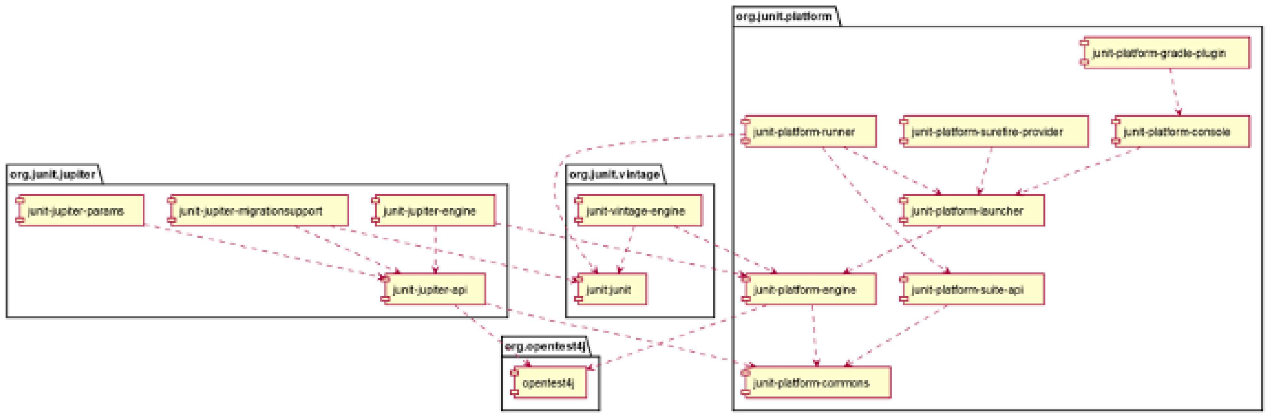
(https://www.eclipse.org/)

Embracing JUnit 5 with Eclipse

JUnit 5 (http://junit.org/junit5/) is out the door as the next generation test framework. It is a fundamentally redesigned version of the most widely used testing library in Java. JUnit 4.0 was first released over a decade ago after the introduction of annotations in Java 5. The world of Java and testing has evolved a lot since then. JUnit 4 was a big ball of mud with a single junit.jar to be used by test developers, testing framework developers, IDE developers, and build tool developers. Over the time, these developers have been accessing internals and duplicating code from JUnit 4 to get things done. This had made it very difficult to maintain and enhance the JUnit framework. So, to take advantage of the new features like lambda expressions from Java 8 and to support the advanced testing needs, JUnit 5 is now available as a modular and extensible test framework for the modern era.



(/community/eclipse_newsletter/2017/october/images/junituserguide.png)

Source: (JUnit 5 User Guide) http://junit.org/junit5/docs/current/user-guide/#dependency-diagram (http://junit.org/junit5/docs/current/user-guide/#dependency-diagram)

How is JUnit 5 different?

JUnit 5 is composed of several different modules from three different sub-projects.

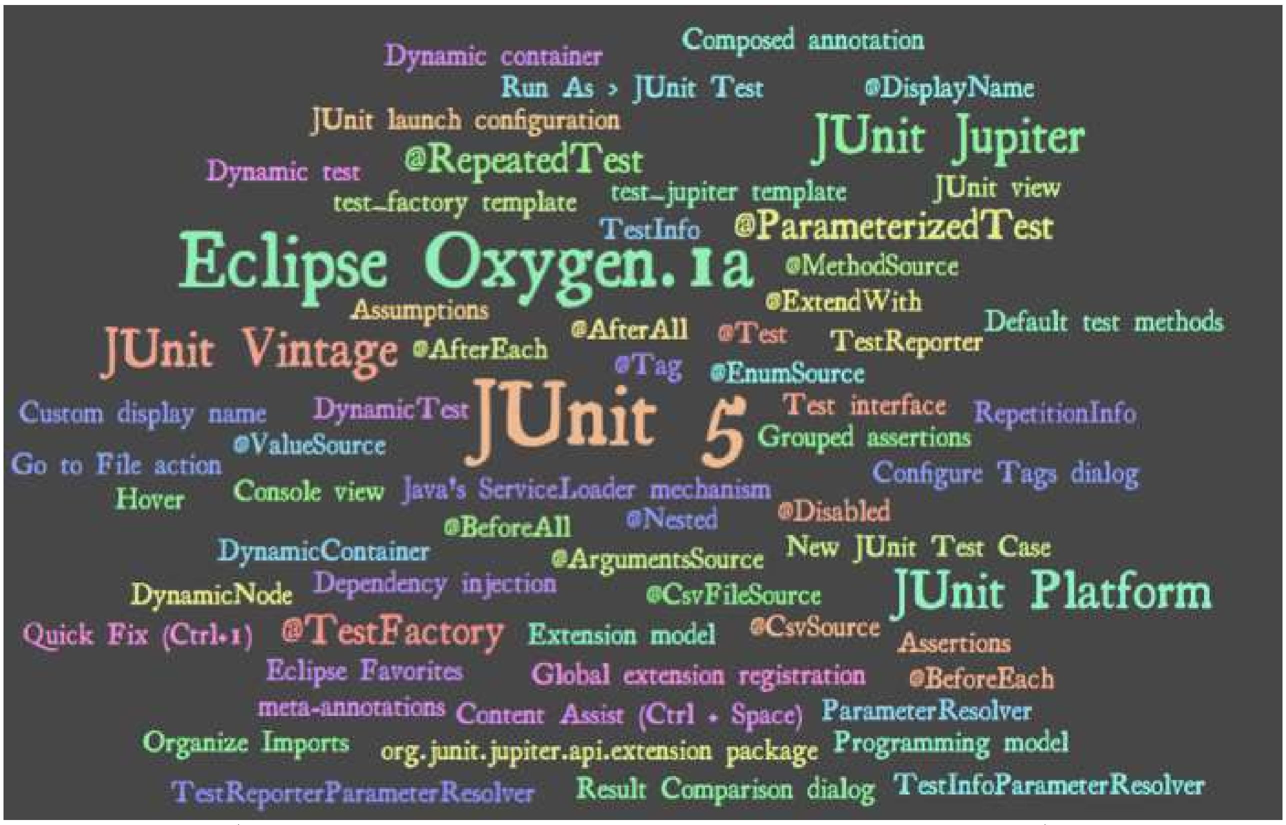
JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

- **JUnit Platform** defines Launcher APIs which are used by IDEs and build tools to launch the framework. It also defines TestEngine APIs which are used to develop testing frameworks that run on the platform.
- **JUnit Jupiter** is the combination of the new programming model and extension model for writing tests and extensions in JUnit 5. It also provides a TestEngine for running Jupiter based tests on the platform.
- **JUnit Vintage** provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform.

Give JUnit 5 a spin

To give JUnit 5 a spin, you have the tooling support in the Eclipse IDE ready at your disposal. Download Eclipse Oxygen.1a (4.7.1a) (https://www.eclipse.org/downloads/eclipse-packages/) now and try it out yourself!

Here is a sneak peek into the major interesting features of JUnit Jupiter with Eclipse support for JUnit 5.

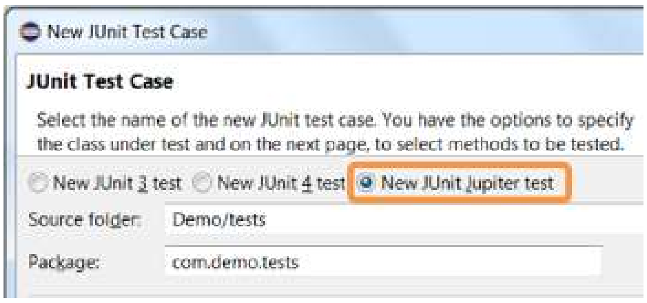


(/community/eclipse_newsletter/2017/october/images/junitwordle.png)

What can I do with JUnit 5?

Create a new JUnit Jupiter test

Create a new JUnit Jupiter test via **New JUnit Test Case** wizard:


New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test New JUnit 4 test [✓ New JUnit Jupiter test]

Source folder: Demo/tests

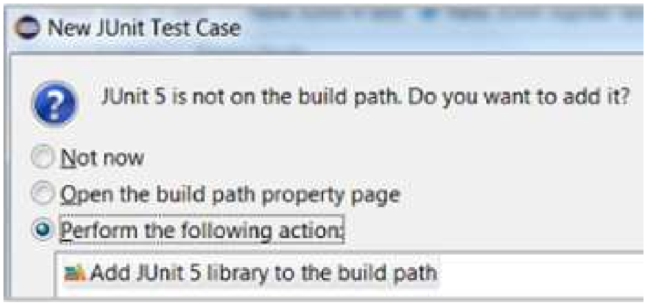
Package: com.demo.tests

(/community/eclipse_newsletter/2017/october/images/junit_test.png)

On this page, you can specify the lifecycle method stubs to be generated for this test case. It also lets you select a class under test and on the next page, you can select the methods from this class to generate test stubs.

Add JUnit 5 library to the build path

- The new JUnit Test Case wizard offers to add it while creating a new JUnit Jupiter test:

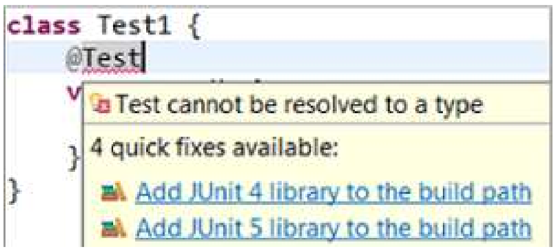

New JUnit Test Case

JUnit 5 is not on the build path. Do you want to add it?

Not now
Open the build path property page
[✓ Perform the following action:]
Add JUnit 5 library to the build path

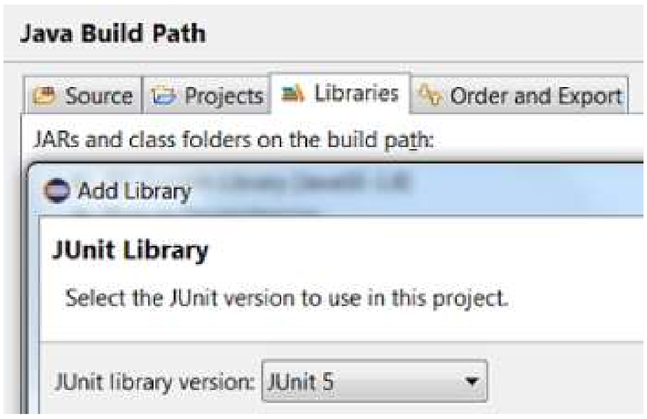
(/community/eclipse_newsletter/2017/october/images/junittestcase.png)

- **Quick Fix (Ctrl+1)** proposal on `@Test`, `@TestFactory`, `@ParameterizedTest`, and `@RepeatedTest` annotations


```
class Test1 {
    @Test
```
Test cannot be resolved to a type
4 quick fixes available:
Add JUnit 4 library to the build path
Add JUnit 5 library to the build path

(/community/eclipse_newsletter/2017/october/images/jtestclass.png)

- Add JUnit library in **Java Build Path** dialog:



JARs and class folders on the build path:

Add Library

JUnit Library

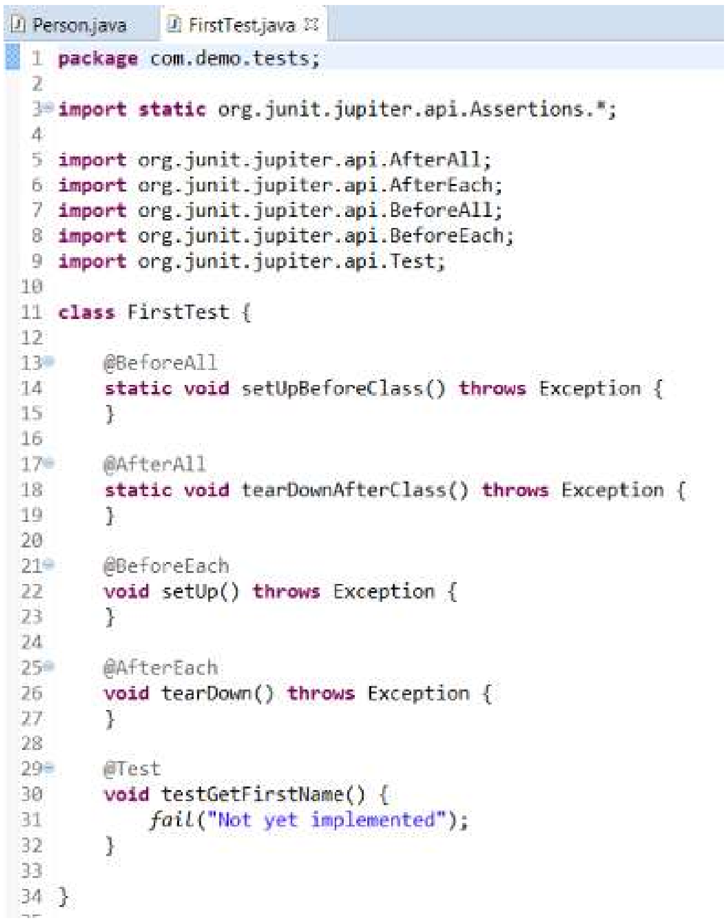
Select the JUnit version to use in this project.

JUnit library version: JUnit 5

(/community/eclipse_newsletter/2017/october/images/buildpath.png)

JUnit Jupiter test case

Here's a simple JUnit Jupiter test case.



```
package com.demo.tests;

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

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class FirstTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
    }

    @BeforeEach
    void setUp() throws Exception {
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void testGetFirstName() {
        fail("Not yet implemented");
    }

}
```

(/community/eclipse_newsletter/2017/october/images/testcase_personjava.png)

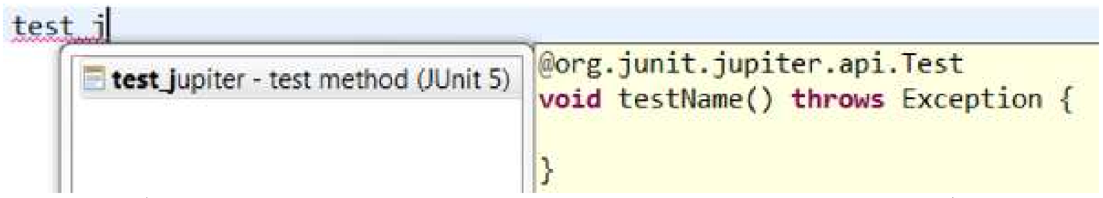
In JUnit Jupiter, test classes and methods can have any access modifier (other than private).

Here is a comparison of the annotations in JUnit 4 and JUnit Jupiter:

JUnit 4	**JUnit Jupiter**
@org.junit.Test	@org.junit.jupiter.api.Test (*No expected* and *timeout* attributes)
@BeforeClass	@BeforeAll
@AfterClass	@AfterAll
@Before	@BeforeEach
@After	@AfterEach
@Ignore	@Disabled

JUnit Jupiter test method

Create a JUnit Jupiter test method in the Eclipse IDE with the new **test_jupiter** template:


test_j

test_jupiter - test method (JUnit 5)

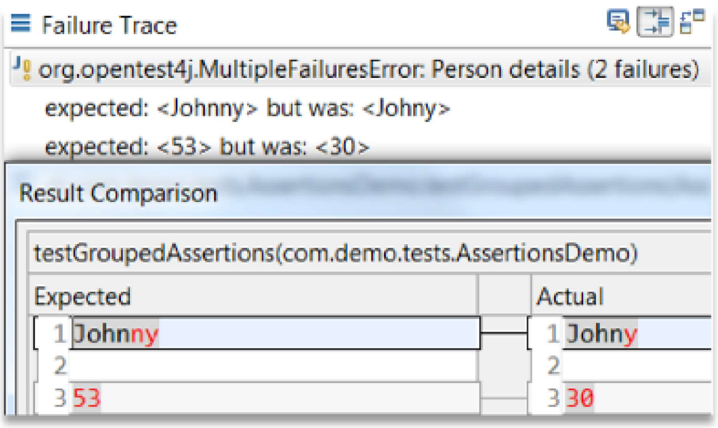
```
@org.junit.jupiter.api.Test
void testName() throws Exception {

}
```

(/community/eclipse_newsletter/2017/october/images/test_jupiter.png)

Assertions and Assumptions

JUnit Jupiter provides assertions and assumptions as static methods in `org.junit.jupiter.api.Assertions` class and `org.junit.jupiter.api.Assumptions` class respectively.

- You can view all the failures from grouped assertions in the **Result Comparison** dialog which can be opened from the Failure Trace section of JUnit view:


Failure Trace

```
org.opentest4j.MultipleFailuresError: Person details (2 failures)
    expected: <Johnny> but was: <Johny>
    expected: <53> but was: <30>
```

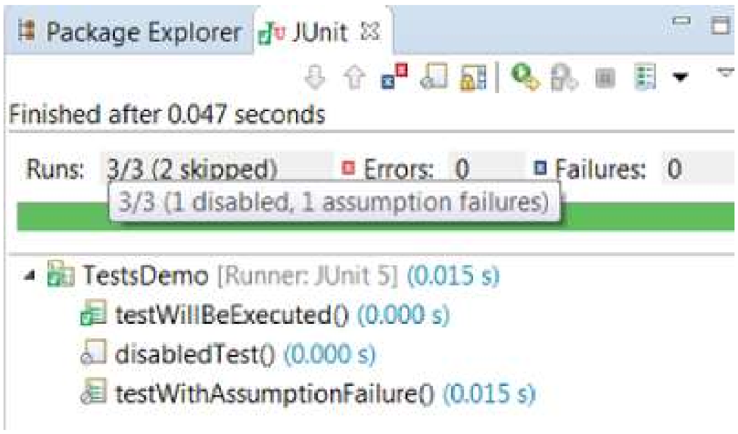
Result Comparison

testGroupedAssertions(com.demo.tests.AssertionsDemo)

Expected	Actual
1 Johnny	1 Johny
2	2
3 53	3 30

(/community/eclipse_newsletter/2017/october/images/failuretrace.png)

- You can view the number of tests with assumption failures on hover in JUnit view:



Finished after 0.047 seconds

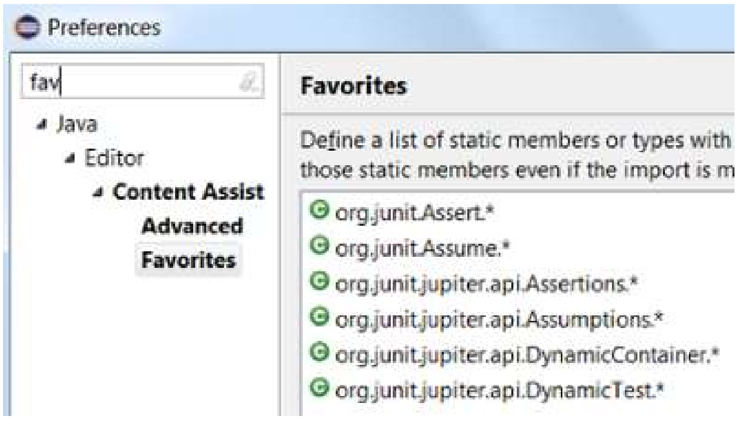
Runs: 3/3 (2 skipped) Errors: 0 Failures: 0

3/3 (1 disabled, 1 assumption failures)

```
TestsDemo [Runner: JUnit 5] (0.015 s)
    testWillBeExecuted() (0.000 s)
    disabledTest() (0.000 s)
    testWithAssumptionFailure() (0.015 s)
```

(/community/eclipse_newsletter/2017/october/images/junitview.png)

- JUnit Jupiter's `Assertions`, `Assumptions`, `DynamicContainer` and `DynamicTest` classes are now added to **Eclipse Favorites** by default:


Preferences

fav

Java > Editor > Content Assist > Advanced, Favorites

Favorites

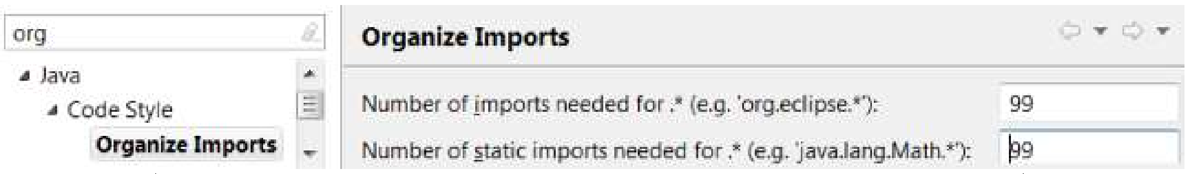
Define a list of static members or types with those static members even if the import is m

- org.junit.Assert.*
- org.junit.Assume.*
- org.junit.jupiter.api.Assertions.*
- org.junit.jupiter.api.Assumptions.*
- org.junit.jupiter.api.DynamicContainer.*
- org.junit.jupiter.api.DynamicTest.*

(/community/eclipse_newsletter/2017/october/images/eclipsefavs.png)

This allows you to quickly import static methods in your code from these favorite classes via **Content Assist (Ctrl + Space)** and **Quick Fix (Ctrl + 1)**.

You can also configure the number of static member imports needed in your code before type.* is used in the **Organize Imports** Preferences:


org

Java > Code Style > Organize Imports

Organize Imports

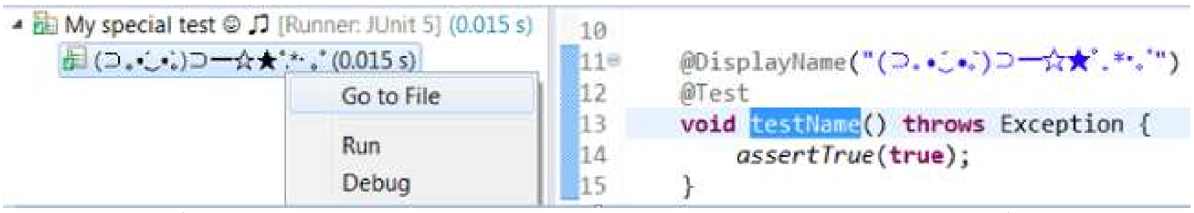
Number of imports needed for .* (e.g. 'org.eclipse.*'): 99

Number of static imports needed for .* (e.g. 'java.lang.Math.*'): 99

(/community/eclipse_newsletter/2017/october/images/organizeimports.png)

Custom display names

JUnit Jupiter's `@DisplayName` annotation allows you to provide custom display names for test classes and test methods which can have spaces, special characters, and even emojis. In Eclipse, you can see the custom test names in JUnit view. You can use the **Go to File** action or you can just double-click on the custom test name to navigate to the corresponding test from JUnit view:

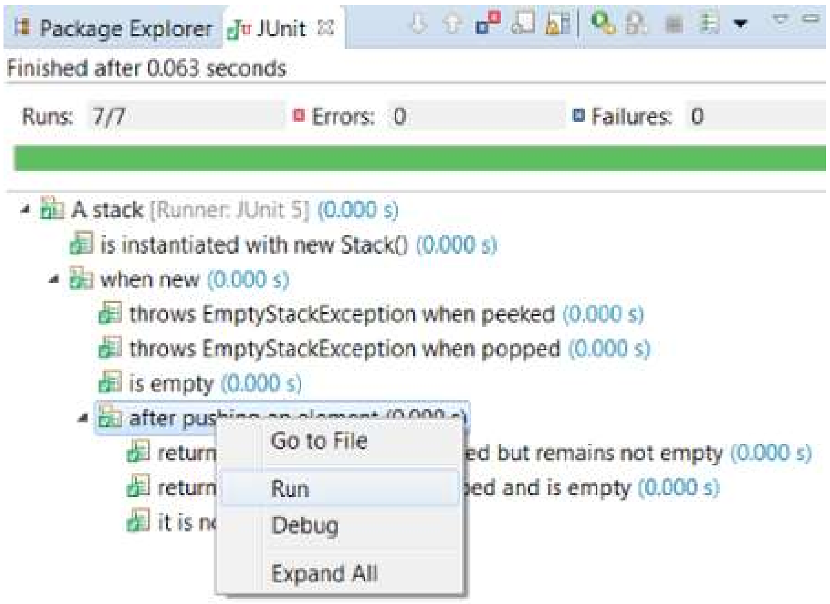

My special test ☺ ♫ [Runner: JUnit 5] (0.015 s)
(つ.•◡•.)つ━☆★'.*·.' (0.015 s)

```
    @DisplayName("(つ.•◡•.)つ━☆★'.*·.'")
    @Test
    void testName() throws Exception {
        assertTrue(true);
    }
```

(/community/eclipse_newsletter/2017/october/images/displayname.png)

Nested test classes

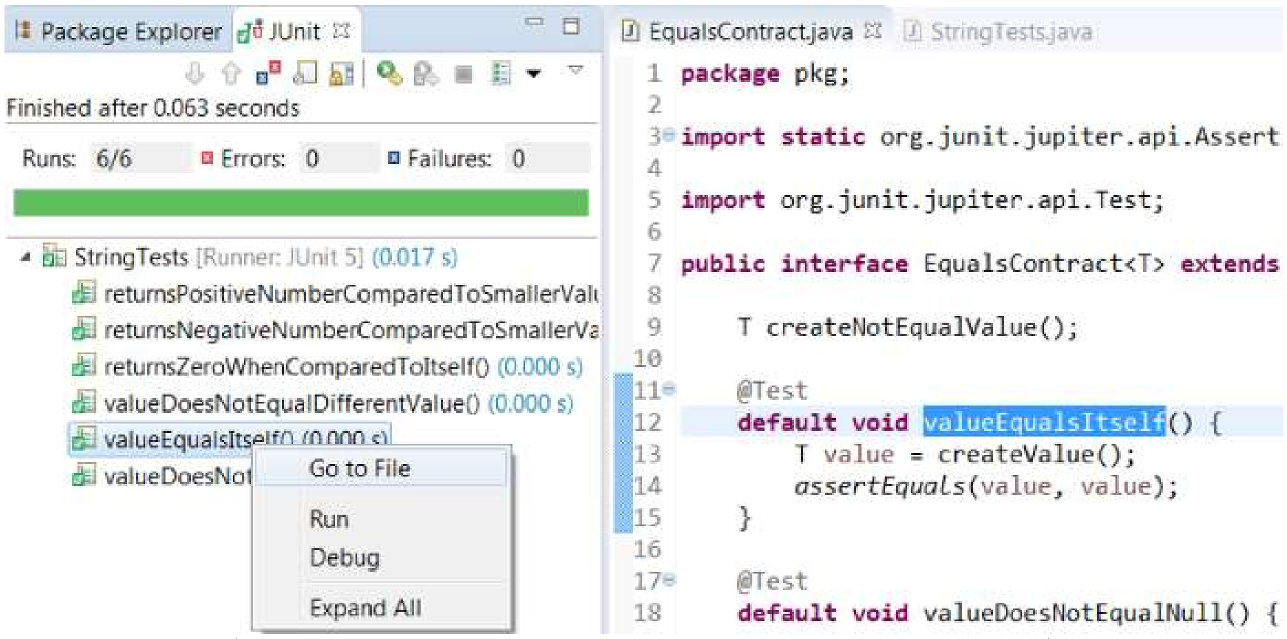
In JUnit Jupiter, non-static nested classes (i.e. inner classes) can serve as `@Nested` test classes for logical grouping of test cases. In Eclipse, you can (re-)run a single nested test class by using the **Run** action in JUnit view or Outline view. You can even right-click on a nested test class name in the editor and use the **Run As → JUnit Test** action:


Package Explorer
JUnit
Finished after 0.063 seconds
Runs: 7/7
Errors: 0
Failures: 0
A stack [Runner: JUnit 5] (0.000 s)
is instantiated with new Stack() (0.000 s)
when new (0.000 s)
throws EmptyStackException when peeked (0.000 s)
throws EmptyStackException when popped (0.000 s)
is empty (0.000 s)
after pus
return
ed but remains not empty (0.000 s)
return
ed and is empty (0.000 s)
it is n
Go to File
Run
Debug
Expand All

(/community/eclipse_newsletter/2017/october/images/runtest.png)

Test interfaces and default methods

JUnit Jupiter introduces the concept of a **test interface** which allows you to pre-implement some test code as default methods in interfaces. These **default test methods** can be inherited by implementing test classes, thereby enabling multiple inheritance in tests. In Eclipse, you can use the **Go to File** action or double-click on a test method in JUnit view to navigate to the corresponding default method inherited by the test class:

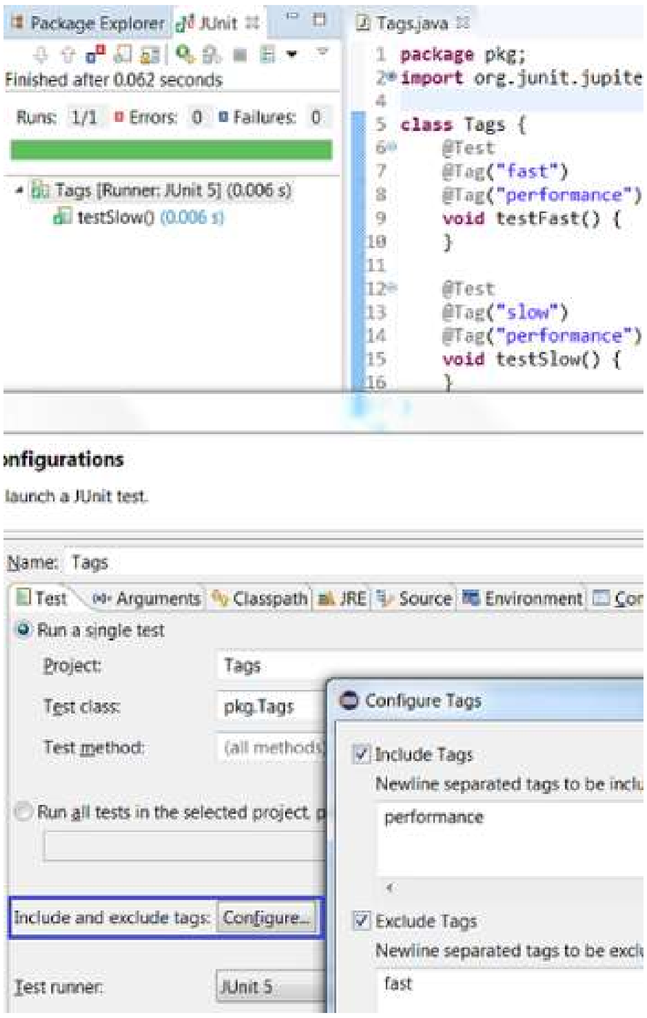

Package Explorer
JUnit
Finished after 0.063 seconds
Runs: 6/6
Errors: 0
Failures: 0
StringTests [Runner: JUnit 5] (0.017 s)
returnsPositiveNumberComparedToSmallerVal
returnsNegativeNumberComparedToSmallerVa
returnsZeroWhenComparedToItself() (0.000 s)
valueDoesNotEqualDifferentValue() (0.000 s)
valueEqualsItself() (0.000 s)
valueDoesNot
Go to File
Run
Debug
Expand All
EqualsContract.java
StringTests.java

```
package pkg;

import static org.junit.jupiter.api.Assert

import org.junit.jupiter.api.Test;

public interface EqualsContract<T> extends

    T createNotEqualValue();

    @Test
    default void valueEqualsItself() {
        T value = createValue();
        assertEquals(value, value);
    }

    @Test
    default void valueDoesNotEqualNull() {
```

(/community/eclipse_newsletter/2017/october/images/testmethod.png)

Tagging and filtering

You can tag test classes and test methods with your own identifiers using the `@Tag` annotation in JUnit Jupiter. These tags can later be used to filter test execution. In Eclipse, you can provide tags to be included in or excluded from a test run via the **Configure Tags** dialog in its JUnit launch configuration:

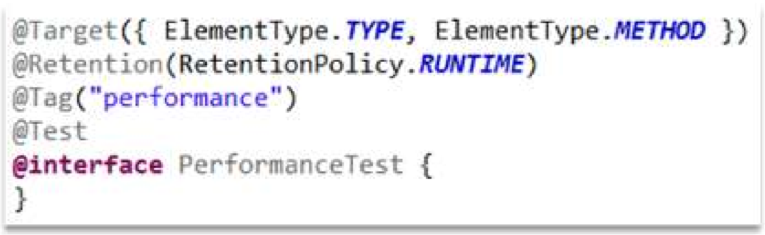


```java
package pkg;
import org.junit.jupite

class Tags {
    @Test
    @Tag("fast")
    @Tag("performance")
    void testFast() {
    }

    @Test
    @Tag("slow")
    @Tag("performance")
    void testSlow() {
    }
```

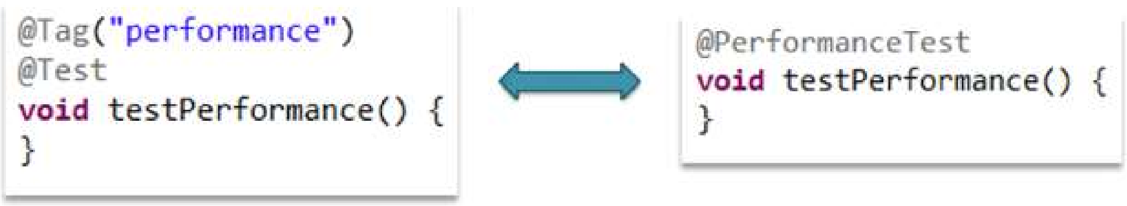
(/community/eclipse_newsletter/2017/october/images/testtags.png)

Meta-annotations and composed annotations

Annotations in JUnit Jupiter can be used as **meta-annotations**. This allows you to create a custom **composed annotation** that will automatically inherit the semantics of its meta-annotations. For example, you can use `@Test` and `@Tag(“performance”)` annotations as meta-annotations and create a composed annotation named `@PerformaceTest` which can be used as a replacement of the other two annotations on tests:



```java
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("performance")
@Test
@interface PerformanceTest {
}
```



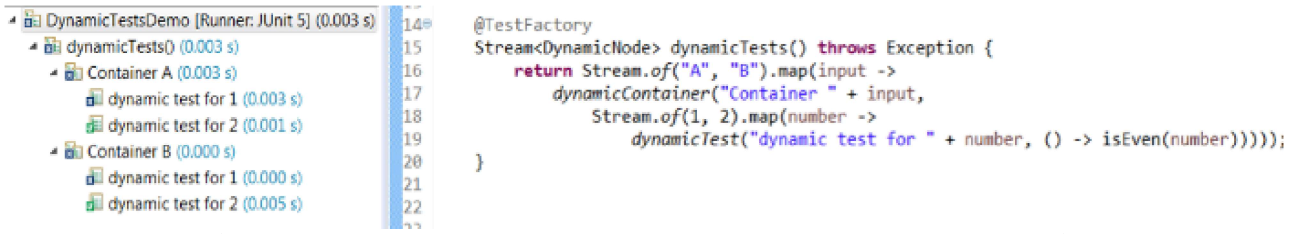
```java
@Tag("performance")
@Test
void testPerformance() {
}
```

```java
@PerformanceTest
void testPerformance() {
}
```

(/community/eclipse_newsletter/2017/october/images/performance.png)

Dynamic tests

JUnit Jupiter introduces a concept of dynamic tests which are generated at runtime by a factory method annotated with `@TestFactory` annotation. The test factory method returns `DynamicNode` instances (`DynamicContainer` and `DynamicTest`). A **dynamic container** is composed of a display name and a list of dynamic nodes. A **dynamic test** is composed of a display name and an Executable. Note that there are no lifecycle callbacks for individual dynamic tests. Dynamic containers can be used for dynamic nesting of nodes:

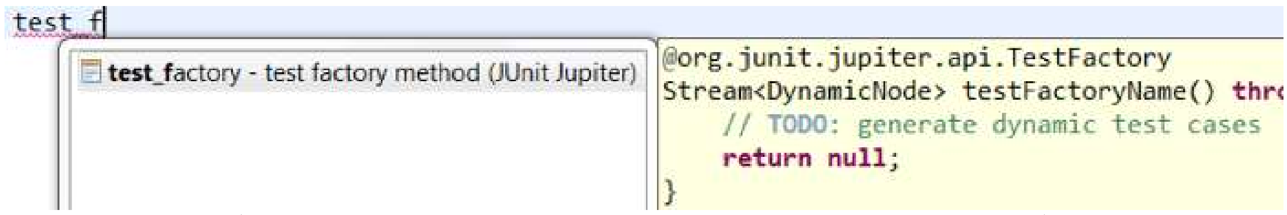


```
DynamicTestsDemo [Runner: JUnit 5] (0.003 s)
  dynamicTests() (0.003 s)
    Container A (0.003 s)
      dynamic test for 1 (0.003 s)
      dynamic test for 2 (0.001 s)
    Container B (0.000 s)
      dynamic test for 1 (0.000 s)
      dynamic test for 2 (0.005 s)
```

```
@TestFactory
Stream<DynamicNode> dynamicTests() throws Exception {
    return Stream.of("A", "B").map(input ->
        dynamicContainer("Container " + input,
            Stream.of(1, 2).map(number ->
                dynamicTest("dynamic test for " + number, () -> isEven(number)))));
}
```

(/community/eclipse_newsletter/2017/october/images/dynamicdemo.png)

In your Eclipse workspace, you can create a test factory method with the new **test_factory** template:



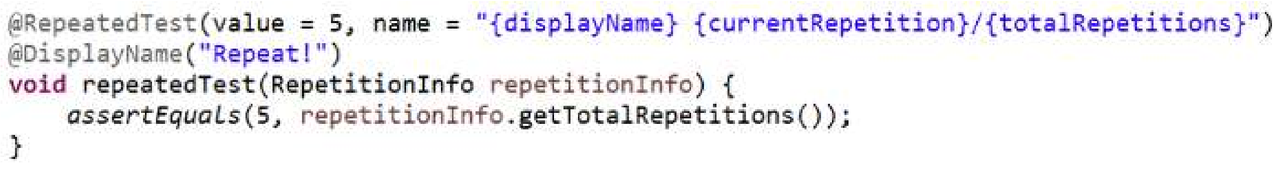
```
test_f
```

```
@org.junit.jupiter.api.TestFactory
Stream<DynamicNode> testFactoryName() thr
    // TODO: generate dynamic test cases
    return null;
}
```

(/community/eclipse_newsletter/2017/october/images/testfactory.png)

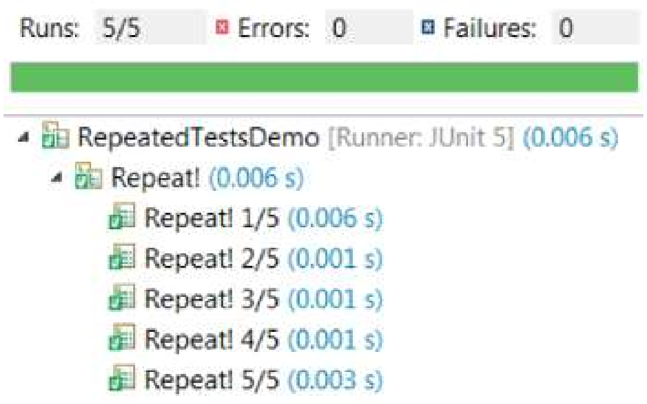
Repeated tests

You can repeat a test in JUnit Jupiter by annotating the test method with `@RepeatedTest` annotation and specifying the number of repetitions. You can optionally specify a custom display name for each repetition and use `RepetitionInfo` as the test method parameter to get information about the current repetition:



```
@RepeatedTest(value = 5, name = "{displayName} {currentRepetition}/{totalRepetitions}")
@DisplayName("Repeat!")
void repeatedTest(RepetitionInfo repetitionInfo) {
    assertEquals(5, repetitionInfo.getTotalRepetitions());
}
```

(/community/eclipse_newsletter/2017/october/images/repeatedtestresult.png)

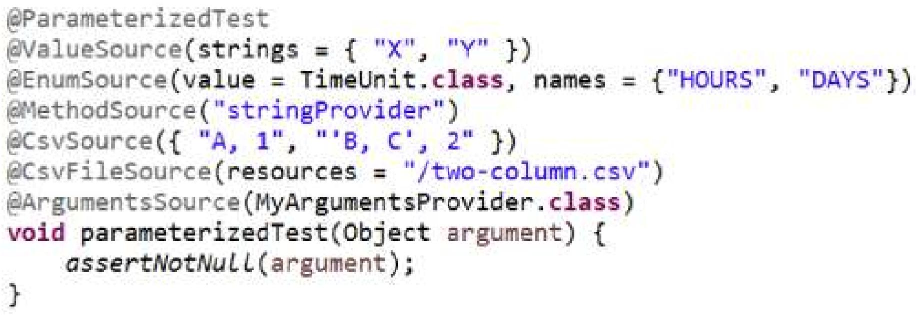


```
Runs: 5/5    Errors: 0    Failures: 0

RepeatedTestsDemo [Runner: JUnit 5] (0.006 s)
  Repeat! (0.006 s)
    Repeat! 1/5 (0.006 s)
    Repeat! 2/5 (0.001 s)
    Repeat! 3/5 (0.001 s)
    Repeat! 4/5 (0.001 s)
    Repeat! 5/5 (0.003 s)
```

(/community/eclipse_newsletter/2017/october/images/repeatedtest.png)

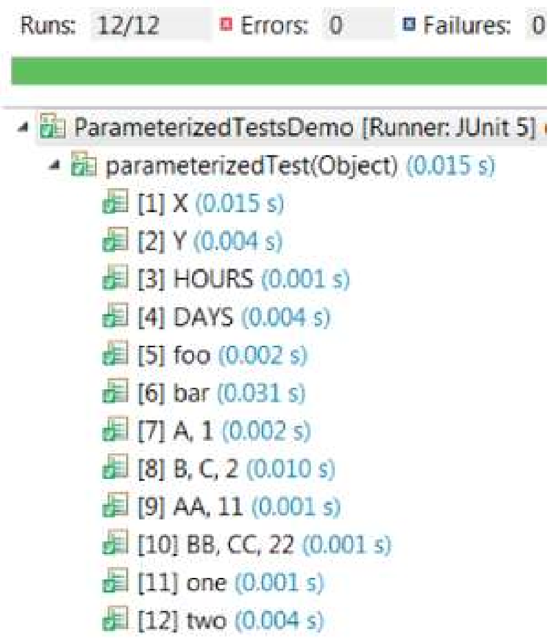
Parameterized tests

You can run a test multiple times with different arguments by annotating the test method with `@ParameterizedTest` annotation and declaring at least one source to provide the arguments. You can optionally specify a custom display name for each run. JUnit Jupiter provides various source annotations which can be seen in this example:



```
@ParameterizedTest
@ValueSource(strings = { "X", "Y" })
@EnumSource(value = TimeUnit.class, names = {"HOURS", "DAYS"})
@MethodSource("stringProvider")
@CsvSource({ "A, 1", "'B, C', 2" })
@CsvFileSource(resources = "/two-column.csv")
@ArgumentsSource(MyArgumentsProvider.class)
void parameterizedTest(Object argument) {
    assertNotNull(argument);
}
```

(/community/eclipse_newsletter/2017/october/images/parametrizedtest.png)



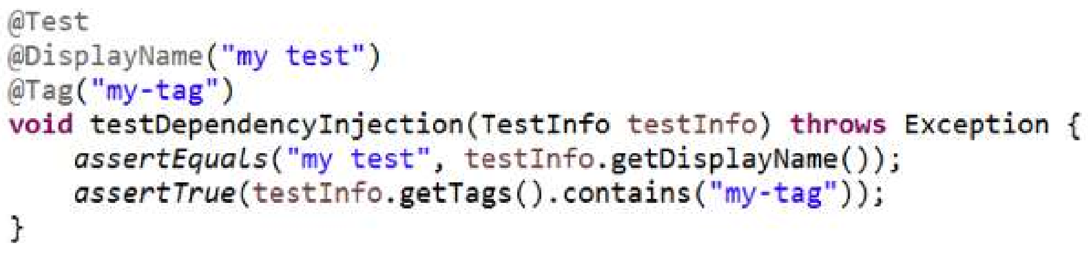
```
Runs: 12/12    Errors: 0    Failures: 0

ParameterizedTestsDemo [Runner: JUnit 5]
  parameterizedTest(Object) (0.015 s)
    [1] X (0.015 s)
    [2] Y (0.004 s)
    [3] HOURS (0.001 s)
    [4] DAYS (0.004 s)
    [5] foo (0.002 s)
    [6] bar (0.031 s)
    [7] A, 1 (0.002 s)
    [8] B, C, 2 (0.010 s)
    [9] AA, 11 (0.001 s)
    [10] BB, CC, 22 (0.001 s)
    [11] one (0.001 s)
    [12] two (0.004 s)
```

(/community/eclipse_newsletter/2017/october/images/parametrizedtestresults.png)

Dependency injection

Dependency injection in JUnit Jupiter allows test constructors and methods to have parameters. These parameters must be dynamically resolved at runtime by a registered parameter resolver. `ParameterResolver` is the extension API to provide parameter resolvers which can be registered via `@ExtendWith` annotation. This is where Jupiter's programming model meets its extension model. JUnit Jupiter provides some built-in resolvers which are registered automatically:

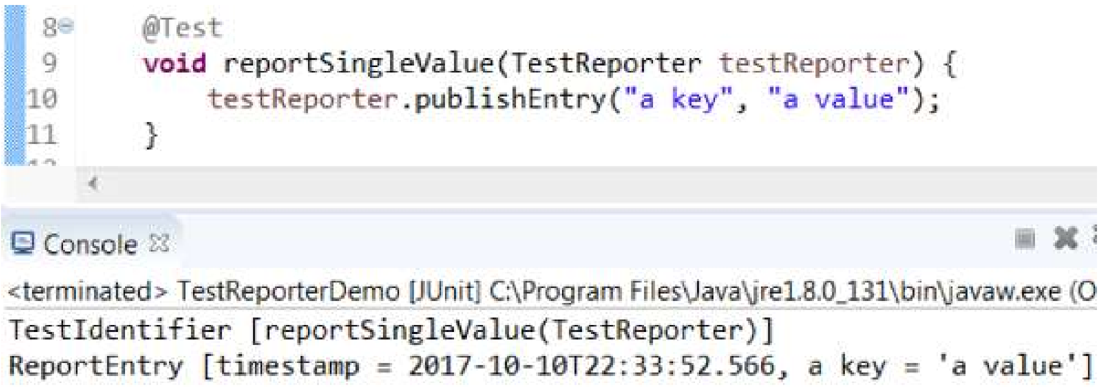
- **TestInfo** (resolved by `TestInfoParameterResolver`) to access information about the currently executing test:



```
@Test
@DisplayName("my test")
@Tag("my-tag")
void testDependencyInjection(TestInfo testInfo) throws Exception {
    assertEquals("my test", testInfo.getDisplayName());
    assertTrue(testInfo.getTags().contains("my-tag"));
}
```

(/community/eclipse_newsletter/2017/october/images/testinfo.png)

- **TestReporter** (resolved by `TestReporterParameterResolver`) to publish additional data about the current test run which can be seen in the **Console view** in the Eclipse IDE:



```
@Test
void reportSingleValue(TestReporter testReporter) {
    testReporter.publishEntry("a key", "a value");
}
```

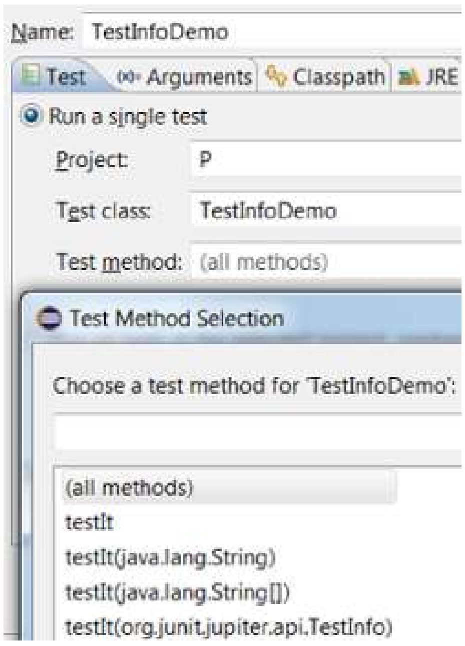
Console

<terminated> TestReporterDemo [JUnit] C:\Program Files\Java\jre1.8.0_131\bin\javaw.exe (O

```
TestIdentifier [reportSingleValue(TestReporter)]
ReportEntry [timestamp = 2017-10-10T22:33:52.566, a key = 'a value']
```

(/community/eclipse_newsletter/2017/october/images/testreporter.png)

In the Eclipse IDE, the **Test Method Selection** dialog in JUnit launch configuration now shows the method parameter types also:



Name: TestInfoDemo

Run a single test

Project: P

Test class: TestInfoDemo

Test method: (all methods)

Test Method Selection

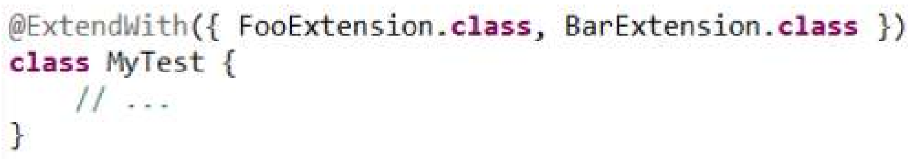
Choose a test method for 'TestInfoDemo':

(all methods)
testIt
testIt(java.lang.String)
testIt(java.lang.String[])
testIt(org.junit.jupiter.api.TestInfo)

(/community/eclipse_newsletter/2017/october/images/testinfodemo.png)

JUnit Jupiter extension model

JUnit Jupiter's extension model provides various extension APIs as small interfaces in `org.junit.jupiter.api.extension` package which can be implemented by extension providers. You can register one or more of these extensions declaratively on a test class, test method, or composed annotation via `@ExtendWith` annotation. JUnit Jupiter also supports global extension registration via Java's `ServiceLoader` mechanism:

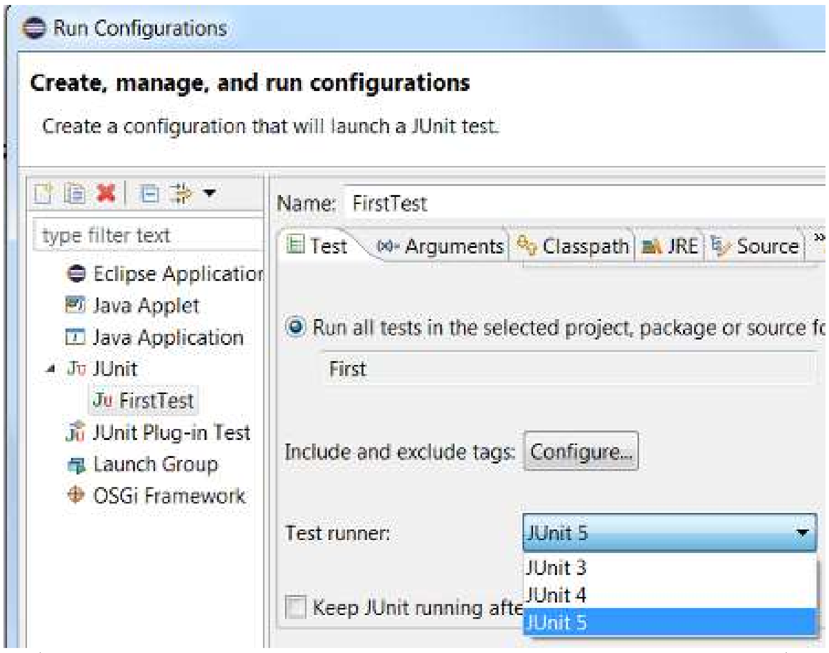


```
@ExtendWith({ FooExtension.class, BarExtension.class })
class MyTest {
    // ...
}
```

(/community/eclipse_newsletter/2017/october/images/extendwith.png)

Note

- If you are using an Eclipse workspace where you were running your JUnit 5 tests via `@RunWith(JUnitPlatform.class)` in the Eclipse IDE without JUnit 5 support then you will have JUnit 4 as the test runner in their launch configurations. Before executing these tests in the Eclipse IDE with JUnit 5 support, you should either change their test runner to JUnit 5 or delete them so that new launch configurations are created with JUnit 5 test runner while running the tests:


Run Configurations

Create, manage, and run configurations

Create a configuration that will launch a JUnit test.

Name: FirstTest

Run all tests in the selected project, package or source fc

First

Include and exclude tags:

Test runner: JUnit 5

Keep JUnit running afte

(/community/eclipse_newsletter/2017/october/images/note1.png)

- We do not support running tests in a setup where an old Eclipse build (not having JUnit 5 support) is using a new Eclipse build (having JUnit 5 support) as target. Also, developers who have the JDT JUnit runtime bundles (`org.eclipse.jdt.junit.runtime` , `org.eclipse.jdt.junit4.runtime`) checked out and pull the latest changes will run into the above issue. You are expected to use a new Eclipse build for the development.

Resources & References

Eclipse Support for JUnit 5

- Download Eclipse Oxygen.1a (4.7.1a) (https://www.eclipse.org/downloads/eclipse-packages)
- Software site repository to update your Eclipse installation (http://download.eclipse.org/releases/oxygen/)
- Report bugs/enhancements in Eclipse support for JUnit 5 (https://bugs.eclipse.org/bugs/enter_bug.cgi?product=JDT&component=UI)

Learn more about the JUnit 5 Project (http://junit.org/junit5)

- User Guide (http://junit.org/junit5/docs/current/user-guide)
- Javadoc (http://junit.org/junit5/docs/current/api)
- GitHub (https://github.com/junit-team/junit5)
- Gitter (https://gitter.im/junit-team/junit5)
- Q&A (http://stackoverflow.com/questions/tagged/junit5)

About the Author

Noopur Gupta
IBM (https://www.ibm.com/)

Twitter (https://twitter.com/noopur2507) LinkedIn (https://www.linkedin.com/in/noopur2507/)

ECLIPSE NEWSLETTER

A fresh new issue delivered monthly

Subscribe

All Eclipse Newsletters available here (/community/eclipse_newsletter)

Back to the top

[illegible]
foundation)