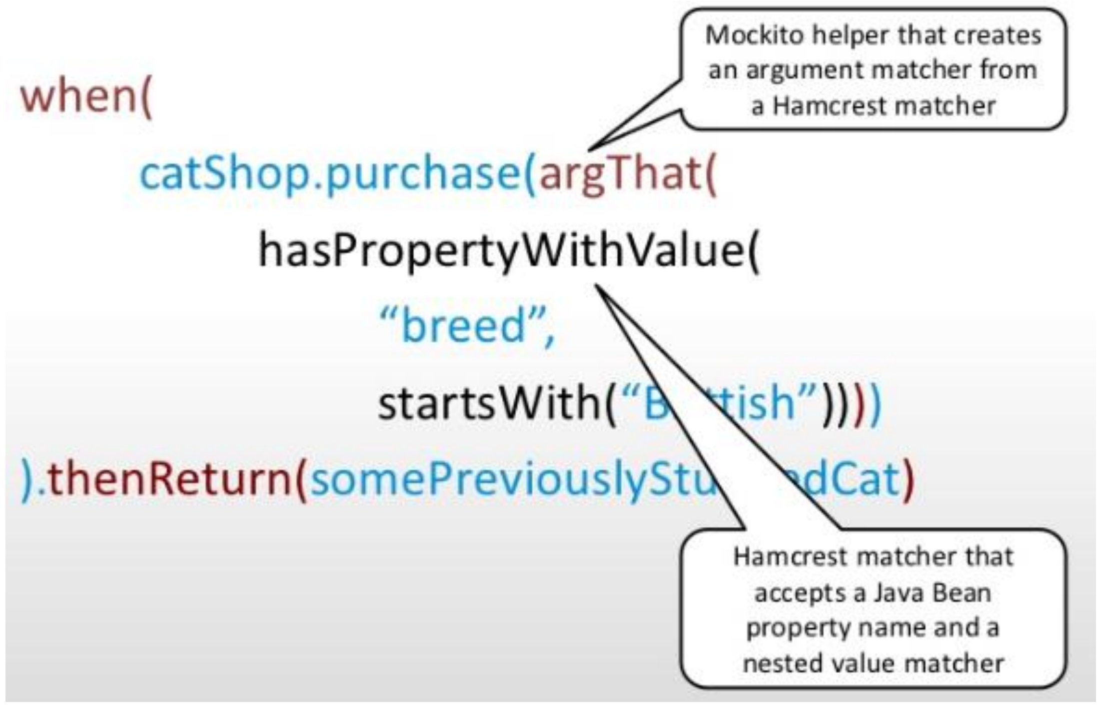
Hamcrest Matchers and jUnit5

Azat Satklichov
azats@seznam.cz,
http://sahet.net/htm/java.html,
https://github.com/azatsatklichov/java-and-ts-tests/tree/master/javatesting

Hamcrest has the target

to make tests self-explanatory and easy to read
/
to make assert statements read like natural language



```
when(
    catShop.purchase(argThat(
        hasPropertyWithValue(
            “breed”,
            startsWith(“B[illegible]tish”))))
).thenReturn(somePreviouslyStu[illegible]dCat)
```

Mockito helper that creates an argument matcher from a Hamcrest matcher
Hamcrest matcher that accepts a Java Bean property name and a nested value matcher

```
<!-- objenesis
Objenisis: Java already supports this
dynamic instantiation of classes using
Class.newInstance(). However, this only
works if the class has an appropriate
constructor.
must require a default constructor.
Objenesis aims to overcome these
restrictions by bypassing the constructor
on object instantiation.
Needing to instantiate an object without
calling the constructor is a fairly
specialized task, however there are
certain cases when this is useful:
-->
```

Using AssertThat Over Other Assert Methods

1-generation: assert(logical statement), **assert(x==y)**

2-generation: traditional assert statements: **verb**, **object,** and **subject** (assert equals expected actual) , **assertEquals(expected, actual);**

3-generation: Hamcrest is typically viewed as a third generation matcher framework.

new syntax, **subject**, **verb**, and **object** (asset that actual is expected)

assertThat(actual, is(equalTo(expected)));

http://hamcrest.org/JavaHamcrest/index

Current version: JavaHamcrest 2.2junit5-uses 2.1, .. , junit4 uses version 1.3,

We use: http://hamcrest.org/JavaHamcrest/javadoc/1.3/

Common Core Matchers

- *is(T)* and *is(Matcher<T>), equalTo(T)*
- *not(T)* and *not(Matcher<T>)*
- *nullValue()* and *nullValue(Class<T>)*, *notNullValue()* and *notNullValue(Class<T>)*
- *instanceOf(Class<?>)*, *isA(Class<T> type)*, *sameInstance()*
- *any(Class<T>), allOf(Matcher<? extends T>…) and anyOf(Matcher<? extends T>…)*
- *hasItem(T)* and *hasItem(Matcher<? extends T>)*
- *hasSize, isIn, , isOneOf, …*

Combining matchers

- *both(Matcher<? extends T>) and either(Matcher<? extends T>)*
- allOf, anyOf, everyItem, …

String Comparison

- containsString, containsStringIgnoringCase, startsWith, endsWithIgnoringCase…

Custom Matchers

***TypeSafeMatcher** - extend this class to create a cutom matcher .*

- E.g. onlyDigitsMatcher, perfectNumberMatcher, divsibleBy, RegEx..... COBOL_KEYWORDS,..

```java
public class PerfectNumberMatcher extends TypeSafeMatcher<Integer> {

  @Override
  protected boolean matchesSafely(Integer num) {
    return isPerfectNumber(num);
  }

  @Override
  public void describeTo(Description description) {
    description.appendText("only digits");
  }
}
```

Combining Matchers, Grouping Matchers, …

Object and Bean Matchers

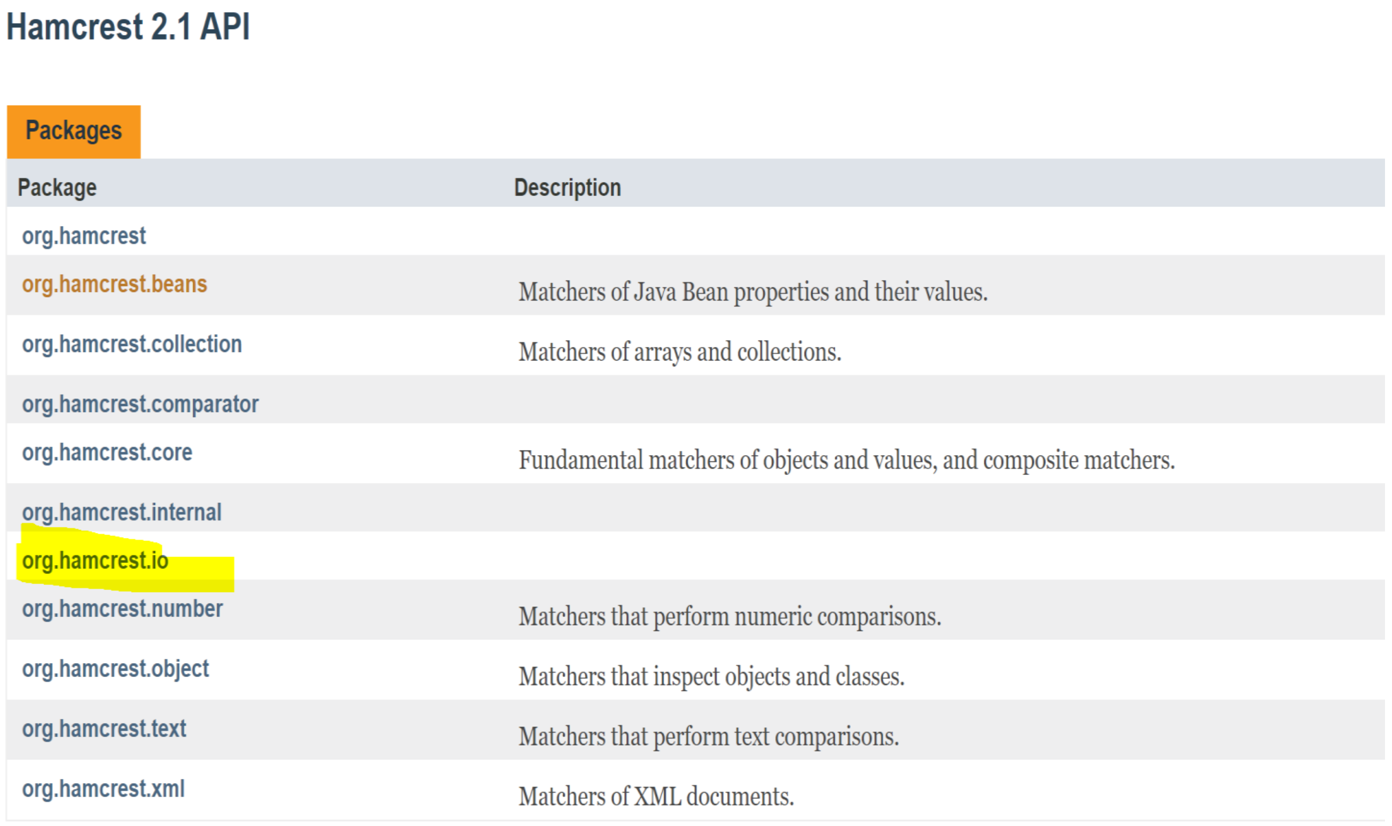
- *hasToString* - `assertThat(processed.getResult().get(0), hasToString("abc "));`
- *typeCompatibleWith* - represents an *is-a* relationship

Bean matchers are very useful to check POJO conditions

- *hasProperty*
- *samePropertyValuesAs*
- *getPropertyDescriptor, propertyDescriptorsFor*

File Matchers

- *aFileNamed(),* aFileWithSize
- aWritableFile, aReadableFile,
- nExistingDirectory(),
- anExistingFileOrDirectory()



Hamcrest 2.1 API

Packages

Package	Description
org.hamcrest	
org.hamcrest.beans	Matchers of Java Bean properties and their values.
org.hamcrest.collection	Matchers of arrays and collections.
org.hamcrest.comparator	
org.hamcrest.core	Fundamental matchers of objects and values, and composite matchers.
org.hamcrest.internal	
org.hamcrest.io	
org.hamcrest.number	Matchers that perform numeric comparisons.
org.hamcrest.object	Matchers that inspect objects and classes.
org.hamcrest.text	Matchers that perform text comparisons.
org.hamcrest.xml	Matchers of XML documents.

```
@Test
public void whenVerifyingFileSize_thenCorrect() {
    File file = new File("src/test/resources/test1.in");

    assertThat(file, aFileWithSize(11));
    assertThat(file, aFileWithSize(greaterThan(1L)));;
}
```

Text Matchers

- *equalToIgnoringCase, equalToIgnoringWhiteSpace*
- *blankString()); blankOrNullString, blankOrNullString());* (NEW Version)
- emptyString()); emptyOrNullString, emptyOrNullString()); (NEW Version)

Pattern Matchers (NEW Version) – matchesPattern

Sub-String Matchers -

- *containsString, containsStringIgnoringCase, stringContainsInOrder, endsWithIgnoringCase,,,*

Number Matchers

- **Proximity Matchers – *closeTo (Double, BigDecimal)***
- **Order Matchers -** *comparesEqualTo, greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo*
- **Order Matchers With *String* Values**
- **Order Matchers With *LocalDate* Values**
- **Order Matchers With *Custom Classes***
- **NaN Matcher -** org.hamcrest.number.IsNaN (New version)

```
assertThat(zero / zero, is(notANumber()));
```

AssertJ - need ?

Why jUnit 5

- *jUnit 4 based on Java 7 (**can't use like Java8 lambdas** for lazy evaluation, etc), and **single jar**.*
- *jUnit5, (Sep.2017, runs on Java 8 or >) is **more granular** (multiple libraries), can import only necessary part. Faster, ..*
- *Consists of three subprojects:* **JUnit 5 = *JUnit Platform + JUnit Jupiter + JUnit Vintage***
- **JUnit Jupiter Engine -** module includes new programming and extension models for writing JUnit 5 tests.
- **Junit Vintage Engine -** Supports running JUnit 3 and JUnit 4 based tests on the JUnit 5 platform.
- In Junit 4, there is no integration support for 3rd party plugins and IDEs. They have to rely on reflection.
- **JUnit Platform** - dedicated sub-project for integration purpose. It defines the TestEngine API for developing a testing framework that runs on the platform. This module scopes all the extension frameworks we might be interested in test execution, discovery, and reporting.
- JUnit 5 can use more than one extension at a time (***multiple test runners***), which JUnit 4 could not (only one runner at a time). This means you can easily combine the Spring extension with other extensions (such as your own custom extension, life cycle callbacks, param-resolver, ..). *e.g. jUnit4, one-runner SpringJUnit4ClassRunner or Parameterized*
- In Junit 4, @RunWith(Suite.class) and @Suite annotation
- In Junit 5, @RunWith(JUnitPlatform.class), @SelectPackages and @SelectClasses
- Features for describing, organizing, and executing tests. For instance, tests get better display names and can be organized hierarchically. @DisplayName
- https://junit.org/junit5/docs/current/user-guide/, (doesn't support test suites and parallel execution yet) https://github.com/junit-team/junit5/issues/744, https://github.com/junit-team/junit5/issues/964
- https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests

jUnit 5 Modules

JUnit Platform

JUnit Jupiter

JUnit Vintage

jUnit 5 Platform

junit-platform-commons

junit-platform-console

junit-platform-console-standalone

junit-platform-engine

junit-platform-launcher

junit-platform-runner

junit-platform-suite-api

junit-platform-surefire-provider

junit-platform-gradle-plugin

jUnit Jupiter

junit-jupiter-api

junit-jupiter-engine

junit-jupiter-params

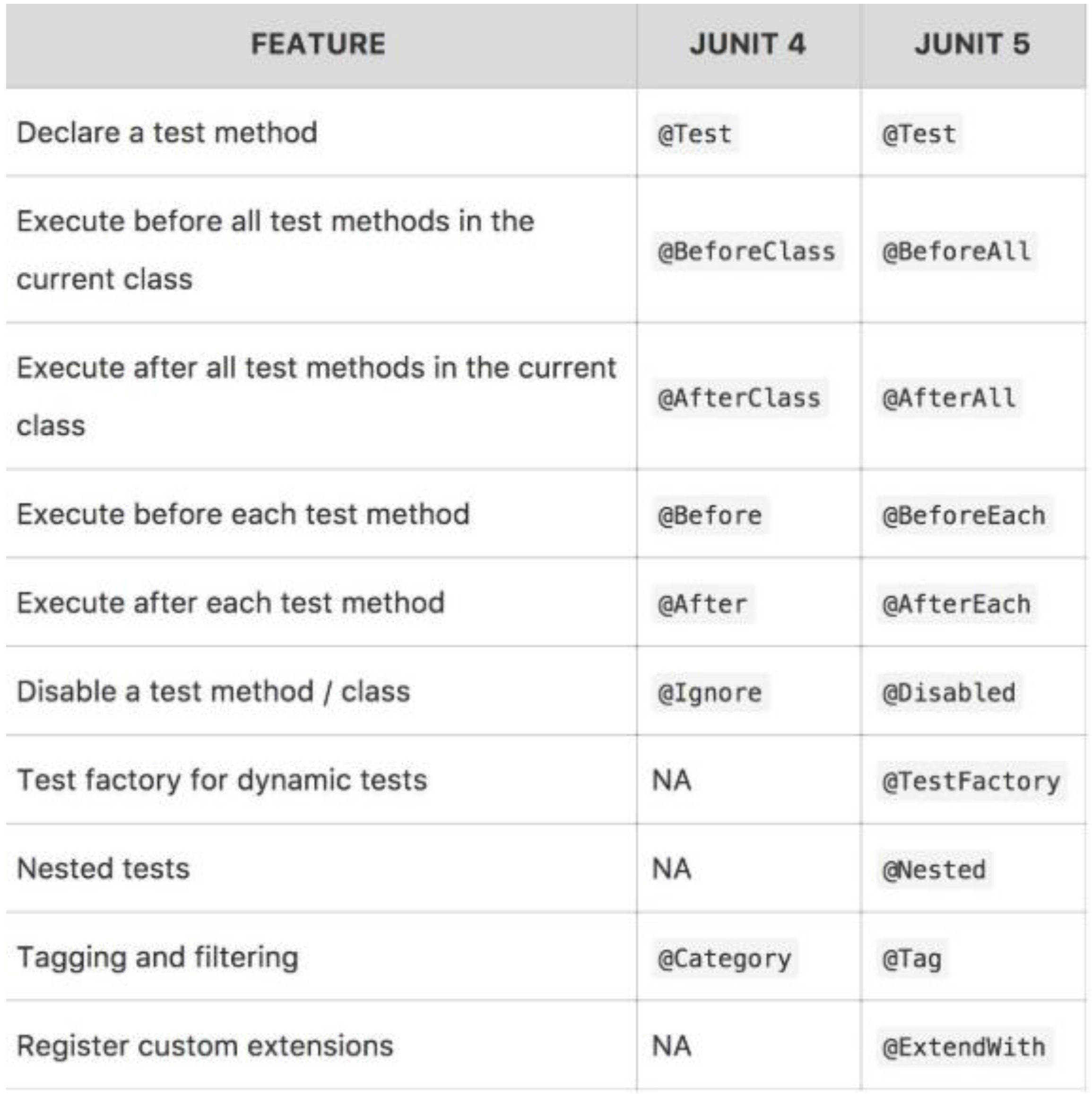
junit-jupiter-migrationsupport

jUnit Vintage

junit-vintage-engine

JUnit 3 or 4 JARs

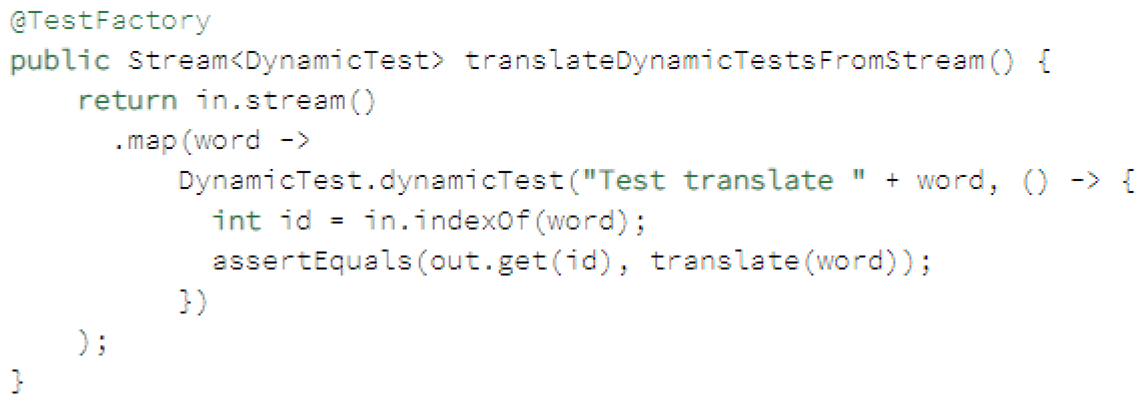
- *jUnit 5 changes on Annotations*



FEATURE	JUNIT 4	JUNIT 5
Declare a test method	`@Test`	`@Test`
Execute before all test methods in the current class	`@BeforeClass`	`@BeforeAll`
Execute after all test methods in the current class	`@AfterClass`	`@AfterAll`
Execute before each test method	`@Before`	`@BeforeEach`
Execute after each test method	`@After`	`@AfterEach`
Disable a test method / class	`@Ignore`	`@Disabled`
Test factory for dynamic tests	NA	`@TestFactory`
Nested tests	NA	`@Nested`
Tagging and filtering	`@Category`	`@Tag`
Register custom extensions	NA	`@ExtendWith`

New Features

- **Display names -** @DisplayName
- **New Assertions**
- **Nested Tests** (express relationships on groups of tests)
- **Parameterized Tests (jUnit4 – Constructor, Field Injection based, @Theory, @DataPoints)**
- **Conditional tests execution -** @Disabled @EnabledOnOs, @DisabledOnOs, @EnabledOnJre, @DisabledOnJre, @EnabledIfSystemProperty, @EnabledIf
- **Test Templates** (@TestTemplate, e.g. similar steps)
- **Meta-annotations (own annotations, Tag..Fast, Prod, ..)**
- **Dynamic Tests -** @TestFactory (not @Test, no life-cycle)



```
@TestFactory
public Stream<DynamicTest> translateDynamicTestsFromStream() {
    return in.stream()
      .map(word ->
          DynamicTest.dynamicTest("Test translate " + word, () -> {
            int id = in.indexOf(word);
            assertEquals(out.get(id), translate(word));
          })
    );
}
```

@DisplayName("☀"), @Rule (ErrorCollector to continue execution on failure) @TempDir) and *@ClassRule, @ExtendWith, @TempDir*

@RepetaedTest(5), @Ex/IncludeTags, @TestInstance [LifeCycle.PER_METHOD (the default). The other is *LifeCycle.PER_CLASS]*

- *Assertions - to validate expected and actual values*

In Junit 4, org.junit.Assert has all assert methods to validate expected and resulted outcomes.

They accept extra parameter for error message as FIRST argument in method signature. e.g.

```
public static void assertEquals(long expected, long actual)
public static void assertEquals(String message, long expected, long actual)
```

*Optional message that an assertion would be printed if it failed is now the last. Also the Supplier (for **lazy evaluation**)*

JUnit 5 assertions methods also have overloaded methods to support passing error message to be printed in case test fails e.g.

```
public static void assertEquals(long expected, long actual)
public static void assertEquals(long expected, long actual, String message)
public static void assertEquals(long expected, long actual, Supplier messageSupplier)
```

```
@Test public void shouldFailWhenNumbersAreDifferent () {
   Assertions.assertTrue(19 == 23,
                         () -> "Numbers" + 19 + " and " + " are not equal ");
}
```

Group Assertions

```
  @Test public void shouldAssertAll() {
      List<Integer> list = List.of(1, 2, 3);
     Assertions.assertAll("Not ordered list",
               () -> Assertions.assertEquals(list.get(0).intValue(), 1),
               () -> Assertions.assertEquals(list.get(1).intValue(), 2),
               () -> Assertions.assertEquals(list.get(2).intValue(), 3));
  }
```

jUnit5 New Assertions

- `assertIterableEquals()` performs a deep verification of two iterables using `equals()`.
- `assertLinesMatch()` verifies that two lists of strings match; it accepts regular expressions in the `expected` argument.
- `assertAll()` groups multiple assertions together. The added benefit is that all assertions are performed, even if individual assertions fail.
- `assertThrows()` and `assertDoesNotThrow()` have replaced the `expected` property in the `@Test` annotation.

```
To test long running tasks:
assertTimeout(),assertTimeoutPreemptively()
```

For `assertTimeoutPreemptively()` `Executable` or `ThrowingSupplier` will be preemptively aborted if the timeout is exceeded, for assertTimeout() not

jUnit 4

```
//multiple runners was problematic, usually required chaining or using an @Rule.
@RunWith(SpringJUnit4ClassRunner.class) public class
MyControllerTest {
  // ...
}
```

```
@Test(expected = ArithmeticException.class)
 void shouldRaiseAnException() throws ArithmeticException {
   // ...
}

@Test void shouldThrowException() {
Try {
  //code may throw exception
  fail(“Not supported");
} catch (UnsupportedOperationException e) {
  assertEquals("Not supported“, exception.getMessage());
}
}
```

```
 @Test(timeout = 10)
  void shouldFailDuetoTimeout() throws InterrptedException {
    Thread.sleep(100);
 }
```

jUnit 5

```
//JUnit 5, you include the Spring extension instead. @ExtendWith is repetable – multiple extensians can be used
@ExtendWith(SpringExtension.class) class MyControllerTest {
 // ...
}
```

```
@Test
void shouldRaiseAnException() throws ArithmeticException {
   Assertions.assertThrows(ArithmeticException.class, () -> {
       //..
   });
}

@Test void shouldThrowException() {
Throwable exception = assertThrows(UnsupportedOperationException.class,
  () -> {
    throw new UnsupportedOperationException("Not supported");
});
   assertEquals(exception.getMessage(), "Not supported");
}
```

```
@Test
void shouldFailDuetoTimeout() throws InterrptedException {
Assertions.assertTimeout(Duration.ofMillies(10),  () ->
Thread.sleep(100));
}
```

- *Assumptions - for stating assumptions about the condition in which a test is meaningful. Assumptions are used whenever it does not make sense to continue execution of a given test method. Don't result in test failure like assertions. Abort the test (seems like executed but actually test is ignored, watch log).*

- *Junit 4, org.junit.Assume contains methods for stating assumptions:*

assumeFalse(), assumeNoException(), assumeNotNull(), assumeThat(), assumeTrue()

```
@Test public void testNothingInParticular() throws Exception {
   Assume.assumeThat("foo", is("bar"));
   assertEquals(...);
}
```

- *In Junit 5, org.junit.jupiter.api.Assumptions contains methods for stating assumptions:* The same assumptions exist, but they now support BooleanSupplier as well as Hamcrest matchers to match conditions

assumeFalse(), assumingThat(), assumeTrue()

```
@Test void testNothingInParticular() throws Exception {
  Assumptions.assumingThat("foo".equals(" bar"), () -> {
     assertEquals(...);
  });
}
```

- *Tagging and Filtering* *- To group tests jUnit4 was using @Category, replaced with @Tag in jUnit5*
- *Dynamic Tests (JUnit 5 Dynamic tests functionality can be achieved by parameterized tests.)*

```
@Tag("annotations")
@Tag("junit5")
@RunWith(JUnitPlatform.class)public class AnnotationTestExampleTest {   /*...*/}
```

We can include/exclude particular tags using the maven-surefire-plugin:

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <configuration>
                <properties>
                    <includeTags>junit5</includeTags>
                </properties>
            </configuration>
        </plugin>
    </plugins>
</build>
```

- *Junit 4, org.junit.Assume contains methods for stating assumptions:*

assumeFalse(), assumeNoException(), assumeNotNull(), assumeThat(), assumeTrue()

```
public class MyUtils {

        public static int add(int x, int y) {
                return x+y;
        }
}
```

```
@TestFactory Stream<DynamicTest>  dynamicTestsExample() {
  List<Integer> input1List = Arrays.asList(1,2,3);
  List<Integer> input2List = Arrays.asList(10,20,30);

   List<DynamicTest> dynamicTests = new ArrayList<>();

    for(int i=0; i < input1List.size(); i++) {
      int x = input1List.get(i);
       int y = input2List.get(i);
      DynamicTest dynamicTest  =
         dynamicTest("Dynamic Test for MyUtils.add("+x+","+y+")",
                () ->{assertEquals(x+y,MyUtils.add(x,y));});
        dynamicTests.add(dynamicTest);
    }
    return dynamicTests.stream();
}
```

jUnit 4 Parameterized Test

```
@Parameters public static Collection<Object[]> data() {
```

Or via feeding input-data @Theory, @DataPoint

```
//Example
@RunWith( Parameterized.class )
public class FooInvariantsTest {

@Parameterized.Parameters public static Collection<Object[]> data(){
  return new Arrays.asList(
       new Object[]{ new CsvFoo() ),
      new Object[]{ new SqlFoo() ),
      new Object[]{ new XmlFoo() ), );
}

private Foo fooUnderTest; public FooInvariantsTest( Foo fooToTest ){
  fooUnderTest = fooToTest;

}

@Test public void testInvariant1(){
...
}

@Test public void testInvariant2(){
 ...
 }
}
```

jUnit 5 Parameterized Test

//JUnit 5 doesn't provide the exact same features than those provided by JUnit 4.

```
private static Stream<Arguments> data() {
```

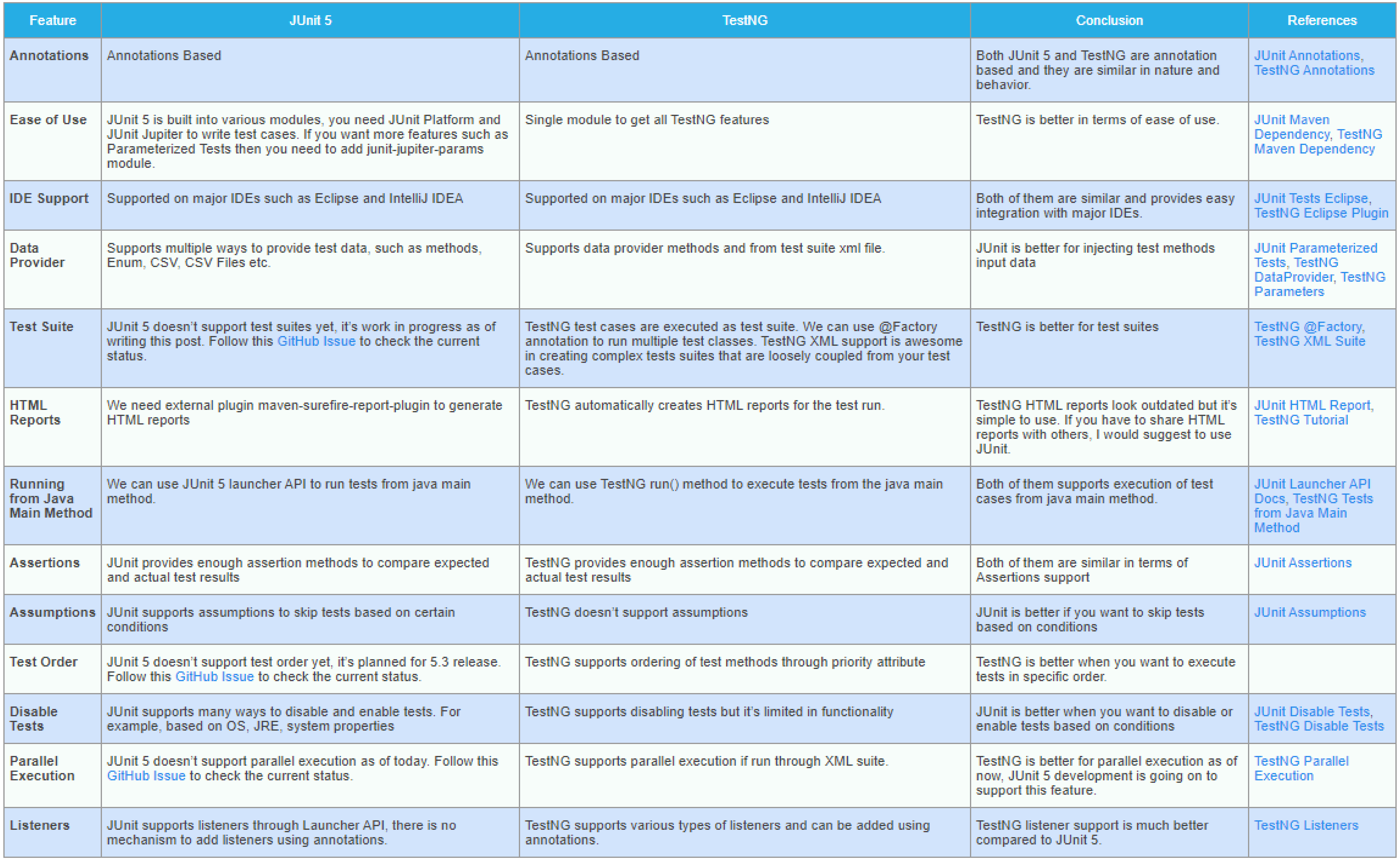
1. `Collection<Object[]>` is become `Stream<Arguments>` that provides more flexibility.
2. `@MethodSource, @ValueSource, @CsvSource, @CsvFileSource, @EnumSource, @argumentsSource`
3. `Null and Empty sources:  @NullSource, #EmptySource, @NullAndEmptySource,`

```
@ParameterizedTest
@NullSource @EmptySource @ValueSource(strings = { " ", " ", "\t", "\n" })
void nullEmptyAndBlankStrings(String text) {
       assertTrue(text == null || text.trim().isEmpty());
}
//Example
public class FooInvariantsTest {

private static Stream<Argument> data() {
  return Stream.of(
      Arguments.of(new CsvFoo()),
      Arguments.of(new SqlFoo() ));
}

@ParameterizedTest
@MethodSource("data")
public void testInvariant1(){
...
}

@ParameterizedTest
@MethodSource("data")
public void testInvariant2(){
 ...
 }
}
```

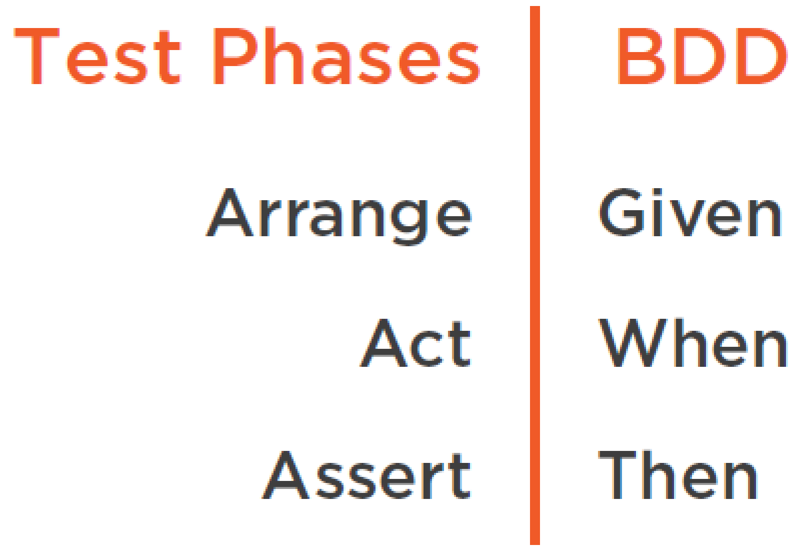


Feature	JUnit 5	TestNG	Conclusion	References	
Annotations	Annotations Based	Annotations Based	Both JUnit 5 and TestNG are annotation based and they are similar in nature and behavior.	JUnit Annotations, TestNG Annotations	↔
Ease of Use	JUnit 5 is built into various modules, you need JUnit Platform and JUnit Jupiter to write test cases. If you want more features such as Parameterized Tests then you need to add junit-jupiter-params module.	Single module to get all TestNG features	TestNG is better in terms of ease of use.	JUnit Maven Dependency, TestNG Maven Dependency	→
IDE Support	Supported on major IDEs such as Eclipse and IntelliJ IDEA	Supported on major IDEs such as Eclipse and IntelliJ IDEA	Both of them are similar and provides easy integration with major IDEs.	JUnit Tests Eclipse, TestNG Eclipse Plugin	↔
Data Provider	Supports multiple ways to provide test data, such as methods, Enum, CSV, CSV Files etc.	Supports data provider methods and from test suite xml file.	JUnit is better for injecting test methods input data	JUnit Parameterized Tests, TestNG DataProvider, TestNG Parameters	←
Test Suite	JUnit 5 doesn't support test suites yet, it's work in progress as of writing this post. Follow this GitHub Issue to check the current status.	TestNG test cases are executed as test suite. We can use @Factory annotation to run multiple test classes. TestNG XML support is awesome in creating complex tests suites that are loosely coupled from your test cases.	TestNG is better for test suites	TestNG @Factory, TestNG XML Suite	→
HTML Reports	We need external plugin maven-surefire-report-plugin to generate HTML reports	TestNG automatically creates HTML reports for the test run.	TestNG HTML reports look outdated but it's simple to use. If you have to share HTML reports with others, I would suggest to use JUnit.	JUnit HTML Report, TestNG Tutorial	←
Running from Java Main Method	We can use JUnit 5 launcher API to run tests from java main method.	We can use TestNG run() method to execute tests from the java main method.	Both of them supports execution of test cases from java main method.	JUnit Launcher API Docs, TestNG Tests from Java Main Method	↔
Assertions	JUnit provides enough assertion methods to compare expected and actual test results	TestNG provides enough assertion methods to compare expected and actual test results	Both of them are similar in terms of Assertions support	JUnit Assertions	↔
Assumptions	JUnit supports assumptions to skip tests based on certain conditions	TestNG doesn't support assumptions	JUnit is better if you want to skip tests based on conditions	JUnit Assumptions	←
Test Order	JUnit 5 doesn't support test order yet, it's planned for 5.3 release. Follow this GitHub Issue to check the current status.	TestNG supports ordering of test methods through priority attribute	TestNG is better when you want to execute tests in specific order.		→
Disable Tests	JUnit supports many ways to disable and enable tests. For example, based on OS, JRE, system properties	TestNG supports disabling tests but it's limited in functionality	JUnit is better when you want to disable or enable tests based on conditions	JUnit Disable Tests, TestNG Disable Tests	←
Parallel Execution	JUnit 5 doesn't support parallel execution as of today. Follow this GitHub Issue to check the current status.	TestNG supports parallel execution if run through XML suite.	TestNG is better for parallel execution as of now, JUnit 5 development is going on to support this feature.	TestNG Parallel Execution	→
Listeners	JUnit supports listeners through Launcher API, there is no mechanism to add listeners using annotations.	TestNG supports various types of listeners and can be added using annotations.	TestNG listener support is much better compared to JUnit 5.	TestNG Listeners	→

Rather than implementing a feature first, then testing it, drive development of the feature from a test

AAA (Arrange -> Act -> Assert) tests, TDD Test Driven Development
https://medium.com/@pjbgf/title-testing-code-ocd-and-the-aaa-pattern-df453975ab80

DDD – Domain Driven Design
BDD – Behavior Driven Development is for UAT (User Acceptance Tests)
An application is specified and designed by describing how it should behave
Given -> when -> Then
https://medium.com/javascript-scene/behavior-driven-development-bdd-and-functional-testing-62084ad7f1f2



Test Phases	BDD
Arrange	Given
Act	When
Assert	Then

Code coverage

A measure used to describe the degree to which the source code of a program is executed when a particular test suite runs.

Code Coverage Tools

- JCov
- OpenClover
- EMMA
- JaCoCo

his can be done via using test replacements (*test doubles*) for the real dependencies. Test doubles can be classified like the following:

A *dummy object* is passed around but never used, i.e., its methods are never called. Such an object can for example be used to fill the parameter list of a method.

Fake objects have working implementations, but are usually simplified. For example, a memory database is used for testing and not a SQL based database.

A *stub* class is an partial implementation for an interface or class with the purpose of using an instance of this stub class during testing. Stubs usually don’t respond to anything outside what’s programmed in for the test. Stubs may also record information about calls.

A *mock object* is a dummy implementation for an interface or a class in which you define the output of certain method calls. Mock objects are configured to perform a certain behavior during a test. They typically record the interaction with the system and tests can validate that.

A ***test fixture*** is a fixed state of a set of objects which are used as a baseline for running tests. Another way to describe this is a **test precondition**.

The percentage of code which is tested by unit tests is typically called ***test coverage***. Jococo, ..

External dependencies should be removed from unit tests, e.g., by replacing the dependency with a test implementation or a (**mock**) object created by a test framework. Mockito, ..

An ***integration*** *test* aims to test the behavior of a component or the integration between a set of components. Term ***functional*** *test* is sometimes used as synonym for integration test. E2E (protractor, selenium, cypress, ..), ..

Performance tests are used to benchmark software components repeatedly. jMeter, ..

A test is a **behavior** test (also called interaction test) if it checks if certain methods were called with the correct input parameters. A behavior test does not validate the result of a method call. **State testing** is about validating the result. Behavior testing is about testing the behavior of the application under test. If you are testing algorithms or system functionality, in most cases you may want to test state and not interactions.

https://howtodoinjava.com/java-best-practices/

https://howtodoinjava.com/junit/junit-creating-temporary-filefolder-using-temporaryfolder-rule/

Test passes but not testing the actual feature
Testing irrelevant things
Testing multiple things in assertions
Test accessing the testee using reflection
Tests swallowing exceptions
Test which depends on excessive setup
Test compatible to only developer's machine
Test filling log files with load of texts