
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.183 s - in io.github.bonigarcia.CsvSourceParameterizedTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

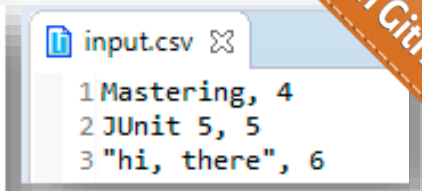
4. Advanced tests - Parameterized tests

mastering-junit5/junit5-parameterized

```
class CsvFileSourceParameterizedTest {

    static final Logger log = getLogger(lookup().lookupClass());

    @ParameterizedTest
    @CsvFileSource(resources = "/input.csv")
    void testWithCsvFileSource(String first, int second) {
        log.debug(
                "Yet another parameterized test with (String) {} and (int) {}
                first, second);

        assertNotNull(first);
        assertNotEquals(0, second);
    }

}
```

@CsvFileSource





Fork me on GitHub

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.CsvFileSourceParameterizedTest
[main] DEBUG Yet another parameterized test with (String) Mastering and (int) 4
[main] DEBUG Yet another parameterized test with (String) JUnit 5 and (int) 5
[main] DEBUG Yet another parameterized test with (String) hi, there and (int) 6
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.187 s - in
io.github.bonigarcia.CsvFileSourceParameterizedTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

4. Advanced tests - Parameterized tests

mastering-junit5/junit5-parameterized


```java
class ArgumentSourceParameterizedTest {

    static final Logger log = getLogger(lookup().lookupClass());

    @ParameterizedTest
    @ArgumentsSource(CustomArgumentsProvider1.class)
    void testWithArgumentsSource(String first, int second) {
        log.debug("Parameterized test with (String) {} and (int) {} ", first,
                second);

        assertNotNull(first);
        assertTrue(second > 0);
    }

}
```

@ArgumentsSource

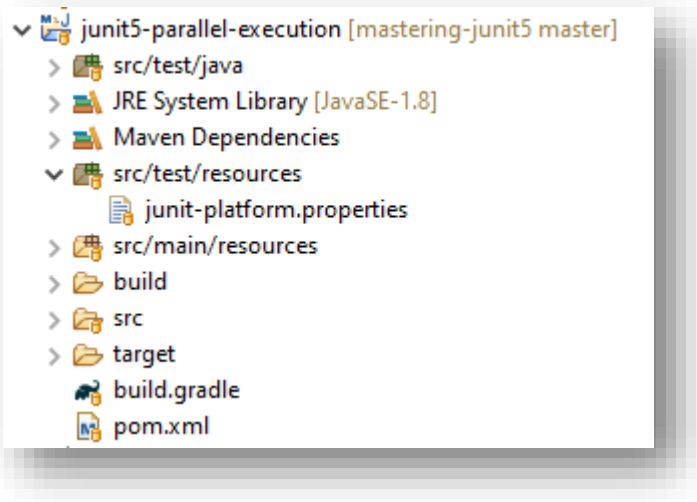
```java
public class CustomArgumentsProvider1 implements ArgumentsProvider {

    static final Logger log = getLogger(lookup().lookupClass());

    @Override
    public Stream<? extends Arguments> provideArguments(
            ExtensionContext context) {
        log.debug("Arguments provider [1] to test {}",
                context.getTestMethod().get().getName());

        return Stream.of(Arguments.of("hello", 1),
                Arguments.of("world", 2));
    }

}
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.github.bonigarcia.ArgumentSourceParameterizedTest
[main] DEBUG Arguments provider [1] to test testWithArgumentsSource
[main] DEBUG Parameterized test with (String) hello and (int) 1
[main] DEBUG Parameterized test with (String) world and (int) 2
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

4. Advanced tests - Parallel execution

- By default, JUnit Jupiter tests are run sequentially in a single thread
- Running tests in parallel is possible since Jupiter 5.3
- This feature is enabled setting the configuration key `junit.jupiter.execution.parallel.enabled` to `true`
- These configuration keys can be defined as properties in the file `junit-platform.properties` (it should be available in the project classpath)

These configuration keys can be overdried passing JVM arguments (`-Dkey=value`), or using configuration properties in Maven or Gradle



4. Advanced tests - Parallel execution

mastering-junit5/junit5-parallel-execution

- There are two parallel execution modes in Jupiter:
  - SAME_THREAD: Force execution in the same thread used by the parent
  - CONCURRENT: Use a different thread for the test execution
- It can be configured for class and test level using configuration keys:
  - `junit.jupiter.execution.parallel.mode.default = concurrent | same_thread`
  - `junit.jupiter.execution.parallel.mode.classes.default = concurrent | same_thread`

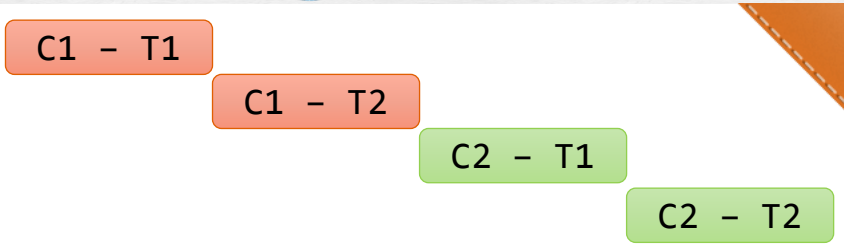
```
class ParallelExecution01Test {

    static final Logger log =
          getLogger(lookup().lookupClass());

    @Test
    void test01() {
        log.debug("01.test01");
    }

    @Test
    void test02() {
        log.debug("01.test02");
    }

}
```

```
class ParallelExecution02Test {

    static final Logger log =
          getLogger(lookup().lookupClass());

    @Test
    void test01() {
        log.debug("02.test01");
    }

    @Test
    void test02() {
        log.debug("02.test02");
    }

}
```

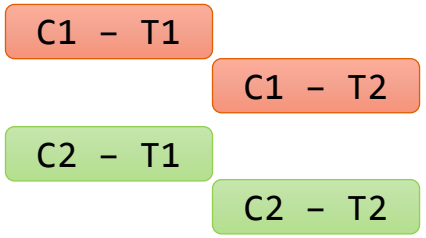
4. Advanced tests - Parallel execution

mastering-junit5/junit5-parallel-execution

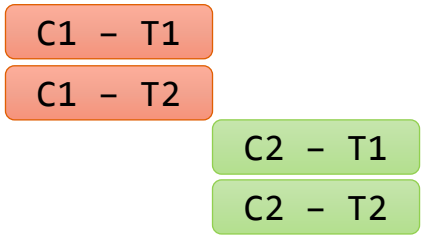

```
junit.jupiter.execution.parallel.enabled = true
junit.jupiter.execution.parallel.mode.default = same_thread
junit.jupiter.execution.parallel.mode.classes.default = same_thread
```



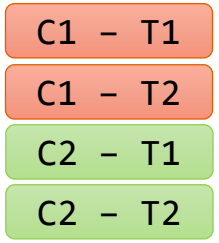

```
junit.jupiter.execution.parallel.enabled = true
junit.jupiter.execution.parallel.mode.default = same_thread
junit.jupiter.execution.parallel.mode.classes.default = concurrent
```




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




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



4. Advanced tests - Parallel execution

mastering-junit5/junit5-parallel-execution

- The size of the thread pool used by JUnit for parallel execution can be customized with the configuration key `junit.jupiter.execution.parallel.config.strategy`
- The possible values for this key are:
  - `dynamic` : use a thread pool based on the number of available processors multiplied by the factor given by the value of `junit.jupiter.execution.parallel.config.dynamic.factor` (`1` by default)
  - `fixed` : use a thread pool based on the number given by the value of `junit.jupiter.execution.parallel.config.fixed.parallelism`

wooclap

(activity 13)

Table of contents

5. Extension model

- The extension model provided by Jupiter allows to add new features on the top of the Jupiter programming model
- Extensions in Jupiter are implemented making use of so-called **extension points**, which are interfaces that allow to implemented different types of operations:
  - Enhance test lifecycle
  - Parameter resolution on test methods
  - Test templates
  - Conditional test execution
  - Exception handling
  - Manage test instances
  - Intercept invocation

5. Extension model - Extension points

- The **extension points** provided by Jupiter are:

Category	Description	Extension point(s)
Test lifecycle callbacks	Used to include custom logic in different moments of the test lifecycle	`BeforeAllCallback`, `BeforeEachCallback`, `BeforeTestExecutionCallback`, `AfterTestExecutionCallback`, `AfterEachCallback`, and `AfterAllCallback`
Parameter resolution	Used in those extensions that require dependency injection (i.e., parameters injected in test methods or constructors)	`ParameterResolver`
Test templates	Used to implement `@TestTemplate` tests (repeated depending on a given context)	`TestTemplateInvocationContextProvider`
Conditional test execution	Used to enable or disable tests depending on custom conditions	`ExecutionCondition`
Exception handling	Used to handle exceptions during the test and its lifecycle (i.e., before and after the test)	`TestExecutionExceptionHandler` and `LifecycleMethodExecutionExceptionHandler`
Test instance	Used to create and process test class instances	`TestInstanceFactory`, `TestInstancePostProcessor`, and `TestInstancePreDestroyCallback`
Intercepting invocations	Used to intercept calls to test code (and decide whether or not these calls proceed)	`InvocationInterceptor`

5. Extension model - Using extensions

- There are 3 ways to use an extension in Jupiter:

1. Declaratively, using the `@ExtendWith` annotation (can be used at the class or method level)
2. Programmatically, using the `@RegisterExtension` annotation (the difference is that we have an instance of the extension available in the tests class)
3. Automatically, using the Java service loading mechanism through the `java.util.ServiceLoader` class

5. Extension model - Using extensions

1. Declaratively (`@ExtendWith`):

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

@ExtendWith(MyExtension.class)
public class MyTest {

    @Test
    public void test1() {
        // ...
    }

    @Test
    public void test2() {
        // ...
    }

}
```

Declared at the class level, the extension will be registered for all tests in the class

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

public class MyTest {

    @ExtendWith(MyExtension.class)
    @Test
    public void test1() {
        // ...
    }

    @Test
    public void test2() {
        // ...
    }

}
```

Declared at test level, the extension will be registered only for said test

5. Extension model - Using extensions

2. Programmatically (`@RegisterExtension`)

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

public class MyTest {

    @RegisterExtension
    MyExtension myExtension = new MyExtension();

    @Test
    public void test1() {
        // ...
    }

    @Test
    public void test2() {
        // ...
    }

}
```

The instance of the extension registered through `@RegisterExtension` can be used programmatically for configuration of the extension or in the tests themselves

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

public class MyTest {

    @RegisterExtension
    static MyExtension myExtension = new MyExtension();

    @Test
    public void test1() {
        // ...
    }

    @Test
    public void test2() {
        // ...
    }

}
```

If declared as `static`, the extension point is limited at the instance level (`BeforeEachCallback`, `AfterEachCallback`, etc.)

5. Extension model - Using extensions

mastering-junit5/junit5-extension-model

3. Automatically (`java.util.ServiceLoader`)
   - To register an extension using this mechanism, first we declare the qualified name of the extension in the following file: `/META-INF/services/org.junit.jupiter.api.extension.Extension`
   - Then, we set to `true` the value of the configuration key `-Djunit.jupiter.extensions.autodetection.enabled`

```
> mvn test -Djunit.jupiter.extensions.autodetection.enabled=true
```

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${maven-surefire-plugin.version}</version>
    <configuration>
        <properties>
            <configurationParameters>
                junit.jupiter.extensions.autodetection.enabled=true
            </configurationParameters>
        </properties>
    </configuration>
</plugin>
```

```
> gradle test -Djunit.jupiter.extensions.autodetection.enabled=true
```

```
test {
    useJUnitPlatform()

    testLogging {
        events "passed", "skipped", "failed"
    }

    testLogging.showStandardStreams = true

    systemProperty 'junit.jupiter.extensions.autodetection.enabled', 'true'
}
```

Maven™

5. Extension model - Third-party extensions

mastering-junit5/junit5-mockito

- **Mockito** extension (unit tests):

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.verifyNoMoreInteractions;
import static org.mockito.Mockito.verifyZeroInteractions;
import static org.mockito.Mockito.when;

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)
class LoginControllerLoginTest {
    // Mocking objects
    @InjectMocks
    LoginController loginController;
    @Mock
    LoginService loginService;
    // Test data
    UserForm userForm = new UserForm("foo", "bar");
```

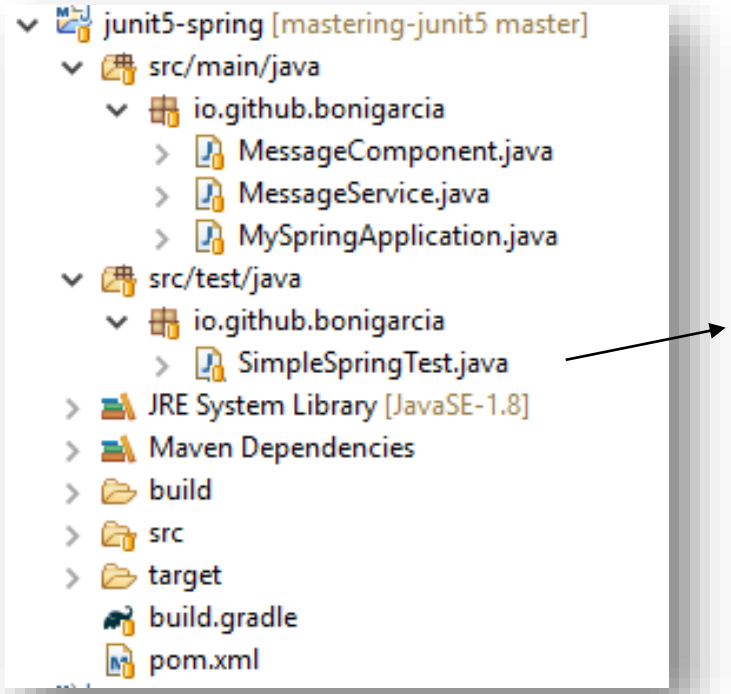
```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.LoginControllerLoginTest
LoginController.login UserForm [username=foo, password=bar]
LoginController.login UserForm [username=foo, password=bar]
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.739 sec -
in io.github.bonigarcia.LoginControllerLoginTest

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

```
    @Test
    void testLoginOk() {
        // Setting expectations (stubbing methods)
        when(loginService.login(userForm)).thenReturn(true);
        // Exercise SUT
        String reseponseLogin = loginController.login(userForm);
        // Verification
        assertEquals("OK", reseponseLogin);
        verify(loginService).login(userForm);
        verifyNoMoreInteractions(loginService);
    }

    @Test
    void testLoginKo() {
        // Setting expectations (stubbing methods)
        when(loginService.login(userForm)).thenReturn(false);

        // Exercise SUT
        String reseponseLogin = loginController.login(userForm);

        // Verification
        assertEquals("KO", reseponseLogin);
        verify(loginService).login(userForm);
        verifyZeroInteractions(loginService);
    }

}
```

5. Extension model - Third-party extensions

mastering-junit5/junit5-spring

- **Spring** extension (integration tests):

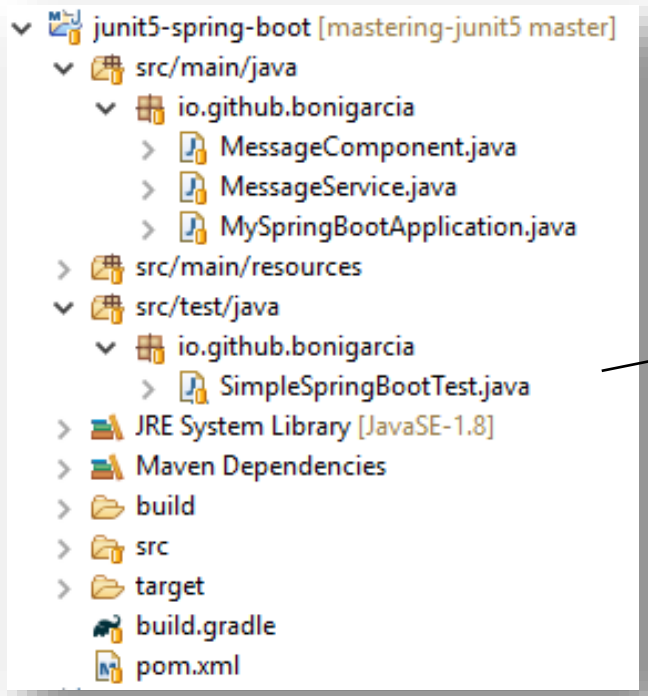



```java
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = { MySpringApplication.class })
class SimpleSpringTest {

    @Autowired
    public MessageComponent messageComponent;

    @Test
    public void test() {
        assertEquals("Hello world!", messageComponent.getMessage());
    }

}
```

5. Extension model - Third-party extensions

mastering-junit5/junit5-spring-boot

- **Spring** extension (integration tests):




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

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(SpringExtension.class)
@SpringBootTest
class SimpleSpringBootTest {
    @Autowired
    public MessageComponent messageComponent;

    @Test
    public void test() {
        assertEquals("Hello world!", messageComponent.getMessage());
    }
}
```

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.SimpleSpringBootTest
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.0.M3)
2017-08-02 00:39:47.904  INFO 14124 --- [           main] i.g.bonigarcia.SimpleSpringBootTest      : Starting SimpleSpringBootTest
on LAPTOP-T9O4060I with PID 14124 (started by boni in D:\dev\mastering-junit5\junit5-spring-boot)
2017-08-02 00:39:47.912  INFO 14124 --- [           main] i.g.bonigarcia.SimpleSpringBootTest      : No active profile set, falling
 back to default profiles: default
2017-08-02 00:39:48.357  INFO 14124 --- [           main] i.g.bonigarcia.MySpringBootApplication   : *** Hello world! ***
2017-08-02 00:39:48.426  INFO 14124 --- [           main] i.g.bonigarcia.SimpleSpringBootTest      : Started SimpleSpringBootTest i
n 0.716 seconds (JVM running for 2.045)
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.395 sec - in io.github.bonigarcia.SimpleSpringBootTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

5. Extension model - Third-party extensions

mastering-junit5/junit5-selenium

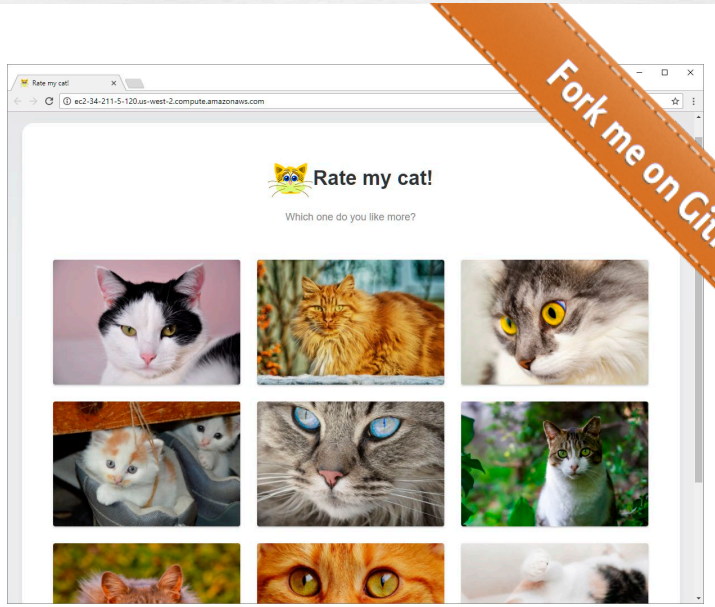
- **Selenium** extension (end-to-end tests):

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

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

import io.github.bonigarcia.seljup.SeleniumJupiter;

@ExtendWith(SeleniumJupiter.class)
public class LocalWebDriverTest {

    @Test
    public void testWithChrome(ChromeDriver chrome) {
        chrome.get("https://bonigarcia.github.io/selenium-jupiter/");

        assertTrue(chrome.getTitle().startsWith("Selenium-Jupiter"));
    }

    @Test
    public void testWithFirefox(FirefoxDriver firefox) {
        firefox.get("http://www.seleniumhq.org/");

        assertTrue(firefox.getTitle().startsWith("Selenium"));
    }

}
```

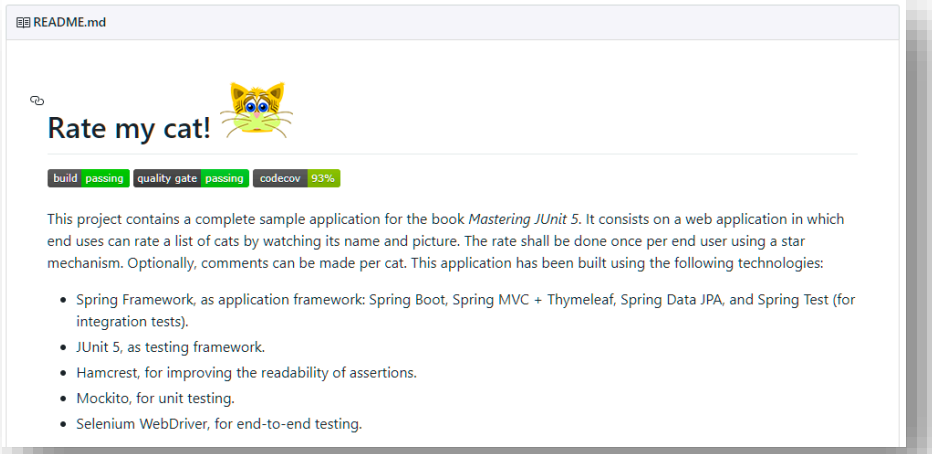
https://bonigarcia.github.io/selenium-jupiter/

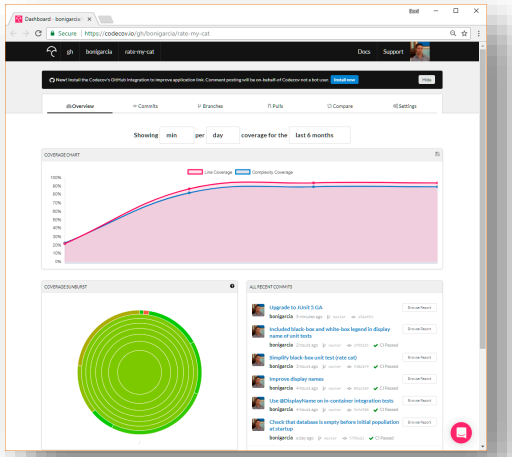
5. Extension model - Third-party extensions

- Complete project example:
  - Web application implemented with Spring-Boot
  - Unit testing with Mockito
  - Integration tests with Spring
  - End-to-end tests with Selenium WebDriver
  - Running tests using GitHub Actions
  - Code analysis using SonarCloud
  - Coverage analysis using Codedov



https://github.com/bonigarcia/rate-my-cat





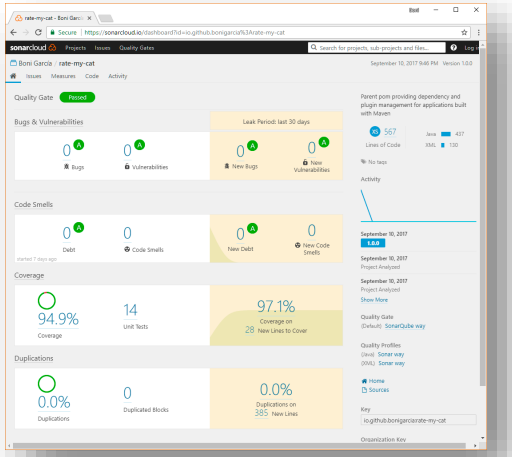


Table of contents

6. Other features

- Repeated tests
- Display names
- Nested tests
- Dynamic tests
- Ordered tests
- Test interfaces and default methods
- Declarative timeouts
- Built-in extensions
- Migration from JUnit 4
- …

https://junit.org/junit5/docs/current/user-guide/

Table of contents

7. Final remarks

- Learning outcomes recap:

1. Describe the modular architecture of JUnit 5
   - Platform (generic executor for tests within the JVM)
   - Vintage (engine for legacy tests, i.e. JUnit 3 and 4)
   - Jupiter (engine for the new programming and extension model, i.e. JUnit 5)
2. Execute basic test cases with JUnit 5
   - Supported IDEs: IntelliJ IDEA, Eclipse, Visual Studio, NetBeans
   - Supported build tools: Maven, Gradle, Ant
   - Test lifecycle: `@BeforeAll`, `@BeforeEach`, `@Test`, `@AfterEach`, `@AfterAll`
3. Develop advanced test cases using Jupiter
   - Assertions, assumptions, disabled tests, tagging and filtering, parameterized tests, parallel execution, …

https://github.com/junit-team/junit5/

A deep dive into JUnit 5

Thanks a lot!

Ministry of Testing
99 Minute Workshop

Boni García

 boni.garcia@uc3m.es  http://bonigarcia.github.io/

 @boni_gg  https://github.com/bonigarcia