JUnit

Version 6.0.1

Table of Contents

Overview . . . 1
Writing Tests . . . 3
- Annotations . . . 3
- Definitions . . . 7
- Test Classes and Methods . . . 7
- Display Names . . . 10
- Assertions . . . 14
- Assumptions . . . 21
- Exception Handling . . . 22
- Disabling Tests . . . 25
- Conditional Test Execution . . . 27
- Tagging and Filtering . . . 35
- Test Execution Order . . . 35
- Test Instance Lifecycle . . . 39
- Nested Tests . . . 40
- Dependency Injection for Constructors and Methods . . . 44
- Test Interfaces and Default Methods . . . 47
- Repeated Tests . . . 50
- Parameterized Classes and Tests . . . 55
- Class Templates . . . 88
- Test Templates . . . 88
- Dynamic Tests . . . 89
- Timeouts . . . 95
- Parallel Execution . . . 99
- Built-in Extensions . . . 108

Migrating from JUnit 4 . . . 115
Running Tests . . . 123
- IDE Support . . . 123
- Build Support . . . 124
- Console Launcher . . . 133
- Discovery Selectors . . . 144
- Configuration Parameters . . . 145
- Tags . . . 146
- Capturing Standard Output/Error . . . 148
- Using Listeners and Interceptors . . . 148
- Stack Trace Pruning . . . 150
- Discovery Issues . . . 150

Extension Model . . . 152

Registering Extensions . . . 152
Conditional Test Execution . . . 160
Test Instance Pre-construct Callback . . . 161
Test Instance Factories . . . 161
Test Instance Post-processing . . . 162
Test Instance Pre-destroy Callback . . . 162
Parameter Resolution . . . 162
Test Result Processing . . . 168
Test Lifecycle Callbacks . . . 169
Exception Handling . . . 171
Pre-Interrupt Callback . . . 173
Intercepting Invocations . . . 173
Providing Invocation Contexts for Class Templates . . . 174
Providing Invocation Contexts for Test Templates . . . 175
Keeping State in Extensions . . . 177
Supported Utilities in Extensions . . . 181
Relative Execution Order of User Code and Extensions . . . 182
Advanced Topics . . . 194
JUnit Platform Reporting . . . 194
JUnit Platform Suite Engine . . . 197
JUnit Platform Test Kit . . . 200
JUnit Platform Launcher API . . . 208
Test Engines . . . 223
API Evolution . . . 226
Release Notes . . . 230
Appendix . . . 238

Overview

The goal of this document is to provide comprehensive reference documentation for programmers writing tests, extension authors, and engine authors as well as build tool and IDE vendors.

What is JUnit?

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

JUnit 6.0.1 = ***JUnit Platform*** + ***JUnit Jupiter*** + ***JUnit Vintage***

The **JUnit Platform** serves as a foundation for launching testing frameworks on the JVM. It also defines the `TestEngine` API for developing a testing framework that runs on the platform. Furthermore, the platform provides a Console Launcher to launch the platform from the command line and the JUnit Platform Suite Engine for running a custom test suite using one or more test engines on the platform. First-class support for the JUnit Platform also exists in popular IDEs (see IntelliJ IDEA, Eclipse, NetBeans, and Visual Studio Code) and build tools (see Gradle, Maven, and Ant).

JUnit Jupiter is the combination of the programming model and extension model for writing JUnit tests and extensions. The Jupiter sub-project 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. It requires JUnit 4.12 or later to be present on the class path or module path. Note, however, that the JUnit Vintage engine is deprecated and should only be used temporarily while migrating tests to JUnit Jupiter or another testing framework with native JUnit Platform support.

Supported Java Versions

JUnit requires Java 17 (or higher) at runtime. However, you can still test code that has been compiled with previous versions of the JDK.

Getting Help

Ask JUnit-related questions on Stack Overflow or use the Q&A category on GitHub Discussions.

Getting Started

Downloading JUnit Artifacts

To find out what artifacts are available for download and inclusion in your project, refer to Dependency Metadata. To set up dependency management for your build, refer to Build Support and the Example Projects.

JUnit Features

To find out what features are available in JUnit 6.0.1 and how to use them, read the corresponding sections of this User Guide, organized by topic.

- Writing Tests in JUnit Jupiter
- Migrating from JUnit 4 to JUnit Jupiter
- Running Tests
- Extension Model for JUnit Jupiter
- Advanced Topics
  - JUnit Platform Launcher API
  - JUnit Platform Test Kit

Example Projects

To see complete, working examples of projects that you can copy and experiment with, the `junit-examples` repository is a good place to start. The `junit-examples` repository hosts a collection of example projects based on JUnit Jupiter, JUnit Vintage, and other testing frameworks. You'll find appropriate build scripts (e.g., `build.gradle`, `pom.xml`, etc.) in the example projects. The links below highlight some of the combinations you can choose from.

- For Gradle and Java, check out the `junit-jupiter-starter-gradle` project.
- For Gradle and Kotlin, check out the `junit-jupiter-starter-gradle-kotlin` project.
- For Gradle and Groovy, check out the `junit-jupiter-starter-gradle-groovy` project.
- For Maven, check out the `junit-jupiter-starter-maven` project.
- For Ant, check out the `junit-jupiter-starter-ant` project.

Writing Tests

The following example provides a glimpse at the minimum requirements for writing a test in JUnit Jupiter. Subsequent sections of this chapter will provide further details on all available features.

A first test case

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

import example.util.Calculator;

import org.junit.jupiter.api.Test;

class MyFirstJUnitJupiterTests {

    private final Calculator calculator = new Calculator();

    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }

}
```

Annotations

JUnit Jupiter supports the following annotations for configuring tests and extending the framework.

Unless otherwise stated, all core annotations are located in the `org.junit.jupiter.api` package in the `junit-jupiter-api` module.

`@Test`

Denotes that a method is a test method. Unlike JUnit 4's `@Test` annotation, this annotation does not declare any attributes, since test extensions in JUnit Jupiter operate based on their own dedicated annotations. Such methods are inherited unless they are overridden.

`@ParameterizedTest`

Denotes that a method is a parameterized test. Such methods are inherited unless they are overridden.

`@RepeatedTest`

Denotes that a method is a test template for a repeated test. Such methods are inherited unless they are overridden.

`@TestFactory`

Denotes that a method is a test factory for dynamic tests. Such methods are inherited unless they are overridden.

`@TestTemplate`

Denotes that a method is a template for a test case designed to be invoked multiple times depending on the number of invocation contexts returned by the registered providers. Such methods are inherited unless they are overridden.

`@TestClassOrder`

Used to configure the test class execution order for `@Nested` test classes in the annotated test class. Such annotations are inherited.

`@TestMethodOrder`

Used to configure the test method execution order for the annotated test class; similar to JUnit 4's `@FixMethodOrder`. Such annotations are inherited.

`@TestInstance`

Used to configure the test instance lifecycle for the annotated test class. Such annotations are inherited.

`@DisplayName`

Declares a custom display name for the test class or test method. Such annotations are not inherited.

`@DisplayNameGeneration`

Declares a custom display name generator for the test class. Such annotations are inherited.

`@BeforeEach`

Denotes that the annotated method should be executed *before* **each** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, or `@TestFactory` method in the current class; analogous to JUnit 4's `@Before`. Such methods are inherited unless they are overridden.

`@AfterEach`

Denotes that the annotated method should be executed *after* **each** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, or `@TestFactory` method in the current class; analogous to JUnit 4's `@After`. Such methods are inherited unless they are overridden.

`@BeforeAll`

Denotes that the annotated method should be executed *before* **all** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, and `@TestFactory` methods in the current top-level or `@Nested` test class; analogous to JUnit 4's `@BeforeClass`. Such methods are inherited unless they are overridden and must be `static` unless the "per-class" test instance lifecycle is used.

`@AfterAll`

Denotes that the annotated method should be executed *after* **all** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, and `@TestFactory` methods in the current top-level or `@Nested` test class; analogous to JUnit 4's `@AfterClass`. Such methods are inherited unless they are overridden and must be `static` unless the "per-class" test instance lifecycle is used.

`@ParameterizedClass`

Denotes that the annotated class is a parameterized class. Such annotations are inherited.

`@BeforeParameterizedClassInvocation`

Denotes that the annotated method should be executed once *before* each invocation of a parameterized class. Such methods are inherited unless they are overridden.

`@AfterParameterizedClassInvocation`

Denotes that the annotated method should be executed once *after* each invocation of a parameterized class. Such methods are inherited unless they are overridden.

`@ClassTemplate`

Denotes that the annotated class is a template for a test class designed to be executed multiple times depending on the number of invocation contexts returned by the registered providers. Such annotations are inherited.

`@Nested`

Denotes that the annotated class is a non-static nested test class. Such annotations are not inherited.

`@Tag`

Used to declare tags for filtering tests, either at the class or method level; analogous to test groups in TestNG or Categories in JUnit 4. Such annotations are inherited at the class level but not at the method level.

`@Disabled`

Used to disable a test class or test method; analogous to JUnit 4's `@Ignore`. Such annotations are not inherited.

`@AutoClose`

Denotes that the annotated field represents a resource that will be automatically closed after test execution. Such fields are inherited.

`@Timeout`

Used to fail a test, test factory, test template, or lifecycle method if its execution exceeds a given duration. Such annotations are inherited.

`@TempDir`

Used to supply a temporary directory via field injection or parameter injection in a test class constructor, lifecycle method, or test method; located in the `org.junit.jupiter.api.io` package. Such fields are inherited.

`@ExtendWith`

Used to register extensions declaratively. Such annotations are inherited.

`@RegisterExtension`

Used to register extensions programmatically via fields. Such fields are inherited.



Some annotations may currently be *experimental*. Consult the table in Experimental APIs for details.

Meta-Annotations and Composed Annotations

JUnit Jupiter annotations can be used as *meta-annotations*. That means that you can define your own *composed annotation* that will automatically *inherit* the semantics of its meta-annotations.

For example, instead of copying and pasting `@Tag("fast")` throughout your code base (see Tagging and Filtering), you can create a custom *composed annotation* named `@Fast` as follows. `@Fast` can then be used as a drop-in replacement for `@Tag("fast")`.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.junit.jupiter.api.Tag;

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("fast")
public @interface Fast {
}
```

The following `@Test` method demonstrates usage of the `@Fast` annotation.

```
@Fast
@Test
void myFastTest() {
    // ...
}
```

You can even take that one step further by introducing a custom `@FastTest` annotation that can be used as a drop-in replacement for `@Tag("fast")` *and* `@Test`.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

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

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Tag("fast")
@Test
public @interface FastTest {
}
```

JUnit automatically recognizes the following as a `@Test` method that is tagged with "fast".

```
@FastTest
void myFastTest() {
    // ...
}
```

Definitions

Platform Concepts

Container

a node in the test tree that contains other containers or tests as its children (e.g. a *test class*).

Test

a node in the test tree that verifies expected behavior when executed (e.g. a `@Test` method).

Jupiter Concepts

Lifecycle Method

any method that is directly annotated or meta-annotated with `@BeforeAll`, `@AfterAll`, `@BeforeEach`, or `@AfterEach`.

Test Class

any top-level class, `static` member class, or `@Nested` class that contains at least one *test method*, i.e. a *container*. Test classes must not be `abstract` and must have a single constructor. Java `record` classes are supported as well.

Test Method

any instance method that is directly annotated or meta-annotated with `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, or `@TestTemplate`. With the exception of `@Test`, these create a *container* in the test tree that groups *tests* or, potentially (for `@TestFactory`), other *containers*.

Test Classes and Methods

Test methods and lifecycle methods may be declared locally within the current test class, inherited from superclasses, or inherited from interfaces (see Test Interfaces and Default Methods). In addition, test methods and lifecycle methods must not be `abstract` and must not return a value (except `@TestFactory` methods which are required to return a value).



Class and method visibility

Test classes, test methods, and lifecycle methods are not required to be `public`, but they must *not* be `private`.

It is generally recommended to omit the `public` modifier for test classes, test methods, and lifecycle methods unless there is a technical reason for doing so – for

example, when a test class is extended by a test class in another package. Another technical reason for making classes and methods `public` is to simplify testing on the module path when using the Java Module System.

Field and method inheritance

Fields in test classes are inherited. For example, a `@TempDir` field from a superclass will always be applied in a subclass.



Test methods and lifecycle methods are inherited unless they are overridden according to the visibility rules of the Java language. For example, a `@Test` method from a superclass will always be applied in a subclass unless the subclass explicitly overrides the method. Similarly, if a package-private `@Test` method is declared in a superclass that resides in a different package than the subclass, that `@Test` method will always be applied in the subclass since the subclass cannot override a package-private method from a superclass in a different package.

See also: Field and Method Search Semantics

The following test class demonstrates the use of `@Test` methods and all supported lifecycle methods. For further information on runtime semantics, see Test Execution Order and Wrapping Behavior of Callbacks.

A standard Java test class

```
import static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeTrue;

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

class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
```

```
        fail("a failing test");
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }

    @Test
    void abortedTest() {
        assumeTrue("abc".contains("Z"));
        fail("test should have been aborted");
    }

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }

}
```

It is also possible to use Java `record` classes as test classes as illustrated by the following example.

A test class written as a Java record

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

import example.util.Calculator;

import org.junit.jupiter.api.Test;

record MyFirstJUnitJupiterRecordTests() {

    @Test
    void addition() {
        assertEquals(2, new Calculator().add(1, 1));
    }

}
```

Test and lifecycle methods may be written in Kotlin and may optionally use the `suspend` keyword for testing code using coroutines.

A test class written in Kotlin

```
import org.junit.jupiter.api.BeforeEach
```

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

@Suppress("JUnitMalformedDeclaration")
class KotlinCoroutinesDemo {
    @BeforeEach
    fun regularSetUp() {
    }

    @BeforeEach
    suspend fun coroutineSetUp() {
    }

    @Test
    fun regularTest() {
    }

    @Test
    suspend fun coroutineTest() {
    }
}
```



Using suspending functions as test or lifecycle methods requires `kotlin-stdlib`, `kotlin-reflect`, and `kotlinx-coroutines-core` to be present on the classpath or module path.

Display Names

Test classes and test methods can declare custom display names via `@DisplayName` — with spaces, special characters, and even emojis — that will be displayed in test reports and by test runners and IDEs.

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

@DisplayName("A special test case")
class DisplayNameDemo {

    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() {
    }

    @Test
    @DisplayName("□°□°□□")
    void testWithDisplayNameContainingSpecialCharacters() {
    }

    @Test
```

```
    @DisplayName("")
    void testWithDisplayNameContainingEmoji() {
    }

}
```



Control characters in text-based arguments in display names for parameterized tests are escaped by default. See Quoted Text-based Arguments for details.

Any remaining ISO control characters in a display name will be replaced as follows.

Original	Replacement	Description
\r	<CR>	Textual representation of a carriage return
\n	<LF>	Textual representation of a line feed
Other control character		Unicode replacement character (U+FFFD)

Display Name Generators

JUnit Jupiter supports custom display name generators that can be configured via the `@DisplayNameGeneration` annotation.

Generators can be created by implementing the `DisplayNameGenerator` API. The following table lists the default display name generators available in Jupiter.

DisplayNameGenerator	Behavior
`Standard`	Matches the standard display name generation behavior in place since JUnit Jupiter was introduced.
`Simple`	Extends the functionality of `Standard` by removing trailing parentheses for methods with no parameters.
`ReplaceUnderscores`	Replaces underscores with spaces.
`IndicativeSentences`	Generates complete sentences by concatenating the names of the test and the enclosing classes.



Values provided via `@DisplayName` annotations always take precedence over display names generated by a `DisplayNameGenerator`.

The following example demonstrates the use of the `ReplaceUnderscores` display name generator.

```
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
class A_year_is_not_supported {

    @Test
    void if_it_is_zero() {
    }

    @DisplayName("A negative value for year is not supported by the leap year
computation.")
    @ParameterizedTest(name = "For example, year {0} is not supported.")
    @ValueSource(ints = { -1, -4 })
    void if_it_is_negative(int year) {
    }

}
```

Running the above test class results in the following display names.

```
A year is not supported ✔
 ├── if it is zero ✔
 └── A negative value for year is not supported by the leap year computation. ✔
    ├── For example, year -1 is not supported. ✔
    └── For example, year -4 is not supported. ✔
```

With the `IndicativeSentences` display name generator, you can customize the separator and the underlying generator by using `@IndicativeSentencesGeneration` as shown in the following example.

```
@IndicativeSentencesGeneration(separator = " -> ", generator = ReplaceUnderscores
.class)
class A_year_is_a_leap_year {

    @Test
    void if_it_is_divisible_by_4_but_not_by_100() {
    }

    @ParameterizedTest(name = "Year {0} is a leap year.")
    @ValueSource(ints = { 2016, 2020, 2048 })
    void if_it_is_one_of_the_following_years(int year) {
    }

}
```

Running the above test class results in the following display names.

```
A year is a leap year ✔
 ├── A year is a leap year -> if it is divisible by 4 but not by 100 ✔
 └── A year is a leap year -> if it is one of the following years ✔
    ├── Year 2016 is a leap year. ✔
    ├── Year 2020 is a leap year. ✔
    └── Year 2048 is a leap year. ✔
```

With `IndicativeSentences`, you can optionally specify custom sentence fragments via the `@SentenceFragment` annotation as demonstrated in the following example.

```
@SentenceFragment("A year is a leap year")
@IndicativeSentencesGeneration
class LeapYearTests {

    @SentenceFragment("if it is divisible by 4 but not by 100")
    @Test
    void divisibleBy4ButNotBy100() {
    }

    @SentenceFragment("if it is one of the following years")
    @ParameterizedTest(name = "{0}")
    @ValueSource(ints = { 2016, 2020, 2048 })
    void validLeapYear(int year) {
    }

}
```

Running the above test class results in the following display names.

```
A year is a leap year ✔
 ├── A year is a leap year, if it is divisible by 4 but not by 100 ✔
 └── A year is a leap year, if it is one of the following years ✔
    ├── 2016 ✔
    ├── 2020 ✔
    └── 2048 ✔
```

Setting the Default Display Name Generator

You can use the `junit.jupiter.displayname.generator.default` configuration parameter to specify the fully qualified class name of the `DisplayNameGenerator` you would like to use by default. Just like for display name generators configured via the `@DisplayNameGeneration` annotation, the supplied class has to implement the `DisplayNameGenerator` interface. The default display name generator will be used for all tests unless the `@DisplayNameGeneration` annotation is present on an enclosing test class or test interface. Values provided via `@DisplayName` annotations always take precedence over

display names generated by a `DisplayNameGenerator`.

For example, to use the `ReplaceUnderscores` display name generator by default, you should set the configuration parameter to the corresponding fully qualified class name (e.g., in `src/test/resources/junit-platform.properties`):

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

Similarly, you can specify the fully qualified name of any custom class that implements `DisplayNameGenerator`.

In summary, the display name for a test class or method is determined according to the following precedence rules:

1. value of the `@DisplayName` annotation, if present
2. by calling the `DisplayNameGenerator` specified in the `@DisplayNameGeneration` annotation, if present
3. by calling the default `DisplayNameGenerator` configured via the configuration parameter, if present
4. by calling `org.junit.jupiter.api.DisplayNameGenerator.Standard`

Assertions

JUnit Jupiter comes with many of the assertion methods that JUnit 4 has and adds a few that lend themselves well to being used with Java lambdas. All JUnit Jupiter assertions are `static` methods in the `org.junit.jupiter.api.Assertions` class.

Assertion methods optionally accept the assertion message as their third parameter, which can be either a `String` or a `Supplier<String>`.

When using a `Supplier<String>` (e.g., a lambda expression), the message is evaluated lazily. This can provide a performance benefit, especially if message construction is complex or time-consuming, as it is only evaluated when the assertion fails.

```
import static java.time.Duration.ofMillis;
import static java.time.Duration.ofMinutes;
import static org.junit.jupiter.api.Assertions.assertAll;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTimeout;
import static org.junit.jupiter.api.Assertions.assertTimeoutPreemptively;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.concurrent.CountDownLatch;
```

```
import example.domain.Person;
import example.util.Calculator;

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

class AssertionsDemo {

    private final Calculator calculator = new Calculator();

    private final Person person = new Person("Jane", "Doe");

    @Test
    void standardAssertions() {
        assertEquals(2, calculator.add(1, 1));
        assertEquals(4, calculator.multiply(2, 2),
                "The optional failure message is now the last parameter");

        // Lazily evaluates generateFailureMessage('a','b').
        assertTrue('a' < 'b', () -> generateFailureMessage('a','b'));
    }

    @Test
    void groupedAssertions() {
        // In a grouped assertion all assertions are executed, and all
        // failures will be reported together.
        assertAll("person",
            () -> assertEquals("Jane", person.getFirstName()),
            () -> assertEquals("Doe", person.getLastName())
        );
    }

    @Test
    void dependentAssertions() {
        // Within a code block, if an assertion fails the
        // subsequent code in the same block will be skipped.
        assertAll("properties",
            () -> {
                String firstName = person.getFirstName();
                assertNotNull(firstName);

                // Executed only if the previous assertion is valid.
                assertAll("first name",
                    () -> assertTrue(firstName.startsWith("J")),
                    () -> assertTrue(firstName.endsWith("e"))
                );
            },
            () -> {
                // Grouped assertion, so processed independently
                // of results of first name assertions.
                String lastName = person.getLastName();
```

```
                assertNotNull(lastName);

                // Executed only if the previous assertion is valid.
                assertAll("last name",
                    () -> assertTrue(lastName.startsWith("D")),
                    () -> assertTrue(lastName.endsWith("e"))
                );
            }
        );
    }

    @Test
    void exceptionTesting() {
        Exception exception = assertThrows(ArithmeticException.class, () ->
            calculator.divide(1, 0));
        assertEquals("/ by zero", exception.getMessage());
    }

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

    @Test
    void timeoutNotExceededWithResult() {
        // The following assertion succeeds, and returns the supplied object.
        String actualResult = assertTimeout(ofMinutes(2), () -> {
            return "a result";
        });
        assertEquals("a result", actualResult);
    }

    @Test
    void timeoutNotExceededWithMethod() {
        // The following assertion invokes a method reference and returns an object.
        String actualGreeting = assertTimeout(ofMinutes(2), AssertionsDemo::greeting);
        assertEquals("Hello, World!", actualGreeting);
    }

    @Test
    void timeoutExceeded() {
        // The following assertion fails with an error message similar to:
        // execution exceeded timeout of 10 ms by 91 ms
        assertTimeout(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms.
            Thread.sleep(100);
        });
    }
```

```
    @Test
    void timeoutExceededWithPreemptiveTermination() {
        // The following assertion fails with an error message similar to:
        // execution timed out after 10 ms
        assertTimeoutPreemptively(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms.
            new CountDownLatch(1).await();
        });
    }

    private static String greeting() {
        return "Hello, World!";
    }

    private static String generateFailureMessage(char a, char b) {
        return "Assertion messages can be lazily evaluated -- "
                + "to avoid constructing complex messages unnecessarily." + (a < b);
    }
}
```



Preemptive Timeouts with `assertTimeoutPreemptively()`

The various `assertTimeoutPreemptively()` methods in the `Assertions` class execute the provided `executable` or `supplier` in a different thread than that of the calling code. This behavior can lead to undesirable side effects if the code that is executed within the `executable` or `supplier` relies on `java.lang.ThreadLocal` storage.

One common example of this is the transactional testing support in the Spring Framework. Specifically, Spring's testing support binds transaction state to the current thread (via a `ThreadLocal`) before a test method is invoked. Consequently, if an `executable` or `supplier` provided to `assertTimeoutPreemptively()` invokes Spring-managed components that participate in transactions, any actions taken by those components will not be rolled back with the test-managed transaction. On the contrary, such actions will be committed to the persistent store (e.g., relational database) even though the test-managed transaction is rolled back.

Similar side effects may be encountered with other frameworks that rely on `ThreadLocal` storage.

Kotlin Assertion Support

JUnit Jupiter also comes with a few assertion methods that lend themselves well to being used in Kotlin. All JUnit Jupiter Kotlin assertions are top-level functions in the `org.junit.jupiter.api` package.

```
import example.domain.Person
import example.util.Calculator
import org.junit.jupiter.api.Assertions.assertEquals
```

```
import org.junit.jupiter.api.Assertions.assertTrue
import org.junit.jupiter.api.Tag
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.assertAll
import org.junit.jupiter.api.assertDoesNotThrow
import org.junit.jupiter.api.assertInstanceOf
import org.junit.jupiter.api.assertNotNull
import org.junit.jupiter.api.assertThrows
import org.junit.jupiter.api.assertTimeout
import org.junit.jupiter.api.assertTimeoutPreemptively
import java.time.Duration

class KotlinAssertionsDemo {
    private val person = Person("Jane", "Doe")
    private val people = setOf(person, Person("John", "Doe"))

    @Test
    fun `exception absence testing`() {
        val calculator = Calculator()
        val result =
            assertDoesNotThrow("Should not throw an exception") {
                calculator.divide(0, 1)
            }
        assertEquals(0, result)
    }

    @Test
    fun `expected exception testing`() {
        val calculator = Calculator()
        val exception =
            assertThrows<ArithmeticException> ("Should throw an exception") {
                calculator.divide(1, 0)
            }
        assertEquals("/ by zero", exception.message)
    }

    @Test
    fun `grouped assertions`() {
        assertAll(
            "Person properties",
            { assertEquals("Jane", person.firstName) },
            { assertEquals("Doe", person.lastName) }
        )
    }

    @Test
    fun `grouped assertions from a stream`() {
        assertAll(
            "People with first name starting with J",
            people
                .stream()
```

```
                .map {
                    // This mapping returns Stream<() -> Unit>
                    { assertTrue(it.firstName.startsWith("J")) }
                }
        )
    }

    @Test
    fun `grouped assertions from a collection`() {
        assertAll(
            "People with last name of Doe",
            people.map { { assertEquals("Doe", it.lastName) } }
        )
    }

    @Test
    fun `timeout not exceeded testing`() {
        val fibonacciCalculator = FibonacciCalculator()
        val result =
            assertTimeout(Duration.ofMillis(1000)) {
                fibonacciCalculator.fib(14)
            }
        assertEquals(377, result)
    }

    @Test
    fun `timeout exceeded with preemptive termination`() {
        // The following assertion fails with an error message similar to:
        // execution timed out after 10 ms
        assertTimeoutPreemptively(Duration.ofMillis(10)) {
            // Simulate task that takes more than 10 ms.
            Thread.sleep(100)
        }
    }

    @Test
    fun `assertNotNull with a smart cast`() {
        val nullablePerson: Person? = person

        assertNotNull(nullablePerson)

        // The compiler smart casts nullablePerson to a non-nullable object.
        // The safe call operator (?.) isn't required.
        assertEquals(person.firstName, nullablePerson.firstName)
        assertEquals(person.lastName, nullablePerson.lastName)
    }

    @Test
    fun `assertInstanceOf with a smart cast`() {
        val maybePerson: Any = person
```

```
        assertInstanceOf<Person>(maybePerson)

        // The compiler smart casts maybePerson to a Person object,
        // allowing to access the Person properties.
        assertEquals(person.firstName, maybePerson.firstName)
        assertEquals(person.lastName, maybePerson.lastName)
    }
}
```

Third-party Assertion Libraries

Even though the assertion facilities provided by JUnit Jupiter are sufficient for many testing scenarios, there are times when more power and additional functionality are desired or required. In such cases, the JUnit team recommends the use of third-party assertion libraries such as AssertJ, Hamcrest, Truth, etc. Developers are therefore free to use the assertion library of their choice.

For example, the following demonstrates how to use the `assertThat()` support from AssertJ in a JUnit Jupiter test. As long as the AssertJ library has been added to the classpath, you can statically import methods such as `assertThat()`, `assertThatException()`, etc. from `org.assertj.core.api.Assertions` and then use them in tests like in the `assertWithAssertJ()` method below.

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

import example.util.Calculator;

import org.junit.jupiter.api.Test;

class AssertJAssertionsDemo {

    private final Calculator calculator = new Calculator();

    @Test
    void assertWithAssertJ() {
        assertThat(calculator.subtract(4, 1)).isEqualTo(3);
    }

}
```



Excluding Jupiter's Assertions From a Project's Classpath

If you would like to enforce that all your tests use a certain third-party assertion library instead of Jupiter's, you can set up a rule using Checkstyle or another static analysis tool that fails the build if Jupiter's `Assertions` class is used.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Checkstyle//DTD Checkstyle Configuration
1.3//EN" "https://checkstyle.org/dtds/configuration_1_3.dtd">
```

```
<module name="Checker">
    <property name="severity" value="error" />
    <module name="TreeWalker">
    <module
name="com.puppycrawl.tools.checkstyle.checks.regexp.RegexpSinglelineJav
aCheck">
            <property name="id" value="jupiterAssertions"/>
            <property name="maximum" value="0"/>
            <property name="format"
value="org\.junit\.jupiter\.api\.(Assertions|Assumptions)\."/>
            <property name="message" value="Jupiter
Assertions/Assumptions should not be used in this project. Please use
... instead."/>
            <property name="ignoreComments" value="true"/>
        </module>
    </module>
</module>
```

Assumptions

Assumptions are typically used whenever it does not make sense to continue execution of a given test — for example, if the test depends on something that does not exist in the current runtime environment.

- When an assumption is valid, the assumption method does not throw an exception, and execution of the test continues as usual.
- When an assumption is invalid, the assumption method throws an exception of type `org.opentest4j.TestAbortedException` to signal that the test should be aborted instead of marked as a failure.

JUnit Jupiter comes with a subset of the *assumption* methods that JUnit 4 provides and adds a few that lend themselves well to being used with Java lambda expressions and method references.

All JUnit Jupiter assumptions are static methods in the `org.junit.jupiter.api.Assumptions` class.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
import static org.junit.jupiter.api.Assumptions.assumingThat;

import example.util.Calculator;

import org.junit.jupiter.api.Test;

class AssumptionsDemo {

    private final Calculator calculator = new Calculator();

    @Test
```

```
    void testOnlyOnCiServer() {
        assumeTrue("CI".equals(System.getenv("ENV")));
        // remainder of test
    }

    @Test
    void testOnlyOnDeveloperWorkstation() {
        assumeTrue("DEV".equals(System.getenv("ENV")),
            () -> "Aborting test: not on developer workstation");
        // remainder of test
    }

    @Test
    void testInAllEnvironments() {
        assumingThat("CI".equals(System.getenv("ENV")),
            () -> {
                // perform these assertions only on the CI server
                assertEquals(2, calculator.divide(4, 2));
            });

        // perform these assertions in all environments
        assertEquals(42, calculator.multiply(6, 7));
    }

}
```



It is also possible to use methods from JUnit 4's `org.junit.Assume` class for assumptions. Specifically, JUnit Jupiter supports JUnit 4's `AssumptionViolatedException` to signal that a test should be aborted instead of marked as a failure.



If you use AssertJ for assertions, you may also wish to use AssertJ for assumptions. To do so, you can statically import the `assumeThat()` method from `org.assertj.core.api.Assumptions` and then use AssertJ's fluent API to specify your assumptions.

Exception Handling

JUnit Jupiter provides robust support for handling test exceptions. This includes the built-in mechanisms for managing test failures due to exceptions, the role of exceptions in implementing assertions and assumptions, and how to specifically assert non-throwing conditions in code.

Uncaught Exceptions

In JUnit Jupiter, if an exception is thrown from a test method, a lifecycle method, or an extension and not caught within that test method, lifecycle method, or extension, the framework will mark the test or test class as failed.



Failed assumptions deviate from this general rule.

In contrast to failed assertions, failed assumptions do not result in a test failure; rather, a failed assumption results in a test being aborted.

See Assumptions for further details and examples.

In the following example, the `failsDueToUncaughtException()` method throws an `ArithmeticException`. Since the exception is not caught within the test method, JUnit Jupiter will mark the test as failed.

```
private final Calculator calculator = new Calculator();

@Test
void failsDueToUncaughtException() {
    // The following throws an ArithmeticException due to division by
    // zero, which causes a test failure.
    calculator.divide(1, 0);
}
```



It's important to note that specifying a `throws` clause in the test method has no effect on the outcome of the test. JUnit Jupiter does not interpret a `throws` clause as an expectation or assertion about what exceptions the test method should throw. A test fails only if an exception is thrown unexpectedly or if an assertion fails.

Failed Assertions

Assertions in JUnit Jupiter are implemented using exceptions. The framework provides a set of assertion methods in the `org.junit.jupiter.api.Assertions` class, which throw `AssertionError` when an assertion fails. This mechanism is a core aspect of how JUnit handles assertion failures as exceptions. See the Assertions section for further information about JUnit Jupiter's assertion support.



Third-party assertion libraries may choose to throw an `AssertionError` to signal a failed assertion; however, they may also choose to throw different types of exceptions to signal failures. See also: Third-party Assertion Libraries.



JUnit Jupiter itself does not differentiate between failed assertions (`AssertionError`) and other types of exceptions. All uncaught exceptions lead to a test failure. However, Integrated Development Environments (IDEs) and other tools may distinguish between these two types of failures by checking whether the thrown exception is an instance of `AssertionError`.

In the following example, the `failsDueToUncaughtAssertionError()` method throws an `AssertionError`. Since the exception is not caught within the test method, JUnit Jupiter will mark the test as failed.

```
private final Calculator calculator = new Calculator();

@Test
void failsDueToUncaughtAssertionError() {
    // The following incorrect assertion will cause a test failure.
    // The expected value should be 2 instead of 99.
    assertEquals(99, calculator.add(1, 1));
}
```

Asserting Expected Exceptions

JUnit Jupiter offers specialized assertions for testing that specific exceptions are thrown under expected conditions. The `assertThrows()` and `assertThrowsExactly()` assertions are critical tools for validating that your code responds correctly to error conditions by throwing the appropriate exceptions.

Using `assertThrows()`

The `assertThrows()` method is used to verify that a particular type of exception is thrown during the execution of a provided executable block. It not only checks for the type of the thrown exception but also its subclasses, making it suitable for more generalized exception handling tests. The `assertThrows()` assertion method returns the thrown exception object to allow performing additional assertions on it.

```
@Test
void testExpectedExceptionIsThrown() {
    // The following assertion succeeds because the code under assertion
    // throws the expected IllegalArgumentException.
    // The assertion also returns the thrown exception which can be used for
    // further assertions like asserting the exception message.
    IllegalArgumentException exception =
        assertThrows(IllegalArgumentException.class, () -> {
            throw new IllegalArgumentException("expected message");
        });
    assertEquals("expected message", exception.getMessage());

    // The following assertion also succeeds because the code under assertion
    // throws IllegalArgumentException which is a subclass of RuntimeException.
    assertThrows(RuntimeException.class, () -> {
        throw new IllegalArgumentException("expected message");
    });
}
```

Using `assertThrowsExactly()`

The `assertThrowsExactly()` method is used when you need to assert that the exception thrown is exactly of a specific type, not allowing for subclasses of the expected exception type. This is useful when precise exception handling behavior needs to be validated. Similar to `assertThrows()`, the

`assertThrowsExactly()` assertion method also returns the thrown exception object to allow performing additional assertions on it.

```
@Test
void testExpectedExceptionIsThrown() {
    // The following assertion succeeds because the code under assertion throws
    // IllegalArgumentException which is exactly equal to the expected type.
    // The assertion also returns the thrown exception which can be used for
    // further assertions like asserting the exception message.
    IllegalArgumentException exception =
        assertThrowsExactly(IllegalArgumentException.class, () -> {
            throw new IllegalArgumentException("expected message");
        });
    assertEquals("expected message", exception.getMessage());

    // The following assertion fails because the assertion expects exactly
    // RuntimeException to be thrown, not subclasses of RuntimeException.
    assertThrowsExactly(RuntimeException.class, () -> {
        throw new IllegalArgumentException("expected message");
    });
}
```

Asserting That no Exception is Expected

Although any exception thrown from a test method will cause the test to fail, there are certain use cases where it can be beneficial to explicitly assert that an exception is *not* thrown for a given code block within a test method. The `assertDoesNotThrow()` assertion can be used when you want to verify that a particular piece of code does not throw any exceptions.

```
@Test
void testExceptionIsNotThrown() {
    assertDoesNotThrow(() -> {
        shouldNotThrowException();
    });
}

void shouldNotThrowException() {
}
```



Third-party assertion libraries often provide similar support. For example, AssertJ has `assertThatNoException().isThrownBy(() → ⋯)`. See also: Third-party Assertion Libraries.

Disabling Tests

Entire test classes or individual test methods may be *disabled* via the `@Disabled` annotation, via one of the annotations discussed in Conditional Test Execution, or via a custom `ExecutionCondition`.

When `@Disabled` is applied at the class level, all test methods within that class are automatically disabled as well.

If a test method is disabled via `@Disabled`, that prevents execution of the test method and method-level lifecycle callbacks such as `@BeforeEach` methods, `@AfterEach` methods, and corresponding extension APIs. However, that does not prevent the test class from being instantiated, and it does not prevent the execution of class-level lifecycle callbacks such as `@BeforeAll` methods, `@AfterAll` methods, and corresponding extension APIs.

Here's a `@Disabled` test class.

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

@Disabled("Disabled until bug #99 has been fixed")
class DisabledClassDemo {

    @Test
    void testWillBeSkipped() {
    }

}
```

And here's a test class that contains a `@Disabled` test method.

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

class DisabledTestsDemo {

    @Disabled("Disabled until bug #42 has been resolved")
    @Test
    void testWillBeSkipped() {
    }

    @Test
    void testWillBeExecuted() {
    }

}
```



`@Disabled` may be declared without providing a *reason*; however, the JUnit team recommends that developers provide a short explanation for why a test class or test method has been disabled. Consequently, the above examples both show the use of a reason — for example, `@Disabled("Disabled until bug #42 has been resolved")`. Some development teams even require the presence of issue tracking numbers in the *reason* for automated traceability, etc.



`@Disabled` is not `@Inherited`. Consequently, if you wish to disable a class whose superclass is `@Disabled`, you must redeclare `@Disabled` on the subclass.

Conditional Test Execution

The `ExecutionCondition` extension API in JUnit Jupiter allows developers to either *enable* or *disable* a test class or test method based on certain conditions *programmatically*. The simplest example of such a condition is the built-in `DisabledCondition` which supports the `@Disabled` annotation (see Disabling Tests).

In addition to `@Disabled`, JUnit Jupiter also supports several other annotation-based conditions in the `org.junit.jupiter.api.condition` package that allow developers to enable or disable test classes and test methods *declaratively*. If you wish to provide details about why they might be disabled, every annotation associated with these built-in conditions has a `disabledReason` attribute available for that purpose.

When multiple `ExecutionCondition` extensions are registered, a test class or test method is disabled as soon as one of the conditions returns *disabled*. If a test class is disabled, all test methods within that class are automatically disabled as well. If a test method is disabled, that prevents execution of the test method and method-level lifecycle callbacks such as `@BeforeEach` methods, `@AfterEach` methods, and corresponding extension APIs. However, that does not prevent the test class from being instantiated, and it does not prevent the execution of class-level lifecycle callbacks such as `@BeforeAll` methods, `@AfterAll` methods, and corresponding extension APIs.

See `ExecutionCondition` and the following sections for details.



Composed Annotations

Note that any of the *conditional* annotations listed in the following sections may also be used as a meta-annotation in order to create a custom *composed annotation*. For example, the `@TestOnMac` annotation in the @EnabledOnOs demo shows how you can combine `@Test` and `@EnabledOnOs` in a single, reusable annotation.



Conditional annotations in JUnit Jupiter are not `@Inherited`. Consequently, if you wish to apply the same semantics to subclasses, each conditional annotation must be redeclared on each subclass.



Unless otherwise stated, each of the *conditional* annotations listed in the following sections can only be declared once on a given test interface, test class, or test method. If a conditional annotation is directly present, indirectly present, or meta-present multiple times on a given element, only the first such annotation discovered by JUnit will be used; any additional declarations will be silently ignored. Note, however, that each conditional annotation may be used in conjunction with other conditional annotations in the `org.junit.jupiter.api.condition` package.

Operating System and Architecture Conditions

A container or test may be enabled or disabled on a particular operating system, architecture, or combination of both via the `@EnabledOnOs` and `@DisabledOnOs` annotations.

Conditional execution based on operating system

```
@Test
@EnabledOnOs(MAC)
void onlyOnMacOs() {
    // ...
}

@TestOnMac
void testOnMac() {
    // ...
}

@Test
@EnabledOnOs({ LINUX, MAC })
void onLinuxOrMac() {
    // ...
}

@Test
@DisabledOnOs(WINDOWS)
void notOnWindows() {
    // ...
}

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Test
@EnabledOnOs(MAC)
@interface TestOnMac {
}
```

Conditional execution based on architecture

```
@Test
@EnabledOnOs(architectures = "aarch64")
void onAarch64() {
    // ...
}

@Test
@DisabledOnOs(architectures = "x86_64")
void notOnX86_64() {
    // ...
}
```

```
@Test
@EnabledOnOs(value = MAC, architectures = "aarch64")
void onNewMacs() {
    // ...
}

@Test
@DisabledOnOs(value = MAC, architectures = "aarch64")
void notOnNewMacs() {
    // ...
}
```

Java Runtime Environment Conditions

A container or test may be enabled or disabled on particular versions of the Java Runtime Environment (JRE) via the `@EnabledOnJre` and `@DisabledOnJre` annotations or on a particular range of versions of the JRE via the `@EnabledForJreRange` and `@DisabledForJreRange` annotations. The range effectively defaults to `JRE.JAVA_8` as the lower bound and `JRE.OTHER` as the upper bound, which allows usage of half open ranges.

The following listing demonstrates the use of these annotations with predefined JRE enum constants.

```
@Test
@EnabledOnJre(JAVA_17)
void onlyOnJava17() {
    // ...
}

@Test
@EnabledOnJre({ JAVA_17, JAVA_21 })
void onJava17And21() {
    // ...
}

@Test
@EnabledForJreRange(min = JAVA_21, max = JAVA_25)
void fromJava21To25() {
    // ...
}

@Test
@EnabledForJreRange(min = JAVA_21)
void onJava21ndHigher() {
    // ...
}

@Test
```

```
@EnabledForJreRange(max = JAVA_18)
void fromJava17To18() {
    // ...
}

@Test
@DisabledOnJre(JAVA_19)
void notOnJava19() {
    // ...
}

@Test
@DisabledForJreRange(min = JAVA_17, max = JAVA_17)
void notFromJava17To19() {
    // ...
}

@Test
@DisabledForJreRange(min = JAVA_19)
void notOnJava19AndHigher() {
    // ...
}

@Test
@DisabledForJreRange(max = JAVA_18)
void notFromJava17To18() {
    // ...
}
```

Since the enum constants defined in JRE are static for any given JUnit release, you might find that you need to configure a Java version that is not supported by the JRE enum. For example, when JUnit Jupiter 5.12 was released the JRE enum defined JAVA_25 as the highest supported Java version. However, you may wish to run your tests against later versions of Java. To support such use cases, you can specify arbitrary Java versions via the versions attributes in @EnabledOnJre and @DisabledOnJre and via the minVersion and maxVersion attributes in @EnabledForJreRange and @DisabledForJreRange.

The following listing demonstrates the use of these annotations with arbitrary Java versions.

```
@Test
@EnabledOnJre(versions = 26)
void onlyOnJava26() {
    // ...
}

@Test
@EnabledOnJre(versions = { 25, 26 })
// Can also be expressed as follows.
// @EnabledOnJre(value = JAVA_25, versions = 26)
```

```
void onJava25And26() {
    // ...
}

@Test
@EnabledForJreRange(minVersion = 26)
void onJava26AndHigher() {
    // ...
}

@Test
@EnabledForJreRange(minVersion = 25, maxVersion = 27)
// Can also be expressed as follows.
// @EnabledForJreRange(min = JAVA_25, maxVersion = 27)
void fromJava25To27() {
    // ...
}

@Test
@DisabledOnJre(versions = 26)
void notOnJava26() {
    // ...
}

@Test
@DisabledOnJre(versions = { 25, 26 })
// Can also be expressed as follows.
// @DisabledOnJre(value = JAVA_25, versions = 26)
void notOnJava25And26() {
    // ...
}

@Test
@DisabledForJreRange(minVersion = 26)
void notOnJava26AndHigher() {
    // ...
}

@Test
@DisabledForJreRange(minVersion = 25, maxVersion = 27)
// Can also be expressed as follows.
// @DisabledForJreRange(min = JAVA_25, maxVersion = 27)
void notFromJava25To27() {
    // ...
}
```

Native Image Conditions

A container or test may be enabled or disabled within a GraalVM native image via the `@EnabledInNativeImage` and `@DisabledInNativeImage` annotations. These annotations are typically

used when running tests within a native image using the Gradle and Maven plug-ins from the GraalVM Native Build Tools project.

```
@Test
@EnabledInNativeImage
void onlyWithinNativeImage() {
    // ...
}

@Test
@DisabledInNativeImage
void neverWithinNativeImage() {
    // ...
}
```

System Property Conditions

A container or test may be enabled or disabled based on the value of the `named` JVM system property via the `@EnabledIfSystemProperty` and `@DisabledIfSystemProperty` annotations. The value supplied via the `matches` attribute will be interpreted as a regular expression.

```
@Test
@EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
void onlyOn64BitArchitectures() {
    // ...
}

@Test
@DisabledIfSystemProperty(named = "ci-server", matches = "true")
void notOnCiServer() {
    // ...
}
```



`@EnabledIfSystemProperty` and `@DisabledIfSystemProperty` are *repeatable annotations*. Consequently, these annotations may be declared multiple times on a test interface, test class, or test method. Specifically, these annotations will be found if they are directly present, indirectly present, or meta-present on a given element.

Environment Variable Conditions

A container or test may be enabled or disabled based on the value of the `named` environment variable from the underlying operating system via the `@EnabledIfEnvironmentVariable` and `@DisabledIfEnvironmentVariable` annotations. The value supplied via the `matches` attribute will be interpreted as a regular expression.

```
@Test
@EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server")
void onlyOnStagingServer() {
    // ...
}

@Test
@DisabledIfEnvironmentVariable(named = "ENV", matches = ".*development.*")
void notOnDeveloperWorkstation() {
    // ...
}
```



`@EnabledIfEnvironmentVariable` and `@DisabledIfEnvironmentVariable` are *repeatable annotations*. Consequently, these annotations may be declared multiple times on a test interface, test class, or test method. Specifically, these annotations will be found if they are directly present, indirectly present, or meta-present on a given element.

Custom Conditions

As an alternative to implementing an `ExecutionCondition`, a container or test may be enabled or disabled based on a *condition method* configured via the `@EnabledIf` and `@DisabledIf` annotations. A condition method must have a `boolean` return type and may accept either no arguments or a single `ExtensionContext` argument.

The following test class demonstrates how to configure a local method named `customCondition` via `@EnabledIf` and `@DisabledIf`.

```
@Test
@EnabledIf("customCondition")
void enabled() {
    // ...
}

@Test
@DisabledIf("customCondition")
void disabled() {
    // ...
}

boolean customCondition() {
    return true;
}
```

Alternatively, the condition method can be located outside the test class. In this case, it must be referenced by its *fully qualified name* as demonstrated in the following example.

```
package example;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.condition.EnabledIf;

class ExternalCustomConditionDemo {

    @Test
    @EnabledIf("example.ExternalCondition#customCondition")
    void enabled() {
        // ...
    }

}

class ExternalCondition {

    static boolean customCondition() {
        return true;
    }

}
```



There are several cases where a condition method would need to be `static`:

- when `@EnabledIf` or `@DisabledIf` is used at class level
- when `@EnabledIf` or `@DisabledIf` is used on a `@ParameterizedTest` or a `@TestTemplate` method
- when the condition method is located in an external class

In any other case, you can use either static methods or instance methods as condition methods.



It is often the case that you can use an existing static method in a utility class as a custom condition.

For example, `java.awt.GraphicsEnvironment` provides a `public static boolean isHeadless()` method that can be used to determine if the current environment does not support a graphical display. Thus, if you have a test that depends on graphical support you can disable it when such support is unavailable as follows.

```
@DisabledIf(value = "java.awt.GraphicsEnvironment#isHeadless",
    disabledReason = "headless environment")
```

Tagging and Filtering

Test classes and methods can be tagged via the `@Tag` annotation. Those tags can later be used to filter test discovery and execution. Please refer to the Tags section for more information about tag support in the JUnit Platform.

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

@Tag("fast")
@Tag("model")
class TaggingDemo {

    @Test
    @Tag("taxes")
    void testingTaxCalculation() {
    }

}
```



See Meta-Annotations and Composed Annotations for examples demonstrating how to create custom annotations for tags.

Test Execution Order

By default, test classes and methods will be ordered using an algorithm that is deterministic but intentionally nonobvious. This ensures that subsequent runs of a test suite execute test classes and test methods in the same order, thereby allowing for repeatable builds.



See Definitions for a definition of *test method* and *test class*.

Method Order

Although true *unit tests* typically should not rely on the order in which they are executed, there are times when it is necessary to enforce a specific test method execution order — for example, when writing *integration tests* or *functional tests* where the sequence of the tests is important, especially in conjunction with `@TestInstance(Lifecycle.PER_CLASS)`.

To control the order in which test methods are executed, annotate your test class or test interface with `@TestMethodOrder` and specify the desired `MethodOrderer` implementation. You can implement your own custom `MethodOrderer` or use one of the following built-in `MethodOrderer` implementations.

- `MethodOrderer.DisplayName`: sorts test methods *alphanumerically* based on their display names (see display name generation precedence rules)
- `MethodOrderer.MethodName`: sorts test methods *alphanumerically* based on their names and formal parameter lists

- `MethodOrderer.OrderAnnotation`: sorts test methods *numerically* based on values specified via the `@Order` annotation
- `MethodOrderer.Random`: orders test methods *pseudo-randomly* and supports configuration of a custom *seed*

The `MethodOrderer` configured on a test class is inherited by the `@Nested` test classes it contains, recursively. If you want to avoid that a `@Nested` test class uses the same `MethodOrderer` as its enclosing class, you can specify `MethodOrderer.Default` together with `@TestMethodOrder`.



See also: Wrapping Behavior of Callbacks

The following example demonstrates how to guarantee that test methods are executed in the order specified via the `@Order` annotation.

```
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;

@TestMethodOrder(OrderAnnotation.class)
class OrderedTestsDemo {

    @Test
    @Order(1)
    void nullValues() {
        // perform assertions against null values
    }

    @Test
    @Order(2)
    void emptyValues() {
        // perform assertions against empty values
    }

    @Test
    @Order(3)
    void validValues() {
        // perform assertions against valid values
    }

}
```

Setting the Default Method Orderer

You can use the `junit.jupiter.testmethod.order.default` configuration parameter to specify the fully qualified class name of the `MethodOrderer` you would like to use by default. Just like for the orderer configured via the `@TestMethodOrder` annotation, the supplied class has to implement the `MethodOrderer` interface. The default orderer will be used for all tests unless the `@TestMethodOrder`

annotation is present on an enclosing test class or test interface.

For example, to use the `MethodOrderer.OrderAnnotation` method orderer by default, you should set the configuration parameter to the corresponding fully qualified class name (e.g., in `src/test/resources/junit-platform.properties`):

```
junit.jupiter.testmethod.order.default = \
    org.junit.jupiter.api.MethodOrderer$OrderAnnotation
```

Similarly, you can specify the fully qualified name of any custom class that implements `MethodOrderer`.

Class Order

Although test classes typically should not rely on the order in which they are executed, there are times when it is desirable to enforce a specific test class execution order. You may wish to execute test classes in a random order to ensure there are no accidental dependencies between test classes, or you may wish to order test classes to optimize build time as outlined in the following scenarios.

- Run previously failing tests and faster tests first: "fail fast" mode
- With parallel execution enabled, schedule longer tests first: "shortest test plan execution duration" mode
- Various other use cases

To configure test class execution order *globally* for the entire test suite, use the `junit.jupiter.testclass.order.default` configuration parameter to specify the fully qualified class name of the `ClassOrderer` you would like to use. The supplied class must implement the `ClassOrderer` interface.

You can implement your own custom `ClassOrderer` or use one of the following built-in `ClassOrderer` implementations.

- `ClassOrderer.ClassName`: sorts test classes *alphanumerically* based on their fully qualified class names
- `ClassOrderer.DisplayName`: sorts test classes *alphanumerically* based on their display names (see display name generation precedence rules)
- `ClassOrderer.OrderAnnotation`: sorts test classes *numerically* based on values specified via the `@Order` annotation
- `ClassOrderer.Random`: orders test classes *pseudo-randomly* and supports configuration of a custom *seed*

For example, for the `@Order` annotation to be honored on *test classes*, you should configure the `ClassOrderer.OrderAnnotation` class orderer using the configuration parameter with the corresponding fully qualified class name (e.g., in `src/test/resources/junit-platform.properties`):

```
junit.jupiter.testclass.order.default = \
```

```
org.junit.jupiter.api.ClassOrderer$OrderAnnotation
```

The configured `ClassOrderer` will be applied to all top-level test classes (including `static` nested test classes) and `@Nested` test classes.



Top-level test classes will be ordered relative to each other; whereas, `@Nested` test classes will be ordered relative to other `@Nested` test classes sharing the same *enclosing class*.

To configure test class execution order *locally* for `@Nested` test classes, declare the `@TestClassOrder` annotation on the enclosing class for the `@Nested` test classes you want to order, and supply a class reference to the `ClassOrderer` implementation you would like to use directly in the `@TestClassOrder` annotation. The configured `ClassOrderer` will be applied recursively to `@Nested` test classes and their `@Nested` test classes. If you want to avoid that a `@Nested` test class uses the same `ClassOrderer` as its enclosing class, you can specify `ClassOrderer.Default` together with `@TestClassOrder`. Note that a local `@TestClassOrder` declaration always overrides an inherited `@TestClassOrder` declaration or a `ClassOrderer` configured globally via the `junit.jupiter.testclass.order.default` configuration parameter.

The following example demonstrates how to guarantee that `@Nested` test classes are executed in the order specified via the `@Order` annotation.

```
import org.junit.jupiter.api.ClassOrderer;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestClassOrder;

@TestClassOrder(ClassOrderer.OrderAnnotation.class)
class OrderedNestedTestClassesDemo {

    @Nested
    @Order(1)
    class PrimaryTests {

        @Test
        void test1() {
        }
    }

    @Nested
    @Order(2)
    class SecondaryTests {

        @Test
        void test2() {
        }
    }
```

```
}
```

Test Instance Lifecycle

In order to allow individual test methods to be executed in isolation and to avoid unexpected side effects due to mutable test instance state, JUnit creates a new instance of each test class before executing each *test method* (see Definitions). This "per-method" test instance lifecycle is the default behavior in JUnit Jupiter and is analogous to all previous versions of JUnit.



Please note that the test class will still be instantiated if a given *test method* is *disabled* via a condition (e.g., `@Disabled`, `@DisabledOnOs`, etc.) even when the "per-method" test instance lifecycle mode is active.

If you would prefer that JUnit Jupiter execute all test methods on the same test instance, annotate your test class with `@TestInstance(Lifecycle.PER_CLASS)`. When using this mode, a new test instance will be created once per test class. Thus, if your test methods rely on state stored in instance variables, you may need to reset that state in `@BeforeEach` or `@AfterEach` methods.

The "per-class" mode has some additional benefits over the default "per-method" mode. Specifically, with the "per-class" mode it becomes possible to declare `@BeforeAll` and `@AfterAll` on non-static methods as well as on interface `default` methods.

If you are authoring tests using the Kotlin programming language, you may also find it easier to implement non-static `@BeforeAll` and `@AfterAll` lifecycle methods as well as `@MethodSource` factory methods by switching to the "per-class" test instance lifecycle mode.

Changing the Default Test Instance Lifecycle

If a test class or test interface is not annotated with `@TestInstance`, JUnit Jupiter will use a *default* lifecycle mode. The standard *default* mode is `PER_METHOD`; however, it is possible to change the *default* for the execution of an entire test plan. To change the default test instance lifecycle mode, set the `junit.jupiter.testinstance.lifecycle.default` *configuration parameter* to the name of an enum constant defined in `TestInstance.Lifecycle`, ignoring case. This can be supplied as a JVM system property, as a *configuration parameter* in the `LauncherDiscoveryRequest` that is passed to the `Launcher`, or via the JUnit Platform configuration file (see Configuration Parameters for details).

For example, to set the default test instance lifecycle mode to `Lifecycle.PER_CLASS`, you can start your JVM with the following system property.

```
-Djunit.jupiter.testinstance.lifecycle.default=per_class
```

Note, however, that setting the default test instance lifecycle mode via the JUnit Platform configuration file is a more robust solution since the configuration file can be checked into a version control system along with your project and can therefore be used within IDEs and your build software.

To set the default test instance lifecycle mode to `Lifecycle.PER_CLASS` via the JUnit Platform configuration file, create a file named `junit-platform.properties` in the root of the class path (e.g.,

`src/test/resources`) with the following content.

```
junit.jupiter.testinstance.lifecycle.default = per_class
```



Changing the *default* test instance lifecycle mode can lead to unpredictable results and fragile builds if not applied consistently. For example, if the build configures "per-class" semantics as the default but tests in the IDE are executed using "per-method" semantics, that can make it difficult to debug errors that occur on the build server. It is therefore recommended to change the default in the JUnit Platform configuration file instead of via a JVM system property.

Nested Tests

`@Nested` tests give the test writer more capabilities to express the relationship among several groups of tests. Such nested tests make use of Java's nested classes and facilitate hierarchical thinking about the test structure. Here's an elaborate example, both as source code and as a screenshot of the execution within an IDE.

Nested test suite for testing a stack

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.EmptyStackException;
import java.util.Stack;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

@DisplayName("A stack")
class TestingAStackDemo {

    @Test
    @DisplayName("is instantiated with new Stack()")
    void isInstantiatedWithNew() {
        new Stack<>();
    }

    @Nested
    @DisplayName("when new")
    class WhenNew {

        Stack<Object> stack;

        @BeforeEach
        void createNewStack() {
```

```
        stack = new Stack<>();
    }

    @Test
    @DisplayName("is empty")
    void isEmpty() {
        assertTrue(stack.isEmpty());
    }

    @Test
    @DisplayName("throws EmptyStackException when popped")
    void throwsExceptionWhenPopped() {
        assertThrows(EmptyStackException.class, stack::pop);
    }

    @Test
    @DisplayName("throws EmptyStackException when peeked")
    void throwsExceptionWhenPeeked() {
        assertThrows(EmptyStackException.class, stack::peek);
    }

    @Nested
    @DisplayName("after pushing an element")
    class AfterPushing {

        String anElement = "an element";

        @BeforeEach
        void pushAnElement() {
            stack.push(anElement);
        }

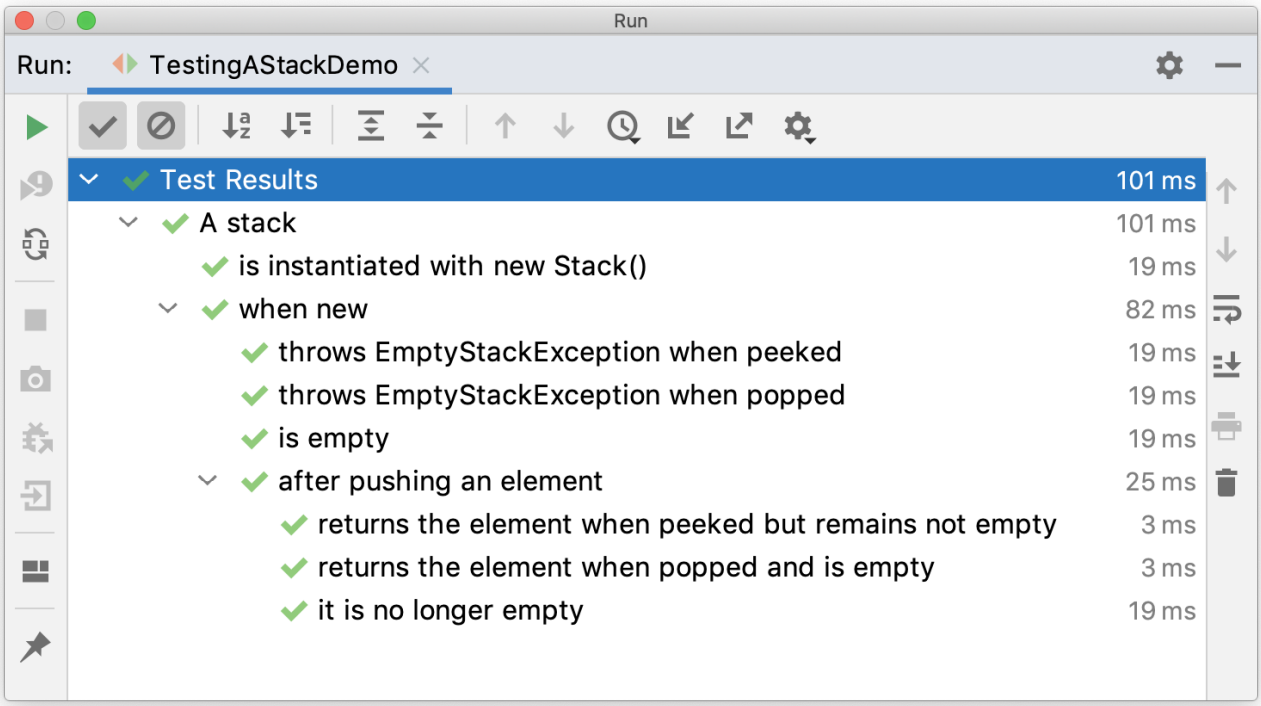
        @Test
        @DisplayName("it is no longer empty")
        void isNotEmpty() {
            assertFalse(stack.isEmpty());
        }

        @Test
        @DisplayName("returns the element when popped and is empty")
        void returnElementWhenPopped() {
            assertEquals(anElement, stack.pop());
            assertTrue(stack.isEmpty());
        }

        @Test
        @DisplayName("returns the element when peeked but remains not empty")
        void returnElementWhenPeeked() {
            assertEquals(anElement, stack.peek());
            assertFalse(stack.isEmpty());
        }
```

```
        }
    }
}
```

When executing this example in an IDE, the test execution tree in the GUI will look similar to the following image.



Executing a nested test in an IDE

In this example, preconditions from outer tests are used in inner tests by defining hierarchical lifecycle methods for the setup code. For example, `createNewStack()` is a `@BeforeEach` lifecycle method that is used in the test class in which it is defined and in all levels in the nesting tree below the class in which it is defined.

The fact that setup code from outer tests is run before inner tests are executed gives you the ability to run all tests independently. You can even run inner tests alone without running the outer tests, because the setup code from the outer tests is always executed.



Only non-static nested classes (i.e. *inner classes*) can serve as `@Nested` test classes. Nesting can be arbitrarily deep, and those inner classes are subject to full lifecycle support, including `@BeforeAll` and `@AfterAll` methods on each level.

Interoperability

`@Nested` may be combined with `@ParameterizedClass` in which case the nested test class is parameterized.

The following example illustrates how to combine `@Nested` with `@ParameterizedClass` and `@ParameterizedTest`.

```
@Execution(SAME_THREAD)
@ParameterizedClass
@ValueSource(strings = { "apple", "banana" })
class FruitTests {

    @Parameter
    String fruit;

    @Nested
    @ParameterizedClass
    @ValueSource(ints = { 23, 42 })
    class QuantityTests {

        @Parameter
        int quantity;

        @ParameterizedTest
        @ValueSource(strings = { "PT1H", "PT2H" })
        void test(Duration duration) {
            assertFruit(fruit);
            assertQuantity(quantity);
            assertFalse(duration.isNegative());
        }
    }
}
```

Executing the above test class yields the following output:

```
FruitTests ✔
├── [1] fruit = "apple" ✔
│   └── QuantityTests ✔
│       ├── [1] quantity = 23 ✔
│       │   └── test(Duration) ✔
│       │       ├── [1] duration = "PT1H" ✔
│       │       └── [2] duration = "PT2H" ✔
│       └── [2] quantity = 42 ✔
│           └── test(Duration) ✔
│               ├── [1] duration = "PT1H" ✔
│               └── [2] duration = "PT2H" ✔
└── [2] fruit = "banana" ✔
    └── QuantityTests ✔
        ├── [1] quantity = 23 ✔
        │   └── test(Duration) ✔
        │       ├── [1] duration = "PT1H" ✔
        │       └── [2] duration = "PT2H" ✔
        └── [2] quantity = 42 ✔
            └── test(Duration) ✔
                ├── [1] duration = "PT1H" ✔
```

```
            └── [2] duration = "PT2H" ✔
```

Dependency Injection for Constructors and Methods

In all prior JUnit versions, test constructors or methods were not allowed to have parameters (at least not with the standard `Runner` implementations). As one of the major changes in JUnit Jupiter, both test constructors and methods are now permitted to have parameters. This allows for greater flexibility and enables *Dependency Injection* for constructors and methods.

`ParameterResolver` defines the API for test extensions that wish to *dynamically* resolve parameters at runtime. If a *test class* constructor, a *test method*, or a *lifecycle method* (see Definitions) accepts a parameter, the parameter must be resolved at runtime by a registered `ParameterResolver`.

There are currently three built-in resolvers that are registered automatically.

- `TestInfoParameterResolver`: if a constructor or method parameter is of type `TestInfo`, the `TestInfoParameterResolver` will supply an instance of `TestInfo` corresponding to the current container or test as the value for the parameter. The `TestInfo` can then be used to retrieve information about the current container or test such as the display name, the test class, the test method, and associated tags. The display name is either a technical name, such as the name of the test class or test method, or a custom name configured via `@DisplayName`.

  `TestInfo` acts as a drop-in replacement for the `TestName` rule from JUnit 4. The following demonstrates how to have `TestInfo` injected into a `@BeforeAll` method, test class constructor, `@BeforeEach` method, and `@Test` method.

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

import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInfo;

@DisplayName("TestInfo Demo")
class TestInfoDemo {

    @BeforeAll
    static void beforeAll(TestInfo testInfo) {
        assertEquals("TestInfo Demo", testInfo.getDisplayName());
    }

    TestInfoDemo(TestInfo testInfo) {
        String displayName = testInfo.getDisplayName();
        assertTrue(displayName.equals("TEST 1") || displayName.equals("test2()"));
    }
```

```
    @BeforeEach
    void init(TestInfo testInfo) {
        String displayName = testInfo.getDisplayName();
        assertTrue(displayName.equals("TEST 1") || displayName.equals("test2()"));
    }

    @Test
    @DisplayName("TEST 1")
    @Tag("my-tag")
    void test1(TestInfo testInfo) {
        assertEquals("TEST 1", testInfo.getDisplayName());
        assertTrue(testInfo.getTags().contains("my-tag"));
    }

    @Test
    void test2() {
    }

}
```

- `RepetitionExtension`: if a method parameter in a `@RepeatedTest`, `@BeforeEach`, or `@AfterEach` method is of type `RepetitionInfo`, the `RepetitionExtension` will supply an instance of `RepetitionInfo`. `RepetitionInfo` can then be used to retrieve information about the current repetition, the total number of repetitions, the number of repetitions that have failed, and the failure threshold for the corresponding `@RepeatedTest`. Note, however, that `RepetitionExtension` is not registered outside the context of a `@RepeatedTest`. See Repeated Test Examples.
- `TestReporterParameterResolver`: if a constructor or method parameter is of type `TestReporter`, the `TestReporterParameterResolver` will supply an instance of `TestReporter`. The `TestReporter` can be used to publish additional data about the current test run or attach files to it. The data can be consumed in a `TestExecutionListener` via the `reportingEntryPublished()` or `fileEntryPublished()` method, respectively. This allows them to be viewed in IDEs or included in reports.

  In JUnit Jupiter you should use `TestReporter` where you used to print information to `stdout` or `stderr` in JUnit 4. Some IDEs print report entries to `stdout` or display them in the user interface for test results.

```
class TestReporterDemo {

    @Test
    void reportSingleValue(TestReporter testReporter) {
        testReporter.publishEntry("a status message");
    }

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

    @Test
```

```
    void reportMultipleKeyValuePairs(TestReporter testReporter) {
        Map<String, String> values = new HashMap<>();
        values.put("user name", "dk38");
        values.put("award year", "1974");

        testReporter.publishEntry(values);
    }

    @Test
    void reportFiles(TestReporter testReporter, @TempDir Path tempDir) throws
Exception {

        testReporter.publishFile("test1.txt", MediaType.TEXT_PLAIN_UTF_8, file ->
Files.write(file, List.of("Test 1")));

        Path existingFile = Files.write(tempDir.resolve("test2.txt"), List.of("Test
2"));
        testReporter.publishFile(existingFile, MediaType.TEXT_PLAIN_UTF_8);

        testReporter.publishDirectory("test3", dir -> {
            Files.write(dir.resolve("nested1.txt"), List.of("Nested content 1"));
            Files.write(dir.resolve("nested2.txt"), List.of("Nested content 2"));
        });

        Path existingDir = Files.createDirectory(tempDir.resolve("test4"));
        Files.write(existingDir.resolve("nested1.txt"), List.of("Nested content 1"));
        Files.write(existingDir.resolve("nested2.txt"), List.of("Nested content 2"));
        testReporter.publishDirectory(existingDir);
    }
}
```



Other parameter resolvers must be explicitly enabled by registering appropriate extensions via `@ExtendWith`.

Check out the `RandomParametersExtension` for an example of a custom `ParameterResolver`. While not intended to be production-ready, it demonstrates the simplicity and expressiveness of both the extension model and the parameter resolution process. `MyRandomParametersTest` demonstrates how to inject random values into `@Test` methods.

```
@ExtendWith(RandomParametersExtension.class)
class MyRandomParametersTest {

    @Test
    void injectsInteger(@Random int i, @Random int j) {
        assertNotEquals(i, j);
    }

    @Test
    void injectsDouble(@Random double d) {
```

```
        assertEquals(0.0, d, 1.0);
    }

}
```

For real-world use cases, check out the source code for the `MockitoExtension` and the `SpringExtension`.

When the type of the parameter to inject is the only condition for your `ParameterResolver`, you can use the generic `TypeBasedParameterResolver` base class. The `supportsParameters` method is implemented behind the scenes and supports parameterized types.

Test Interfaces and Default Methods

JUnit Jupiter allows `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, `@TestTemplate`, `@BeforeEach`, and `@AfterEach` to be declared on interface `default` methods. `@BeforeAll` and `@AfterAll` can either be declared on `static` methods in a test interface or on interface `default` methods *if* the test interface or test class is annotated with `@TestInstance(Lifecycle.PER_CLASS)` (see Test Instance Lifecycle). Here are some examples.

```
@TestInstance(Lifecycle.PER_CLASS)
interface TestLifecycleLogger {

    Logger logger = Logger.getLogger(TestLifecycleLogger.class.getName());

    @BeforeAll
    default void beforeAllTests() {
        logger.info("Before all tests");
    }

    @AfterAll
    default void afterAllTests() {
        logger.info("After all tests");
    }

    @BeforeEach
    default void beforeEachTest(TestInfo testInfo) {
        logger.info(() -> "About to execute [%s]".formatted(
            testInfo.getDisplayName()));
    }

    @AfterEach
    default void afterEachTest(TestInfo testInfo) {
        logger.info(() -> "Finished executing [%s]".formatted(
            testInfo.getDisplayName()));
    }

}
```

```
interface TestInterfaceDynamicTestsDemo {

    @TestFactory
    default Stream<DynamicTest> dynamicTestsForPalindromes() {
        return Stream.of("racecar", "radar", "mom", "dad")
            .map(text -> dynamicTest(text, () -> assertTrue(isPalindrome(text))));
    }

}
```

`@ExtendWith` and `@Tag` can be declared on a test interface so that classes that implement the interface automatically inherit its tags and extensions. See Before and After Test Execution Callbacks for the source code of the TimingExtension.

```
@Tag("timed")
@ExtendWith(TimingExtension.class)
interface TimeExecutionLogger {
}
```

In your test class you can then implement these test interfaces to have them applied.

```
class TestInterfaceDemo implements TestLifecycleLogger,
        TimeExecutionLogger, TestInterfaceDynamicTestsDemo {

    @Test
    void isEqualValue() {
        assertEquals(1, "a".length(), "is always equal");
    }

}
```

Running the `TestInterfaceDemo` results in output similar to the following:

```
INFO  example.TestLifecycleLogger - Before all tests
INFO  example.TestLifecycleLogger - About to execute [dynamicTestsForPalindromes()]
INFO  example.TimingExtension - Method [dynamicTestsForPalindromes] took 19 ms.
INFO  example.TestLifecycleLogger - Finished executing [dynamicTestsForPalindromes()]
INFO  example.TestLifecycleLogger - About to execute [isEqualValue()]
INFO  example.TimingExtension - Method [isEqualValue] took 1 ms.
INFO  example.TestLifecycleLogger - Finished executing [isEqualValue()]
INFO  example.TestLifecycleLogger - After all tests
```

Another possible application of this feature is to write tests for interface contracts. For example, you can write tests for how implementations of `Object.equals` or `Comparable.compareTo` should behave as follows.

```
public interface Testable<T> {

    T createValue();

}
```

```
public interface EqualsContract<T> extends Testable<T> {

    T createNotEqualValue();

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

    @Test
    default void valueDoesNotEqualNull() {
        T value = createValue();
        assertNotEquals(null, value);
    }

    @Test
    default void valueDoesNotEqualDifferentValue() {
        T value = createValue();
        T differentValue = createNotEqualValue();
        assertNotEquals(value, differentValue);
        assertNotEquals(differentValue, value);
    }

}
```

```
public interface ComparableContract<T extends Comparable<T>> extends Testable<T> {

    T createSmallerValue();

    @Test
    default void returnsZeroWhenComparedToItself() {
        T value = createValue();
        assertEquals(0, value.compareTo(value));
    }

    @Test
    default void returnsPositiveNumberWhenComparedToSmallerValue() {
        T value = createValue();
        T smallerValue = createSmallerValue();
        assertTrue(value.compareTo(smallerValue) > 0);
    }
```

```
    @Test
    default void returnsNegativeNumberWhenComparedToLargerValue() {
        T value = createValue();
        T smallerValue = createSmallerValue();
        assertTrue(smallerValue.compareTo(value) < 0);
    }

}
```

In your test class you can then implement both contract interfaces thereby inheriting the corresponding tests. Of course you'll have to implement the abstract methods.

```
class StringTests implements ComparableContract<String>, EqualsContract<String> {

    @Override
    public String createValue() {
        return "banana";
    }

    @Override
    public String createSmallerValue() {
        return "apple"; // 'a' < 'b' in "banana"
    }

    @Override
    public String createNotEqualValue() {
        return "cherry";
    }

}
```



The above tests are merely meant as examples and therefore not complete.

Repeated Tests

JUnit Jupiter provides the ability to repeat a test a specified number of times by annotating a method with `@RepeatedTest` and specifying the total number of repetitions desired. Each invocation of a repeated test behaves like the execution of a regular `@Test` method with full support for the same lifecycle callbacks and extensions.

The following example demonstrates how to declare a test named `repeatedTest()` that will be automatically repeated 10 times.

```
@RepeatedTest(10)
void repeatedTest() {
    // ...
```

```
}
```

`@RepeatedTest` can be configured with a failure threshold which signifies the number of failures after which remaining repetitions will be automatically skipped. Set the `failureThreshold` attribute to a positive number less than the total number of repetitions in order to skip the invocations of remaining repetitions after the specified number of failures has been encountered.

For example, if you are using `@RepeatedTest` to repeatedly invoke a test that you suspect to be *flaky*, a single failure is sufficient to demonstrate that the test is flaky, and there is no need to invoke the remaining repetitions. To support that specific use case, set `failureThreshold = 1`. You can alternatively set the threshold to a number greater than 1 depending on your use case.

By default, the `failureThreshold` attribute is set to `Integer.MAX_VALUE`, signaling that no failure threshold will be applied, which effectively means that the specified number of repetitions will be invoked regardless of whether any repetitions fail.



If the repetitions of a `@RepeatedTest` method are executed in parallel, no guarantees can be made regarding the failure threshold. It is therefore recommended that a `@RepeatedTest` method be annotated with `@Execution(SAME_THREAD)` when parallel execution is configured. See Parallel Execution for further details.

In addition to specifying the number of repetitions and failure threshold, a custom display name can be configured for each repetition via the `name` attribute of the `@RepeatedTest` annotation. Furthermore, the display name can be a pattern composed of a combination of static text and dynamic placeholders. The following placeholders are currently supported.

- `{displayName}`: display name of the `@RepeatedTest` method
- `{currentRepetition}`: the current repetition count
- `{totalRepetitions}`: the total number of repetitions

The default display name for a given repetition is generated based on the following pattern: `"repetition {currentRepetition} of {totalRepetitions}"`.Thus, the display names for individual repetitions of the previous `repeatedTest()` example would be: `repetition 1 of 10`, `repetition 2 of 10`, etc.If you would like the display name of the `@RepeatedTest` method included in the name of each repetition, you can define your own custom pattern or use the predefined `RepeatedTest.LONG_DISPLAY_NAME` pattern.The latter is equal to `"{displayName} :: repetition {currentRepetition} of {totalRepetitions}"` which results in display names for individual repetitions like `repeatedTest() :: repetition 1 of 10`, `repeatedTest() :: repetition 2 of 10`, etc.

In order to retrieve information about the current repetition, the total number of repetitions, the number of repetitions that have failed, and the failure threshold, a developer can choose to have an instance of `RepetitionInfo` injected into a `@RepeatedTest`, `@BeforeEach`, or `@AfterEach` method.

Repeated Test Examples

The `RepeatedTestsDemo` class at the end of this section demonstrates several examples of repeated tests.

The `repeatedTest()` method is identical to the example from the previous section; whereas, `repeatedTestWithRepetitionInfo()` demonstrates how to have an instance of `RepetitionInfo` injected into a test to access the total number of repetitions for the current repeated test.

`repeatedTestWithFailureThreshold()` demonstrates how to set a failure threshold and simulates an unexpected failure for every second repetition.The resulting behavior can be viewed in the `ConsoleLauncher` output at the end of this section.

The next two methods demonstrate how to include a custom `@DisplayName` for the `@RepeatedTest` method in the display name of each repetition. `customDisplayName()` combines a custom display name with a custom pattern and then uses `TestInfo` to verify the format of the generated display name. `Repeat!` is the `{displayName}` which comes from the `@DisplayName` declaration, and `1/1` comes from `{currentRepetition}/{totalRepetitions}`.In contrast, `customDisplayNameWithLongPattern()` uses the aforementioned predefined `RepeatedTest.LONG_DISPLAY_NAME` pattern.

`repeatedTestInGerman()` demonstrates the ability to translate display names of repeated tests into foreign languages — in this case German, resulting in names for individual repetitions such as: `Wiederholung 1 von 5`, `Wiederholung 2 von 5`, etc.

Since the `beforeEach()` method is annotated with `@BeforeEach` it will get executed before each repetition of each repeated test. By having the `TestInfo` and `RepetitionInfo` injected into the method, we see that it's possible to obtain information about the currently executing repeated test. Executing `RepeatedTestsDemo` with the `INFO` log level enabled results in the following output.

```
INFO: About to execute repetition 1 of 10 for repeatedTest
INFO: About to execute repetition 2 of 10 for repeatedTest
INFO: About to execute repetition 3 of 10 for repeatedTest
INFO: About to execute repetition 4 of 10 for repeatedTest
INFO: About to execute repetition 5 of 10 for repeatedTest
INFO: About to execute repetition 6 of 10 for repeatedTest
INFO: About to execute repetition 7 of 10 for repeatedTest
INFO: About to execute repetition 8 of 10 for repeatedTest
INFO: About to execute repetition 9 of 10 for repeatedTest
INFO: About to execute repetition 10 of 10 for repeatedTest
INFO: About to execute repetition 1 of 5 for repeatedTestWithRepetitionInfo
INFO: About to execute repetition 2 of 5 for repeatedTestWithRepetitionInfo
INFO: About to execute repetition 3 of 5 for repeatedTestWithRepetitionInfo
INFO: About to execute repetition 4 of 5 for repeatedTestWithRepetitionInfo
INFO: About to execute repetition 5 of 5 for repeatedTestWithRepetitionInfo
INFO: About to execute repetition 1 of 8 for repeatedTestWithFailureThreshold
INFO: About to execute repetition 2 of 8 for repeatedTestWithFailureThreshold
INFO: About to execute repetition 3 of 8 for repeatedTestWithFailureThreshold
INFO: About to execute repetition 4 of 8 for repeatedTestWithFailureThreshold
INFO: About to execute repetition 1 of 1 for customDisplayName
INFO: About to execute repetition 1 of 1 for customDisplayNameWithLongPattern
INFO: About to execute repetition 1 of 5 for repeatedTestInGerman
INFO: About to execute repetition 2 of 5 for repeatedTestInGerman
INFO: About to execute repetition 3 of 5 for repeatedTestInGerman
INFO: About to execute repetition 4 of 5 for repeatedTestInGerman
```

```
INFO: About to execute repetition 5 of 5 for repeatedTestInGerman
```

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

import java.util.logging.Logger;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.RepetitionInfo;
import org.junit.jupiter.api.TestInfo;

class RepeatedTestsDemo {

    private Logger logger = // ...

    @BeforeEach
    void beforeEach(TestInfo testInfo, RepetitionInfo repetitionInfo) {
        int currentRepetition = repetitionInfo.getCurrentRepetition();
        int totalRepetitions = repetitionInfo.getTotalRepetitions();
        String methodName = testInfo.getTestMethod().get().getName();
        logger.info("About to execute repetition %d of %d for %s".formatted( //
            currentRepetition, totalRepetitions, methodName));
    }

    @RepeatedTest(10)
    void repeatedTest() {
        // ...
    }

    @RepeatedTest(5)
    void repeatedTestWithRepetitionInfo(RepetitionInfo repetitionInfo) {
        assertEquals(5, repetitionInfo.getTotalRepetitions());
    }

    @RepeatedTest(value = 8, failureThreshold = 2)
    void repeatedTestWithFailureThreshold(RepetitionInfo repetitionInfo) {
        // Simulate unexpected failure every second repetition
        if (repetitionInfo.getCurrentRepetition() % 2 == 0) {
            fail("Boom!");
        }
    }

    @RepeatedTest(value = 1, name = "{displayName}
{currentRepetition}/{totalRepetitions}")
    @DisplayName("Repeat!")
    void customDisplayName(TestInfo testInfo) {
        assertEquals("Repeat! 1/1", testInfo.getDisplayName());
    }
```

```
    @RepeatedTest(value = 1, name = RepeatedTest.LONG_DISPLAY_NAME)
    @DisplayName("Details...")
    void customDisplayNameWithLongPattern(TestInfo testInfo) {
        assertEquals("Details... :: repetition 1 of 1", testInfo.getDisplayName());
    }

    @RepeatedTest(value = 5, name = "Wiederholung {currentRepetition} von
{totalRepetitions}")
    void repeatedTestInGerman() {
        // ...
    }

}
```

When using the `ConsoleLauncher` with the unicode theme enabled, execution of `RepeatedTestsDemo` results in the following output to the console.

```
├── RepeatedTestsDemo ✔
│   ├── repeatedTest() ✔
│   │   ├── repetition 1 of 10 ✔
│   │   ├── repetition 2 of 10 ✔
│   │   ├── repetition 3 of 10 ✔
│   │   ├── repetition 4 of 10 ✔
│   │   ├── repetition 5 of 10 ✔
│   │   ├── repetition 6 of 10 ✔
│   │   ├── repetition 7 of 10 ✔
│   │   ├── repetition 8 of 10 ✔
│   │   ├── repetition 9 of 10 ✔
│   │   └── repetition 10 of 10 ✔
│   ├── repeatedTestWithRepetitionInfo(RepetitionInfo) ✔
│   │   ├── repetition 1 of 5 ✔
│   │   ├── repetition 2 of 5 ✔
│   │   ├── repetition 3 of 5 ✔
│   │   ├── repetition 4 of 5 ✔
│   │   └── repetition 5 of 5 ✔
│   ├── repeatedTestWithFailureThreshold(RepetitionInfo) ✔
│   │   ├── repetition 1 of 8 ✔
│   │   ├── repetition 2 of 8 ✘ Boom!
│   │   ├── repetition 3 of 8 ✔
│   │   ├── repetition 4 of 8 ✘ Boom!
│   │   ├── repetition 5 of 8 ↯ Failure threshold [2] exceeded
│   │   ├── repetition 6 of 8 ↯ Failure threshold [2] exceeded
│   │   ├── repetition 7 of 8 ↯ Failure threshold [2] exceeded
│   │   └── repetition 8 of 8 ↯ Failure threshold [2] exceeded
│   ├── Repeat! ✔
│   │   └── Repeat! 1/1 ✔
│   ├── Details... ✔
│   │   └── Details... :: repetition 1 of 1 ✔
│   └── repeatedTestInGerman() ✔
```

```
|      ├── Wiederholung 1 von 5 ✔
|      ├── Wiederholung 2 von 5 ✔
|      ├── Wiederholung 3 von 5 ✔
|      ├── Wiederholung 4 von 5 ✔
|      └── Wiederholung 5 von 5 ✔
```

Parameterized Classes and Tests

Parameterized tests make it possible to run a test method multiple times with different arguments. They are declared just like regular `@Test` methods but use the `@ParameterizedTest` annotation instead.

Parameterized classes make it possible to run *all* tests in a test class, including Nested Tests, multiple times with different arguments. They are declared just like regular test classes and may contain any supported test method type (including `@ParameterizedTest`) but annotated with the `@ParameterizedClass` annotation.



Parameterized classes are currently an *experimental* feature. You're invited to give it a try and provide feedback to the JUnit team so they can improve and eventually promote this feature.

Regardless of whether you are parameterizing a test method or a test class, you must declare at least one source that will provide the arguments for each invocation and then consume the arguments in the parameterized method or class, respectively.

The following example demonstrates a parameterized test that uses the `@ValueSource` annotation to specify a `String` array as the source of arguments.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

When executing the above parameterized test method, each invocation will be reported separately. For instance, the `ConsoleLauncher` will print output similar to the following.

```
palindromes(String) ✔
 ├── [1] candidate = "racecar" ✔
 ├── [2] candidate = "radar" ✔
 └── [3] candidate = "able was I ere I saw elba" ✔
```

The same `@ValueSource` annotation can be used to specify the source of arguments for a `@ParameterizedClass`.

```
@ParameterizedClass
```

```
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
class PalindromeTests {

    @Parameter
    String candidate;

    @Test
    void palindrome() {
        assertTrue(StringUtils.isPalindrome(candidate));
    }

    @Test
    void reversePalindrome() {
        String reverseCandidate = new StringBuilder(candidate).reverse().toString();
        assertTrue(StringUtils.isPalindrome(reverseCandidate));
    }
}
```

When executing the above parameterized test class, each invocation will be reported separately. For instance, the `ConsoleLauncher` will print output similar to the following.

```
PalindromeTests ✔
 ├── [1] candidate = "racecar" ✔
 │   ├── palindrome() ✔
 │   └── reversePalindrome() ✔
 ├── [2] candidate = "radar" ✔
 │   ├── palindrome() ✔
 │   └── reversePalindrome() ✔
 └── [3] candidate = "able was I ere I saw elba" ✔
    ├── palindrome() ✔
    └── reversePalindrome() ✔
```

Required Setup

In order to use parameterized classes or tests you need to add a dependency on the `junit-jupiter-params` artifact. Please refer to Dependency Metadata for details.

Consuming Arguments

Parameterized Tests

Parameterized test methods *consume* arguments directly from the configured source (see Sources of Arguments) following a one-to-one correlation between argument source index and method parameter index (see examples in @CsvSource). However, a parameterized test method may also choose to *aggregate* arguments from the source into a single object passed to the method (see Argument Aggregation). Additional arguments may also be provided by a `ParameterResolver` (e.g., to obtain an instance of `TestInfo`, `TestReporter`, etc.). Specifically, a parameterized test method must declare formal parameters according to the following rules.

- Zero or more *indexed parameters* must be declared first.
- Zero or more *aggregators* must be declared next.
- Zero or more arguments supplied by a `ParameterResolver` must be declared last.

In this context, an *indexed parameter* is an argument for a given index in the `Arguments` provided by an `ArgumentsProvider` that is passed as an argument to the parameterized method at the same index in the method's formal parameter list. An *aggregator* is any parameter of type `ArgumentsAccessor` or any parameter annotated with `@AggregateWith`.

Parameterized Classes

Parameterized classes *consume* arguments directly from the configured source (see Sources of Arguments); either via their unique constructor or via field injection. If a `@Parameter`-annotated field is declared in the parameterized class or one of its superclasses, field injection will be used. Otherwise, constructor injection will be used.

Constructor Injection



Constructor injection can only be used with the (default) `PER_METHOD` test instance lifecycle mode. Please use field injection with the `PER_CLASS` mode instead.

For constructor injection, the same rules apply as defined for parameterized tests above. In the following example, two arguments are injected into the constructor of the test class.

```
@ParameterizedClass
@CsvSource({ "apple, 23", "banana, 42" })
class FruitTests {

    final String fruit;
    final int quantity;

    FruitTests(String fruit, int quantity) {
        this.fruit = fruit;
        this.quantity = quantity;
    }

    @Test
    void test() {
        assertFruit(fruit);
        assertQuantity(quantity);
    }

    @Test
    void anotherTest() {
        // ...
    }
}
```

You may use *records* to implement parameterized classes that avoid the boilerplate code of declaring a test class constructor.

```
@ParameterizedClass
@CsvSource({ "apple, 23", "banana, 42" })
record FruitTests(String fruit, int quantity) {

    @Test
    void test() {
        assertFruit(fruit);
        assertQuantity(quantity);
    }

    @Test
    void anotherTest() {
        // ...
    }
}
```

Field Injection

For field injection, the following rules apply for fields annotated with `@Parameter`.

- Zero or more *indexed parameters* may be declared; each must have a unique index specified in its `@Parameter(index)` annotation. The index may be omitted if there is only one indexed parameter. If there are at least two indexed parameter declarations, there must be declarations for all indexes from 0 to the largest declared index.
- Zero or more *aggregators* may be declared; each without specifying an index in its `@Parameter` annotation.
- Zero or more other fields may be declared as usual as long as they're not annotated with `@Parameter`.

In this context, an *indexed parameter* is an argument for a given index in the `Arguments` provided by an `ArgumentsProvider` that is injected into a field annotated with `@Parameter(index)`. An *aggregator* is any `@Parameter`-annotated field of type ArgumentsAccessor or any field annotated with @AggregateWith.

The following example demonstrates how to use field injection to consume multiple arguments in a parameterized class.

```
@ParameterizedClass
@CsvSource({ "apple, 23", "banana, 42" })
class FruitTests {

    @Parameter(0)
    String fruit;

    @Parameter(1)
```

```
    int quantity;

    @Test
    void test() {
        assertFruit(fruit);
        assertQuantity(quantity);
    }

    @Test
    void anotherTest() {
        // ...
    }
}
```

If field injection is used, no constructor parameters will be resolved with arguments from the source. Other `ParameterResolver` extensions may resolve constructor parameters as usual, though.

Lifecycle Methods

`@BeforeParameterizedClassInvocation` and `@AfterParameterizedClassInvocation` can also be used to consume arguments if their `injectArguments` attribute is set to `true` (the default). If so, their method signatures must follow the same rules apply as defined for parameterized tests and additionally use the same parameter types as the *indexed parameters* of the parameterized test class. Please refer to the Javadoc of `@BeforeParameterizedClassInvocation` and `@AfterParameterizedClassInvocation` for details and to the Lifecycle section for an example.



AutoCloseable arguments

Arguments that implement `java.lang.AutoCloseable` (or `java.io.Closeable` which extends `java.lang.AutoCloseable`) will be automatically closed after the parameterized class or test invocation.

To prevent this from happening, set the `autoCloseArguments` attribute in `@ParameterizedTest` to `false`. Specifically, if an argument that implements `AutoCloseable` is reused for multiple invocations of the same parameterized class or test method, you must specify the `autoCloseArguments = false` on the `@ParameterizedClass` or `@ParameterizedTest` annotation to ensure that the argument is not closed between invocations.

Other Extensions

Other extensions can access the parameters and resolved arguments of a parameterized test or class by retrieving a `ParameterInfo` object from the `Store`. Please refer to the Javadoc of `ParameterInfo` for details.

Sources of Arguments

Out of the box, JUnit Jupiter provides quite a few *source* annotations. Each of the following subsections provides a brief overview and an example for each of them. Please refer to the Javadoc in the `org.junit.jupiter.params.provider` package for additional information.



All source annotations in this section are applicable to both `@ParameterizedClass` and `@ParameterizedTest`. For the sake of brevity, the examples in this section will only show how to use them with `@ParameterizedTest` methods.

@ValueSource

`@ValueSource` is one of the simplest possible sources. It lets you specify a single array of literal values and can only be used for providing a single argument per parameterized test invocation.

The following types of literal values are supported by `@ValueSource`.

- `short`
- `byte`
- `int`
- `long`
- `float`
- `double`
- `char`
- `boolean`
- `java.lang.String`
- `java.lang.Class`

For example, the following `@ParameterizedTest` method will be invoked three times, with the values `1`, `2`, and `3` respectively.

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertTrue(argument > 0 && argument < 4);
}
```

Null and Empty Sources

In order to check corner cases and verify proper behavior of our software when it is supplied *bad input*, it can be useful to have `null` and *empty* values supplied to our parameterized tests. The following annotations serve as sources of `null` and empty values for parameterized tests that accept a single argument.

- `@NullSource`: provides a single `null` argument to the annotated `@ParameterizedClass` or `@ParameterizedTest`.
  - `@NullSource` cannot be used for a parameter that has a primitive type.
- `@EmptySource`: provides a single *empty* argument to the annotated `@ParameterizedClass` or `@ParameterizedTest` for parameters of the following types: `java.lang.String`, `java.util.Collection` (and concrete subtypes with a `public` no-arg constructor), `java.util.List`,

`java.util.Set`, `java.util.SortedSet`, `java.util.NavigableSet`, `java.util.Map` (and concrete subtypes with a `public` no-arg constructor), `java.util.SortedMap`, `java.util.NavigableMap`, primitive arrays (e.g., `int[]`, `char[][]`, etc.), object arrays (e.g., `String[]`, `Integer[][]`, etc.).

- `@NullAndEmptySource`: a *composed annotation* that combines the functionality of `@NullSource` and `@EmptySource`.

If you need to supply multiple varying types of *blank* strings to a parameterized class or test, you can achieve that using `@ValueSource` — for example, `@ValueSource(strings = {" ", "   ", "\t", "\n"})`.

You can also combine `@NullSource`, `@EmptySource`, and `@ValueSource` to test a wider range of `null`, *empty*, and *blank* input. The following example demonstrates how to achieve this for strings.

```
@ParameterizedTest
@NullSource
@EmptySource
@ValueSource(strings = { " ", "   ", "\t", "\n" })
void nullEmptyAndBlankStrings(String text) {
    assertTrue(text == null || text.isBlank());
}
```

Making use of the composed `@NullAndEmptySource` annotation simplifies the above as follows.

```
@ParameterizedTest
@NullAndEmptySource
@ValueSource(strings = { " ", "   ", "\t", "\n" })
void nullEmptyAndBlankStrings(String text) {
    assertTrue(text == null || text.isBlank());
}
```



Both variants of the `nullEmptyAndBlankStrings(String)` parameterized test method result in six invocations: 1 for `null`, 1 for the empty string, and 4 for the explicit blank strings supplied via `@ValueSource`.

@EnumSource

`@EnumSource` provides a convenient way to use `Enum` constants.

```
@ParameterizedTest
@EnumSource(ChronoUnit.class)
void testWithEnumSource(TemporalUnit unit) {
    assertNotNull(unit);
}
```

The annotation's `value` attribute is optional. When omitted, the declared type of the first parameter is used. The test will fail if it does not reference an enum type. Thus, the `value` attribute is required

in the above example because the method parameter is declared as `TemporalUnit`, i.e. the interface implemented by `ChronoUnit`, which isn't an enum type. Changing the method parameter type to `ChronoUnit` allows you to omit the explicit enum type from the annotation as follows.

```
@ParameterizedTest
@EnumSource
void testWithEnumSourceWithAutoDetection(ChronoUnit unit) {
    assertNotNull(unit);
}
```

The annotation provides an optional `names` attribute that lets you specify which constants shall be used, like in the following example.

```
@ParameterizedTest
@EnumSource(names = { "DAYS", "HOURS" })
void testWithEnumSourceInclude(ChronoUnit unit) {
    assertTrue(EnumSet.of(ChronoUnit.DAYS, ChronoUnit.HOURS).contains(unit));
}
```

In addition to `names`, you can use the `from` and `to` attributes to specify a range of constants. The range starts from the constant specified in the `from` attribute and includes all subsequent constants up to and including the one specified in the `to` attribute, based on the natural order of the enum constants.

If `from` and `to` attributes are omitted, they default to the first and last constants in the enum type, respectively. If all `names`, `from`, and `to` attributes are omitted, all constants will be used. The following example demonstrates how to specify a range of constants.

```
@ParameterizedTest
@EnumSource(from = "HOURS", to = "DAYS")
void testWithEnumSourceRange(ChronoUnit unit) {
    assertTrue(EnumSet.of(ChronoUnit.HOURS, ChronoUnit.HALF_DAYS, ChronoUnit.DAYS
).contains(unit));
}
```

The `@EnumSource` annotation also provides an optional `mode` attribute that enables fine-grained control over which constants are passed to the test method. For example, you can exclude names from the enum constant pool or specify regular expressions as in the following examples.

```
@ParameterizedTest
@EnumSource(mode = EXCLUDE, names = { "ERAS", "FOREVER" })
void testWithEnumSourceExclude(ChronoUnit unit) {
    assertFalse(EnumSet.of(ChronoUnit.ERAS, ChronoUnit.FOREVER).contains(unit));
}
```

```
@ParameterizedTest
@EnumSource(mode = MATCH_ALL, names = "^.*DAYS$")
void testWithEnumSourceRegex(ChronoUnit unit) {
    assertTrue(unit.name().endsWith("DAYS"));
}
```

You can also combine `mode` with the `from`, `to` and `names` attributes to define a range of constants while excluding specific values from that range as shown below.

```
@ParameterizedTest
@EnumSource(from = "HOURS", to = "DAYS", mode = EXCLUDE, names = { "HALF_DAYS" })
void testWithEnumSourceRangeExclude(ChronoUnit unit) {
    assertTrue(EnumSet.of(ChronoUnit.HOURS, ChronoUnit.DAYS).contains(unit));
    assertFalse(EnumSet.of(ChronoUnit.HALF_DAYS).contains(unit));
}
```

@MethodSource

`@MethodSource` allows you to refer to one or more *factory* methods of the test class or external classes.

Factory methods within the test class must be `static` unless the test class is annotated with `@TestInstance(Lifecycle.PER_CLASS)`; whereas, factory methods in external classes must always be `static`.

Each factory method must generate a *stream* of *arguments*, and each set of arguments within the stream will be provided as the physical arguments for individual invocations of the annotated `@ParameterizedClass` or `@ParameterizedTest`. Generally speaking this translates to a `Stream` of `Arguments` (i.e., `Stream<Arguments>`); however, the actual concrete return type can take on many forms. In this context, a "stream" is anything that JUnit can reliably convert into a `Stream`, such as `Stream`, `DoubleStream`, `LongStream`, `IntStream`, `Collection`, `Iterator`, `Iterable`, an array of objects or primitives, or any type that provides an `iterator(): Iterator` method (such as, for example, a `kotlin.sequences.Sequence`). The "arguments" within the stream can be supplied as an instance of `Arguments`, an array of objects (e.g., `Object[]`), or a single value if the parameterized class or test method accepts a single argument.

If the return type is `Stream` or one of the primitive streams, JUnit will properly close it by calling `BaseStream.close()`, making it safe to use a resource such as `Files.lines()`.

If you only need a single parameter, you can return a `Stream` of instances of the parameter type as demonstrated in the following example.

```
@ParameterizedTest
@MethodSource("stringProvider")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull(argument);
}
```

```
static Stream<String> stringProvider() {
    return Stream.of("apple", "banana");
}
```

For a `@ParameterizedClass`, providing a factory method name via `@MethodSource` is mandatory. For a `@ParameterizedTest`, if you do not explicitly provide a factory method name, JUnit Jupiter will search for a *factory* method with the same name as the current `@ParameterizedTest` method by convention. This is demonstrated in the following example.

```
@ParameterizedTest
@MethodSource
void testWithDefaultLocalMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> testWithDefaultLocalMethodSource() {
    return Stream.of("apple", "banana");
}
```

Streams for primitive types (`DoubleStream`, `IntStream`, and `LongStream`) are also supported as demonstrated by the following example.

```
@ParameterizedTest
@MethodSource("range")
void testWithRangeMethodSource(int argument) {
    assertNotEquals(9, argument);
}

static IntStream range() {
    return IntStream.range(0, 20).skip(10);
}
```

If a parameterized class or test method declares multiple parameters, you need to return a collection, stream, or array of `Arguments` instances or object arrays as shown below (see the Javadoc for `@MethodSource` for further details on supported return types). Note that `arguments(Object…)` is a static factory method defined in the `Arguments` interface. In addition, `Arguments.of(Object…)` may be used as an alternative to `arguments(Object…)`.

```
@ParameterizedTest
@MethodSource("stringIntAndListProvider")
void testWithMultiArgMethodSource(String str, int num, List<String> list) {
    assertEquals(5, str.length());
    assertTrue(num >=1 && num <=2);
    assertEquals(2, list.size());
}
```

```
static Stream<Arguments> stringIntAndListProvider() {
    return Stream.of(
        arguments("apple", 1, Arrays.asList("a", "b")),
        arguments("lemon", 2, Arrays.asList("x", "y"))
    );
}
```

An external, `static` *factory* method can be referenced by providing its *fully qualified method name* as demonstrated in the following example.

```
package example;

import java.util.stream.Stream;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;

class ExternalMethodSourceDemo {

    @ParameterizedTest
    @MethodSource("example.StringsProviders#tinyStrings")
    void testWithExternalMethodSource(String tinyString) {
        // test with tiny string
    }
}

class StringsProviders {

    static Stream<String> tinyStrings() {
        return Stream.of(".", "oo", "OOO");
    }
}
```

Factory methods can declare parameters, which will be provided by registered implementations of the `ParameterResolver` extension API. In the following example, the factory method is referenced by its name since there is only one such method in the test class. If there are several local methods with the same name, parameters can also be provided to differentiate them – for example, `@MethodSource("factoryMethod()")` or `@MethodSource("factoryMethod(java.lang.String)")`. Alternatively, the factory method can be referenced by its fully qualified method name, e.g. `@MethodSource("example.MyTests#factoryMethod(java.lang.String)")`.

```
@RegisterExtension
static final IntegerResolver integerResolver = new IntegerResolver();

@ParameterizedTest
@MethodSource("factoryMethodWithArguments")
void testWithFactoryMethodWithArguments(String argument) {
    assertTrue(argument.startsWith("2"));
```

```
}

static Stream<Arguments> factoryMethodWithArguments(int quantity) {
    return Stream.of(
            arguments(quantity + " apples"),
            arguments(quantity + " lemons")
    );
}

static class IntegerResolver implements ParameterResolver {

    @Override
    public boolean supportsParameter(ParameterContext parameterContext,
            ExtensionContext extensionContext) {

        return parameterContext.getParameter().getType() == int.class;
    }

    @Override
    public Object resolveParameter(ParameterContext parameterContext,
            ExtensionContext extensionContext) {

        return 2;
    }

}
```

@FieldSource

`@FieldSource` allows you to refer to one or more fields of the test class or external classes.

Fields within the test class must be `static` unless the test class is annotated with `@TestInstance(Lifecycle.PER_CLASS)`; whereas, fields in external classes must always be `static`.

Each field must be able to supply a *stream* of arguments, and each set of "arguments" within the "stream" will be provided as the physical arguments for individual invocations of the annotated `@ParameterizedClass` or `@ParameterizedTest`.

In this context, a "stream" is anything that JUnit can reliably convert to a `Stream`; however, the actual concrete field type can take on many forms. Generally speaking this translates to a `Collection`, an `Iterable`, a `Supplier` of a stream (`Stream`, `DoubleStream`, `LongStream`, or `IntStream`), a `Supplier` of an `Iterator`, an array of objects or primitives, or any type that provides an `iterator(): Iterator` method (such as, for example, a `kotlin.sequences.Sequence`). Each set of "arguments" within the "stream" can be supplied as an instance of `Arguments`, an array of objects (for example, `Object[]`, `String[]`, etc.), or a single value if the parameterized class or test method accepts a single argument.



In contrast to the supported return types for `@MethodSource` factory methods, the value of a `@FieldSource` field cannot be an instance of `Stream`, `DoubleStream`, `LongStream`, `IntStream`, or `Iterator`, since the values of such types are *consumed* the

> first time they are processed. However, if you wish to use one of these types, you can wrap it in a `Supplier` — for example, `Supplier<IntStream>`.

If the `Supplier` return type is `Stream` or one of the primitive streams, JUnit will properly close it by calling `BaseStream.close()`, making it safe to use a resource such as `Files.lines()`.

Please note that a one-dimensional array of objects supplied as a set of "arguments" will be handled differently than other types of arguments. Specifically, all the elements of a one-dimensional array of objects will be passed as individual physical arguments to the `@ParameterizedClass` or `@ParameterizedTest`. See the Javadoc for `@FieldSource` for further details.

For a `@ParameterizedClass`, providing a field name via `@FieldSource` is mandatory. For a `@ParameterizedTest`, if you do not explicitly provide a field name, JUnit Jupiter will search in the test class for a field that has the same name as the current `@ParameterizedTest` method by convention. This is demonstrated in the following example. This parameterized test method will be invoked twice: with the values `"apple"` and `"banana"`.

```
@ParameterizedTest
@FieldSource
void arrayOfFruits(String fruit) {
    assertFruit(fruit);
}

static final String[] arrayOfFruits = { "apple", "banana" };
```

The following example demonstrates how to provide a single explicit field name via `@FieldSource`. This parameterized test method will be invoked twice: with the values `"apple"` and `"banana"`.

```
@ParameterizedTest
@FieldSource("listOfFruits")
void singleFieldSource(String fruit) {
    assertFruit(fruit);
}

static final List<String> listOfFruits = Arrays.asList("apple", "banana");
```

The following example demonstrates how to provide multiple explicit field names via `@FieldSource`. This example uses the `listOfFruits` field from the previous example as well as the `additionalFruits` field. Consequently, this parameterized test method will be invoked four times: with the values `"apple"`, `"banana"`, `"cherry"`, and `"dewberry"`.

```
@ParameterizedTest
@FieldSource({ "listOfFruits", "additionalFruits" })
void multipleFieldSources(String fruit) {
    assertFruit(fruit);
}
```

```
static final Collection<String> additionalFruits = Arrays.asList("cherry",
"dewberry");
```

It is also possible to provide a `Stream`, `DoubleStream`, `IntStream`, `LongStream`, or `Iterator` as the source of arguments via a `@FieldSource` field as long as the stream or iterator is wrapped in a `java.util.function.Supplier`. The following example demonstrates how to provide a `Supplier` of a `Stream` of named arguments. This parameterized test method will be invoked twice: with the values `"apple"` and `"banana"` and with display names `"Apple"` and `"Banana"`, respectively.

```
@ParameterizedTest
@FieldSource
void namedArgumentsSupplier(String fruit) {
    assertFruit(fruit);
}

static final Supplier<Stream<Arguments>> namedArgumentsSupplier = () -> Stream.of(
    arguments(named("Apple", "apple")),
    arguments(named("Banana", "banana"))
);
```



Note that `arguments(Object…)` is a static factory method defined in the `org.junit.jupiter.params.provider.Arguments` interface.

Similarly, `named(String, Object)` is a static factory method defined in the `org.junit.jupiter.api.Named` interface.

If a parameterized class or test method declares multiple parameters, the corresponding `@FieldSource` field must be able to provide a collection, stream supplier, or array of `Arguments` instances or object arrays as shown below (see the Javadoc for `@FieldSource` for further details on supported types).

```
@ParameterizedTest
@FieldSource("stringIntAndListArguments")
void testWithMultiArgFieldSource(String str, int num, List<String> list) {
    assertEquals(5, str.length());
    assertTrue(num >=1 && num <=2);
    assertEquals(2, list.size());
}

static List<Arguments> stringIntAndListArguments = Arrays.asList(
    arguments("apple", 1, Arrays.asList("a", "b")),
    arguments("lemon", 2, Arrays.asList("x", "y"))
);
```



Note that `arguments(Object…)` is a static factory method defined in the `org.junit.jupiter.params.provider.Arguments` interface.

An external, `static` `@FieldSource` field can be referenced by providing its *fully qualified field name* as demonstrated in the following example.

```
@ParameterizedTest
@FieldSource("example.FruitUtils#tropicalFruits")
void testWithExternalFieldSource(String tropicalFruit) {
    // test with tropicalFruit
}
```

@CsvSource

`@CsvSource` allows you to express argument lists as comma-separated values (i.e., CSV `String` literals). Each string provided via the `value` attribute in `@CsvSource` represents a CSV record and results in one invocation of the parameterized class or test. The first record may optionally be used to supply CSV headers (see the Javadoc for the `useHeadersInDisplayName` attribute for details and an example).

```
@ParameterizedTest
@CsvSource({
    "apple,         1",
    "banana,        2",
    "'lemon, lime', 0xF1",
    "strawberry,    700_000"
})
void testWithCsvSource(String fruit, int rank) {
    assertNotNull(fruit);
    assertNotEquals(0, rank);
}
```

The default delimiter is a comma (`,`), but you can use another character by setting the `delimiter` attribute. Alternatively, the `delimiterString` attribute allows you to use a `String` delimiter instead of a single character. However, both delimiter attributes cannot be set simultaneously.

By default, `@CsvSource` uses a single quote (`'`) as its quote character, but this can be changed via the `quoteCharacter` attribute. See the `'lemon, lime'` value in the example above and in the table below. An empty, quoted value (`''`) results in an empty `String` unless the `emptyValue` attribute is set; whereas, an entirely *empty* value is interpreted as a `null` reference. By specifying one or more `nullValues`, a custom value can be interpreted as a `null` reference (see the `NIL` example in the table below). An `ArgumentConversionException` is thrown if the target type of a `null` reference is a primitive type.



An *unquoted* empty value will always be converted to a `null` reference regardless of any custom values configured via the `nullValues` attribute.

Except within a quoted string, leading and trailing whitespace in a CSV column is trimmed by default. This behavior can be changed by setting the `ignoreLeadingAndTrailingWhitespace` attribute to `true`.

Example Input	Resulting Argument List
`@CsvSource({ "apple, banana" })`	`"apple", "banana"`
`@CsvSource({ "apple, 'lemon, lime'" })`	`"apple", "lemon, lime"`
`@CsvSource({ "apple, ''" })`	`"apple", ""`
`@CsvSource({ "apple, " })`	`"apple", null`
`@CsvSource(value = { "apple, banana, NIL" }, nullValues = "NIL")`	`"apple", "banana", null`
`@CsvSource(value = { " apple , banana" }, ignoreLeadingAndTrailingWhitespace = false)`	`" apple ", " banana"`

If the programming language you are using supports Java *text blocks* or equivalent multi-line string literals, you can alternatively use the `textBlock` attribute of `@CsvSource`. Each record within a text block represents a CSV record and results in one invocation of the parameterized class or test. The first record may optionally be used to supply CSV headers by setting the `useHeadersInDisplayName` attribute to `true` as in the example below.

Using a text block, the previous example can be implemented as follows.

```
@ParameterizedTest
@CsvSource(useHeadersInDisplayName = true, textBlock = """
    FRUIT,         RANK
    apple,         1
    banana,        2
    'lemon, lime', 0xF1
    strawberry,    700_000
    """)
void testWithCsvSource(String fruit, int rank) {
    // ...
}
```

The generated display names for the previous example include the CSV header names.

```
[1] FRUIT = "apple", RANK = "1"
[2] FRUIT = "banana", RANK = "2"
[3] FRUIT = "lemon, lime", RANK = "0xF1"
[4] FRUIT = "strawberry", RANK = "700_000"
```

In contrast to CSV records supplied via the `value` attribute, a text block can contain comments. Any line beginning with the value of the `commentCharacter` attribute (`#` by default) will be treated as a comment and ignored. Note that there is one exception to this rule: if the comment character appears within a quoted field, it loses its special meaning.

The comment character must be the first character on the line without any leading whitespace. It is therefore recommended that the closing text block delimiter (`"""`) be placed either at the end of the last line of input or on the following line, left aligned with the rest of the input (as can be seen in

the example below which demonstrates formatting similar to a table).

```
@ParameterizedTest
@CsvSource(delimiter = '|', quoteCharacter = '"', textBlock = """
    #-------------------------
    #    FRUIT     |     RANK
    #-------------------------
         apple     |      1
    #-------------------------
         banana    |      2
    #-------------------------
      "lemon lime" |     0xF1
    #-------------------------
       strawberry  |    700_000
    #-------------------------
    """)
void testWithCsvSource(String fruit, int rank) {
    // ...
}
```



Java's text block feature automatically removes *incidental whitespace* when the code is compiled. However other JVM languages such as Groovy and Kotlin do not. Thus, if you are using a programming language other than Java and your text block contains comments or new lines within quoted strings, you will need to ensure that there is no leading whitespace within your text block.

@CsvFileSource

`@CsvFileSource` lets you use comma-separated value (CSV) files from the classpath or the local file system. Each record from a CSV file results in one invocation of the parameterized class or test. The first record may optionally be used to supply CSV headers. You can instruct JUnit to ignore the headers via the `numLinesToSkip` attribute. If you would like for the headers to be used in the display names, you can set the `useHeadersInDisplayName` attribute to `true`. The examples below demonstrate the use of `numLinesToSkip` and `useHeadersInDisplayName`.

The default delimiter is a comma (`,`), but you can use another character by setting the `delimiter` attribute. Alternatively, the `delimiterString` attribute allows you to use a `String` delimiter instead of a single character. However, both delimiter attributes cannot be set simultaneously.



Comments in CSV files

Any line beginning with the value of the `commentCharacter` attribute (`#` by default) will be interpreted as a comment and will be ignored.

```
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSourceFromClasspath(String country, int reference) {
    assertNotNull(country);
```

```
    assertNotEquals(0, reference);
}

@ParameterizedTest
@CsvFileSource(files = "src/test/resources/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSourceFromFile(String country, int reference) {
    assertNotNull(country);
    assertNotEquals(0, reference);
}

@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", useHeadersInDisplayName = true)
void testWithCsvFileSourceAndHeaders(String country, int reference) {
    assertNotNull(country);
    assertNotEquals(0, reference);
}
```

two-column.csv

```
COUNTRY, REFERENCE
Sweden, 1
Poland, 2
"United States of America", 3
France, 700_000
```

The following listing shows the generated display names for the first two parameterized test methods above.

```
[1] country = "Sweden", reference = "1"
[2] country = "Poland", reference = "2"
[3] country = "United States of America", reference = "3"
[4] country = "France", reference = "700_000"
```

The following listing shows the generated display names for the last parameterized test method above that uses CSV header names.

```
[1] COUNTRY = "Sweden", REFERENCE = "1"
[2] COUNTRY = "Poland", REFERENCE = "2"
[3] COUNTRY = "United States of America", REFERENCE = "3"
[4] COUNTRY = "France", REFERENCE = "700_000"
```

In contrast to the default syntax used in `@CsvSource`, `@CsvFileSource` uses a double quote (") as the quote character by default, but this can be changed via the `quoteCharacter` attribute. See the `"United States of America"` value in the example above. An empty, quoted value ("") results in an empty `String` unless the `emptyValue` attribute is set; whereas, an entirely *empty* value is interpreted as a `null` reference. By specifying one or more `nullValues`, a custom value can be interpreted as a `null` reference. An `ArgumentConversionException` is thrown if the target type of a `null` reference is a

primitive type.



An *unquoted* empty value will always be converted to a `null` reference regardless of any custom values configured via the `nullValues` attribute.

Except within a quoted string, leading and trailing whitespace in a CSV column is trimmed by default. This behavior can be changed by setting the `ignoreLeadingAndTrailingWhitespace` attribute to `true`.

@ArgumentsSource

`@ArgumentsSource` can be used to specify a custom, reusable `ArgumentsProvider`. Note that an implementation of `ArgumentsProvider` must be declared as either a top-level class or as a `static` nested class.

```
@ParameterizedTest
@ArgumentsSource(MyArgumentsProvider.class)
void testWithArgumentsSource(String argument) {
    assertNotNull(argument);
}
```

```
public class MyArgumentsProvider implements ArgumentsProvider {

    @Override
    public Stream<? extends Arguments> provideArguments(ParameterDeclarations
parameters,
            ExtensionContext context) {
        return Stream.of("apple", "banana").map(Arguments::of);
    }
}
```

If you wish to implement a custom `ArgumentsProvider` that also consumes an annotation (like built-in providers such as `ValueArgumentsProvider` or `CsvArgumentsProvider`), you have the possibility to extend the `AnnotationBasedArgumentsProvider` class.

Moreover, `ArgumentsProvider` implementations may declare constructor parameters in case they need to be resolved by a registered `ParameterResolver` as demonstrated in the following example.

```
public class MyArgumentsProviderWithConstructorInjection implements ArgumentsProvider
{

    private final TestInfo testInfo;

    public MyArgumentsProviderWithConstructorInjection(TestInfo testInfo) {
        this.testInfo = testInfo;
    }
```

```
    @Override
    public Stream<? extends Arguments> provideArguments(ParameterDeclarations
parameters,
            ExtensionContext context) {
        return Stream.of(Arguments.of(testInfo.getDisplayName()));
    }
}
```

Multiple sources using repeatable annotations

Repeatable annotations provide a convenient way to specify multiple sources from different providers.

```
@DisplayName("A parameterized test that makes use of repeatable annotations")
@ParameterizedTest
@MethodSource("someProvider")
@MethodSource("otherProvider")
void testWithRepeatedAnnotation(String argument) {
    assertNotNull(argument);
}

static Stream<String> someProvider() {
    return Stream.of("foo");
}

static Stream<String> otherProvider() {
    return Stream.of("bar");
}
```

Following the above parameterized test, a test case will run for each argument:

```
[1] foo
[2] bar
```

The following annotations are repeatable:

- `@ValueSource`
- `@EnumSource`
- `@MethodSource`
- `@FieldSource`
- `@CsvSource`
- `@CsvFileSource`
- `@ArgumentsSource`

Argument Count Validation

By default, when an arguments source provides more arguments than the test method needs, those additional arguments are ignored and the test executes as usual. This can lead to bugs where arguments are never passed to the parameterized class or method.

To prevent this, you can set argument count validation to 'strict'. Then, any additional arguments will cause an error instead.

To change this behavior for all tests, set the `junit.jupiter.params.argumentCountValidation` configuration parameter to `strict`. To change this behavior for a single parameterized class or test method, use the `argumentCountValidation` attribute of the `@ParameterizedClass` or `@ParameterizedTest` annotation:

```
@ParameterizedTest(argumentCountValidation = ArgumentCountValidationMode.STRICT)
@CsvSource({ "42, -666" })
void testWithArgumentCountValidation(int number) {
    assertTrue(number > 0);
}
```

Argument Conversion

Widening Conversion

JUnit Jupiter supports Widening Primitive Conversion for arguments supplied to a `@ParameterizedClass` or `@ParameterizedTest`. For example, a parameterized class or test method annotated with `@ValueSource(ints = { 1, 2, 3 })` can be declared to accept not only an argument of type `int` but also an argument of type `long`, `float`, or `double`.

Implicit Conversion

To support use cases like `@CsvSource`, JUnit Jupiter provides a number of built-in implicit type converters. The conversion process depends on the declared type of each method parameter.

For example, if a `@ParameterizedClass` or `@ParameterizedTest` declares a parameter of type `TimeUnit` and the actual type supplied by the declared source is a `String`, the string will be automatically converted into the corresponding `TimeUnit` enum constant.

```
@ParameterizedTest
@ValueSource(strings = "SECONDS")
void testWithImplicitArgumentConversion(ChronoUnit argument) {
    assertNotNull(argument.name());
}
```

`String` instances are implicitly converted to the following target types.



Decimal, hexadecimal, and octal `String` literals will be converted to their integral

> types: `byte`, `short`, `int`, `long`, and their boxed counterparts.

Target Type	Example
`boolean`/`Boolean`	`"true"` → `true` *(only accepts values 'true' or 'false', case-insensitive)*
`byte`/`Byte`	`"15"`, `"0xF"`, or `"017"` → `(byte) 15`
`char`/`Character`	`"o"` → `'o'`
`short`/`Short`	`"15"`, `"0xF"`, or `"017"` → `(short) 15`
`int`/`Integer`	`"15"`, `"0xF"`, or `"017"` → `15`
`long`/`Long`	`"15"`, `"0xF"`, or `"017"` → `15L`
`float`/`Float`	`"1.0"` → `1.0f`
`double`/`Double`	`"1.0"` → `1.0d`
`Enum` subclass	`"SECONDS"` → `TimeUnit.SECONDS`
`java.io.File`	`"/path/to/file"` → `new File("/path/to/file")`
`java.lang.Class`	`"java.lang.Integer"` → `java.lang.Integer.class` *(use `$` for nested classes, e.g. `"java.lang.Thread$State"`)*
`java.lang.Class`	`"byte"` → `byte.class` *(primitive types are supported)*
`java.lang.Class`	`"char[]"` → `char[].class` *(array types are supported)*
`java.math.BigDecimal`	`"123.456e789"` → `new BigDecimal("123.456e789")`
`java.math.BigInteger`	`"1234567890123456789"` → `new BigInteger("1234567890123456789")`
`java.net.URI`	`"https://junit.org/"` → `URI.create("https://junit.org/")`
`java.net.URL`	`"https://junit.org/"` → `URI.create("https://junit.org/").toURL()`
`java.nio.charset.Charset`	`"UTF-8"` → `Charset.forName("UTF-8")`
`java.nio.file.Path`	`"/path/to/file"` → `Paths.get("/path/to/file")`

Target Type	Example
java.time.Duration	"PT3S" → Duration.ofSeconds(3)
java.time.Instant	"1970-01-01T00:00:00Z" → Instant.ofEpochMilli(0)
java.time.LocalDateTime	"2017-03-14T12:34:56.789" → LocalDateTime.of(2017, 3, 14, 12, 34, 56, 789_000_000)
java.time.LocalDate	"2017-03-14" → LocalDate.of(2017, 3, 14)
java.time.LocalTime	"12:34:56.789" → LocalTime.of(12, 34, 56, 789_000_000)
java.time.MonthDay	"--03-14" → MonthDay.of(3, 14)
java.time.OffsetDateTime	"2017-03-14T12:34:56.789Z" → OffsetDateTime.of(2017, 3, 14, 12, 34, 56, 789_000_000, ZoneOffset.UTC)
java.time.OffsetTime	"12:34:56.789Z" → OffsetTime.of(12, 34, 56, 789_000_000, ZoneOffset.UTC)
java.time.Period	"P2M6D" → Period.of(0, 2, 6)
java.time.YearMonth	"2017-03" → YearMonth.of(2017, 3)
java.time.Year	"2017" → Year.of(2017)
java.time.ZonedDateTime	"2017-03-14T12:34:56.789Z" → ZonedDateTime.of(2017, 3, 14, 12, 34, 56, 789_000_000, ZoneOffset.UTC)
java.time.ZoneId	"Europe/Berlin" → ZoneId.of("Europe/Berlin")
java.time.ZoneOffset	"+02:30" → ZoneOffset.ofHoursMinutes(2, 30)
java.util.Currency	"JPY" → Currency.getInstance("JPY")
java.util.Locale	"en-US" → Locale.forLanguageTag("en-US")
java.util.UUID	"d043e930-7b3b-48e3-bdbe-5a3ccfb833db" → UUID.fromString("d043e930-7b3b-48e3-bdbe-5a3ccfb833db")

Fallback String-to-Object Conversion

In addition to implicit conversion from strings to the target types listed in the above table, JUnit Jupiter also provides a fallback mechanism for automatic conversion from a `String` to a given target type if the target type declares exactly one suitable *factory method* or a *factory constructor* as defined below.

- *factory method*: a non-private, `static` method declared in the target type that accepts either a single `String` argument or a single `CharSequence` argument and returns an instance of the target type. The name of the method can be arbitrary and need not follow any particular convention.
- *factory constructor*: a non-private constructor in the target type that accepts a either a single `String` argument or a single `CharSequence` argument. Note that the target type must be declared as either a top-level class or as a `static` nested class.



If multiple *factory methods* are discovered, they will be ignored. If a *factory method* and a *factory constructor* are discovered, the factory method will be used instead of the constructor.

For example, in the following `@ParameterizedTest` method, the `Book` argument will be created by invoking the `Book.fromTitle(String)` factory method and passing `"42 Cats"` as the title of the book.

```
@ParameterizedTest
@ValueSource(strings = "42 Cats")
void testWithImplicitFallbackArgumentConversion(Book book) {
    assertEquals("42 Cats", book.getTitle());
}
```

```
public class Book {

    private final String title;

    private Book(String title) {
        this.title = title;
    }

    public static Book fromTitle(String title) {
        return new Book(title);
    }

    public String getTitle() {
        return this.title;
    }
}
```

Explicit Conversion

Instead of relying on implicit argument conversion, you may explicitly specify an `ArgumentConverter`

to use for a certain parameter using the `@ConvertWith` annotation like in the following example. Note that an implementation of `ArgumentConverter` must be declared as either a top-level class or as a `static` nested class.

```
@ParameterizedTest
@EnumSource(ChronoUnit.class)
void testWithExplicitArgumentConversion(
        @ConvertWith(ToStringArgumentConverter.class) String argument) {

    assertNotNull(ChronoUnit.valueOf(argument));
}
```

```
public class ToStringArgumentConverter extends SimpleArgumentConverter {

    @Override
    protected Object convert(Object source, Class<?> targetType) {
        assertEquals(String.class, targetType, "Can only convert to String");
        if (source instanceof Enum<?> constant) {
            return constant.name();
        }
        return String.valueOf(source);
    }
}
```

If the converter is only meant to convert one type to another, you can extend `TypedArgumentConverter` to avoid boilerplate type checks.

```
public class ToLengthArgumentConverter extends TypedArgumentConverter<String, Integer>
{

    protected ToLengthArgumentConverter() {
        super(String.class, Integer.class);
    }

    @Override
    protected Integer convert(String source) {
        return (source != null ? source.length() : 0);
    }

}
```

Explicit argument converters are meant to be implemented by test and extension authors. Thus, `junit-jupiter-params` only provides a single explicit argument converter that may also serve as a reference implementation: `JavaTimeArgumentConverter`. It is used via the composed annotation `JavaTimeConversionPattern`.

```
@ParameterizedTest
@ValueSource(strings = { "01.01.2017", "31.12.2017" })
void testWithExplicitJavaTimeConverter(
        @JavaTimeConversionPattern("dd.MM.yyyy") LocalDate argument) {

    assertEquals(2017, argument.getYear());
}
```

If you wish to implement a custom `ArgumentConverter` that also consumes an annotation (like `JavaTimeArgumentConverter`), you have the possibility to extend the `AnnotationBasedArgumentConverter` class.

Argument Aggregation

By default, each *argument* provided to a `@ParameterizedClass` or `@ParameterizedTest` corresponds to a single method parameter. Consequently, argument sources which are expected to supply a large number of arguments can lead to large constructor or method signatures, respectively.

In such cases, an `ArgumentsAccessor` can be used instead of multiple parameters. Using this API, you can access the provided arguments through a single argument passed to your test method. In addition, type conversion is supported as discussed in Implicit Conversion.

Besides, you can retrieve the current test invocation index with `ArgumentsAccessor.getInvocationIndex()`.

```
@ParameterizedTest
@CsvSource({
    "Jane, Doe, F, 1990-05-20",
    "John, Doe, M, 1990-10-22"
})
void testWithArgumentsAccessor(ArgumentsAccessor arguments) {
    Person person = new Person(
                                arguments.getString(0),
                                arguments.getString(1),
                                arguments.get(2, Gender.class),
                                arguments.get(3, LocalDate.class));

    if (person.getFirstName().equals("Jane")) {
        assertEquals(Gender.F, person.getGender());
    }
    else {
        assertEquals(Gender.M, person.getGender());
    }
    assertEquals("Doe", person.getLastName());
    assertEquals(1990, person.getDateOfBirth().getYear());
}
```

An instance of `ArgumentsAccessor` *is automatically injected into any parameter of type*

`ArgumentsAccessor`.

Custom Aggregators

Apart from direct access to the arguments of a `@ParameterizedClass` or `@ParameterizedTest` using an `ArgumentsAccessor`, JUnit Jupiter also supports the usage of custom, reusable *aggregators*.

To use a custom aggregator, implement the `ArgumentsAggregator` interface and register it via the `@AggregateWith` annotation on a compatible parameter of the `@ParameterizedClass` or `@ParameterizedTest`. The result of the aggregation will then be provided as an argument for the corresponding parameter when the parameterized test is invoked. Note that an implementation of `ArgumentsAggregator` must be declared as either a top-level class or as a `static` nested class.

```
@ParameterizedTest
@CsvSource({
    "Jane, Doe, F, 1990-05-20",
    "John, Doe, M, 1990-10-22"
})
void testWithArgumentsAggregator(@AggregateWith(PersonAggregator.class) Person person)
{
    // perform assertions against person
}
```

```
public class PersonAggregator extends SimpleArgumentsAggregator {
    @Override
    protected Person aggregateArguments(ArgumentsAccessor arguments, Class<?>
targetType,
            AnnotatedElementContext context, int parameterIndex) {
        return new Person(
                            arguments.getString(0),
                            arguments.getString(1),
                            arguments.get(2, Gender.class),
                            arguments.get(3, LocalDate.class));
    }
}
```

If you find yourself repeatedly declaring `@AggregateWith(MyTypeAggregator.class)` for multiple parameterized classes or methods across your codebase, you may wish to create a custom *composed annotation* such as `@CsvToMyType` that is meta-annotated with `@AggregateWith(MyTypeAggregator.class)`. The following example demonstrates this in action with a custom `@CsvToPerson` annotation.

```
@ParameterizedTest
@CsvSource({
    "Jane, Doe, F, 1990-05-20",
    "John, Doe, M, 1990-10-22"
})
```

```
void testWithCustomAggregatorAnnotation(@CsvToPerson Person person) {
    // perform assertions against person
}
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
@AggregateWith(PersonAggregator.class)
public @interface CsvToPerson {
}
```

Customizing Display Names

By default, the display name of a parameterized class or test invocation contains the invocation index and a comma-separated list of the `String` representations of all arguments for that specific invocation. If parameter names are present in the bytecode, each argument will be preceded by its parameter name and an equals sign (unless the argument is only available via an `ArgumentsAccessor` or `ArgumentAggregator`) – for example, `firstName = "Jane"`.



To ensure that parameter names are present in the bytecode, test code must be compiled with the `-parameters` compiler flag for Java or with the `-java-parameters` compiler flag for Kotlin.

However, you can customize invocation display names via the `name` attribute of the `@ParameterizedClass` or `@ParameterizedTest` annotation as in the following example.

```
@DisplayName("Display name of container")
@ParameterizedTest(name = "{index} ==> the rank of {0} is {1}")
@CsvSource({ "apple, 1", "banana, 2", "'lemon, lime', 3" })
void testWithCustomDisplayNames(String fruit, int rank) {
}
```

When executing the above method using the `ConsoleLauncher` you will see output similar to the following.

```
Display name of container ✔
├── 1 ==> the rank of "apple" is "1" ✔
├── 2 ==> the rank of "banana" is "2" ✔
└── 3 ==> the rank of "lemon, lime" is "3" ✔
```



Please note that `name` is a `MessageFormat` pattern. Thus, a single quote (`'`) needs to be represented as a doubled single quote (`''`) in order to be displayed.

The following placeholders are supported within custom display names.

Placeholder	Description
`{displayName}`	the display name of the method
`{index}`	the current invocation index (1-based)
`{arguments}`	the complete, comma-separated arguments list
`{argumentsWithNames}`	the complete, comma-separated arguments list with parameter names
`{argumentSetName}`	the name of the argument set
`{argumentSetNameOrArgumentsWithNames}`	`{argumentSetName}` or `{argumentsWithNames}`, depending on how the arguments are supplied
`{0}`, `{1}`, …	an individual argument



When including arguments in display names, their string representations are truncated if they exceed the configured maximum length. The limit is configurable via the `junit.jupiter.params.displayname.argument.maxlength` configuration parameter and defaults to 512 characters.

When using `@MethodSource`, `@FieldSource`, or `@ArgumentsSource`, you can provide custom names for individual arguments or custom names for entire sets of arguments.

Use the `Named` API to provide a custom name for an individual argument, and the custom name will be used if the argument is included in the invocation display name, like in the example below.

```
@DisplayName("A parameterized test with named arguments")
@ParameterizedTest(name = "{index}: {0}")
@MethodSource("namedArguments")
void testWithNamedArguments(File file) {
}

static Stream<Arguments> namedArguments() {
    return Stream.of(
        arguments(named("An important file", new File("path1"))),
        arguments(named("Another file", new File("path2")))
    );
}
```

When executing the above method using the `ConsoleLauncher` you will see output similar to the following.

```
A parameterized test with named arguments ✔
├── 1: An important file ✔
└── 2: Another file ✔
```



Note that `arguments(Object…)` is a static factory method defined in the `org.junit.jupiter.params.provider.Arguments` interface.

Similarly, `named(String, Object)` is a static factory method defined in the `org.junit.jupiter.api.Named` interface.

Use the `ArgumentSet` API to provide a custom name for the entire set of arguments, and the custom name will be used as the display name, like in the example below.

```
@DisplayName("A parameterized test with named argument sets")
@ParameterizedTest
@FieldSource("argumentSets")
void testWithArgumentSets(File file1, File file2) {
}

static List<Arguments> argumentSets = Arrays.asList(
    argumentSet("Important files", new File("path1"), new File("path2")),
    argumentSet("Other files", new File("path3"), new File("path4"))
);
```

When executing the above method using the `ConsoleLauncher` you will see output similar to the following.

```
A parameterized test with named argument sets ✔
├── [1] Important files ✔
└── [2] Other files ✔
```



Note that `argumentSet(String, Object…)` is a static factory method defined in the `org.junit.jupiter.params.provider.Arguments` interface.

Quoted Text-based Arguments

As of JUnit Jupiter 6.0, text-based arguments in display names for parameterized tests are quoted by default. In this context, any `CharSequence` (such as a `String`) or `Character` is considered text. A `CharSequence` is wrapped in double quotes (`"`), and a `Character` is wrapped in single quotes (`'`).

Special characters will be escaped in the quoted text. For example, carriage returns and line feeds will be escaped as `\\r` and `\\n`, respectively.



This feature can be disabled by setting the `quoteTextArguments` attributes in `@ParameterizedClass` and `@ParameterizedTest` to `false`.

For example, given a string argument `"line 1\nline 2"`, the physical representation in the display name will be `"\"line 1\\nline 2\""` which is printed as `"line 1\nline 2"`. Similarly, given a string argument `"\t"`, the physical representation in the display name will be `"\"\\t\""` which is printed

as `"\t"` instead of a blank string or invisible tab character. The same applies for a character argument `'\t'`, whose physical representation in the display name would be `"'\\t'"` which is printed as `'\t'`.

For a concrete example, if you run the first `nullEmptyAndBlankStrings(String text)` parameterized test method from the Null and Empty Sources section above, the following display names are generated.

```
[1] text = null
[2] text = ""
[3] text = " "
[4] text = "   "
[5] text = "\t"
[6] text = "\n"
```

If you run the first `testWithCsvSource(String fruit, int rank)` parameterized test method from the @CsvSource section above, the following display names are generated.

```
[1] fruit = "apple", rank = "1"
[2] fruit = "banana", rank = "2"
[3] fruit = "lemon, lime", rank = "0xF1"
[4] fruit = "strawberry", rank = "700_000"
```



The original source arguments are quoted when generating a display name, and this occurs before any implicit or explicit argument conversion is performed.

For example, if a parameterized test accepts `3.14` as a `float` argument that was converted from `"3.14"` as an input string, `"3.14"` will be present in the display name instead of `3.14`. You can see the effect of this with the `rank` values in the above example.

Default Display Name Pattern

If you'd like to set a default name pattern for all parameterized classes and tests in your project, you can declare the `junit.jupiter.params.displayname.default` configuration parameter in the `junit-platform.properties` file as demonstrated in the following example (see Configuration Parameters for other options).

```
junit.jupiter.params.displayname.default = {index}
```

Precedence Rules

The display name for a parameterized class or test is determined according to the following precedence rules:

1. `name` attribute in `@ParameterizedClass` or `@ParameterizedTest`, if present

2. value of the `junit.jupiter.params.displayname.default` configuration parameter, if present
3. `DEFAULT_DISPLAY_NAME` constant defined in `org.junit.jupiter.params.ParameterizedInvocationConstants`

Lifecycle and Interoperability

Parameterized Tests

Each invocation of a parameterized test has the same lifecycle as a regular `@Test` method. For example, `@BeforeEach` methods will be executed before each invocation. Similar to Dynamic Tests, invocations will appear one by one in the test tree of an IDE. You may at will mix regular `@Test` methods and `@ParameterizedTest` methods within the same test class.

You may use `ParameterResolver` extensions with `@ParameterizedTest` methods. However, method parameters that are resolved by argument sources need to come first in the parameter list. Since a test class may contain regular tests as well as parameterized tests with different parameter lists, values from argument sources are not resolved for lifecycle methods (e.g. `@BeforeEach`) and test class constructors.

```
@BeforeEach
void beforeEach(TestInfo testInfo) {
    // ...
}

@ParameterizedTest
@ValueSource(strings = "apple")
void testWithRegularParameterResolver(String argument, TestReporter testReporter) {
    testReporter.publishEntry("argument", argument);
}

@AfterEach
void afterEach(TestInfo testInfo) {
    // ...
}
```

Parameterized Classes

Each invocation of a parameterized class has the same lifecycle as a regular test class. For example, `@BeforeAll` methods will be executed *once* before all invocations and `@BeforeEach` methods will be executed before each *test method* invocation. Similar to Dynamic Tests, invocations will appear one by one in the test tree of an IDE.

You may use `ParameterResolver` extensions with `@ParameterizedClass` constructors. However, if constructor injection is used, constructor parameters that are resolved by argument sources need to come first in the parameter list. Values from argument sources are not resolved for regular lifecycle methods (e.g. `@BeforeEach`).

In addition to regular lifecycle methods, parameterized classes may declare

`@BeforeParameterizedClassInvocation` and `@AfterParameterizedClassInvocation` lifecycle methods that are called once before/after each invocation of the parameterized class. These methods must be `static` unless the parameterized class is configured to use `@TestInstance(Lifecycle.PER_CLASS)` (see Test Instance Lifecycle).

These lifecycle methods may optionally declare parameters that are resolved depending on the setting of the `injectArguments` annotation attribute. If it is set to `false`, the parameters must be resolved by other registered ParameterResolver extensions. If the attribute is set to `true` (the default), the method may declare parameters that match the arguments of the parameterized class (see the Javadoc of `@BeforeParameterizedClassInvocation` and `@AfterParameterizedClassInvocation` for details). This may, for example, be used to initialize the used arguments as demonstrated by the following example.

Using parameterized class lifecycle methods

```
@ParameterizedClass
@MethodSource("textFiles")
class TextFileTests {

    static List<TextFile> textFiles() {
        return List.of(
            new TextFile("file1", "first content"),
            new TextFile("file2", "second content")
        );
    }

    @Parameter
    TextFile textFile;

    @BeforeParameterizedClassInvocation
    static void beforeInvocation(TextFile textFile, @TempDir Path tempDir) throws
Exception {
        var filePath = tempDir.resolve(textFile.fileName); ①
        textFile.path = Files.writeString(filePath, textFile.content);
    }

    @SuppressWarnings("DataFlowIssue")
    @AfterParameterizedClassInvocation
    static void afterInvocation(TextFile textFile) throws Exception {
        var actualContent = Files.readString(textFile.path); ③
        assertEquals(textFile.content, actualContent, "Content must not have
changed");
        // Custom cleanup logic, if necessary
        // File will be deleted automatically by @TempDir support
    }

    @SuppressWarnings("DataFlowIssue")
    @Test
    void test() {
        assertTrue(Files.exists(textFile.path)); ②
    }
```

```
    @Test
    void anotherTest() {
        // ...
    }

    static class TextFile {

        final String fileName;
        final String content;
        Path path;

        TextFile(String fileName, String content) {
            this.fileName = fileName;
            this.content = content;
        }

        @Override
        public String toString() {
            return fileName;
        }
    }
}
```

① Initialization of the argument *before* each invocation of the parameterized class

② Usage of the previously initialized argument in a test method

③ Validation and cleanup of the argument *after* each invocation of the parameterized class

Class Templates

A `@ClassTemplate` is not a regular test class but rather a template for the contained test cases. As such, it is designed to be invoked multiple times depending on invocation contexts returned by the registered providers. Thus, it must be used in conjunction with a registered `ClassTemplateInvocationContextProvider` extension. Each invocation of a class template behaves like the execution of a regular test class with full support for the same lifecycle callbacks and extensions. Please refer to Providing Invocation Contexts for Class Templates for usage examples.



Parameterized Classes are a built-in specialization of class templates.

Test Templates

A `@TestTemplate` method is not a regular test case but rather a template for a test case. As such, it is designed to be invoked multiple times depending on the number of invocation contexts returned by the registered providers. Thus, it must be used in conjunction with a registered `TestTemplateInvocationContextProvider` extension. Each invocation of a test template method behaves like the execution of a regular `@Test` method with full support for the same lifecycle callbacks and extensions. Please refer to Providing Invocation Contexts for Test Templates for usage

examples.



Repeated Tests and Parameterized Tests are built-in specializations of test templates.

Dynamic Tests

The standard `@Test` annotation in JUnit Jupiter described in Annotations is very similar to the `@Test` annotation in JUnit 4. Both describe methods that implement test cases. These test cases are static in the sense that they are fully specified at compile time, and their behavior cannot be changed by anything happening at runtime. *Assumptions provide a basic form of dynamic behavior but are intentionally rather limited in their expressiveness.*

In addition to these standard tests a completely new kind of test programming model has been introduced in JUnit Jupiter. This new kind of test is a *dynamic test* which is generated at runtime by a factory method that is annotated with `@TestFactory`.

In contrast to `@Test` methods, a `@TestFactory` method is not itself a test case but rather a factory for test cases. Thus, a dynamic test is the product of a factory. Technically speaking, a `@TestFactory` method must return a single `DynamicNode` or a *stream* of `DynamicNode` instances or any of its subclasses. In this context, a "stream" is anything that JUnit can reliably convert into a `Stream`, such as `Stream`, `Collection`, `Iterator`, `Iterable`, an array of objects, or any type that provides an `iterator(): Iterator` method (such as, for example, a `kotlin.sequences.Sequence`).

Instantiable subclasses of `DynamicNode` are `DynamicContainer` and `DynamicTest`. `DynamicContainer` instances are composed of a *display name* and a list of dynamic child nodes, enabling the creation of arbitrarily nested hierarchies of dynamic nodes. `DynamicTest` instances will be executed lazily, enabling dynamic and even non-deterministic generation of test cases.

Any `Stream` returned by a `@TestFactory` will be properly closed by calling `stream.close()`, making it safe to use a resource such as `Files.lines()`.

As with `@Test` methods, `@TestFactory` methods must not be `private` or `static` and may optionally declare parameters to be resolved by `ParameterResolvers`.

A `DynamicTest` is a test case generated at runtime. It is composed of a *display name* and an `Executable`. `Executable` is a `@FunctionalInterface` which means that the implementations of dynamic tests can be provided as *lambda expressions* or *method references*.



Dynamic Test Lifecycle

The execution lifecycle of a dynamic test is quite different than it is for a standard `@Test` case. Specifically, there are no lifecycle callbacks for individual dynamic tests. This means that `@BeforeEach` and `@AfterEach` methods and their corresponding extension callbacks are executed for the `@TestFactory` method but not for each *dynamic test*. In other words, if you access fields from the test instance within a lambda expression for a dynamic test, those fields will not be reset by callback methods or extensions between the execution of individual dynamic tests generated by the same `@TestFactory` method.

Dynamic Test Examples

The following `DynamicTestsDemo` class demonstrates several examples of test factories and dynamic tests.

The first method returns an invalid return type and will cause a warning to be reported by JUnit during test discovery. Such methods are not executed.

The next six methods demonstrate the generation of a `Collection`, `Iterable`, `Iterator`, array, or `Stream` of `DynamicTest` instances. Most of these examples do not really exhibit dynamic behavior but merely demonstrate the supported return types in principle. However, `dynamicTestsFromStream()` and `dynamicTestsFromIntStream()` demonstrate how to generate dynamic tests for a given set of strings or a range of input numbers.

The next method is truly dynamic in nature. `generateRandomNumberOfTests()` implements an `Iterator` that generates random numbers, a display name generator, and a test executor and then provides all three to `DynamicTest.stream()`. Although the non-deterministic behavior of `generateRandomNumberOfTests()` is of course in conflict with test repeatability and should thus be used with care, it serves to demonstrate the expressiveness and power of dynamic tests.

The next method is similar to `generateRandomNumberOfTests()` in terms of flexibility; however, `dynamicTestsFromStreamFactoryMethod()` generates a stream of dynamic tests from an existing `Stream` via the `DynamicTest.stream()` factory method.

For demonstration purposes, the `dynamicNodeSingleTest()` method generates a single `DynamicTest` instead of a stream, and the `dynamicNodeSingleContainer()` method generates a nested hierarchy of dynamic tests utilizing `DynamicContainer`.

```
import static example.util.StringUtils.isPalindrome;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.DynamicContainer.dynamicContainer;
import static org.junit.jupiter.api.DynamicTest.dynamicTest;

import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;
import java.util.Random;
import java.util.function.Function;
import java.util.stream.IntStream;
import java.util.stream.Stream;

import example.util.Calculator;

import org.junit.jupiter.api.DynamicNode;
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.Tag;
```

```
import org.junit.jupiter.api.TestFactory;
import org.junit.jupiter.api.function.ThrowingConsumer;

class DynamicTestsDemo {

    private final Calculator calculator = new Calculator();

    // This method will not be executed but produce a warning
    @TestFactory
    List<String> dynamicTestsWithInvalidReturnType() {
        return Arrays.asList("Hello");
    }

    @TestFactory
    Collection<DynamicTest> dynamicTestsFromCollection() {
        return Arrays.asList(
            dynamicTest("1st dynamic test", () -> assertTrue(isPalindrome("madam"))),
            dynamicTest("2nd dynamic test", () -> assertEquals(4, calculator.multiply
(2, 2)))
        );
    }

    @TestFactory
    Iterable<DynamicTest> dynamicTestsFromIterable() {
        return Arrays.asList(
            dynamicTest("3rd dynamic test", () -> assertTrue(isPalindrome("madam"))),
            dynamicTest("4th dynamic test", () -> assertEquals(4, calculator.multiply
(2, 2)))
        );
    }

    @TestFactory
    Iterator<DynamicTest> dynamicTestsFromIterator() {
        return Arrays.asList(
            dynamicTest("5th dynamic test", () -> assertTrue(isPalindrome("madam"))),
            dynamicTest("6th dynamic test", () -> assertEquals(4, calculator.multiply
(2, 2)))
        ).iterator();
    }

    @TestFactory
    DynamicTest[] dynamicTestsFromArray() {
        return new DynamicTest[] {
            dynamicTest("7th dynamic test", () -> assertTrue(isPalindrome("madam"))),
            dynamicTest("8th dynamic test", () -> assertEquals(4, calculator.multiply
(2, 2)))
        };
    }

    @TestFactory
    Stream<DynamicTest> dynamicTestsFromStream() {
```

```
        return Stream.of("racecar", "radar", "mom", "dad")
            .map(text -> dynamicTest(text, () -> assertTrue(isPalindrome(text))));
    }

    @TestFactory
    Stream<DynamicTest> dynamicTestsFromIntStream() {
        // Generates tests for the first 10 even integers.
        return IntStream.iterate(0, n -> n + 2).limit(10)
            .mapToObj(n -> dynamicTest("test" + n, () -> assertEquals(0, n % 2)));
    }

    @TestFactory
    Stream<DynamicTest> generateRandomNumberOfTests() {

        // Generates random positive integers between 0 and 100 until
        // a number evenly divisible by 7 is encountered.
        Iterator<Integer> inputGenerator = new Iterator<>() {

            Random random = new Random();
            int current;

            @Override
            public boolean hasNext() {
                current = random.nextInt(100);
                return current % 7 != 0;
            }

            @Override
            public Integer next() {
                return current;
            }
        };

        // Generates display names like: input:5, input:37, input:85, etc.
        Function<Integer, String> displayNameGenerator = (input) -> "input:" + input;

        // Executes tests based on the current input value.
        ThrowingConsumer<Integer> testExecutor = (input) -> assertTrue(input % 7 !=
0);

        // Returns a stream of dynamic tests.
        return DynamicTest.stream(inputGenerator, displayNameGenerator, testExecutor);
    }

    @TestFactory
    Stream<DynamicTest> dynamicTestsFromStreamFactoryMethod() {
        // Stream of palindromes to check
        Stream<String> inputStream = Stream.of("racecar", "radar", "mom", "dad");

        // Generates display names like: racecar is a palindrome
        Function<String, String> displayNameGenerator = text -> text + " is a
```

```
palindrome";

        // Executes tests based on the current input value.
        ThrowingConsumer<String> testExecutor = text -> assertTrue(isPalindrome(
text));

        // Returns a stream of dynamic tests.
        return DynamicTest.stream(inputStream, displayNameGenerator, testExecutor);
    }

    @TestFactory
    Stream<DynamicNode> dynamicTestsWithContainers() {
        return Stream.of("A", "B", "C")
            .map(input -> dynamicContainer("Container " + input, Stream.of(
                dynamicTest("not null", () -> assertNotNull(input)),
                dynamicContainer("properties", Stream.of(
                    dynamicTest("length > 0", () -> assertTrue(input.length() > 0)),
                    dynamicTest("not empty", () -> assertFalse(input.isEmpty()))
                ))
            )));
    }

    @TestFactory
    DynamicNode dynamicNodeSingleTest() {
        return dynamicTest("'pop' is a palindrome", () -> assertTrue(isPalindrome
("pop")));
    }

    @TestFactory
    DynamicNode dynamicNodeSingleContainer() {
        return dynamicContainer("palindromes",
            Stream.of("racecar", "radar", "mom", "dad")
                .map(text -> dynamicTest(text, () -> assertTrue(isPalindrome(text)))
        ));
    }
}
```

Dynamic Tests and Named

In some cases, it can be more natural to specify inputs together with a descriptive name using the Named API and the corresponding `stream()` factory methods on `DynamicTest` as shown in the first example below. The second example takes it one step further and allows to provide the code block that should be executed by implementing the `Executable` interface along with `Named` via the `NamedExecutable` base class.

```
import static example.util.StringUtils.isPalindrome;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Named.named;
```

```
import java.util.stream.Stream;

import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.NamedExecutable;
import org.junit.jupiter.api.TestFactory;

public class DynamicTestsNamedDemo {

    @TestFactory
    Stream<DynamicTest> dynamicTestsFromStreamFactoryMethodWithNames() {
        // Stream of palindromes to check
        var inputStream = Stream.of(
            named("racecar is a palindrome", "racecar"),
            named("radar is also a palindrome", "radar"),
            named("mom also seems to be a palindrome", "mom"),
            named("dad is yet another palindrome", "dad")
        );

        // Returns a stream of dynamic tests.
        return DynamicTest.stream(inputStream, text -> assertTrue(isPalindrome(
text)));
    }

    @TestFactory
    Stream<DynamicTest> dynamicTestsFromStreamFactoryMethodWithNamedExecutables() {
        // Stream of palindromes to check
        var inputStream = Stream.of("racecar", "radar", "mom", "dad")
                .map(PalindromeNamedExecutable::new);

        // Returns a stream of dynamic tests based on NamedExecutables.
        return DynamicTest.stream(inputStream);
    }

    record PalindromeNamedExecutable(String text) implements NamedExecutable {

        @Override
        public String getName() {
            return "'%s' is a palindrome".formatted(text);
        }

        @Override
        public void execute() {
            assertTrue(isPalindrome(text));
        }
    }
}
```

URI Test Sources for Dynamic Tests

The JUnit Platform provides `TestSource`, a representation of the source of a test or container used to

navigate to its location by IDEs and build tools.

The `TestSource` for a dynamic test or dynamic container can be constructed from a `java.net.URI` which can be supplied via the `DynamicTest.dynamicTest(String, URI, Executable)` or `DynamicContainer.dynamicContainer(String, URI, Stream)` factory method, respectively. The `URI` will be converted to one of the following `TestSource` implementations.

`ClasspathResourceSource`

If the `URI` contains the `classpath` scheme — for example, `classpath:/test/foo.xml?line=20,column=2`.

`DirectorySource`

If the `URI` represents a directory present in the file system.

`FileSource`

If the `URI` represents a file present in the file system.

`MethodSource`

If the `URI` contains the `method` scheme and the fully qualified method name (FQMN) — for example, `method:org.junit.Foo#bar(java.lang.String, java.lang.String[])`. Please refer to the Javadoc for `DiscoverySelectors`.https://docs.junit.org/6.0.1/api/org.junit.platform.engine/org/junit/platform/engine/discovery/DiscoverySelectors.html#selectMethod(java.lang.String)[selectMethod] for the supported formats for a FQMN.

`ClassSource`

If the `URI` contains the `class` scheme and the fully qualified class name — for example, `class:org.junit.Foo?line=42`.

`UriSource`

If none of the above `TestSource` implementations are applicable.

Timeouts

The `@Timeout` annotation allows one to declare that a test, test factory, test template, or lifecycle method should fail if its execution time exceeds a given duration. The time unit for the duration defaults to seconds but is configurable.

The following example shows how `@Timeout` is applied to lifecycle and test methods.

```
class TimeoutDemo {

    @BeforeEach
    @Timeout(5)
    void setUp() {
        // fails if execution time exceeds 5 seconds
    }
```

```
    @Test
    @Timeout(value = 500, unit = TimeUnit.MILLISECONDS)
    void failsIfExecutionTimeExceeds500Milliseconds() {
        // fails if execution time exceeds 500 milliseconds
    }

    @Test
    @Timeout(value = 500, unit = TimeUnit.MILLISECONDS, threadMode = ThreadMode
.SEPARATE_THREAD)
    void failsIfExecutionTimeExceeds500MillisecondsInSeparateThread() {
        // fails if execution time exceeds 500 milliseconds, the test code is executed
in a separate thread
    }

}
```

To apply the same timeout to all test methods within a test class and all of its `@Nested` classes, you can declare the `@Timeout` annotation at the class level. It will then be applied to all test, test factory, and test template methods within that class and its `@Nested` classes unless overridden by a `@Timeout` annotation on a specific method or `@Nested` class. Please note that `@Timeout` annotations declared at the class level are not applied to lifecycle methods.

Declaring `@Timeout` on a `@TestFactory` method checks that the factory method returns within the specified duration but does not verify the execution time of each individual `DynamicTest` generated by the factory. Please use `assertTimeout()` or `assertTimeoutPreemptively()` for that purpose.

If `@Timeout` is present on a `@TestTemplate` method — for example, a `@RepeatedTest` or `@ParameterizedTest` — each invocation will have the given timeout applied to it.

Thread mode

The timeout can be applied using one of the following three thread modes: `SAME_THREAD`, `SEPARATE_THREAD`, or `INFERRED`.

When `SAME_THREAD` is used, the execution of the annotated method proceeds in the main thread of the test. If the timeout is exceeded, the main thread is interrupted from another thread. This is done to ensure interoperability with frameworks such as Spring that make use of mechanisms that are sensitive to the currently running thread — for example, `ThreadLocal` transaction management.

On the contrary when `SEPARATE_THREAD` is used, like the `assertTimeoutPreemptively()` assertion, the execution of the annotated method proceeds in a separate thread, this can lead to undesirable side effects, see Preemptive Timeouts with `assertTimeoutPreemptively()`.

When `INFERRED` (default) thread mode is used, the thread mode is resolved via the `junit.jupiter.execution.timeout.thread.mode.default` configuration parameter. If the provided configuration parameter is invalid or not present then `SAME_THREAD` is used as fallback.

Default Timeouts

The following configuration parameters can be used to specify default timeouts for all methods of a certain category unless they or an enclosing test class is annotated with `@Timeout`:

`junit.jupiter.execution.timeout.default`
: Default timeout for all testable and lifecycle methods

`junit.jupiter.execution.timeout.testable.method.default`
: Default timeout for all testable methods

`junit.jupiter.execution.timeout.test.method.default`
: Default timeout for `@Test` methods

`junit.jupiter.execution.timeout.testtemplate.method.default`
: Default timeout for `@TestTemplate` methods

`junit.jupiter.execution.timeout.testfactory.method.default`
: Default timeout for `@TestFactory` methods

`junit.jupiter.execution.timeout.lifecycle.method.default`
: Default timeout for all lifecycle methods

`junit.jupiter.execution.timeout.beforeall.method.default`
: Default timeout for `@BeforeAll` methods

`junit.jupiter.execution.timeout.beforeeach.method.default`
: Default timeout for `@BeforeEach` methods

`junit.jupiter.execution.timeout.aftereach.method.default`
: Default timeout for `@AfterEach` methods

`junit.jupiter.execution.timeout.afterall.method.default`
: Default timeout for `@AfterAll` methods

More specific configuration parameters override less specific ones. For example, `junit.jupiter.execution.timeout.test.method.default` overrides `junit.jupiter.execution.timeout.testable.method.default` which overrides `junit.jupiter.execution.timeout.default`.

The values of such configuration parameters must be in the following, case-insensitive format: `<number> [ns|µs|ms|s|m|h|d]`. The space between the number and the unit may be omitted. Specifying no unit is equivalent to using seconds.

Example timeout configuration parameter values

Parameter value	Equivalent annotation
`42`	`@Timeout(42)`
`42 ns`	`@Timeout(value = 42, unit = NANOSECONDS)`

Parameter value	Equivalent annotation
`42 µs`	`@Timeout(value = 42, unit = MICROSECONDS)`
`42 ms`	`@Timeout(value = 42, unit = MILLISECONDS)`
`42 s`	`@Timeout(value = 42, unit = SECONDS)`
`42 m`	`@Timeout(value = 42, unit = MINUTES)`
`42 h`	`@Timeout(value = 42, unit = HOURS)`
`42 d`	`@Timeout(value = 42, unit = DAYS)`

Using @Timeout for Polling Tests

When dealing with asynchronous code, it is common to write tests that poll while waiting for something to happen before performing any assertions. In some cases you can rewrite the logic to use a `CountDownLatch` or another synchronization mechanism, but sometimes that is not possible — for example, if the subject under test sends a message to a channel in an external message broker and assertions cannot be performed until the message has been successfully sent through the channel. Asynchronous tests like these require some form of timeout to ensure they don't hang the test suite by executing indefinitely, as would be the case if an asynchronous message never gets successfully delivered.

By configuring a timeout for an asynchronous test that polls, you can ensure that the test does not execute indefinitely. The following example demonstrates how to achieve this with JUnit Jupiter's `@Timeout` annotation. This technique can be used to implement "poll until" logic very easily.

```
@Test
@Timeout(5) // Poll at most 5 seconds
void pollUntil() throws InterruptedException {
    while (asynchronousResultNotAvailable()) {
        Thread.sleep(250); // custom poll interval
    }
    // Obtain the asynchronous result and perform assertions
}
```



If you need more control over polling intervals and greater flexibility with asynchronous tests, consider using a dedicated library such as Awaitility.

Debugging Timeouts

Registered Pre-Interrupt Callback extensions are called prior to invoking `Thread.interrupt()` on the thread that is executing the timed out method. This allows to inspect the application state and output additional information that might be helpful for diagnosing the cause of a timeout.

Thread Dump on Timeout

JUnit registers a default implementation of the Pre-Interrupt Callback extension point that dumps the stacks of all threads to `System.out` if enabled by setting the `junit.jupiter.execution.timeout.threaddump.enabled` configuration parameter to `true`.

Disable @Timeout Globally

When stepping through your code in a debug session, a fixed timeout limit may influence the result of the test, e.g. mark the test as failed although all assertions were met.

JUnit Jupiter supports the `junit.jupiter.execution.timeout.mode` configuration parameter to configure when timeouts are applied. There are three modes: `enabled`, `disabled`, and `disabled_on_debug`. The default mode is `enabled`. A VM runtime is considered to run in debug mode when one of its input parameters starts with `-agentlib:jdwp` or `-Xrunjdwp`. This heuristic is queried by the `disabled_on_debug` mode.

Parallel Execution

By default, JUnit Jupiter tests are run sequentially in a single thread; however, running tests in parallel — for example, to speed up execution — is available as an opt-in feature. To enable parallel execution, set the `junit.jupiter.execution.parallel.enabled` configuration parameter to `true` — for example, in `junit-platform.properties` (see Configuration Parameters for other options).

Please note that enabling this property is only the first step required to execute tests in parallel. If enabled, test classes and methods will still be executed sequentially by default. Whether or not a node in the test tree is executed concurrently is controlled by its execution mode. The following two modes are available.

`SAME_THREAD`
: Force execution in the same thread used by the parent. For example, when used on a test method, the test method will be executed in the same thread as any `@BeforeAll` or `@AfterAll` methods of the containing test class.

`CONCURRENT`
: Execute concurrently unless a resource lock forces execution in the same thread.

By default, nodes in the test tree use the `SAME_THREAD` execution mode. You can change the default by setting the `junit.jupiter.execution.parallel.mode.default` configuration parameter. Alternatively, you can use the `@Execution` annotation to change the execution mode for the annotated element and its subelements (if any) which allows you to activate parallel execution for individual test classes, one by one.

Configuration parameters to execute all tests in parallel

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

The default execution mode is applied to all nodes of the test tree with a few notable exceptions, namely test classes that use the `Lifecycle.PER_CLASS` mode or a `MethodOrderer`. In the former case, test authors have to ensure that the test class is thread-safe; in the latter, concurrent execution might conflict with the configured execution order. Thus, in both cases, test methods in such test classes are only executed concurrently if the `@Execution(CONCURRENT)` annotation is present on the test class or method.

You can use the `@Execution` annotation to explicitly configure the execution mode for a test class or method:

```
@Execution(ExecutionMode.CONCURRENT)
class ExplicitExecutionModeDemo {

    @Test
    void testA() {
        // concurrent
    }

    @Test
    @Execution(ExecutionMode.SAME_THREAD)
    void testB() {
        // overrides to same_thread
    }

}
```

This allows test classes or methods to opt in or out of concurrent execution regardless of the globally configured default.

When parallel execution is enabled and a default `ClassOrderer` is registered (see Class Order for details), top-level test classes will initially be sorted accordingly and scheduled in that order. However, they are not guaranteed to be started in exactly that order since the threads they are executed on are not controlled directly by JUnit.

All nodes of the test tree that are configured with the `CONCURRENT` execution mode will be executed fully in parallel according to the provided configuration while observing the declarative synchronization mechanism. Please note that Capturing Standard Output/Error needs to be enabled separately.

In addition, you can configure the default execution mode for top-level classes by setting the `junit.jupiter.execution.parallel.mode.classes.default` configuration parameter. By combining both configuration parameters, you can configure classes to run in parallel but their methods in the same thread:

Configuration parameters to execute top-level classes in parallel but methods in same thread

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

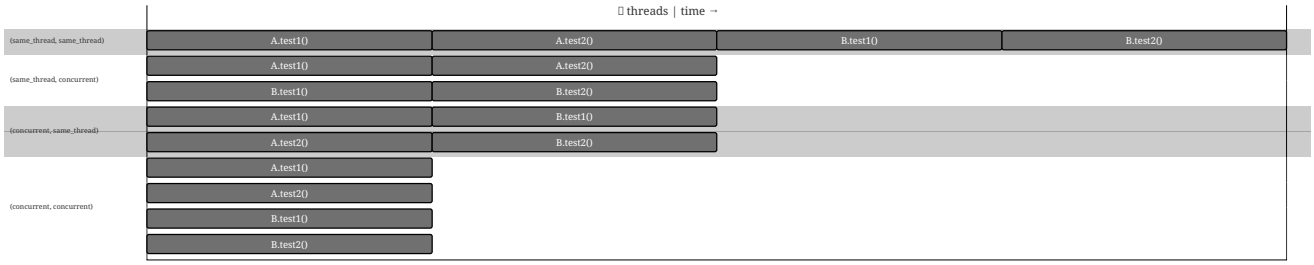
The opposite combination will run all methods within one class in parallel, but top-level classes will run sequentially:

Configuration parameters to execute top-level classes sequentially but their methods in parallel

```
junit.jupiter.execution.parallel.enabled = true
```

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

The following diagram illustrates how the execution of two top-level test classes `A` and `B` with two test methods per class behaves for all four combinations of `junit.jupiter.execution.parallel.mode.default` and `junit.jupiter.execution.parallel.mode.classes.default` (see labels in first column).



Default execution mode configuration combinations

If the `junit.jupiter.execution.parallel.mode.classes.default` configuration parameter is not explicitly set, the value for `junit.jupiter.execution.parallel.mode.default` will be used instead.

Configuration

Properties such as the desired parallelism and the maximum pool size can be configured using a `ParallelExecutionConfigurationStrategy`. The JUnit Platform provides two implementations out of the box: `dynamic` and `fixed`. Alternatively, you may implement a `custom` strategy.

To select a strategy, set the `junit.jupiter.execution.parallel.config.strategy` configuration parameter to one of the following options.

`dynamic`

Computes the desired parallelism based on the number of available processors/cores multiplied by the `junit.jupiter.execution.parallel.config.dynamic.factor` configuration parameter (defaults to `1`). The optional `junit.jupiter.execution.parallel.config.dynamic.max-pool-size-factor` configuration parameter can be used to limit the maximum number of threads.

`fixed`

Uses the mandatory `junit.jupiter.execution.parallel.config.fixed.parallelism` configuration parameter as the desired parallelism. The optional `junit.jupiter.execution.parallel.config.fixed.max-pool-size` configuration parameter can be used to limit the maximum number of threads.

`custom`

Allows you to specify a custom `ParallelExecutionConfigurationStrategy` implementation via the mandatory `junit.jupiter.execution.parallel.config.custom.class` configuration parameter to determine the desired configuration.

If no configuration strategy is set, JUnit Jupiter uses the `dynamic` configuration strategy with a factor of `1`. Consequently, the desired parallelism will be equal to the number of available processors/cores.



Parallelism alone does not imply maximum number of concurrent threads

By default JUnit Jupiter does not guarantee that the number of concurrently executing tests will not exceed the configured parallelism. For example, when using one of the synchronization mechanisms described in the next section, the `ForkJoinPool` that is used behind the scenes may spawn additional threads to ensure execution continues with sufficient parallelism. If you require such guarantees, it is possible to limit the maximum number of concurrent threads by controlling the maximum pool size of the `dynamic`, `fixed` and `custom` strategies.

Relevant properties

The following table lists relevant properties for configuring parallel execution. See Configuration Parameters for details on how to set such properties.

Property	Description	Supported Values	Default Value
`junit.jupiter.execution.parallel.enabled`	Enable parallel test execution	• `true` • `false`	`false`
`junit.jupiter.execution.parallel.mode.default`	Default execution mode of nodes in the test tree	• `concurrent` • `same_thread`	`same_thread`
`junit.jupiter.execution.parallel.mode.classes.default`	Default execution mode of top-level classes	• `concurrent` • `same_thread`	`same_thread`
`junit.jupiter.execution.parallel.config.strategy`	Execution strategy for desired parallelism and maximum pool size	• `dynamic` • `fixed` • `custom`	`dynamic`
`junit.jupiter.execution.parallel.config.dynamic.factor`	Factor to be multiplied by the number of available processors/cores to determine the desired parallelism for the `dynamic` configuration strategy	a positive decimal number	`1.0`
`junit.jupiter.execution.parallel.config.dynamic.max-pool-size-factor`	Factor to be multiplied by the number of available processors/cores and the value of `junit.jupiter.execution.parallel.config.dynamic.factor` to determine the desired parallelism for the `dynamic` configuration strategy	a positive decimal number, must be greater than or equal to `1.0`	256 + the value of `junit.jupiter.execution.parallel.config.dynamic.factor` multiplied by the number of available processors/cores
`junit.jupiter.execution.parallel.config.dynamic.saturate`	Disable saturation of the underlying fork-join pool for the `dynamic` configuration strategy	• `true` • `false`	`true`
`junit.jupiter.execution.parallel.config.fixed.parallelism`	Desired parallelism for the `fixed` configuration strategy	a positive integer	no default value
`junit.jupiter.execution.parallel.config.fixed.max-pool-size`	Desired maximum pool size of the underlying fork-join pool for the `fixed` configuration strategy	a positive integer, must be greater than or equal to `junit.jupiter.execution.parallel.config.fixed.parallelism`	256 + the value of `junit.jupiter.execution.parallel.config.fixed.parallelism`

Property	Description	Supported Values	Default Value
`junit.jupiter.execution.parallel.config.fixed.saturate`	Disable saturation of the underlying fork-join pool for the `fixed` configuration strategy	• `true` • `false`	`true`
`junit.jupiter.execution.parallel.config.custom.class`	Fully qualified class name of the *ParallelExecutionConfigurationStrategy* to be used for the `custom` configuration strategy	for example, *org.example.CustomStrategy*	no default value

Synchronization

In addition to controlling the execution mode using the `@Execution` annotation, JUnit Jupiter provides another annotation-based declarative synchronization mechanism. The `@ResourceLock` annotation allows you to declare that a test class or method uses a specific shared resource that requires synchronized access to ensure reliable test execution. The shared resource is identified by a unique name which is a `String`. The name can be user-defined or one of the predefined constants in `Resources`: `SYSTEM_PROPERTIES`, `SYSTEM_OUT`, `SYSTEM_ERR`, `LOCALE`, or `TIME_ZONE`.

In addition to declaring these shared resources statically, the `@ResourceLock` annotation has a `providers` attribute that allows registering implementations of the `ResourceLocksProvider` interface that can add shared resources dynamically at runtime. Note that resources declared statically with `@ResourceLock` annotation are combined with resources added dynamically by `ResourceLocksProvider` implementations.

If the tests in the following example were run in parallel *without* the use of `@ResourceLock`, they would be *flaky*. Sometimes they would pass, and at other times they would fail due to the inherent race condition of writing and then reading the same JVM System Property.

When access to shared resources is declared using the `@ResourceLock` annotation, the JUnit Jupiter engine uses this information to ensure that no conflicting tests are run in parallel. This guarantee extends to lifecycle methods of a test class or method. For example, if a test method is annotated with a `@ResourceLock` annotation, the "lock" will be acquired before any `@BeforeEach` methods are executed and released after all `@AfterEach` methods have been executed.



Running tests in isolation

If most of your test classes can be run in parallel without any synchronization but you have some test classes that need to run in isolation, you can mark the latter with the `@Isolated` annotation. Tests in such classes are executed sequentially without any other tests running at the same time.

In addition to the `String` that uniquely identifies the shared resource, you may specify an access mode. Two tests that require `READ` access to a shared resource may run in parallel with each other but not while any other test that requires `READ_WRITE` access to the same shared resource is running.

Declaring shared resources "statically" with `@ResourceLock` annotation

```
@Execution(CONCURRENT)
class StaticSharedResourcesDemo {

    private Properties backup;

    @BeforeEach
    void backup() {
        backup = new Properties();
        backup.putAll(System.getProperties());
    }

    @AfterEach
    void restore() {
        System.setProperties(backup);
    }

    @Test
    @ResourceLock(value = SYSTEM_PROPERTIES, mode = READ)
    void customPropertyIsNotSetByDefault() {
        assertNull(System.getProperty("my.prop"));
    }

    @Test
    @ResourceLock(value = SYSTEM_PROPERTIES, mode = READ_WRITE)
    void canSetCustomPropertyToApple() {
        System.setProperty("my.prop", "apple");
        assertEquals("apple", System.getProperty("my.prop"));
    }

    @Test
    @ResourceLock(value = SYSTEM_PROPERTIES, mode = READ_WRITE)
    void canSetCustomPropertyToBanana() {
        System.setProperty("my.prop", "banana");
        assertEquals("banana", System.getProperty("my.prop"));
    }

}
```

Adding shared resources "dynamically" with `ResourceLocksProvider` implementation

```
@Execution(CONCURRENT)
@ResourceLock(providers = DynamicSharedResourcesDemo.Provider.class)
class DynamicSharedResourcesDemo {

    private Properties backup;

    @BeforeEach
    void backup() {
```

```
        backup = new Properties();
        backup.putAll(System.getProperties());
    }

    @AfterEach
    void restore() {
        System.setProperties(backup);
    }

    @Test
    void customPropertyIsNotSetByDefault() {
        assertNull(System.getProperty("my.prop"));
    }

    @Test
    void canSetCustomPropertyToApple() {
        System.setProperty("my.prop", "apple");
        assertEquals("apple", System.getProperty("my.prop"));
    }

    @Test
    void canSetCustomPropertyToBanana() {
        System.setProperty("my.prop", "banana");
        assertEquals("banana", System.getProperty("my.prop"));
    }

    static class Provider implements ResourceLocksProvider {

        @Override
        public Set<Lock> provideForMethod(List<Class<?>> enclosingInstanceTypes,
Class<?> testClass,
                Method testMethod) {
            ResourceAccessMode mode = testMethod.getName().startsWith("canSet") ?
READ_WRITE : READ;
            return Set.of(new Lock(SYSTEM_PROPERTIES, mode));
        }
    }

}
```

Also, "static" shared resources can be declared for *direct* child nodes via the `target` attribute in the `@ResourceLock` annotation, the attribute accepts a value from the `ResourceLockTarget` enum.

Specifying `target = CHILDREN` in a class-level `@ResourceLock` annotation has the same semantics as adding an annotation with the same `value` and `mode` to each test method and nested test class declared in this class.

This may improve parallelization when a test class declares a `READ` lock, but only a few methods hold a `READ_WRITE` lock.

Tests in the following example would run in the `SAME_THREAD` if the `@ResourceLock` didn't have `target = CHILDREN`. This is because the test class declares a `READ` shared resource, but one test method holds a `READ_WRITE` lock, which would force the `SAME_THREAD` execution mode for all the test methods.

Declaring shared resources for child nodes with `target` attribute

```
@Execution(CONCURRENT)
@ResourceLock(value = "a", mode = READ, target = CHILDREN)
public class ChildrenSharedResourcesDemo {

    @ResourceLock(value = "a", mode = READ_WRITE)
    @Test
    void test1() throws InterruptedException {
        Thread.sleep(2000L);
    }

    @Test
    void test2() throws InterruptedException {
        Thread.sleep(2000L);
    }

    @Test
    void test3() throws InterruptedException {
        Thread.sleep(2000L);
    }

    @Test
    void test4() throws InterruptedException {
        Thread.sleep(2000L);
    }

    @Test
    void test5() throws InterruptedException {
        Thread.sleep(2000L);
    }

}
```

Built-in Extensions

While the JUnit team encourages reusable extensions to be packaged and maintained in separate libraries, JUnit Jupiter includes a few user-facing extension implementations that are considered so generally useful that users shouldn't have to add another dependency.

The @TempDir Extension

The built-in `TempDirectory` extension is used to create and clean up a temporary directory for an individual test or all tests in a test class. It is registered by default. To use it, annotate a non-final, unassigned field of type `java.nio.file.Path` or `java.io.File` with `@TempDir` or add a parameter of

type `java.nio.file.Path` or `java.io.File` annotated with `@TempDir` to a test class constructor, lifecycle method, or test method.

For example, the following test declares a parameter annotated with `@TempDir` for a single test method, creates and writes to a file in the temporary directory, and checks its content.

A test method that requires a temporary directory

```
@Test
void writeItemsToFile(@TempDir Path tempDir) throws IOException {
    Path file = tempDir.resolve("test.txt");

    new ListWriter(file).write("a", "b", "c");

    assertEquals(List.of("a,b,c"), Files.readAllLines(file));
}
```

You can inject multiple temporary directories by specifying multiple annotated parameters.

A test method that requires multiple temporary directories

```
@Test
void copyFileFromSourceToTarget(@TempDir Path source, @TempDir Path target) throws
IOException {
    Path sourceFile = source.resolve("test.txt");
    new ListWriter(sourceFile).write("a", "b", "c");

    Path targetFile = Files.copy(sourceFile, target.resolve("test.txt"));

    assertNotEquals(sourceFile, targetFile);
    assertEquals(List.of("a,b,c"), Files.readAllLines(targetFile));
}
```

The following example stores a *shared* temporary directory in a `static` field. This allows the same `sharedTempDir` to be used in all lifecycle methods and test methods of the test class. For better isolation, you should use an instance field or constructor injection so that each test method uses a separate directory.

A test class that shares a temporary directory across test methods

```
class SharedTempDirectoryDemo {

    @TempDir
    static Path sharedTempDir;

    @Test
    void writeItemsToFile() throws IOException {
        Path file = sharedTempDir.resolve("test.txt");

        new ListWriter(file).write("a", "b", "c");
```

```
        assertEquals(List.of("a,b,c"), Files.readAllLines(file));
    }

    @Test
    void anotherTestThatUsesTheSameTempDir() {
        // use sharedTempDir
    }

}
```

The `@TempDir` annotation has an optional `cleanup` attribute that can be set to either `NEVER`, `ON_SUCCESS`, or `ALWAYS`. If the cleanup mode is set to `NEVER`, the temporary directory will not be deleted after the test completes. If it is set to `ON_SUCCESS`, the temporary directory will only be deleted after the test if the test completed successfully.

The default cleanup mode is `ALWAYS`. You can use the `junit.jupiter.tempdir.cleanup.mode.default` configuration parameter to override this default.

A test class with a temporary directory that doesn't get cleaned up

```
class CleanupModeDemo {

    @Test
    void fileTest(@TempDir(cleanup = ON_SUCCESS) Path tempDir) {
        // perform test
    }

}
```

`@TempDir` supports the programmatic creation of temporary directories via the optional `factory` attribute. This is typically used to gain control over the temporary directory creation, like defining the parent directory or the file system that should be used.

Factories can be created by implementing `TempDirFactory`. Implementations must provide a no-args constructor and should not make any assumptions regarding when and how many times they are instantiated, but they can assume that their `createTempDirectory(…)` and `close()` methods will both be called once per instance, in this order, and from the same thread.

The default implementation available in Jupiter delegates directory creation to `java.nio.file.Files::createTempDirectory` which uses the default file system and the system's temporary directory as the parent directory. It passes `junit-` as the prefix string of the generated directory name to help identify it as a created by JUnit.

The following example defines a factory that uses the test name as the directory name prefix instead of the `junit` constant value.

A test class with a temporary directory having the test name as the directory name prefix

```
class TempDirFactoryDemo {
```

```
    @Test
    void factoryTest(@TempDir(factory = Factory.class) Path tempDir) {
        assertTrue(tempDir.getFileName().toString().startsWith("factoryTest"));
    }

    static class Factory implements TempDirFactory {

        @Override
        public Path createTempDirectory(AnnotatedElementContext elementContext,
ExtensionContext extensionContext)
                throws IOException {
            return Files.createTempDirectory(extensionContext.getRequiredTestMethod
().getName());
        }

    }

}
```

It is also possible to use an in-memory file system like `Jimfs` for the creation of the temporary directory. The following example demonstrates how to achieve that.

A test class with a temporary directory created with the Jimfs in-memory file system

```
class InMemoryTempDirDemo {

    @Test
    void test(@TempDir(factory = JimfsTempDirFactory.class) Path tempDir) {
        // perform test
    }

    static class JimfsTempDirFactory implements TempDirFactory {

        private final FileSystem fileSystem = Jimfs.newFileSystem(Configuration.
unix());

        @Override
        public Path createTempDirectory(AnnotatedElementContext elementContext,
ExtensionContext extensionContext)
                throws IOException {
            return Files.createTempDirectory(fileSystem.getPath("/"), "junit-");
        }

        @Override
        public void close() throws IOException {
            fileSystem.close();
        }

    }
```

```
}
```

`@TempDir` can also be used as a meta-annotation to reduce repetition. The following code listing shows how to create a custom `@JimfsTempDir` annotation that can be used as a drop-in replacement for `@TempDir(factory = JimfsTempDirFactory.class)`.

A custom annotation meta-annotated with `@TempDir`

```
@Target({ ElementType.ANNOTATION_TYPE, ElementType.FIELD, ElementType.PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
@TempDir(factory = JimfsTempDirFactory.class)
@interface JimfsTempDir {
}
```

The following example demonstrates how to use the custom `@JimfsTempDir` annotation.

A test class using the custom annotation

```
class JimfsTempDirAnnotationDemo {

    @Test
    void test(@JimfsTempDir Path tempDir) {
        // perform test
    }

}
```

Meta-annotations or additional annotations on the field or parameter the `TempDir` annotation is declared on might expose additional attributes to configure the factory. Such annotations and related attributes can be accessed via the `AnnotatedElementContext` parameter of the `createTempDirectory(…)` method.

You can use the `junit.jupiter.tempdir.factory.default` configuration parameter to specify the fully qualified class name of the `TempDirFactory` you would like to use by default. Just like for factories configured via the `factory` attribute of the `@TempDir` annotation, the supplied class has to implement the `TempDirFactory` interface. The default factory will be used for all `@TempDir` annotations unless the `factory` attribute of the annotation specifies a different factory.

In summary, the factory for a temporary directory is determined according to the following precedence rules:

1. The `factory` attribute of the `@TempDir` annotation, if present
2. The default `TempDirFactory` configured via the configuration parameter, if present
3. Otherwise, `org.junit.jupiter.api.io.TempDirFactory$Standard` will be used.

The @AutoClose Extension

The built-in `AutoCloseExtension` automatically closes resources associated with fields. It is registered by default. To use it, annotate a field in a test class with `@AutoClose`.

`@AutoClose` fields may be either `static` or non-static. If the value of an `@AutoClose` field is `null` when it is evaluated the field will be ignored, but a warning message will be logged to inform you.

By default, `@AutoClose` expects the value of the annotated field to implement a `close()` method that will be invoked to close the resource. However, developers can customize the name of the close method via the `value` attribute. For example, `@AutoClose("shutdown")` instructs JUnit to look for a `shutdown()` method to close the resource.

`@AutoClose` fields are inherited from superclasses. Furthermore, `@AutoClose` fields from subclasses will be closed before `@AutoClose` fields in superclasses.

When multiple `@AutoClose` fields exist within a given test class, the order in which the resources are closed depends on an algorithm that is deterministic but intentionally nonobvious. This ensures that subsequent runs of a test suite close resources in the same order, thereby allowing for repeatable builds.

The `AutoCloseExtension` implements the `AfterAllCallback` and `TestInstancePreDestroyCallback` extension APIs. Consequently, a `static` `@AutoClose` field will be closed after all tests in the current test class have completed, effectively after `@AfterAll` methods have executed for the test class. A non-static `@AutoClose` field will be closed before the current test class instance is destroyed. Specifically, if the test class is configured with `@TestInstance(Lifecycle.PER_METHOD)` semantics, a non-static `@AutoClose` field will be closed after the execution of each test method, test factory method, or test template method. However, if the test class is configured with `@TestInstance(Lifecycle.PER_CLASS)` semantics, a non-static `@AutoClose` field will not be closed until the current test class instance is no longer needed, which means after `@AfterAll` methods and after all `static` `@AutoClose` fields have been closed.

The following example demonstrates how to annotate an instance field with `@AutoClose` so that the resource is automatically closed after test execution. In this example, we assume that the default `@TestInstance(Lifecycle.PER_METHOD)` semantics apply.

A test class using `@AutoClose` to close a resource

```
class AutoCloseDemo {

    @AutoClose ①
    WebClient webClient = new WebClient(); ②

    String serverUrl = // specify server URL ...

    @Test
    void getProductList() {
        // Use WebClient to connect to web server and verify response
        assertEquals(200, webClient.get(serverUrl + "/products").getResponseStatus());
    }
```

```
}
```

① Annotate an instance field with `@AutoClose`.

② `WebClient` implements `java.lang.AutoCloseable` which defines a `close()` method that will be invoked after each `@Test` method.

Migrating from JUnit 4

Although the JUnit Jupiter programming model and extension model do not support JUnit 4 features such as `Rules` and `Runners` natively, it is not expected that source code maintainers will need to update all of their existing tests, test extensions, and custom build test infrastructure to migrate to JUnit Jupiter.

Instead, JUnit provides a gentle migration path via a *JUnit Vintage test engine* which allows existing tests based on JUnit 3 and JUnit 4 to be executed using the JUnit Platform infrastructure. Since all classes and annotations specific to JUnit Jupiter reside under the `org.junit.jupiter` base package, having both JUnit 4 and JUnit Jupiter in the classpath does not lead to any conflicts. It is therefore safe to maintain existing JUnit 4 tests alongside JUnit Jupiter tests and migrate them gradually.

Running JUnit 4 Tests on the JUnit Platform

The JUnit Vintage engine is deprecated and should only be used temporarily while migrating tests to JUnit Jupiter or another testing framework with native JUnit Platform support.



By default, if the JUnit Vintage engine is registered and discovers at least one test class, it reports a discovery issue of INFO severity. You can prevent this discovery issue from being reported by setting the `junit.vintage.discovery.issue.reporting.enabled` configuration parameter to `false`.

Make sure that the `junit-vintage-engine` artifact is in your test runtime path. In that case JUnit 3 and JUnit 4 tests will automatically be picked up by the JUnit Platform launcher.

See the example projects in the `junit-examples` repository to find out how this is done with Gradle and Maven.

Categories Support

For test classes or methods that are annotated with `@Category`, the *JUnit Vintage test engine* exposes the category's fully qualified class name as a tag for the corresponding test class or test method. For example, if a test method is annotated with `@Category(Example.class)`, it will be tagged with `"com.acme.Example"`. Similar to the `Categories` runner in JUnit 4, this information can be used to filter the discovered tests before executing them (see Running Tests for details).

Parallel Execution

The JUnit Vintage test engine supports parallel execution of top-level test classes and test methods, allowing existing JUnit 3 and JUnit 4 tests to benefit from improved performance through concurrent test execution. It can be enabled and configured using the following configuration parameters:

`junit.vintage.execution.parallel.enabled=true|false`

Enable/disable parallel execution (defaults to `false`). Requires opt-in for `classes` or `methods` to be executed in parallel using the configuration parameters below.

`junit.vintage.execution.parallel.classes=true|false`

Enable/disable parallel execution of test classes (defaults to `false`).

`junit.vintage.execution.parallel.methods=true|false`

Enable/disable parallel execution of test methods (defaults to `false`).

`junit.vintage.execution.parallel.pool-size=<number>`

Specifies the size of the thread pool to be used for parallel execution. By default, the number of available processors is used.

Parallelization at Class Level

Let's assume we have two test classes `FooTest` and `BarTest` with each class containing three unit tests. Now, let's enable parallel execution of test classes:

```
junit.vintage.execution.parallel.enabled=true
junit.vintage.execution.parallel.classes=true
```

With this setup, the `VintageTestEngine` will use two different threads, one for each test class:

```
ForkJoinPool-1-worker-1 - BarTest::test1
ForkJoinPool-1-worker-2 - FooTest::test1
ForkJoinPool-1-worker-1 - BarTest::test2
ForkJoinPool-1-worker-2 - FooTest::test2
ForkJoinPool-1-worker-1 - BarTest::test3
ForkJoinPool-1-worker-2 - FooTest::test3
```

Parallelization at Method Level

Alternatively, we can enable parallel test execution at a method level, rather than the class level:

```
junit.vintage.execution.parallel.enabled=true
junit.vintage.execution.parallel.methods=true
```

Therefore, the test methods within each class will be executed in parallel, while different test classes will be executed sequentially:

```
ForkJoinPool-1-worker-1 - BarTest::test1
ForkJoinPool-1-worker-2 - BarTest::test2
ForkJoinPool-1-worker-3 - BarTest::test3
```

```
ForkJoinPool-1-worker-3 - FooTest::test1
ForkJoinPool-1-worker-2 - FooTest::test2
ForkJoinPool-1-worker-1 - FooTest::test3
```

Full Parallelization

Finally, we can also enable parallelization at both class and method level:

```
junit.vintage.execution.parallel.enabled=true
junit.vintage.execution.parallel.classes=true
junit.vintage.execution.parallel.methods=true
```

With these properties set, the `VintageTestEngine` will execute all tests classes and methods in parallel, potentially significantly reducing the overall test suite execution time:

```
ForkJoinPool-1-worker-6 - FooTest::test2
ForkJoinPool-1-worker-7 - BarTest::test3
ForkJoinPool-1-worker-3 - FooTest::test1
ForkJoinPool-1-worker-8 - FooTest::test3
ForkJoinPool-1-worker-5 - BarTest::test2
ForkJoinPool-1-worker-4 - BarTest::test1
```

Configuring the Pool Size

The default thread pool size is equal to the number of available processors. However, we can also configure the pool size explicitly:

```
junit.vintage.execution.parallel.enabled=true
junit.vintage.execution.parallel.classes=true
junit.vintage.execution.parallel.methods=true
junit.vintage.execution.parallel.pool-size=4
```

For instance, if we update our previous example that uses full parallelization and configure the pool size to four, we can expect to see our six test methods executed with a parallelism of four:

```
ForkJoinPool-1-worker-2 - FooTest::test1
ForkJoinPool-1-worker-4 - BarTest::test2
ForkJoinPool-1-worker-3 - BarTest::test1
ForkJoinPool-1-worker-4 - BarTest::test3
ForkJoinPool-1-worker-2 - FooTest::test2
ForkJoinPool-1-worker-3 - FooTest::test3
```

As we can see, even though we set the thread pool size was four, only three threads were used in this case. This happens because the pool adjusts the number of active threads based on workload and system needs.

Sequential Execution

On the other hand, if we disable parallel execution, the `VintageTestEngine` will execute all tests sequentially, regardless of the other properties:

```
junit.vintage.execution.parallel.enabled=false
junit.vintage.execution.parallel.classes=true
junit.vintage.execution.parallel.methods=true
```

Similarly, tests will be executed sequentially if you enable parallel execution in general but enable neither class-level nor method-level parallelization.

Migration Tips

The following are topics that you should be aware of when migrating existing JUnit 4 tests to JUnit Jupiter.

- Annotations reside in the `org.junit.jupiter.api` package.
- Assertions reside in `org.junit.jupiter.api.Assertions`.
  - Note that you may continue to use assertion methods from `org.junit.Assert` or any other assertion library such as AssertJ, Hamcrest, Truth, etc.
- Assumptions reside in `org.junit.jupiter.api.Assumptions`.
  - Note that JUnit Jupiter supports methods from JUnit 4's `org.junit.Assume` class for assumptions. Specifically, JUnit Jupiter supports JUnit 4's `AssumptionViolatedException` to signal that a test should be aborted instead of marked as a failure.
- `@Before` and `@After` no longer exist; use `@BeforeEach` and `@AfterEach` instead.
- `@BeforeClass` and `@AfterClass` no longer exist; use `@BeforeAll` and `@AfterAll` instead.
- `@Ignore` no longer exists: use `@Disabled` or one of the other built-in execution conditions instead
  - See also JUnit 4 @Ignore Support.
- `@Category` no longer exists; use `@Tag` instead.
- `@RunWith` no longer exists; superseded by `@ExtendWith`.
  - For `@RunWith(Enclosed.class)` use `@Nested`.
  - For `@RunWith(Parameterized.class)` see Parameterized test classes.
- `@Rule` and `@ClassRule` no longer exist; superseded by `@ExtendWith` and `@RegisterExtension`.
  - See also Limited JUnit 4 Rule Support.
- `@Test(expected = ⋯)` and the `ExpectedException` rule no longer exist; use `Assertions.assertThrows(⋯)` instead.
  - See Limited JUnit 4 Rule Support if you still need to use `ExpectedException`.
- Assertions and assumptions in JUnit Jupiter accept the failure message as their last argument instead of the first one.

- See Failure Message Arguments for details.

Parameterized test classes

Unless `@UseParametersRunnerFactory` is used, a JUnit 4 parameterized test class can be converted into a JUnit Jupiter `@ParameterizedClass` by following these steps:

1. Replace `@RunWith(Parameterized.class)` with `@ParameterizedClass`.
2. Add a class-level `@MethodSource("methodName")` annotation where `methodName` is the name of the method annotated with `@Parameters` and remove the `@Parameters` annotation from the method.
3. Replace `@BeforeParam` and `@AfterParam` with `@BeforeParameterizedClassInvocation` and `@AfterParameterizedClassInvocation`, respectively, if there are any methods with such annotations.
4. Change the imports of the `@Test` and `@Parameter` annotations to use the `org.junit.jupiter.params` package.
5. Change assertions etc. to use the `org.junit.jupiter.api` package as usual.
6. Optionally, remove all `public` modifiers from the class and its methods and fields.

Before

```
@RunWith(Parameterized.class)
public class JUnit4ParameterizedClassTests {

    @Parameterized.Parameters
    public static Iterable<Object[]> data() {
        return Arrays.asList(new Object[][] { { 1, "foo" }, { 2, "bar" } });
    }

    @Parameterized.Parameter(0)
    public int number;

    @Parameterized.Parameter(1)
    public String text;

    @Parameterized.BeforeParam
    public static void before(int number, String text) {
    }

    @Parameterized.AfterParam
    public static void after() {
    }

    @org.junit.Test
    public void someTest() {
    }

    @org.junit.Test
```

```
    public void anotherTest() {
    }
}
```

After

```
@ParameterizedClass
@MethodSource("data")
class JupiterParameterizedClassTests {

    static Iterable<Object[]> data() {
        return Arrays.asList(new Object[][] { { 1, "foo" }, { 2, "bar" } });
    }

    @org.junit.jupiter.params.Parameter(0)
    int number;

    @org.junit.jupiter.params.Parameter(1)
    String text;

    @BeforeParameterizedClassInvocation
    static void before(int number, String text) {
    }

    @AfterParameterizedClassInvocation
    static void after() {
    }

    @org.junit.jupiter.api.Test
    void someTest() {
    }

    @org.junit.jupiter.api.Test
    void anotherTest() {
    }
}
```

Limited JUnit 4 Rule Support



JUnit 4 rule support is deprecated for removal since version 6.0.0. Please migrate to the corresponding APIs and extensions provided by JUnit Jupiter.

As stated above, JUnit Jupiter does not and will not support JUnit 4 rules natively. The JUnit team realizes, however, that many organizations, especially large ones, are likely to have large JUnit 4 code bases that make use of custom rules. To serve these organizations and enable a gradual migration path the JUnit team has decided to support a selection of JUnit 4 rules verbatim within JUnit Jupiter. This support is based on adapters and is limited to those rules that are semantically

compatible to the JUnit Jupiter extension model, i.e. those that do not completely change the overall execution flow of the test.

The `junit-jupiter-migrationsupport` module from JUnit Jupiter currently supports the following three `Rule` types including subclasses of these types:

- `org.junit.rules.ExternalResource` (including `org.junit.rules.TemporaryFolder`)
- `org.junit.rules.Verifier` (including `org.junit.rules.ErrorCollector`)
- `org.junit.rules.ExpectedException`

As in JUnit 4, Rule-annotated fields as well as methods are supported. By using these class-level extensions on a test class such `Rule` implementations in legacy code bases can be *left unchanged* including the JUnit 4 rule import statements.

This limited form of `Rule` support can be switched on by the class-level annotation `@EnableRuleMigrationSupport`. This annotation is a *composed annotation* which enables all rule migration support extensions: `VerifierSupport`, `ExternalResourceSupport`, and `ExpectedExceptionSupport`. You may alternatively choose to annotate your test class with `@EnableJUnit4MigrationSupport` which registers migration support for rules *and* JUnit 4's `@Ignore` annotation (see JUnit 4 @Ignore Support).

However, if you intend to develop a new extension for JUnit Jupiter please use the new extension model of JUnit Jupiter instead of the rule-based model of JUnit 4.

JUnit 4 @Ignore Support



JUnit 4 `@Ignore` *support* is deprecated for removal since version 6.0.0. Please use JUnit Jupiter's `@Disabled` annotation instead.

In order to provide a smooth migration path from JUnit 4 to JUnit Jupiter, the `junit-jupiter-migrationsupport` module provides support for JUnit 4's `@Ignore` annotation analogous to Jupiter's `@Disabled` annotation.

To use `@Ignore` with JUnit Jupiter based tests, configure a *test* dependency on the `junit-jupiter-migrationsupport` module in your build and then annotate your test class with `@ExtendWith(IgnoreCondition.class)` or `@EnableJUnit4MigrationSupport` (which automatically registers the `IgnoreCondition` along with Limited JUnit 4 Rule Support). The `IgnoreCondition` is an `ExecutionCondition` that disables test classes or test methods that are annotated with `@Ignore`.

```
import org.junit.Ignore;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.migrationsupport.EnableJUnit4MigrationSupport;

// @ExtendWith(IgnoreCondition.class)
@SuppressWarnings("removal")
@EnableJUnit4MigrationSupport
class IgnoredTestsDemo {
```

```
    @Ignore
    @Test
    void testWillBeIgnored() {
    }

    @Test
    void testWillBeExecuted() {
    }
}
```

Failure Message Arguments

The `Assumptions` and `Assertions` classes in JUnit Jupiter declare arguments in a different order than in JUnit 4. In JUnit 4 assertion and assumption methods accept the failure message as the first argument; whereas, in JUnit Jupiter assertion and assumption methods accept the failure message as the last argument.

For instance, the method `assertEquals` in JUnit 4 is declared as `assertEquals(String message, Object expected, Object actual)`, but in JUnit Jupiter it is declared as `assertEquals(Object expected, Object actual, String message)`. The rationale for this is that a failure message is *optional*, and optional arguments should be declared after required arguments in a method signature.

The methods affected by this change are the following:

- Assertions
  - `assertTrue`
  - `assertFalse`
  - `assertNull`
  - `assertNotNull`
  - `assertEquals`
  - `assertNotEquals`
  - `assertArrayEquals`
  - `assertSame`
  - `assertNotSame`
  - `assertThrows`
- Assumptions
  - `assumeTrue`
  - `assumeFalse`

Running Tests

This section explains how to run tests from IDEs and build tools.

IDE Support

IntelliJ IDEA

IntelliJ IDEA supports running tests on the JUnit Platform since version 2016.2. For more information, please consult this IntelliJ IDEA resource. Note, however, that it is recommended to use IDEA 2017.3 or newer since more recent versions of IDEA download the following JARs automatically based on the API version used in the project: `junit-platform-launcher`, `junit-jupiter-engine`, and `junit-vintage-engine`.

In order to use a different JUnit version (e.g., 6.0.1), you may need to include the corresponding versions of the `junit-platform-launcher`, `junit-jupiter-engine`, and `junit-vintage-engine` JARs in the classpath.

Additional Gradle Dependencies

```
testImplementation(platform("org.junit:junit-bom:6.0.1"))
testRuntimeOnly("org.junit.platform:junit-platform-launcher")
testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine")
testRuntimeOnly("org.junit.vintage:junit-vintage-engine")
```

Additional Maven Dependencies

```
<!-- ... -->
<dependencies>
    <dependency>
        <groupId>org.junit.platform</groupId>
        <artifactId>junit-platform-launcher</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.junit</groupId>
```

```
            <artifactId>junit-bom</artifactId>
            <version>6.0.1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Eclipse

Eclipse IDE offers support for the JUnit Platform since the Eclipse Oxygen.1a (4.7.1a) release.

For more information on using JUnit Platform in Eclipse consult the official *Eclipse support for JUnit 5* section of the Eclipse Project Oxygen.1a (4.7.1a) - New and Noteworthy documentation.

NetBeans

NetBeans offers support for JUnit Jupiter and the JUnit Platform since the Apache NetBeans 10.0 release.

For more information consult the JUnit 5 section of the Apache NetBeans 10.0 release notes.

Visual Studio Code

Visual Studio Code supports JUnit Jupiter and the JUnit Platform via the Java Test Runner extension which is installed by default as part of the Java Extension Pack.

For more information consult the *Testing* section of the Java in Visual Studio Code documentation.

Other IDEs

If you are using an editor or IDE other than one of those listed in the previous sections, and it doesn't support running tests on the JUnit Platform, you can use the Console Launcher to run them from the command line.

Build Support

Gradle

Starting with version 4.6, Gradle provides native support for executing tests on the JUnit Platform. To enable it, you need to specify `useJUnitPlatform()` within a `test` task declaration in `build.gradle`:

```
test {
    useJUnitPlatform()
}
```

Filtering by tags, tag expressions, or engines is also supported:

```
test {
    useJUnitPlatform {
        includeTags("fast", "smoke & feature-a")
        // excludeTags("slow", "ci")
        includeEngines("junit-jupiter")
        // excludeEngines("junit-vintage")
    }
}
```

Please refer to the official Gradle documentation for a comprehensive list of options.

Aligning dependency versions



See Spring Boot for details on how to override the version of JUnit used in your Spring Boot application.

Unless you're using Spring Boot which defines its own way of managing dependencies, it is recommended to use the JUnit Platform Bill of Materials (BOM) to align the versions of all JUnit artifacts.

Explicit platform dependency on the BOM

```
dependencies {
    testImplementation(platform("org.junit:junit-bom:6.0.1"))
    testImplementation("org.junit.jupiter:junit-jupiter")
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")
}
```

Using the BOM allows you to omit the version when declaring dependencies on all artifacts with the `org.junit.platform`, `org.junit.jupiter`, and `org.junit.vintage` group IDs.

Since all JUnit artifacts declare a platform dependency on the BOM, you usually don't need to declare an explicit dependency on it yourself. Instead, it's sufficient to declare *one* regular dependency that includes a version number. Gradle will then pull in the BOM automatically so you can omit the version for all other JUnit artifacts.

Implicit platform dependency on the BOM

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:6.0.1") ①
    testRuntimeOnly("org.junit.platform:junit-platform-launcher") ②
}
```

① Dependency declaration with explicit version. Pulls in the `junit-bom` automatically.

② Dependency declaration without version. The version is supplied by the `junit-bom`.



Declaring a dependency on junit-platform-launcher

> Even though pre-8.0 versions of Gradle don't require declaring an explicit dependency on `junit-platform-launcher`, it is recommended to do so to ensure the versions of JUnit artifacts on the test runtime classpath are aligned.
>
> Moreover, doing so is recommended and in some cases even required when importing the project into an IDE like Eclipse or IntelliJ IDEA.

Configuring Test Engines

In order to run any tests at all, a `TestEngine` implementation must be on the classpath.

To configure support for JUnit Jupiter based tests, configure a `testImplementation` dependency on the dependency-aggregating JUnit Jupiter artifact similar to the following.

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:6.0.1")
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")
}
```

Alternatively, you can use Gradle's JVM Test Suite support.

Kotlin DSL

```
testing {
    suites {
        named<JvmTestSuite>("test") {
            useJUnitJupiter("6.0.1")
        }
    }
}
```

Groovy DSL

```
testing {
    suites {
        test {
            useJUnitJupiter("6.0.1")
        }
    }
}
```

The JUnit Platform can run JUnit 4 based tests as long as you configure a `testImplementation` dependency on JUnit 4 and a `testRuntimeOnly` dependency on the JUnit Vintage `TestEngine` implementation similar to the following.

```
dependencies {
    testImplementation("junit:junit:4.13.2")
    testRuntimeOnly("org.junit.vintage:junit-vintage-engine:6.0.1")
```

```
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")
}
```

Configuration Parameters

The standard Gradle `test` task currently does not provide a dedicated DSL to set JUnit Platform configuration parameters to influence test discovery and execution. However, you can provide configuration parameters within the build script via system properties (as shown below) or via the `junit-platform.properties` file.

```
test {
    // ...
    systemProperty("junit.jupiter.conditions.deactivate", "*")
    systemProperty("junit.jupiter.extensions.autodetection.enabled", true)
    systemProperty("junit.jupiter.testinstance.lifecycle.default", "per_class")
    // ...
}
```

Configuring Logging (optional)

JUnit uses the Java Logging APIs in the `java.util.logging` package (a.k.a. *JUL*) to emit warnings and debug information. Please refer to the official documentation of `LogManager` for configuration options.

Alternatively, it's possible to redirect log messages to other logging frameworks such as Log4j or Logback. To use a logging framework that provides a custom implementation of `LogManager`, set the `java.util.logging.manager` system property to the *fully qualified class name* of the `LogManager` implementation to use. The example below demonstrates how to configure Log4j 2.x (see Log4j JDK Logging Adapter for details).

```
test {
    systemProperty("java.util.logging.manager",
"org.apache.logging.log4j.jul.LogManager")
    // Avoid overhead (see
https://logging.apache.org/log4j/2.x/manual/jmx.html#enabling-jmx)
    systemProperty("log4j2.disableJmx", "true")
}
```

Other logging frameworks provide different means to redirect messages logged using `java.util.logging`. For example, for Logback you can use the JUL to SLF4J Bridge by adding it as a dependency to the test runtime classpath.

Maven

Maven Surefire and Maven Failsafe provide native support for executing tests on the JUnit Platform. The `pom.xml` file in the `junit-jupiter-starter-maven` project demonstrates how to use the Maven Surefire plugin and can serve as a starting point for configuring your Maven build.



Minimum required version of Maven Surefire/Failsafe

As of JUnit 6.0, the minimum required version of Maven Surefire/Failsafe is 3.0.0.

Aligning dependency versions

Unless you're using Spring Boot which defines its own way of managing dependencies, it is recommended to use the JUnit Platform Bill of Materials (BOM) to align the versions of all JUnit artifacts.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.junit</groupId>
            <artifactId>junit-bom</artifactId>
            <version>6.0.1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Using the BOM allows you to omit the version when declaring dependencies on all artifacts with the `org.junit.platform`, `org.junit.jupiter`, and `org.junit.vintage` group IDs.



See Spring Boot for details on how to override the version of JUnit used in your Spring Boot application.

Configuring Test Engines

In order to have Maven Surefire or Maven Failsafe run any tests at all, at least one `TestEngine` implementation must be added to the test classpath.

To configure support for JUnit Jupiter based tests, configure `test` scoped dependencies on the JUnit Jupiter API and the JUnit Jupiter `TestEngine` implementation similar to the following.

```
<!-- ... -->
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>6.0.1</version> <!-- can be omitted when using the BOM -->
        <scope>test</scope>
    </dependency>
    <!-- ... -->
</dependencies>
<build>
    <plugins>
```

```
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.5.4</version>
        </plugin>
        <plugin>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>3.5.4</version>
        </plugin>
    </plugins>
</build>
<!-- ... -->
```

Maven Surefire and Maven Failsafe can run JUnit 4 based tests alongside Jupiter tests as long as you configure `test` scoped dependencies on JUnit 4 and the JUnit Vintage `TestEngine` implementation similar to the following.

```
<!-- ... -->
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
        <version>6.0.1</version> <!-- can be omitted when using the BOM -->
        <scope>test</scope>
    </dependency>
    <!-- ... -->
</dependencies>
<!-- ... -->
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.5.4</version>
        </plugin>
        <plugin>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>3.5.4</version>
        </plugin>
    </plugins>
</build>
<!-- ... -->
```

Filtering by Test Class Names

The Maven Surefire Plugin will scan for test classes whose fully qualified names match the following patterns.

- `**/Test*.java`
- `**/*Test.java`
- `**/*Tests.java`
- `**/*TestCase.java`

Moreover, it will exclude all nested classes (including static member classes) by default.

Note, however, that you can override this default behavior by configuring explicit `include` and `exclude` rules in your `pom.xml` file. For example, to keep Maven Surefire from excluding static member classes, you can override its exclude rules as follows.

Overriding exclude rules of Maven Surefire

```
<!-- ... -->
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.5.4</version>
            <configuration>
                <excludes>
                    <exclude/>
                </excludes>
            </configuration>
        </plugin>
    </plugins>
</build>
<!-- ... -->
```

Please see the Inclusions and Exclusions of Tests documentation for Maven Surefire for details.

Filtering by Tags

You can filter tests by tags or tag expressions using the following configuration properties.

- to include *tags* or *tag expressions*, use `groups`.
- to exclude *tags* or *tag expressions*, use `excludedGroups`.

```
<!-- ... -->
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.5.4</version>
```

```
            <configuration>
                <groups>acceptance | !feature-a</groups>
                <excludedGroups>integration, regression</excludedGroups>
            </configuration>
        </plugin>
    </plugins>
</build>
<!-- ... -->
```

Configuration Parameters

You can set JUnit Platform configuration parameters to influence test discovery and execution by declaring the `configurationParameters` property and providing key-value pairs using the Java `Properties` file syntax (as shown below) or via the `junit-platform.properties` file.

```
<!-- ... -->
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.5.4</version>
            <configuration>
                <properties>
                    <configurationParameters>
                        junit.jupiter.conditions.deactivate = *
                        junit.jupiter.extensions.autodetection.enabled = true
                        junit.jupiter.testinstance.lifecycle.default = per_class
                    </configurationParameters>
                </properties>
            </configuration>
        </plugin>
    </plugins>
</build>
<!-- ... -->
```

Ant

Starting with version `1.10.3`, Ant has a `junitlauncher` task that provides native support for launching tests on the JUnit Platform. The `junitlauncher` task is solely responsible for launching the JUnit Platform and passing it the selected collection of tests. The JUnit Platform then delegates to registered test engines to discover and execute the tests.

The `junitlauncher` task attempts to align as closely as possible with native Ant constructs such as resource collections for allowing users to select the tests that they want executed by test engines. This gives the task a consistent and natural feel when compared to many other core Ant tasks.

Starting with version `1.10.6` of Ant, the `junitlauncher` task supports forking the tests in a separate JVM.

The `build.xml` file in the `junit-jupiter-starter-ant` project demonstrates how to use the task and can serve as a starting point.

Basic Usage

The following example demonstrates how to configure the `junitlauncher` task to select a single test class (i.e., `com.example.project.CalculatorTests`).

```
<path id="test.classpath">
    <!-- The location where you have your compiled classes -->
    <pathelement location="${build.classes.dir}" />
</path>

<!-- ... -->

<junitlauncher>
    <classpath refid="test.classpath" />
    <test name="com.example.project.CalculatorTests" />
</junitlauncher>
```

The `test` element allows you to specify a single test class that you want to be selected and executed. The `classpath` element allows you to specify the classpath to be used to launch the JUnit Platform. This classpath will also be used to locate test classes that are part of the execution.

The following example demonstrates how to configure the `junitlauncher` task to select test classes from multiple locations.

```
<path id="test.classpath">
    <!-- The location where you have your compiled classes -->
    <pathelement location="${build.classes.dir}" />
</path>
<!-- ... -->
<junitlauncher>
    <classpath refid="test.classpath" />
    <testclasses outputdir="${output.dir}">
        <fileset dir="${build.classes.dir}">
            <include name="org/example/**/demo/**/" />
        </fileset>
        <fileset dir="${some.other.dir}">
            <include name="org/myapp/**/" />
        </fileset>
    </testclasses>
</junitlauncher>
```

In the above example, the `testclasses` element allows you to select multiple test classes that reside in different locations.

For further details on usage and configuration options please refer to the official Ant

documentation for the `junitlauncher` task.

Spring Boot

Spring Boot provides automatic support for managing the version of JUnit used in your project. In addition, the `spring-boot-starter-test` artifact automatically includes testing libraries such as JUnit Jupiter, AssertJ, Mockito, etc.

If your build relies on dependency management support from Spring Boot, you should not import JUnit's Bill of Materials (BOM) in your build script since that would result in duplicate (and potentially conflicting) management of JUnit dependencies.

If you need to override the version of a dependency used in your Spring Boot application, you have to override the exact name of the version property defined in the BOM used by the Spring Boot plugin. For example, the name of the JUnit Jupiter version property in Spring Boot is `junit-jupiter.version`. The mechanism for changing a dependency version is documented for both Gradle and Maven.

With Gradle you can override the JUnit Jupiter version by including the following in your `build.gradle` file.

```
ext['junit-jupiter.version'] = '6.0.1'
```

With Maven you can override the JUnit Jupiter version by including the following in your `pom.xml` file.

```
<properties>
    <junit-jupiter.version>6.0.1</junit-jupiter.version>
</properties>
```

Console Launcher

The `ConsoleLauncher` is a command-line Java application that lets you launch the JUnit Platform from the console. For example, it can be used to run JUnit Vintage and JUnit Jupiter tests and print test execution results to the console.

An executable *Fat JAR* (`junit-platform-console-standalone-6.0.1.jar`) that contains the contents of all of its dependencies is published in the Maven Central repository under the junit-platform-console-standalone directory. It contains the contents of the following artifacts:

- `junit:junit:4.13.2`
- `org.apiguardian:apiguardian-api:1.1.2`
- `org.hamcrest:hamcrest-core:1.3`
- `org.junit.jupiter:junit-jupiter-api:6.0.1`
- `org.junit.jupiter:junit-jupiter-engine:6.0.1`

- org.junit.jupiter:junit-jupiter-params:6.0.1
- org.junit.platform:junit-platform-commons:6.0.1
- org.junit.platform:junit-platform-console:6.0.1
- org.junit.platform:junit-platform-engine:6.0.1
- org.junit.platform:junit-platform-launcher:6.0.1
- org.junit.platform:junit-platform-reporting:6.0.1
- org.junit.platform:junit-platform-suite-api:6.0.1
- org.junit.platform:junit-platform-suite-engine:6.0.1
- org.junit.vintage:junit-vintage-engine:6.0.1
- org.opentest4j.reporting:open-test-reporting-tooling-spi:0.2.4
- org.opentest4j:opentest4j:1.3.0



Since the junit-platform-console-standalone JAR contains the contents of all of its dependencies, its Maven POM does not declare any dependencies.

Furthermore, it is not very likely that you would need to include a dependency on the junit-platform-console-standalone artifact in your project's Maven POM or Gradle build script. On the contrary, the executable junit-platform-console-standalone JAR is typically invoked directly from the command line or a shell script without a build script.

If you need to declare dependencies in your build script on some of the artifacts contained in the junit-platform-console-standalone artifact, you should declare dependencies only on the JUnit artifacts that are used in your project. To simplify dependency management of JUnit artifacts in your build, you may wish to use the junit-jupiter aggregator artifact or junit-bom. See Dependency Metadata for details.

You can run the standalone ConsoleLauncher as shown below.

```
$ java -jar junit-platform-console-standalone-6.0.1.jar execute <OPTIONS>

├─ JUnit Vintage
│  └─ example.JUnit4Tests
│     └─ standardJUnit4Test ✔
└─ JUnit Jupiter
   ├─ StandardTests
   │  ├─ succeedingTest() ✔
   │  └─ skippedTest() ↓ for demonstration purposes
   └─ A special test case
      ├─ Custom test name containing spaces ✔
      ├─ ╯°□°)╯ ✔
      └─ ↓ ✔

Test run finished after 64 ms
```

```
[         5 containers found      ]
[         0 containers skipped    ]
[         5 containers started    ]
[         0 containers aborted    ]
[         5 containers successful ]
[         0 containers failed     ]
[         6 tests found           ]
[         1 tests skipped         ]
[         5 tests started         ]
[         0 tests aborted         ]
[         5 tests successful      ]
[         0 tests failed          ]
```

You can also run the standalone `ConsoleLauncher` as shown below (for example, to include all jars in a directory):

```
$ java -cp classes:testlib/* org.junit.platform.console.ConsoleLauncher <OPTIONS>
```

Subcommands and Options

The `ConsoleLauncher` provides the following subcommands:

```
Usage: junit [OPTIONS] COMMAND
Launches the JUnit Platform for test discovery and execution.
      [@<filename>...]   One or more argument files containing options.
  -h, --help             Display help information.
      --version          Display version information.
      --disable-ansi-colors
                         Disable ANSI colors in output (not supported by all
terminals).
Commands:
  discover  Discover tests
  execute   Execute tests
  engines   List available test engines

For more information, please refer to the JUnit User Guide at
https://docs.junit.org/6.0.1/user-guide/
```

Discovering tests

```
Usage: junit discover [OPTIONS]
Discover tests
      [@<filename>...]       One or more argument files containing options.
      --disable-ansi-colors  Disable ANSI colors in output (not supported by all
terminals).
      --disable-banner       Disable print out of the welcome message.
  -h, --help                 Display help information.
```

```
      --version              Display version information.

SELECTORS

      --scan-classpath, --scan-class-path[=PATH]
                             Scan all directories on the classpath or explicit
classpath
                               roots. Without arguments, only directories on the
system
                               classpath as well as additional classpath entries
supplied via
                               -cp (directories and JAR files) are scanned. Explicit
classpath
                               roots that are not on the classpath will be silently
ignored.
                               This option can be repeated.
      --scan-modules         Scan all resolved modules for test discovery.
  -u, --select-uri=URI...    Select a URI for test discovery. This option can be
repeated.
  -f, --select-file=FILE...  Select a file for test discovery. The line and column
numbers can
                               be provided as URI query parameters (e.g. foo.txt?
                               line=12&column=34). This option can be repeated.
  -d, --select-directory=DIR...
                             Select a directory for test discovery. This option can be
                               repeated.
  -o, --select-module=NAME...
                             Select single module for test discovery. This option can
be
                               repeated.
  -p, --select-package=PKG...
                             Select a package for test discovery. This option can be
repeated.
  -c, --select-class=CLASS...
                             Select a class for test discovery. This option can be
repeated.
  -m, --select-method=NAME...
                             Select a method for test discovery. This option can be
repeated.
  -r, --select-resource=RESOURCE...
                             Select a classpath resource for test discovery. This
option can
                               be repeated.
  -i, --select-iteration=PREFIX:VALUE[INDEX(..INDEX)?(,INDEX(..INDEX)?)*]...
                             Select iterations for test discovery via a prefixed
identifier
                               and a list of indexes or index ranges (e.g.
method:com.acme.
                               Foo#m()[1..2] selects the first and second iteration of
the m()
                               method in the com.acme.Foo class). This option can be
```

```
repeated.
      --uid, --select-unique-id=UNIQUE-ID...
                              Select a unique id for test discovery. This option can be
                                repeated.
      --select=PREFIX:VALUE...
                              Select via a prefixed identifier (e.g.
method:com.acme.Foo#m
                                selects the m() method in the com.acme.Foo class). This
option
                                can be repeated.

  For more information on selectors including syntax examples, see
  https://docs.junit.org/6.0.1/user-guide/#running-tests-discovery-selectors

FILTERS

  -n, --include-classname=PATTERN
                              Provide a regular expression to include only classes
whose fully
                                qualified names match. To avoid loading classes
unnecessarily,
                                the default pattern only includes class names that
begin with
                                "Test" or end with "Test" or "Tests". When this option
is
                                repeated, all patterns will be combined using OR
semantics.
                                Default: ^(Test.*|.+[.$]Test.*|.*Tests?)$
  -N, --exclude-classname=PATTERN
                              Provide a regular expression to exclude those classes
whose fully
                                qualified names match. When this option is repeated,
all
                                patterns will be combined using OR semantics.
      --include-package=PKG   Provide a package to be included in the test run. This
option can
                                be repeated.
      --exclude-package=PKG   Provide a package to be excluded from the test run. This
option
                                can be repeated.
      --include-methodname=PATTERN
                              Provide a regular expression to include only methods
whose fully
                                qualified names without parameters match. When this
option is
                                repeated, all patterns will be combined using OR
semantics.
      --exclude-methodname=PATTERN
                              Provide a regular expression to exclude those methods
whose fully
                                qualified names without parameters match. When this
```

```
option is
                              repeated, all patterns will be combined using OR
semantics.
  -t, --include-tag=TAG      Provide a tag or tag expression to include only tests
whose tags
                              match. When this option is repeated, all patterns will
be
                              combined using OR semantics.
  -T, --exclude-tag=TAG      Provide a tag or tag expression to exclude those tests
whose tags
                              match. When this option is repeated, all patterns will
be
                              combined using OR semantics.
  -e, --include-engine=ID    Provide the ID of an engine to be included in the test
run. This
                              option can be repeated.
  -E, --exclude-engine=ID    Provide the ID of an engine to be excluded from the test
run.
                              This option can be repeated.

RUNTIME CONFIGURATION

      -cp, --classpath, --class-path=PATH
                            Provide additional classpath entries -- for example, for
adding
                              engines and their dependencies. This option can be
repeated.
      --config-resource=PATH Set configuration parameters for test discovery and
execution via
                              a classpath resource. This option can be repeated.
      --config=KEY=VALUE     Set a configuration parameter for test discovery and
execution.
                              This option can be repeated.

CONSOLE OUTPUT

      --color-palette=FILE   Specify a path to a properties file to customize ANSI
style of
                              output (not supported by all terminals).
      --single-color         Style test output using only text attributes, no color
(not
                              supported by all terminals).
      --details=MODE         Select an output details mode for when tests are
executed. Use
                              one of: none, summary, flat, tree, verbose, testfeed.
If 'none'
                              is selected, then only the summary and test failures
are shown.
                              Default: tree.
      --details-theme=THEME  Select an output details tree theme for when tests are
executed.
```

```
                              Use one of: ascii, unicode. Default is detected based
on
                              default character encoding.
      --redirect-stdout=FILE Redirect test output to stdout to a file.
      --redirect-stderr=FILE Redirect test output to stderr to a file.

For more information, please refer to the JUnit User Guide at
https://docs.junit.org/6.0.1/user-guide/
```

Executing tests



Exit Code

On successful runs, the `ConsoleLauncher` exits with a status code of `0`. All non-zero codes indicate an error of some sort. For example, status code `1` is returned if any containers or tests failed. If no tests are discovered and the `--fail-if-no-tests` command-line option is supplied, the `ConsoleLauncher` exits with a status code of `2`. Unexpected or invalid user input yields a status code of `3`. An exit code of `-1` indicates an unspecified error condition.

```
Usage: junit execute [OPTIONS]
Execute tests
      [@<filename>...]       One or more argument files containing options.
      --disable-ansi-colors  Disable ANSI colors in output (not supported by all
terminals).
      --disable-banner       Disable print out of the welcome message.
  -h, --help                 Display help information.
      --version              Display version information.

SELECTORS

      --scan-classpath, --scan-class-path[=PATH]
                             Scan all directories on the classpath or explicit
classpath
                               roots. Without arguments, only directories on the
system
                               classpath as well as additional classpath entries
supplied via
                               -cp (directories and JAR files) are scanned. Explicit
classpath
                               roots that are not on the classpath will be silently
ignored.
                               This option can be repeated.
      --scan-modules         Scan all resolved modules for test discovery.
  -u, --select-uri=URI...    Select a URI for test discovery. This option can be
repeated.
  -f, --select-file=FILE...  Select a file for test discovery. The line and column
numbers can
                               be provided as URI query parameters (e.g. foo.txt?
                               line=12&column=34). This option can be repeated.
```

```
  -d, --select-directory=DIR...
                            Select a directory for test discovery. This option can be
                              repeated.
  -o, --select-module=NAME...
                            Select single module for test discovery. This option can
be
                              repeated.
  -p, --select-package=PKG...
                            Select a package for test discovery. This option can be
repeated.
  -c, --select-class=CLASS...
                            Select a class for test discovery. This option can be
repeated.
  -m, --select-method=NAME...
                            Select a method for test discovery. This option can be
repeated.
  -r, --select-resource=RESOURCE...
                            Select a classpath resource for test discovery. This
option can
                              be repeated.
  -i, --select-iteration=PREFIX:VALUE[INDEX(..INDEX)?(,INDEX(..INDEX)?)*]...
                            Select iterations for test discovery via a prefixed
identifier
                              and a list of indexes or index ranges (e.g.
method:com.acme.
                              Foo#m()[1..2] selects the first and second iteration of
the m()
                              method in the com.acme.Foo class). This option can be
repeated.
      --uid, --select-unique-id=UNIQUE-ID...
                            Select a unique id for test discovery. This option can be
                              repeated.
      --select=PREFIX:VALUE...
                            Select via a prefixed identifier (e.g.
method:com.acme.Foo#m
                              selects the m() method in the com.acme.Foo class). This
option
                              can be repeated.

  For more information on selectors including syntax examples, see
  https://docs.junit.org/6.0.1/user-guide/#running-tests-discovery-selectors

FILTERS

  -n, --include-classname=PATTERN
                            Provide a regular expression to include only classes
whose fully
                              qualified names match. To avoid loading classes
unnecessarily,
                              the default pattern only includes class names that
begin with
```

```
                              "Test" or end with "Test" or "Tests". When this option
is
                              repeated, all patterns will be combined using OR
semantics.
                              Default: ^(Test.*|.+[.$]Test.*|.*Tests?)$
  -N, --exclude-classname=PATTERN
                            Provide a regular expression to exclude those classes
whose fully
                              qualified names match. When this option is repeated,
all
                              patterns will be combined using OR semantics.
      --include-package=PKG  Provide a package to be included in the test run. This
option can
                              be repeated.
      --exclude-package=PKG  Provide a package to be excluded from the test run. This
option
                              can be repeated.
      --include-methodname=PATTERN
                            Provide a regular expression to include only methods
whose fully
                              qualified names without parameters match. When this
option is
                              repeated, all patterns will be combined using OR
semantics.
      --exclude-methodname=PATTERN
                            Provide a regular expression to exclude those methods
whose fully
                              qualified names without parameters match. When this
option is
                              repeated, all patterns will be combined using OR
semantics.
  -t, --include-tag=TAG      Provide a tag or tag expression to include only tests
whose tags
                              match. When this option is repeated, all patterns will
be
                              combined using OR semantics.
  -T, --exclude-tag=TAG      Provide a tag or tag expression to exclude those tests
whose tags
                              match. When this option is repeated, all patterns will
be
                              combined using OR semantics.
  -e, --include-engine=ID    Provide the ID of an engine to be included in the test
run. This
                              option can be repeated.
  -E, --exclude-engine=ID    Provide the ID of an engine to be excluded from the test
run.
                              This option can be repeated.

RUNTIME CONFIGURATION

      -cp, --classpath, --class-path=PATH
```

```
                            Provide additional classpath entries -- for example, for
adding
                              engines and their dependencies. This option can be
repeated.
      --config-resource=PATH Set configuration parameters for test discovery and
execution via
                              a classpath resource. This option can be repeated.
      --config=KEY=VALUE     Set a configuration parameter for test discovery and
execution.
                              This option can be repeated.

CONSOLE OUTPUT

      --color-palette=FILE   Specify a path to a properties file to customize ANSI
style of
                              output (not supported by all terminals).
      --single-color         Style test output using only text attributes, no color
(not
                              supported by all terminals).
      --details=MODE         Select an output details mode for when tests are
executed. Use
                              one of: none, summary, flat, tree, verbose, testfeed.
If 'none'
                              is selected, then only the summary and test failures
are shown.
                              Default: tree.
      --details-theme=THEME  Select an output details tree theme for when tests are
executed.
                              Use one of: ascii, unicode. Default is detected based
on
                              default character encoding.
      --redirect-stdout=FILE Redirect test output to stdout to a file.
      --redirect-stderr=FILE Redirect test output to stderr to a file.

REPORTING

      --fail-if-no-tests     Fail and return exit status code 2 if no tests are found.
      --fail-fast            Stops test execution after the first failed test.
      --reports-dir=DIR      Enable report output into a specified local directory
(will be
                              created if it does not exist).

For more information, please refer to the JUnit User Guide at
https://docs.junit.org/6.0.1/user-guide/
```

Listing test engines

```
Usage: junit engines [OPTIONS]
List available test engines
      [@<filename>...]   One or more argument files containing options.
```

```
      --disable-ansi-colors
                          Disable ANSI colors in output (not supported by all
terminals).
      --disable-banner    Disable print out of the welcome message.
  -h, --help              Display help information.
      --version           Display version information.

For more information, please refer to the JUnit User Guide at
https://docs.junit.org/6.0.1/user-guide/
```

Argument Files (@-files)

On some platforms you may run into system limitations on the length of a command line when creating a command line with lots of options or with long arguments.

The `ConsoleLauncher` supports *argument files*, also known as *@-files*. Argument files are files that themselves contain arguments to be passed to the command. When the underlying picocli command line parser encounters an argument beginning with the character `@`, it expands the contents of that file into the argument list.

The arguments within a file can be separated by spaces or newlines. If an argument contains embedded whitespace, the whole argument should be wrapped in double or single quotes — for example, `"-f=My Files/Stuff.java"`.

If the argument file does not exist or cannot be read, the argument will be treated literally and will not be removed. This will likely result in an "unmatched argument" error message. You can troubleshoot such errors by executing the command with the `picocli.trace` system property set to `DEBUG`.

Multiple *@-files* may be specified on the command line. The specified path may be relative to the current directory or absolute.

You can pass a real parameter with an initial `@` character by escaping it with an additional `@` symbol. For example, `@@somearg` will become `@somearg` and will not be subject to expansion.

Redirecting Standard Output/Error to Files

You can redirect the `System.out` (stdout) and `System.err` (stderr) output streams to files using the `--redirect-stdout` and `--redirect-stderr` options:

```
$ java -jar junit-platform-console-standalone-6.0.1.jar <OPTIONS> \
  --redirect-stdout=stdout.txt \
  --redirect-stderr=stderr.txt
```



If the `--redirect-stdout` and `--redirect-stderr` arguments point to the same file, both output streams will be redirected to that file.

The default charset is used for writing to the files.

Color Customization

The colors used in the output of the `ConsoleLauncher` can be customized. The option `--single-color` will apply a built-in monochrome style, while `--color-palette` will accept a properties file to override the ANSI SGR color styling. The properties file below demonstrates the default style:

```
SUCCESSFUL = 32
ABORTED = 33
FAILED = 31
SKIPPED = 35
CONTAINER = 35
TEST = 34
DYNAMIC = 35
REPORTED = 37
```

Discovery Selectors

The JUnit Platform provides a rich set of discovery selectors that can be used to specify which tests should be discovered or executed.

Discovery selectors can be created programmatically using the factory methods in the `DiscoverySelectors` class, specified declaratively via annotations when using the JUnit Platform Suite Engine, via options of the Console Launcher, or generically as strings via their identifiers.

The following discovery selectors are provided out of the box:

Java Type	API	Annotation	Console Launcher	Identifier
`ClasspathResourceSelector`	`selectClasspathResource`	`@SelectClasspathResource`	`--select-resource /foo.csv`	`resource:/foo.csv`
`ClasspathRootSelector`	`selectClasspathRoots`	—	`--scan-classpath bin`	`classpath-root:bin`
`ClassSelector`	`selectClass`	`@SelectClasses`	`--select-class com.acme.Foo`	`class:com.acme.Foo`
`DirectorySelector`	`selectDirectory`	`@SelectDirectories`	`--select-directory foo/bar`	`directory:foo/bar`
`FileSelector`	`selectFile`	`@SelectFile`	`--select-file dir/foo.txt`	`file:dir/foo.txt`
`IterationSelector`	`selectIteration`	`@Select("<identifier>")`	`--select-iteration method=com.acme.Foo#m[1..2]`	`iteration:method:com.acme.Foo#m[1..2]`
`MethodSelector`	`selectMethod`	`@SelectMethod`	`--select-method com.acme.Foo#m`	`method:com.acme.Foo#m`
`ModuleSelector`	`selectModule`	`@SelectModules`	`--select-module com.acme`	`module:com.acme`

Java Type	API	Annotation	Console Launcher	Identifier
NestedClassSelector	selectNestedClass	@Select("<identifier>")	--select <identifier>	nested-class:com.acme.Foo/Bar
NestedMethodSelector	selectNestedMethod	@Select("<identifier>")	--select <identifier>	nested-method:com.acme.Foo/Bar#m
PackageSelector	selectPackage	@SelectPackages	--select-package com.acme.foo	package:com.acme.foo
UniqueIdSelector	selectUniqueId	@Select("<identifier>")	--select-unique-id <identifier>	uid:[engine:Foo]/[segment:Bar]
UriSelector	selectUri	@SelectUris	--select-uri file:///foo.txt	uri:file:///foo.txt

Configuration Parameters

In addition to instructing the platform which test classes and test engines to include, which packages to scan, etc., it is sometimes necessary to provide additional custom configuration parameters that are specific to a particular test engine, listener, or registered extension. For example, the JUnit Jupiter `TestEngine` supports *configuration parameters* for the following use cases.

- Changing the Default Test Instance Lifecycle
- Enabling Automatic Extension Detection
- Deactivating Conditions
- Setting the Default Display Name Generator

Configuration Parameters are text-based key-value pairs that can be supplied to test engines running on the JUnit Platform via one of the following mechanisms.

1. The `configurationParameter()` and `configurationParameters()` methods in `LauncherDiscoveryRequestBuilder` which is used to build a request supplied to the Launcher API. When running tests via one of the tools provided by the JUnit Platform you can specify configuration parameters as follows:
    - Console Launcher: use the `--config` command-line option.
    - Gradle: use the `systemProperty` or `systemProperties` DSL.
    - Maven Surefire provider: use the `configurationParameters` property.
2. The `configurationParametersResources()` method in `LauncherDiscoveryRequestBuilder`. When running tests via the Console Launcher you can specify custom configuration files using the `--config-resource` command-line option.
3. JVM system properties.
4. The JUnit Platform default configuration file: a file named `junit-platform.properties` in the root of the class path that follows the syntax rules for Java `Properties` files.



Configuration parameters are looked up in the exact order defined above. Consequently, configuration parameters supplied directly to the `Launcher` take precedence over those supplied via custom configuration files, system properties, and the default configuration file. Similarly, configuration parameters supplied via system properties take precedence over those supplied via the default configuration file.

Pattern Matching Syntax

This section describes the pattern matching syntax that is applied to the *configuration parameters* used for the following features.

- Deactivating Conditions
- Deactivating a TestExecutionListener
- Stack Trace Pruning
- Filtering Auto-detected Extensions

If the value for the given *configuration parameter* consists solely of an asterisk (`*`), the pattern will match against all candidate classes. Otherwise, the value will be treated as a comma-separated list of patterns where each pattern will be matched against the fully qualified class name (*FQCN*) of each candidate class. Any dot (`.`) in a pattern will match against a dot (`.`) or a dollar sign (`$`) in a FQCN. Any asterisk (`*`) will match against one or more characters in a FQCN. All other characters in a pattern will be matched one-to-one against a FQCN.

Examples:

- `*`: matches all candidate classes.
- `org.junit.*`: matches all candidate classes under the `org.junit` base package and any of its subpackages.
- `*.MyCustomImpl`: matches every candidate class whose simple class name is exactly `MyCustomImpl`.
- `*System*`: matches every candidate class whose FQCN contains `System`.
- `*System*, *Unit*`: matches every candidate class whose FQCN contains `System` or `Unit`.
- `org.example.MyCustomImpl`: matches the candidate class whose FQCN is exactly `org.example.MyCustomImpl`.
- `org.example.MyCustomImpl, org.example.TheirCustomImpl`: matches candidate classes whose FQCN is exactly `org.example.MyCustomImpl` or `org.example.TheirCustomImpl`.

Tags

Tags are a JUnit Platform concept for marking and filtering tests. The programming model for adding tags to containers and tests is defined by the testing framework. For example, in JUnit Jupiter based tests, the `@Tag` annotation (see Tagging and Filtering) should be used. For JUnit 4 based tests, the Vintage engine maps `@Category` annotations to tags (see Categories Support). Other testing frameworks may define their own annotation or other means for users to specify tags.

Syntax Rules for Tags

Regardless how a tag is specified, the JUnit Platform enforces the following rules:

- A tag must not be `null` or *blank*.
- A *stripped* tag must not contain whitespace.
- A *stripped* tag must not contain ISO control characters.
- A *stripped* tag must not contain any of the following *reserved characters*.
  - `,`: *comma*
  - `(`: *left parenthesis*
  - `)`: *right parenthesis*
  - `&`: *ampersand*
  - `|`: *vertical bar*
  - `!`: *exclamation point*



In the above context, "stripped" means that leading and trailing whitespace characters have been removed using `java.lang.String.strip()`.

Tag Expressions

Tag expressions are boolean expressions with the operators `!`, `&` and `|`. In addition, `(` and `)` can be used to adjust for operator precedence.

Two special expressions are supported, `any()` and `none()`, which select all tests *with* any tags at all, and all tests *without* any tags, respectively. These special expressions may be combined with other expressions just like normal tags.

Operators (in descending order of precedence)

Operator	Meaning	Associativity
`!`	not	right
`&`	and	left
`|`	or	left

If you are tagging your tests across multiple dimensions, tag expressions help you to select which tests to execute. When tagging by test type (e.g., *micro*, *integration*, *end-to-end*) and feature (e.g., **product**, **catalog**, **shipping**), the following tag expressions can be useful.

Tag Expression	Selection
`product`	all tests for **product**
`catalog | shipping`	all tests for **catalog** plus all tests for **shipping**
<code>catalog & shipping</code>	all tests for the intersection between **catalog** and **shipping**
<code>product & !end-to-end</code>	all tests for **product**, but not the *end-to-end* tests

Tag Expression	Selection
<code>(micro | integration) & (product | shipping)</code>	all *micro* or *integration* tests for **product** or **shipping**

Capturing Standard Output/Error

The JUnit Platform provides opt-in support for capturing output printed to `System.out` and `System.err`. To enable it, set the `junit.platform.output.capture.stdout` and/or `junit.platform.output.capture.stderr` configuration parameter to `true`. In addition, you may configure the maximum number of buffered bytes to be used per executed test or container using `junit.platform.output.capture.maxBuffer`.

If enabled, the JUnit Platform captures the corresponding output and publishes it as a report entry using the `stdout` or `stderr` keys to all registered `TestExecutionListener` instances immediately before reporting the test or container as finished.

Please note that the captured output will only contain output emitted by the thread that was used to execute a container or test. Any output by other threads will be omitted because particularly when executing tests in parallel it would be impossible to attribute it to a specific test or container.

Using Listeners and Interceptors

The JUnit Platform provides the following listener APIs that allow JUnit, third parties, and custom user code to react to events fired at various points during the discovery and execution of a `TestPlan`.

- `LauncherSessionListener`: receives events when a `LauncherSession` is opened and closed.
- `LauncherInterceptor`: intercepts test discovery and execution in the context of a `LauncherSession`.
- `LauncherDiscoveryListener`: receives events that occur during test discovery.
- `TestExecutionListener`: receives events that occur during test execution.

The `LauncherSessionListener` API is typically implemented by build tools or IDEs and registered automatically for you in order to support some feature of the build tool or IDE.

The `LauncherDiscoveryListener` and `TestExecutionListener` APIs are often implemented in order to produce some form of report or to display a graphical representation of the test plan in an IDE. Such listeners may be implemented and automatically registered by a build tool or IDE, or they may be included in a third-party library – potentially registered for you automatically. You can also implement and register your own listeners.

For details on registering and configuring listeners, see the following sections of this guide.

- Registering a LauncherSessionListener
- Registering a LauncherInterceptor
- Registering a LauncherDiscoveryListener
- Registering a TestExecutionListener

- Configuring a TestExecutionListener
- Deactivating a TestExecutionListener

The JUnit Platform provides the following listeners which you may wish to use with your test suite.

JUnit Platform Reporting

`LegacyXmlReportGeneratingListener` can be used via the Console Launcher or registered manually to generate XML reports compatible with the de facto standard for JUnit 4 based test reports.

`OpenTestReportGeneratingListener` generates an XML report in the event-based format specified by Open Test Reporting. It is auto-registered and can be enabled and configured via Configuration Parameters.

See JUnit Platform Reporting for details.

Flight Recorder Support

`FlightRecordingExecutionListener` and `FlightRecordingDiscoveryListener` that generate Java Flight Recorder events during test discovery and execution.

`LoggingListener`

`TestExecutionListener` for logging informational messages for all events via a `BiConsumer` that consumes `Throwable` and `Supplier<String>`.

`SummaryGeneratingListener`

`TestExecutionListener` that generates a summary of the test execution which can be printed via a `PrintWriter`.

`UniqueIdTrackingListener`

`TestExecutionListener` that that tracks the unique IDs of all tests that were skipped or executed during the execution of the `TestPlan` and generates a file containing the unique IDs once execution of the `TestPlan` has finished.

Flight Recorder Support

The JUnit Platform provides opt-in support for generating Flight Recorder events. JEP 328 describes the Java Flight Recorder (JFR) as follows.

> Flight Recorder records events originating from applications, the JVM, and the OS. Events are stored in a single file that can be attached to bug reports and examined by support engineers, allowing after-the-fact analysis of issues in the period leading up to a problem.

In order to record Flight Recorder events generated while running tests, you need to start flight recording when launching a test suite via the following java command line option.

```
-XX:StartFlightRecording:filename=...
```

Please consult the manual of your build tool for the appropriate commands.

To analyze the recorded events, use the jfr command line tool shipped with recent JDKs or open the recording file with JDK Mission Control.

Stack Trace Pruning

The JUnit Platform provides built-in support for pruning stack traces produced by failing tests. This feature is enabled by default but can be disabled by setting the `junit.platform.stacktrace.pruning.enabled` *configuration parameter* to `false`.

When enabled, all calls from the `org.junit`, `jdk.internal.reflect`, and `sun.reflect` packages are removed from the stack trace, unless the calls occur after the test itself or any of its ancestors. For that reason, calls to `org.junit.jupiter.api.Assertions` or `org.junit.jupiter.api.Assumptions` will never be excluded.

In addition, all elements prior to and including the first call from the JUnit Platform `Launcher` will be removed.

Discovery Issues

Test engines may encounter issues during test discovery. For example, the declaration of a test class or method may be invalid. To avoid such issues from going unnoticed, the JUnit Platform provides a mechanism for test engines to report them with different severity levels:

INFO

Indicates that the engine encountered something that could be potentially problematic, but could also happen due to a valid setup or configuration.

WARNING

Indicates that the engine encountered something that is problematic and might lead to unexpected behavior or will be removed or changed in a future release.

ERROR

Indicates that the engine encountered something that is definitely problematic and will lead to unexpected behavior.

If an engine reports an issue with a severity equal to or higher than a configurable *critical* severity, its tests will not be executed. Instead, the engine will be reported as failed during execution with a `DiscoveryIssueException` listing all critical issues. Non-critical issues will be logged but will not prevent the engine from executing its tests. The `junit.platform.discovery.issue.severity.critical` configuration parameter can be used to set the critical severity level. Currently, the default value is `ERROR` but it may be changed in a future release.



To surface all discovery issues in your project, it is recommended to set the `junit.platform.discovery.issue.severity.critical` configuration parameter to `INFO`.

In addition, registered `LauncherDiscoveryListener` implementations can receive discovery issues via the `issueEncountered()` method. This allows IDEs and build tools to report issues to the user in a more user-friendly way. For example, IDEs may choose to display all issues in a list or table.

Extension Model

In contrast to the competing `Runner`, `TestRule`, and `MethodRule` extension points in JUnit 4, the JUnit Jupiter extension model consists of a single, coherent concept: the `Extension` API. Note, however, that `Extension` itself is just a marker interface.

Registering Extensions

Extensions can be registered *declaratively* via `@ExtendWith`, *programmatically* via `@RegisterExtension`, or *automatically* via Java's `ServiceLoader` mechanism.

Declarative Extension Registration

Developers can register one or more extensions *declaratively* by annotating a test interface, test class, test method, or custom *composed annotation* with `@ExtendWith(…)` and supplying class references for the extensions to register. `@ExtendWith` may also be declared on fields or on parameters in test class constructors, in test methods, and in `@BeforeAll`, `@AfterAll`, `@BeforeEach`, and `@AfterEach` lifecycle methods.

For example, to register a `WebServerExtension` for a particular test method, you would annotate the test method as follows. We assume the `WebServerExtension` starts a local web server and injects the server's URL into parameters annotated with `@WebServerUrl`.

```
@Test
@ExtendWith(WebServerExtension.class)
void getProductList(@WebServerUrl String serverUrl) {
    WebClient webClient = new WebClient();
    // Use WebClient to connect to web server using serverUrl and verify response
    assertEquals(200, webClient.get(serverUrl + "/products").getResponseStatus());
}
```

To register the `WebServerExtension` for all tests in a particular class and its subclasses, you would annotate the test class as follows.

```
@ExtendWith(WebServerExtension.class)
class MyTests {
    // ...
}
```

Multiple extensions can be registered together like this:

```
@ExtendWith({ DatabaseExtension.class, WebServerExtension.class })
class MyFirstTests {
    // ...
}
```

As an alternative, multiple extensions can be registered separately like this:

```
@ExtendWith(DatabaseExtension.class)
@ExtendWith(WebServerExtension.class)
class MySecondTests {
    // ...
}
```



Extension Registration Order

Extensions registered declaratively via `@ExtendWith` at the class level, method level, or parameter level will be executed in the order in which they are declared in the source code. For example, the execution of tests in both `MyFirstTests` and `MySecondTests` will be extended by the `DatabaseExtension` and `WebServerExtension`, **in exactly that order**.

If you wish to combine multiple extensions in a reusable way, you can define a custom *composed annotation* and use `@ExtendWith` as a *meta-annotation* as in the following code listing. Then `@DatabaseAndWebServerExtension` can be used in place of `@ExtendWith({ DatabaseExtension.class, WebServerExtension.class })`.

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@ExtendWith({ DatabaseExtension.class, WebServerExtension.class })
public @interface DatabaseAndWebServerExtension {
}
```

The above examples demonstrate how `@ExtendWith` can be applied at the class level or at the method level; however, for certain use cases it makes sense for an extension to be registered declaratively at the field or parameter level. Consider a `RandomNumberExtension` which generates random numbers that can be injected into a field or via a parameter in a constructor, test method, or lifecycle method. If the extension provides a `@Random` annotation that is meta-annotated with `@ExtendWith(RandomNumberExtension.class)` (see listing below), the extension can be used transparently as in the following `RandomNumberDemo` example.

```
@Target({ ElementType.FIELD, ElementType.PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
@ExtendWith(RandomNumberExtension.class)
public @interface Random {
}
```

```
class RandomNumberDemo {

    // Use static randomNumber0 field anywhere in the test class,
    // including @BeforeAll or @AfterEach lifecycle methods.
    @Random
```

```
    private static Integer randomNumber0;

    // Use randomNumber1 field in test methods and @BeforeEach
    // or @AfterEach lifecycle methods.
    @Random
    private int randomNumber1;

    RandomNumberDemo(@Random int randomNumber2) {
        // Use randomNumber2 in constructor.
    }

    @BeforeEach
    void beforeEach(@Random int randomNumber3) {
        // Use randomNumber3 in @BeforeEach method.
    }

    @Test
    void test(@Random int randomNumber4) {
        // Use randomNumber4 in test method.
    }

}
```

The following code listing provides an example of how one might choose to implement such a `RandomNumberExtension`. This implementation works for the use cases in `RandomNumberDemo`; however, it may not prove robust enough to cover all use cases — for example, the random number generation support is limited to integers; it uses `java.util.Random` instead of `java.security.SecureRandom`; etc. In any case, it is important to note which extension APIs are implemented and for what reasons.

Specifically, `RandomNumberExtension` implements the following extension APIs:

- `BeforeAllCallback`: to support static field injection
- `TestInstancePostProcessor`: to support non-static field injection
- `ParameterResolver`: to support constructor and method injection

```
import static
org.junit.platform.commons.support.AnnotationSupport.findAnnotatedFields;

import java.lang.reflect.Field;
import java.util.function.Predicate;

import org.jspecify.annotations.Nullable;
import org.junit.jupiter.api.extension.BeforeAllCallback;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.api.extension.ParameterContext;
import org.junit.jupiter.api.extension.ParameterResolver;
import org.junit.jupiter.api.extension.TestInstancePostProcessor;
import org.junit.platform.commons.support.ModifierSupport;
```

```
class RandomNumberExtension
        implements BeforeAllCallback, TestInstancePostProcessor, ParameterResolver {

    private final java.util.Random random = new java.util.Random(System.nanoTime());

    /**
     * Inject a random integer into static fields that are annotated with
     * {@code @Random} and can be assigned an integer value.
     */
    @Override
    public void beforeAll(ExtensionContext context) {
        Class<?> testClass = context.getRequiredTestClass();
        injectFields(testClass, null, ModifierSupport::isStatic);
    }

    /**
     * Inject a random integer into non-static fields that are annotated with
     * {@code @Random} and can be assigned an integer value.
     */
    @Override
    public void postProcessTestInstance(Object testInstance, ExtensionContext context)
{
        Class<?> testClass = context.getRequiredTestClass();
        injectFields(testClass, testInstance, ModifierSupport::isNotStatic);
    }

    /**
     * Determine if the parameter is annotated with {@code @Random} and can be
     * assigned an integer value.
     */
    @Override
    public boolean supportsParameter(ParameterContext pc, ExtensionContext ec) {
        return pc.isAnnotated(Random.class) && isInteger(pc.getParameter().getType());
    }

    /**
     * Resolve a random integer.
     */
    @Override
    public Integer resolveParameter(ParameterContext pc, ExtensionContext ec) {
        return this.random.nextInt();
    }

    private void injectFields(Class<?> testClass, @Nullable Object testInstance,
            Predicate<Field> predicate) {

        predicate = predicate.and(field -> isInteger(field.getType()));
        findAnnotatedFields(testClass, Random.class, predicate)
            .forEach(field -> {
                try {
```

```
                    field.setAccessible(true);
                    field.set(testInstance, this.random.nextInt());
                }
                catch (Exception ex) {
                    throw new RuntimeException(ex);
                }
            });
    }

    private static boolean isInteger(Class<?> type) {
        return type == Integer.class || type == int.class;
    }

}
```



Extension Registration Order for `@ExtendWith` on Fields

Extensions registered declaratively via `@ExtendWith` on fields will be ordered relative to `@RegisterExtension` fields and other `@ExtendWith` fields using an algorithm that is deterministic but intentionally nonobvious. However, `@ExtendWith` fields can be ordered using the `@Order` annotation. See the Extension Registration Order tip for `@RegisterExtension` fields for details.



Extension Inheritance

Extensions registered declaratively via `@ExtendWith` on fields in superclasses will be inherited.

See Extension Inheritance for details.



`@ExtendWith` fields may be either `static` or non-static. The documentation on Static Fields and Instance Fields for `@RegisterExtension` fields also applies to `@ExtendWith` fields.

Programmatic Extension Registration

Developers can register extensions *programmatically* by annotating fields in test classes with `@RegisterExtension`.

When an extension is registered *declaratively* via `@ExtendWith`, it can typically only be configured via annotations. In contrast, when an extension is registered via `@RegisterExtension`, it can be configured *programmatically* — for example, in order to pass arguments to the extension's constructor, a static factory method, or a builder API.



Extension Registration Order

By default, extensions registered programmatically via `@RegisterExtension` or declaratively via `@ExtendWith` on fields will be ordered using an algorithm that is deterministic but intentionally nonobvious. This ensures that subsequent runs of a test suite execute extensions in the same order, thereby allowing for repeatable

builds. However, there are times when extensions need to be registered in an explicit order. To achieve that, annotate `@RegisterExtension` fields or `@ExtendWith` fields with `@Order`.

Any `@RegisterExtension` field or `@ExtendWith` field not annotated with `@Order` will be ordered using the *default* order which has a value of `Integer.MAX_VALUE / 2`. This allows `@Order` annotated extension fields to be explicitly ordered before or after non-annotated extension fields. Extensions with an explicit order value less than the default order value will be registered before non-annotated extensions. Similarly, extensions with an explicit order value greater than the default order value will be registered after non-annotated extensions. For example, assigning an extension an explicit order value that is greater than the default order value allows *before* callback extensions to be registered last and *after* callback extensions to be registered first, relative to other programmatically registered extensions.



Extension Inheritance

Extensions registered via `@RegisterExtension` or `@ExtendWith` on fields in superclasses will be inherited.

See Extension Inheritance for details.



`@RegisterExtension` fields must not be `null` (at evaluation time) but may be either `static` or non-static.

Static Fields

If a `@RegisterExtension` field is `static`, the extension will be registered after extensions that are registered at the class level via `@ExtendWith`. Such *static extensions* are not limited in which extension APIs they can implement. Extensions registered via static fields may therefore implement class-level and instance-level extension APIs such as `BeforeAllCallback`, `AfterAllCallback`, `TestInstancePostProcessor`, and `TestInstancePreDestroyCallback` as well as method-level extension APIs such as `BeforeEachCallback`, etc.

In the following example, the `server` field in the test class is initialized programmatically by using a builder pattern supported by the `WebServerExtension`. The configured `WebServerExtension` will be automatically registered as an extension at the class level — for example, in order to start the server before all tests in the class and then stop the server after all tests in the class have completed. In addition, static lifecycle methods annotated with `@BeforeAll` or `@AfterAll` as well as `@BeforeEach`, `@AfterEach`, and `@Test` methods can access the instance of the extension via the `server` field if necessary.

Registering an extension via a static field in Java

```
class WebServerDemo {

    @RegisterExtension
    static WebServerExtension server = WebServerExtension.builder()
        .enableSecurity(false)
```

```
        .build();

    @Test
    void getProductList() {
        WebClient webClient = new WebClient();
        String serverUrl = server.getServerUrl();
        // Use WebClient to connect to web server using serverUrl and verify response
        assertEquals(200, webClient.get(serverUrl + "/products").getResponseStatus());
    }

}
```

Static Fields in Kotlin

The Kotlin programming language does not have the concept of a `static` field. However, the compiler can be instructed to generate a `private static` field using the `@JvmStatic` annotation in Kotlin. If you want the Kotlin compiler to generate a `public static` field, you can use the `@JvmField` annotation instead.

The following example is a version of the `WebServerDemo` from the previous section that has been ported to Kotlin.

Registering an extension via a static field in Kotlin

```
class KotlinWebServerDemo {
    companion object {
        @JvmField
        @RegisterExtension
        val server =
            WebServerExtension
                .builder()
                .enableSecurity(false)
                .build()!!
    }

    @Test
    fun getProductList() {
        // Use WebClient to connect to web server using serverUrl and verify response
        val webClient = WebClient()
        val serverUrl = server.serverUrl
        assertEquals(200, webClient.get("$serverUrl/products").responseStatus)
    }
}
```

Instance Fields

If a `@RegisterExtension` field is non-static (i.e., an instance field), the extension will be registered after the test class has been instantiated and after each registered `TestInstancePostProcessor` has been given a chance to post-process the test instance (potentially injecting the instance of the extension to be used into the annotated field). Thus, if such an *instance extension* implements class-

level or instance-level extension APIs such as `BeforeAllCallback`, `AfterAllCallback`, or `TestInstancePostProcessor`, those APIs will not be honored. Instance extensions will be registered *before* extensions that are registered at the method level via `@ExtendWith`.

In the following example, the `docs` field in the test class is initialized programmatically by invoking a custom `lookUpDocsDir()` method and supplying the result to the static `forPath()` factory method in the `DocumentationExtension`. The configured `DocumentationExtension` will be automatically registered as an extension at the method level. In addition, `@BeforeEach`, `@AfterEach`, and `@Test` methods can access the instance of the extension via the `docs` field if necessary.

An extension registered via an instance field

```
class DocumentationDemo {

    static Path lookUpDocsDir() {
        // return path to docs dir
    }

    @RegisterExtension
    DocumentationExtension docs = DocumentationExtension.forPath(lookUpDocsDir());

    @Test
    void generateDocumentation() {
        // use this.docs ...
    }
}
```

Automatic Extension Registration

In addition to declarative extension registration and programmatic extension registration support using annotations, JUnit Jupiter also supports *global extension registration* via Java's `ServiceLoader` mechanism, allowing third-party extensions to be auto-detected and automatically registered based on what is available in the classpath.

Specifically, a custom extension can be registered by supplying its fully qualified class name in a file named `org.junit.jupiter.api.extension.Extension` within the `/META-INF/services` folder in its enclosing JAR file.

Enabling Automatic Extension Detection

Auto-detection is an advanced feature and is therefore not enabled by default. To enable it, set the `junit.jupiter.extensions.autodetection.enabled` *configuration parameter* to `true`. This can be supplied as a JVM system property, as a *configuration parameter* in the `LauncherDiscoveryRequest` that is passed to the `Launcher`, or via the JUnit Platform configuration file (see Configuration Parameters for details).

For example, to enable auto-detection of extensions, you can start your JVM with the following system property.

```
-Djunit.jupiter.extensions.autodetection.enabled=true
```

When auto-detection is enabled, extensions discovered via the `ServiceLoader` mechanism will be added to the extension registry after JUnit Jupiter's global extensions (e.g., support for `TestInfo`, `TestReporter`, etc.).

Filtering Auto-detected Extensions

The list of auto-detected extensions can be filtered using include and exclude patterns via the following configuration parameters:

`junit.jupiter.extensions.autodetection.include=<patterns>`

Comma-separated list of *include* patterns for auto-detected extensions.

`junit.jupiter.extensions.autodetection.exclude=<patterns>`

Comma-separated list of *exclude* patterns for auto-detected extensions.

Include patterns are applied *before* exclude patterns. If both include and exclude patterns are provided, only extensions that match at least one include pattern and do not match any exclude pattern will be auto-detected.

See Pattern Matching Syntax for details on the pattern syntax.

Extension Inheritance

Registered extensions are inherited within test class hierarchies with top-down semantics. Similarly, extensions registered at the class-level are inherited at the method-level. This applies to all extensions, independent of how they are registered (declaratively or programmatically).

This means that extensions registered declaratively via `@ExtendWith` on a superclass will be registered before extensions registered declaratively via `@ExtendWith` on a subclass.

Similarly, extensions registered programmatically via `@RegisterExtension` or `@ExtendWith` on fields in a superclass will be registered before extensions registered programmatically via `@RegisterExtension` or `@ExtendWith` on fields in a subclass, unless `@Order` is used to alter that behavior (see Extension Registration Order for details).



A specific extension implementation can only be registered once for a given extension context and its parent contexts. Consequently, any attempt to register a duplicate extension implementation will be ignored.

Conditional Test Execution

`ExecutionCondition` defines the `Extension` API for programmatic, *conditional test execution*.

An `ExecutionCondition` is *evaluated* for each container (e.g., a test class) to determine if all the tests it contains should be executed based on the supplied `ExtensionContext`. Similarly, an `ExecutionCondition` is *evaluated* for each test to determine if a given test method should be executed based on the supplied `ExtensionContext`.

When multiple `ExecutionCondition` extensions are registered, a container or test is disabled as soon

as one of the conditions returns *disabled*. Thus, there is no guarantee that a condition is evaluated because another extension might have already caused a container or test to be disabled. In other words, the evaluation works like the short-circuiting boolean OR operator.

See the source code of `DisabledCondition` and `@Disabled` for concrete examples.

Deactivating Conditions

Sometimes it can be useful to run a test suite *without* certain conditions being active. For example, you may wish to run tests even if they are annotated with `@Disabled` in order to see if they are still *broken*. To do this, provide a pattern for the `junit.jupiter.conditions.deactivate` *configuration parameter* to specify which conditions should be deactivated (i.e., not evaluated) for the current test run. The pattern can be supplied as a JVM system property, as a *configuration parameter* in the `LauncherDiscoveryRequest` that is passed to the `Launcher`, or via the JUnit Platform configuration file (see Configuration Parameters for details).

For example, to deactivate JUnit's `@Disabled` condition, you can start your JVM with the following system property.

```
-Djunit.jupiter.conditions.deactivate=org.junit.*DisabledCondition
```

Pattern Matching Syntax

Refer to Pattern Matching Syntax for details.

Test Instance Pre-construct Callback

`TestInstancePreConstructCallback` defines the API for `Extensions` that wish to be invoked *prior* to test instances being constructed (by a constructor call or via `TestInstanceFactory`).

This extension provides a symmetric call to `TestInstancePreDestroyCallback` and is useful in combination with other extensions to prepare constructor parameters or keeping track of test instances and their lifecycle.



Accessing the test-scoped `ExtensionContext`

You may override the `getTestInstantiationExtensionContextScope(⋯)` method to return `TEST_METHOD` to make test-specific data available to your extension implementation or if you want to keep state on the test method level.

Test Instance Factories

`TestInstanceFactory` defines the API for `Extensions` that wish to *create* test class instances.

Common use cases include acquiring the test instance from a dependency injection framework or invoking a static factory method to create the test class instance.

If no `TestInstanceFactory` is registered, the framework will invoke the *sole* constructor for the test class to instantiate it, potentially resolving constructor arguments via registered `ParameterResolver` extensions.

Extensions that implement `TestInstanceFactory` can be registered on test interfaces, top-level test classes, or `@Nested` test classes.



Registering multiple extensions that implement `TestInstanceFactory` for any single class will result in an exception being thrown for all tests in that class, in any subclass, and in any nested class. Note that any `TestInstanceFactory` registered in a superclass or *enclosing* class (i.e., in the case of a `@Nested` test class) is *inherited*. It is the user's responsibility to ensure that only a single `TestInstanceFactory` is registered for any specific test class.



Accessing the test-scoped `ExtensionContext`

You may override the `getTestInstantiationExtensionContextScope(…)` method to return `TEST_METHOD` to make test-specific data available to your extension implementation or if you want to keep state on the test method level.

Test Instance Post-processing

`TestInstancePostProcessor` defines the API for `Extensions` that wish to *post process* test instances.

Common use cases include injecting dependencies into the test instance, invoking custom initialization methods on the test instance, etc.

For a concrete example, consult the source code for the `MockitoExtension` and the `SpringExtension`.



Accessing the test-scoped `ExtensionContext`

You may override the `getTestInstantiationExtensionContextScope(…)` method to return `TEST_METHOD` to make test-specific data available to your extension implementation or if you want to keep state on the test method level.

Test Instance Pre-destroy Callback

`TestInstancePreDestroyCallback` defines the API for `Extensions` that wish to process test instances *after* they have been used in tests and *before* they are destroyed.

Common use cases include cleaning dependencies that have been injected into the test instance, invoking custom de-initialization methods on the test instance, etc.

Parameter Resolution

`ParameterResolver` defines the `Extension` API for dynamically resolving parameters at runtime.

If a *test class* constructor, *test method*, or *lifecycle method* (see Definitions) declares a parameter, the parameter must be *resolved* at runtime by a `ParameterResolver`. A `ParameterResolver` can either be built-in (see `TestInfoParameterResolver`) or registered by the user. Generally speaking, parameters may be resolved by *name*, *type*, *annotation*, or any combination thereof.

If you wish to implement a custom `ParameterResolver` that resolves parameters based solely on the

type of the parameter, you may find it convenient to extend the `TypeBasedParameterResolver` which serves as a generic adapter for such use cases.

For concrete examples, consult the source code for `CustomTypeParameterResolver`, `CustomAnnotationParameterResolver`, and `MapOfListsTypeBasedParameterResolver`.



Due to a bug in the byte code generated by `javac` on JDK versions prior to JDK 9, looking up annotations on parameters directly via the core `java.lang.reflect.Parameter` API will always fail for *inner class* constructors (e.g., a constructor in a `@Nested` test class).

The `ParameterContext` API supplied to `ParameterResolver` implementations therefore includes the following convenience methods for correctly looking up annotations on parameters. Extension authors are strongly encouraged to use these methods instead of those provided in `java.lang.reflect.Parameter` in order to avoid this bug in the JDK.

- `boolean isAnnotated(Class<? extends Annotation> annotationType)`
- `Optional<A> findAnnotation(Class<A> annotationType)`
- `List<A> findRepeatableAnnotations(Class<A> annotationType)`



Accessing the test-scoped `ExtensionContext`

You may override the `getTestInstantiationExtensionContextScope(⋯)` method to return `TEST_METHOD` to support injecting test specific data into constructor parameters of the test class instance. Doing so causes a test-specific `ExtensionContext` to be used while resolving constructor parameters, unless the test instance lifecycle is set to `PER_CLASS`.



Parameter resolution for methods called from extensions

Other extensions can also leverage registered `ParameterResolvers` for method and constructor invocations, using the `ExecutableInvoker` available via the `getExecutableInvoker()` method in the `ExtensionContext`.

Parameter Conflicts

If multiple implementations of `ParameterResolver` that support the same type are registered for a test, a `ParameterResolutionException` will be thrown, with a message to indicate that competing resolvers have been discovered. See the following example:

Conflicting parameter resolution due to multiple resolvers claiming support for integers

```
public class ParameterResolverConflictDemo {

    @Test
    @ExtendWith({ FirstIntegerResolver.class, SecondIntegerResolver.class })
    void testInt(int i) {
        // Test will not run due to ParameterResolutionException
```

```
        assertEquals(1, i);
    }

    static class FirstIntegerResolver implements ParameterResolver {

        @Override
        public boolean supportsParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
            return parameterContext.getParameter().getType() == int.class;
        }

        @Override
        public Object resolveParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
            return 1;
        }
    }

    static class SecondIntegerResolver implements ParameterResolver {

        @Override
        public boolean supportsParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
            return parameterContext.getParameter().getType() == int.class;
        }

        @Override
        public Object resolveParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
            return 2;
        }
    }
}
```

If the conflicting `ParameterResolver` implementations are applied to different test methods as shown in the following example, no conflict occurs.

Fine-grained registration to avoid conflict

```
public class ParameterResolverNoConflictDemo {

    @Test
    @ExtendWith(FirstIntegerResolver.class)
    void firstResolution(int i) {
        assertEquals(1, i);
    }

    @Test
    @ExtendWith(SecondIntegerResolver.class)
    void secondResolution(int i) {
```

```
        assertEquals(2, i);
    }

    static class FirstIntegerResolver implements ParameterResolver {

        @Override
        public boolean supportsParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
            return parameterContext.getParameter().getType() == int.class;
        }

        @Override
        public Object resolveParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
            return 1;
        }
    }

    static class SecondIntegerResolver implements ParameterResolver {

        @Override
        public boolean supportsParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
            return parameterContext.getParameter().getType() == int.class;
        }

        @Override
        public Object resolveParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
            return 2;
        }
    }
}
```

If the conflicting `ParameterResolver` implementations need to be applied to the same test method, you can implement a custom type or custom annotation as illustrated by `CustomTypeParameterResolver` and `CustomAnnotationParameterResolver`, respectively.

Custom type to resolve duplicate types

```
public class ParameterResolverCustomTypeDemo {

    @Test
    @ExtendWith({ FirstIntegerResolver.class, SecondIntegerResolver.class })
    void testInt(Integer i, WrappedInteger wrappedInteger) {
        assertEquals(1, i);
        assertEquals(2, wrappedInteger.value);
    }

    static class FirstIntegerResolver implements ParameterResolver {
```

```
        @Override
        public boolean supportsParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
            return parameterContext.getParameter().getType().equals(Integer.class);
        }

        @Override
        public Object resolveParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
            return 1;
        }
    }

    static class SecondIntegerResolver implements ParameterResolver {

        @Override
        public boolean supportsParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
            return parameterContext.getParameter().getType().equals(WrappedInteger
.class);
        }

        @Override
        public Object resolveParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
            return new WrappedInteger(2);
        }
    }

    static class WrappedInteger {

        private final int value;

        WrappedInteger(int value) {
            this.value = value;
        }

    }
}
```

A custom annotation makes the duplicate type distinguishable from its counterpart:

Custom annotation to resolve duplicate types

```
public class ParameterResolverCustomAnnotationDemo {

    @Test
    void testInt(@FirstInteger Integer first, @SecondInteger Integer second) {
        assertEquals(1, first);
        assertEquals(2, second);
```

```
    }

    @Target(ElementType.PARAMETER)
    @Retention(RetentionPolicy.RUNTIME)
    @ExtendWith(FirstInteger.Extension.class)
    public @interface FirstInteger {

        class Extension implements ParameterResolver {

            @Override
            public boolean supportsParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
                return parameterContext.getParameter().getType().equals(Integer.class)
                        && !parameterContext.isAnnotated(SecondInteger.class);
            }

            @Override
            public Object resolveParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
                return 1;
            }
        }
    }

    @Target(ElementType.PARAMETER)
    @Retention(RetentionPolicy.RUNTIME)
    @ExtendWith(SecondInteger.Extension.class)
    public @interface SecondInteger {

        class Extension implements ParameterResolver {

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

            @Override
            public Object resolveParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
                return 2;
            }
        }
    }
}
```

JUnit includes some built-in parameter resolvers that can cause conflicts if a resolver attempts to claim their supported types. For example, `TestInfo` provides metadata about tests. See Dependency Injection for Constructors and Methods for details. Third-party frameworks such as Spring may also define parameter resolvers. Apply one of the techniques in this section to resolve any conflicts.

Parameterized tests are another potential source of conflict. Ensure that tests annotated with `@ParameterizedTest` are not also annotated with `@Test` and see Consuming Arguments for more details.

Test Result Processing

`TestWatcher` defines the API for extensions that wish to process the results of *test method* executions. Specifically, a `TestWatcher` will be invoked with contextual information for the following events.

- `testDisabled`: invoked after a disabled *test method* has been skipped
- `testSuccessful`: invoked after a *test method* has completed successfully
- `testAborted`: invoked after a *test method* has been aborted
- `testFailed`: invoked after a *test method* has failed



In contrast to the definition of "test method" presented in Definitions, in this context *test method* refers to any `@Test` method or `@TestTemplate` method (for example, a `@RepeatedTest` or `@ParameterizedTest`).

Extensions implementing this interface can be registered at the class level, instance level, or method level. When registered at the class level, a `TestWatcher` will be invoked for any contained *test method* including those in `@Nested` classes. When registered at the method level, a `TestWatcher` will only be invoked for the *test method* for which it was registered.



If a `TestWatcher` is registered via a non-static (instance) field – for example, using `@RegisterExtension` – and the test class is configured with `@TestInstance(Lifecycle.PER_METHOD)` semantics (which is the default lifecycle mode), the `TestWatcher` will **not** be invoked with events for `@TestTemplate` methods (for example, `@RepeatedTest` or `@ParameterizedTest`).

To ensure that a `TestWatcher` is invoked for all *test methods* in a given class, it is therefore recommended that the `TestWatcher` be registered at the class level with `@ExtendWith` or via a `static` field with `@RegisterExtension` or `@ExtendWith`.

If there is a failure at the class level — for example, an exception thrown by a `@BeforeAll` method — no test results will be reported. Similarly, if the test class is disabled via an `ExecutionCondition` — for example, `@Disabled` — no test results will be reported.

In contrast to other Extension APIs, a `TestWatcher` is not permitted to adversely influence the execution of tests. Consequently, any exception thrown by a method in the `TestWatcher` API will be logged at `WARNING` level and will not be allowed to propagate or fail test execution.



Any instances of `ExtensionContext.Store.CloseableResource` stored in the `Store` of the provided `ExtensionContext` will be closed *before* methods in the `TestWatcher` API are invoked (see Keeping State in Extensions). You can use the parent context's `Store` to work with such resources.

Test Lifecycle Callbacks

The following interfaces define the APIs for extending tests at various points in the test execution lifecycle. Consult the following sections for examples and the Javadoc for each of these interfaces in the `org.junit.jupiter.api.extension` package for further details.

- `BeforeAllCallback`
  - `BeforeClassTemplateInvocationCallback` (only applicable for class templates)
    - `BeforeEachCallback`
      - `BeforeTestExecutionCallback`
      - `AfterTestExecutionCallback`
    - `AfterEachCallback`
  - `AfterClassTemplateInvocationCallback` (only applicable for class templates)
- `AfterAllCallback`



Implementing Multiple Extension APIs

Extension developers may choose to implement any number of these interfaces within a single extension. Consult the source code of the `SpringExtension` for a concrete example.

Before and After Test Execution Callbacks

`BeforeTestExecutionCallback` and `AfterTestExecutionCallback` define the APIs for `Extensions` that wish to add behavior that will be executed *immediately before* and *immediately after* a test method is executed, respectively. As such, these callbacks are well suited for timing, tracing, and similar use cases. If you need to implement callbacks that are invoked *around* `@BeforeEach` and `@AfterEach` methods, implement `BeforeEachCallback` and `AfterEachCallback` instead.

The following example shows how to use these callbacks to calculate and log the execution time of a test method. `TimingExtension` implements both `BeforeTestExecutionCallback` and `AfterTestExecutionCallback` in order to time and log the test execution.

An extension that times and logs the execution of test methods

```
import java.lang.reflect.Method;
import java.util.logging.Logger;

import org.junit.jupiter.api.extension.AfterTestExecutionCallback;
import org.junit.jupiter.api.extension.BeforeTestExecutionCallback;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.api.extension.ExtensionContext.Namespace;
import org.junit.jupiter.api.extension.ExtensionContext.Store;

public class TimingExtension implements BeforeTestExecutionCallback,
AfterTestExecutionCallback {
```

```
    private static final Logger logger = Logger.getLogger(TimingExtension.class
.getName());

    private static final String START_TIME = "start time";

    @Override
    public void beforeTestExecution(ExtensionContext context) {
        getStore(context).put(START_TIME, System.currentTimeMillis());
    }

    @Override
    public void afterTestExecution(ExtensionContext context) {
        Method testMethod = context.getRequiredTestMethod();
        long startTime = getStore(context).remove(START_TIME, long.class);
        long duration = System.currentTimeMillis() - startTime;

        logger.info(() ->
            "Method [%s] took %s ms.".formatted(testMethod.getName(), duration));
    }

    private Store getStore(ExtensionContext context) {
        return context.getStore(Namespace.create(getClass(), context
.getRequiredTestMethod()));
    }

}
```

Since the `TimingExtensionTests` class registers the `TimingExtension` via `@ExtendWith`, its tests will have this timing applied when they execute.

A test class that uses the example TimingExtension

```
@ExtendWith(TimingExtension.class)
class TimingExtensionTests {

    @Test
    void sleep20ms() throws Exception {
        Thread.sleep(20);
    }

    @Test
    void sleep50ms() throws Exception {
        Thread.sleep(50);
    }

}
```

The following is an example of the logging produced when `TimingExtensionTests` is run.

```
INFO: Method [sleep20ms] took 24 ms.
INFO: Method [sleep50ms] took 53 ms.
```

Exception Handling

Exceptions thrown during the test execution may be intercepted and handled accordingly before propagating further, so that certain actions like error logging or resource releasing may be defined in specialized `Extensions`. JUnit Jupiter offers API for `Extensions` that wish to handle exceptions thrown during `@Test` methods via `TestExecutionExceptionHandler` and for those thrown during one of test lifecycle methods (`@BeforeAll`, `@BeforeEach`, `@AfterEach` and `@AfterAll`) via `LifecycleMethodExecutionExceptionHandler`.

The following example shows an extension which will swallow all instances of `IOException` but rethrow any other type of exception.

An exception handling extension that filters IOExceptions in test execution

```
public class IgnoreIOExceptionExtension implements TestExecutionExceptionHandler {

    @Override
    public void handleTestExecutionException(ExtensionContext context, Throwable
throwable)
            throws Throwable {

        if (throwable instanceof IOException) {
            return;
        }
        throw throwable;
    }
}
```

Another example shows how to record the state of an application under test exactly at the point of unexpected exception being thrown during setup and cleanup. Note that unlike relying on lifecycle callbacks, which may or may not be executed depending on the test status, this solution guarantees execution immediately after failing `@BeforeAll`, `@BeforeEach`, `@AfterEach` or `@AfterAll`.

An exception handling extension that records application state on error

```
class RecordStateOnErrorExtension implements LifecycleMethodExecutionExceptionHandler
{

    @Override
    public void handleBeforeAllMethodExecutionException(ExtensionContext context,
Throwable ex)
            throws Throwable {
        memoryDumpForFurtherInvestigation("Failure recorded during class setup");
        throw ex;
    }
```

```
    @Override
    public void handleBeforeEachMethodExecutionException(ExtensionContext context,
Throwable ex)
            throws Throwable {
        memoryDumpForFurtherInvestigation("Failure recorded during test setup");
        throw ex;
    }

    @Override
    public void handleAfterEachMethodExecutionException(ExtensionContext context,
Throwable ex)
            throws Throwable {
        memoryDumpForFurtherInvestigation("Failure recorded during test cleanup");
        throw ex;
    }

    @Override
    public void handleAfterAllMethodExecutionException(ExtensionContext context,
Throwable ex)
            throws Throwable {
        memoryDumpForFurtherInvestigation("Failure recorded during class cleanup");
        throw ex;
    }
}
```

Multiple execution exception handlers may be invoked for the same lifecycle method in order of declaration. If one of the handlers swallows the handled exception, subsequent ones will not be executed, and no failure will be propagated to JUnit engine, as if the exception was never thrown. Handlers may also choose to rethrow the exception or throw a different one, potentially wrapping the original.

Extensions implementing `LifecycleMethodExecutionExceptionHandler` that wish to handle exceptions thrown during `@BeforeAll` or `@AfterAll` need to be registered on a class level, while handlers for `BeforeEach` and `AfterEach` may be also registered for individual test methods.

Registering multiple exception handling extensions

```
// Register handlers for @Test, @BeforeEach, @AfterEach as well as @BeforeAll and
@AfterAll
@ExtendWith(ThirdExecutedHandler.class)
class MultipleHandlersTestCase {

    // Register handlers for @Test, @BeforeEach, @AfterEach only
    @ExtendWith(SecondExecutedHandler.class)
    @ExtendWith(FirstExecutedHandler.class)
    @Test
    void testMethod() {
    }
```

```
}
```

Pre-Interrupt Callback

`PreInterruptCallback` defines the API for `Extensions` that wish to react on timeouts before the `Thread.interrupt()` is called.

Please refer to Debugging Timeouts for additional information.

Intercepting Invocations

`InvocationInterceptor` defines the API for `Extensions` that wish to intercept calls to test code.

The following example shows an extension that executes all test methods in Swing's Event Dispatch Thread.

An extension that executes tests in a user-defined thread

```
public class SwingEdtInterceptor implements InvocationInterceptor {

    @Override
    public void interceptTestMethod(Invocation<Void> invocation,
            ReflectiveInvocationContext<Method> invocationContext,
            ExtensionContext extensionContext) throws Throwable {

        AtomicReference<Throwable> throwable = new AtomicReference<>();

        SwingUtilities.invokeAndWait(() -> {
            try {
                invocation.proceed();
            }
            catch (Throwable t) {
                throwable.set(t);
            }
        });
        Throwable t = throwable.get();
        if (t != null) {
            throw t;
        }
    }
}
```



Accessing the test-scoped `ExtensionContext`

You may override the `getTestInstantiationExtensionContextScope(…)` method to return `TEST_METHOD` to make test-specific data available to your extension implementation of `interceptTestClassConstructor` or if you want to keep state on the test method level.

Providing Invocation Contexts for Class Templates

A `@ClassTemplate` class can only be executed when at least one `ClassTemplateInvocationContextProvider` is registered. Each such provider is responsible for providing a `Stream` of `ClassTemplateInvocationContext` instances. Each context may specify a custom display name and a list of additional extensions that will only be used for the next invocation of the `@ClassTemplate`.

The following example shows how to write a class template as well as how to register and implement a `ClassTemplateInvocationContextProvider`.

A class template with accompanying extension

```
@ClassTemplate
@ExtendWith(ClassTemplateDemo.MyClassTemplateInvocationContextProvider.class)
class ClassTemplateDemo {

    static final List<String> WELL_KNOWN_FRUITS
        = List.of("apple", "banana", "lemon");

    private String fruit;

    @Test
    void notNull() {
        assertNotNull(fruit);
    }

    @Test
    void wellKnown() {
        assertTrue(WELL_KNOWN_FRUITS.contains(fruit));
    }

    public class MyClassTemplateInvocationContextProvider
            implements ClassTemplateInvocationContextProvider {

        @Override
        public boolean supportsClassTemplate(ExtensionContext context) {
            return true;
        }

        @Override
        public Stream<ClassTemplateInvocationContext>
                provideClassTemplateInvocationContexts(ExtensionContext context) {

            return Stream.of(invocationContext("apple"), invocationContext("banana"));
        }

        private ClassTemplateInvocationContext invocationContext(String parameter) {
            return new ClassTemplateInvocationContext() {
                @Override
```

```
                public String getDisplayName(int invocationIndex) {
                    return parameter;
                }

                @Override
                public List<Extension> getAdditionalExtensions() {
                    return List.of(new TestInstancePostProcessor() {
                        @Override
                        public void postProcessTestInstance(
                                Object testInstance, ExtensionContext context) {
                            ((ClassTemplateDemo) testInstance).fruit = parameter;
                        }
                    });
                }
            };
        }
    }
}
```

In this example, the class template will be invoked twice, meaning all test methods in the class template will be executed twice. The display names of the invocations will be `apple` and `banana` as specified by the invocation context. Each invocation registers a custom `TestInstancePostProcessor` which is used to inject a value into a field. The output when using the `ConsoleLauncher` is as follows.

```
└── ClassTemplateDemo ✔
   ├── apple ✔
   │   ├── notNull() ✔
   │   └── wellKnown() ✔
   └── banana ✔
      ├── notNull() ✔
      └── wellKnown() ✔
```

The `ClassTemplateInvocationContextProvider` extension API is primarily intended for implementing different kinds of tests that rely on repetitive invocation of *all* test methods in a test class albeit in different contexts — for example, with different parameters, by preparing the test class instance differently, or multiple times without modifying the context. Please refer to the implementations of Parameterized Classes which uses this extension point to provide its functionality.

Providing Invocation Contexts for Test Templates

A `@TestTemplate` method can only be executed when at least one `TestTemplateInvocationContextProvider` is registered. Each such provider is responsible for providing a `Stream` of `TestTemplateInvocationContext` instances. Each context may specify a custom display name and a list of additional extensions that will only be used for the next invocation of the `@TestTemplate` method.

The following example shows how to write a test template as well as how to register and implement a `TestTemplateInvocationContextProvider`.

A test template with accompanying extension

```
final List<String> fruits = Arrays.asList("apple", "banana", "lemon");

@TestTemplate
@ExtendWith(MyTestTemplateInvocationContextProvider.class)
void testTemplate(String fruit) {
    assertTrue(fruits.contains(fruit));
}

public class MyTestTemplateInvocationContextProvider
        implements TestTemplateInvocationContextProvider {

    @Override
    public boolean supportsTestTemplate(ExtensionContext context) {
        return true;
    }

    @Override
    public Stream<TestTemplateInvocationContext>
provideTestTemplateInvocationContexts(
            ExtensionContext context) {

        return Stream.of(invocationContext("apple"), invocationContext("banana"));
    }

    private TestTemplateInvocationContext invocationContext(String parameter) {
        return new TestTemplateInvocationContext() {
            @Override
            public String getDisplayName(int invocationIndex) {
                return parameter;
            }

            @Override
            public List<Extension> getAdditionalExtensions() {
                return List.of(new ParameterResolver() {
                    @Override
                    public boolean supportsParameter(ParameterContext
parameterContext,
                            ExtensionContext extensionContext) {
                        return parameterContext.getParameter().getType().equals(
String.class);
                    }

                    @Override
                    public Object resolveParameter(ParameterContext parameterContext,
                            ExtensionContext extensionContext) {
                        return parameter;
                    }
                });
            }
```

```
        };
    }
}
```

In this example, the test template will be invoked twice. The display names of the invocations will be `apple` and `banana` as specified by the invocation context. Each invocation registers a custom `ParameterResolver` which is used to resolve the method parameter. The output when using the `ConsoleLauncher` is as follows.

```
└── testTemplate(String) ✔
    ├── apple ✔
    └── banana ✔
```

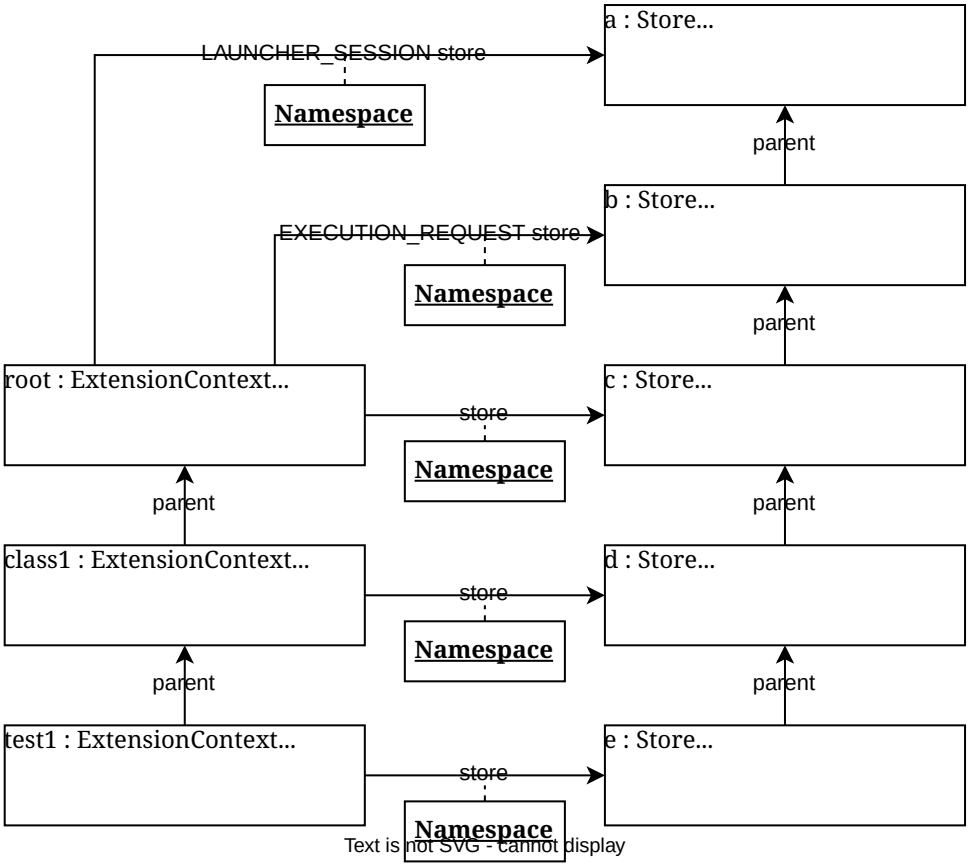
The `TestTemplateInvocationContextProvider` extension API is primarily intended for implementing different kinds of tests that rely on repetitive invocation of a test-like method albeit in different contexts — for example, with different parameters, by preparing the test class instance differently, or multiple times without modifying the context. Please refer to the implementations of Repeated Tests or Parameterized Tests which use this extension point to provide their functionality.

Keeping State in Extensions

Usually, an extension is instantiated only once. So the question becomes relevant: How do you keep the state from one invocation of an extension to the next? The `ExtensionContext` API provides a `Store` exactly for this purpose. Extensions may put values into a store for later retrieval.



See the `TimingExtension` for an example of using the `Store` with a method-level scope.



The `ExtensionContext` and `Store` hierarchy

As illustrated by the diagram above, stores are hierarchical in nature. When looking up a value, if no value is stored in the current `ExtensionContext` for the supplied key, the stores of the context's ancestors will be queried for a value with the same key in the `Namespace` used to create this store. The root `ExtensionContext` represents the engine level so its `Store` may be used to store or cache values that are used by multiple test classes or extension. The `StoreScope` enum allows to go beyond even that and access the stores on the level of the current `LauncherExecutionRequest` or `LauncherSession` which can be used to share data across test engines or inject data from a registered `LauncherSessionListener`, respectively. Please consult the Javadoc of `ExtensionContext`, `Store`, and `StoreScope` for details.



Resource management via `AutoCloseable`

An extension context store is bound to its extension context lifecycle. When an extension context lifecycle ends it closes its associated store.

All stored values that are instances of `AutoCloseable` are notified by an invocation of their `close()` method in the inverse order they were added in (unless the `junit.jupiter.extensions.store.close.autocloseable.enabled` configuration parameter is set to `false`).

Versions prior to 5.13 only supported `CloseableResource`, which is deprecated but still available for backward compatibility.

An example implementation of `AutoCloseable` is shown below, using an `HttpServer` resource.

`HttpServer` *resource implementing* `AutoCloseable`

```
class HttpServerResource implements AutoCloseable {

    private final HttpServer httpServer;

    HttpServerResource(int port) throws IOException {
        InetAddress loopbackAddress = InetAddress.getLoopbackAddress();
        this.httpServer = HttpServer.create(new InetSocketAddress(loopbackAddress,
port), 0);
    }

    HttpServer getHttpServer() {
        return httpServer;
    }

    void start() {
        // Example handler
        httpServer.createContext("/example", exchange -> {
            String body = "This is a test";
            exchange.sendResponseHeaders(200, body.length());
            try (OutputStream os = exchange.getResponseBody()) {
                os.write(body.getBytes(UTF_8));
            }
        });
        httpServer.setExecutor(null);
        httpServer.start();
    }

    @Override
    public void close() {
        httpServer.stop(0);
    }
}
```

This resource can then be stored in the desired `ExtensionContext`. It may be stored at class or method level, if desired, but this may add unnecessary overhead for this type of resource. For this example it might be prudent to store it at root level and instantiate it lazily to ensure it's only created once per test run and reused across different test classes and methods.

Lazily storing in root context with `Store.computeIfAbsent`

```
public class HttpServerExtension implements ParameterResolver {

    @Override
    public boolean supportsParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
        return HttpServer.class.equals(parameterContext.getParameter().getType());
    }
```

```
    @Override
    public Object resolveParameter(ParameterContext parameterContext, ExtensionContext
extensionContext) {

        ExtensionContext rootContext = extensionContext.getRoot();
        ExtensionContext.Store store = rootContext.getStore(Namespace.GLOBAL);
        Class<HttpServerResource> key = HttpServerResource.class;
        HttpServerResource resource = store.computeIfAbsent(key, __ -> {
            try {
                HttpServerResource serverResource = new HttpServerResource(0);
                serverResource.start();
                return serverResource;
            }
            catch (IOException e) {
                throw new UncheckedIOException("Failed to create HttpServerResource",
e);
            }
        }, HttpServerResource.class);
        return resource.getHttpServer();
    }
}
```

A test case using the `HttpServerExtension`

```
@ExtendWith(HttpServerExtension.class)
public class HttpServerDemo {

    @Test
    void httpCall(HttpServer server) throws Exception {
        String hostName = server.getAddress().getHostName();
        int port = server.getAddress().getPort();
        String rawUrl = "http://%s:%d/example".formatted(hostName, port);
        URL requestUrl = URI.create(rawUrl).toURL();

        String responseBody = sendRequest(requestUrl);

        assertEquals("This is a test", responseBody);
    }

    private static String sendRequest(URL url) throws IOException {
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        int contentLength = connection.getContentLength();
        try (InputStream response = url.openStream()) {
            byte[] content = new byte[contentLength];
            assertEquals(contentLength, response.read(content));
            return new String(content, UTF_8);
        }
    }
}
```

Migration Note for Resource Cleanup

The framework automatically closes resources stored in the `ExtensionContext.Store` that implement `AutoCloseable`. In versions prior to 5.13, only resources implementing `Store.CloseableResource` were automatically closed.

If you're developing an extension that needs to support both JUnit Jupiter 5.13+ and earlier versions and your extension stores resources that need to be cleaned up, you should implement both interfaces:



```
public class MyResource implements Store.CloseableResource,
AutoCloseable {
    @Override
    public void close() throws Exception {
        // Resource cleanup code
    }
}
```

This ensures that your resource will be properly closed regardless of which JUnit Jupiter version is being used.

Supported Utilities in Extensions

The `junit-platform-commons` artifact provides *maintained* utilities for working with annotations, classes, reflection, classpath scanning, and conversion tasks. These utilities can be found in the `org.junit.platform.commons.support` and its subpackages. `TestEngine` and `Extension` authors are encouraged to use these supported utilities in order to align with the behavior of the JUnit Platform and JUnit Jupiter.

Annotation Support

`AnnotationSupport` provides static utility methods that operate on annotated elements (e.g., packages, annotations, classes, interfaces, constructors, methods, and fields). These include methods to check whether an element is annotated or meta-annotated with a particular annotation, to search for specific annotations, and to find annotated methods and fields in a class or interface. Some of these methods search on implemented interfaces and within class hierarchies to find annotations. Consult the Javadoc for `AnnotationSupport` for further details.



The `isAnnotated()` methods do not find repeatable annotations. To check for repeatable annotations, use one of the `findRepeatableAnnotations()` methods and verify that the returned list is not empty.



See also: Field and Method Search Semantics

Class Support

`ClassSupport` provides static utility methods for working with classes (i.e., instances of

`java.lang.Class`). Consult the Javadoc for `ClassSupport` for further details.

Reflection Support

`ReflectionSupport` provides static utility methods that augment the standard JDK reflection and class-loading mechanisms. These include methods to scan the classpath in search of classes matching specified predicates, to load and create new instances of a class, and to find and invoke methods. Some of these methods traverse class hierarchies to locate matching methods. Consult the Javadoc for `ReflectionSupport` for further details.



See also: Field and Method Search Semantics

Modifier Support

`ModifierSupport` provides static utility methods for working with member and class modifiers — for example, to determine if a member is declared as `public`, `private`, `abstract`, `static`, etc. Consult the Javadoc for `ModifierSupport` for further details.

Conversion Support

`ConversionSupport` (in the `org.junit.platform.commons.support.conversion` package) provides support for converting from strings to primitive types and their corresponding wrapper types, date and time types from the `java.time` package, and some additional common Java types such as `File`, `BigDecimal`, `BigInteger`, `Currency`, `Locale`, `URI`, `URL`, `UUID`, etc. Consult the Javadoc for `ConversionSupport` for further details.

Field and Method Search Semantics

Various methods in `AnnotationSupport` and `ReflectionSupport` use search algorithms that traverse type hierarchies to locate matching fields and methods – for example, `AnnotationSupport.findAnnotatedFields(…)`, `ReflectionSupport.findMethods(…)`, etc.

The field and method search algorithms adhere to standard Java semantics regarding whether a given field or method is visible or overridden according to the rules of the Java language.

Relative Execution Order of User Code and Extensions

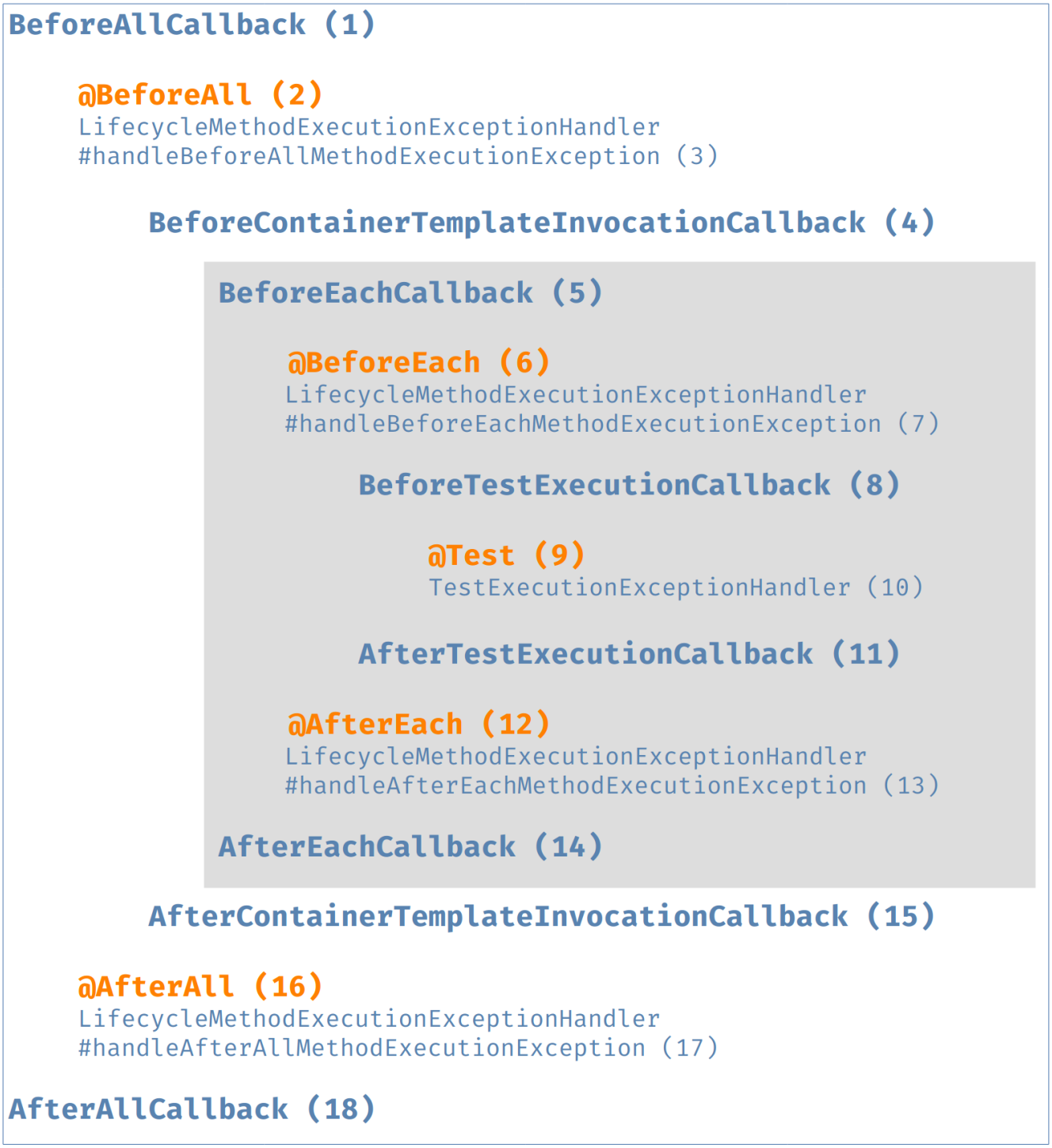
When executing a test class that contains one or more test methods, a number of extension callbacks are called in addition to the user-supplied test and lifecycle methods.



See also: Test Execution Order

User and Extension Code

The following diagram illustrates the relative order of user-supplied code and extension code. User-supplied test and lifecycle methods are shown in orange, with callback code implemented by extensions shown in blue. The grey box denotes the execution of a single test method and will be repeated for every test method in the test class.



User code and extension code

The following table further explains the sixteen steps in the User code and extension code diagram.

1. **interface** `org.junit.jupiter.api.extension.BeforeAllCallback`
   extension code executed before all tests of the container are executed
2. **annotation** `org.junit.jupiter.api.BeforeAll`
   user code executed before all tests of the container are executed
3. **interface** `org.junit.jupiter.api.extension.LifecycleMethodExecutionExceptionHandler#handleBeforeAllMethodExecutionException`
   extension code for handling exceptions thrown from `@BeforeAll` methods

4. **interface** `org.junit.jupiter.api.extension.BeforeClassTemplateInvocationCallback`
   extension code executed before each class template invocation is executed (only applicable if the test class is a class template)
5. **interface** `org.junit.jupiter.api.extension.BeforeEachCallback`
   extension code executed before each test is executed
6. **annotation** `org.junit.jupiter.api.BeforeEach`
   user code executed before each test is executed
7. **interface** `org.junit.jupiter.api.extension.LifecycleMethodExecutionExceptionHandler#handleBeforeEachMethodExecutionException`
   extension code for handling exceptions thrown from `@BeforeEach` methods
8. **interface** `org.junit.jupiter.api.extension.BeforeTestExecutionCallback`
   extension code executed immediately before a test is executed
9. **annotation** `org.junit.jupiter.api.Test`
   user code of the actual test method
10. **interface** `org.junit.jupiter.api.extension.TestExecutionExceptionHandler`
    extension code for handling exceptions thrown during a test
11. **interface** `org.junit.jupiter.api.extension.AfterTestExecutionCallback`
    extension code executed immediately after test execution and its corresponding exception handlers
12. **annotation** `org.junit.jupiter.api.AfterEach`
    user code executed after each test is executed
13. **interface** `org.junit.jupiter.api.extension.LifecycleMethodExecutionExceptionHandler#handleAfterEachMethodExecutionException`
    extension code for handling exceptions thrown from `@AfterEach` methods
14. **interface** `org.junit.jupiter.api.extension.AfterEachCallback`
    extension code executed after each test is executed
15. **interface** `org.junit.jupiter.api.extension.AfterClassTemplateInvocationCallback`
    extension code executed after each class template invocation is executed (only applicable if the test class is a class template)
16. **annotation** `org.junit.jupiter.api.AfterAll`
    user code executed after all tests of the container are executed
17. **interface** `org.junit.jupiter.api.extension.LifecycleMethodExecutionExceptionHandler#handleAfterAllMethodExecutionException`
    extension code for handling exceptions thrown from `@AfterAll` methods
18. **interface** `org.junit.jupiter.api.extension.AfterAllCallback`
    extension code executed after all tests of the container are executed

In the simplest case only the actual test method will be executed (step 9); all other steps are optional depending on the presence of user code or extension support for the corresponding lifecycle callback. For further details on the various lifecycle callbacks please consult the respective Javadoc for each annotation and extension.

All invocations of user code methods in the above table can additionally be intercepted by

implementing `InvocationInterceptor`.

Wrapping Behavior of Callbacks

JUnit Jupiter always guarantees *wrapping* behavior for multiple registered extensions that implement lifecycle callbacks such as `BeforeAllCallback`, `AfterAllCallback`, `BeforeClassTemplateInvocationCallback`, `AfterClassTemplateInvocationCallback`, `BeforeEachCallback`, `AfterEachCallback`, `BeforeTestExecutionCallback`, and `AfterTestExecutionCallback`.

That means that, given two extensions `Extension1` and `Extension2` with `Extension1` registered before `Extension2`, any "before" callbacks implemented by `Extension1` are guaranteed to execute **before** any "before" callbacks implemented by `Extension2`. Similarly, given the two same two extensions registered in the same order, any "after" callbacks implemented by `Extension1` are guaranteed to execute **after** any "after" callbacks implemented by `Extension2`. `Extension1` is therefore said to *wrap* `Extension2`.

JUnit Jupiter also guarantees *wrapping* behavior within class and interface hierarchies for user-supplied *lifecycle methods* (see Definitions).

- `@BeforeAll` methods are inherited from superclasses as long as they are not *overridden*. Furthermore, `@BeforeAll` methods from superclasses will be executed **before** `@BeforeAll` methods in subclasses.
    - Similarly, `@BeforeAll` methods declared in an interface are inherited as long as they are not *overridden*, and `@BeforeAll` methods from an interface will be executed **before** `@BeforeAll` methods in the class that implements the interface.
- `@AfterAll` methods are inherited from superclasses as long as they are not *overridden*. Furthermore, `@AfterAll` methods from superclasses will be executed **after** `@AfterAll` methods in subclasses.
    - Similarly, `@AfterAll` methods declared in an interface are inherited as long as they are not *overridden*, and `@AfterAll` methods from an interface will be executed **after** `@AfterAll` methods in the class that implements the interface.
- `@BeforeEach` methods are inherited from superclasses as long as they are not *overridden*. Furthermore, `@BeforeEach` methods from superclasses will be executed **before** `@BeforeEach` methods in subclasses.
    - Similarly, `@BeforeEach` methods declared as interface default methods are inherited as long as they are not *overridden*, and `@BeforeEach` default methods will be executed **before** `@BeforeEach` methods in the class that implements the interface.
- `@AfterEach` methods are inherited from superclasses as long as they are not *overridden*. Furthermore, `@AfterEach` methods from superclasses will be executed **after** `@AfterEach` methods in subclasses.
    - Similarly, `@AfterEach` methods declared as interface default methods are inherited as long as they are not *overridden*, and `@AfterEach` default methods will be executed **after** `@AfterEach` methods in the class that implements the interface.

The following examples demonstrate this behavior. Please note that the examples do not actually do anything realistic. Instead, they mimic common scenarios for testing interactions with the

database. All methods imported statically from the `Logger` class log contextual information in order to help us better understand the execution order of user-supplied callback methods and callback methods in extensions.

Extension1

```
import static example.callbacks.Logger.afterEachCallback;
import static example.callbacks.Logger.beforeEachCallback;

import org.junit.jupiter.api.extension.AfterEachCallback;
import org.junit.jupiter.api.extension.BeforeEachCallback;
import org.junit.jupiter.api.extension.ExtensionContext;

public class Extension1 implements BeforeEachCallback, AfterEachCallback {

    @Override
    public void beforeEach(ExtensionContext context) {
        beforeEachCallback(this);
    }

    @Override
    public void afterEach(ExtensionContext context) {
        afterEachCallback(this);
    }

}
```

Extension2

```
import static example.callbacks.Logger.afterEachCallback;
import static example.callbacks.Logger.beforeEachCallback;

import org.junit.jupiter.api.extension.AfterEachCallback;
import org.junit.jupiter.api.extension.BeforeEachCallback;
import org.junit.jupiter.api.extension.ExtensionContext;

public class Extension2 implements BeforeEachCallback, AfterEachCallback {

    @Override
    public void beforeEach(ExtensionContext context) {
        beforeEachCallback(this);
    }

    @Override
    public void afterEach(ExtensionContext context) {
        afterEachCallback(this);
    }

}
```

AbstractDatabaseTests

```
import static example.callbacks.Logger.afterAllMethod;
import static example.callbacks.Logger.afterEachMethod;
import static example.callbacks.Logger.beforeAllMethod;
import static example.callbacks.Logger.beforeEachMethod;

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

/**
 * Abstract base class for tests that use the database.
 */
abstract class AbstractDatabaseTests {

    @BeforeAll
    static void createDatabase() {
        beforeAllMethod(AbstractDatabaseTests.class.getSimpleName() +
".createDatabase()");
    }

    @BeforeEach
    void connectToDatabase() {
        beforeEachMethod(AbstractDatabaseTests.class.getSimpleName() +
".connectToDatabase()");
    }

    @AfterEach
    void disconnectFromDatabase() {
        afterEachMethod(AbstractDatabaseTests.class.getSimpleName() +
".disconnectFromDatabase()");
    }

    @AfterAll
    static void destroyDatabase() {
        afterAllMethod(AbstractDatabaseTests.class.getSimpleName() +
".destroyDatabase()");
    }

}
```

DatabaseTestsDemo

```
import static example.callbacks.Logger.afterEachMethod;
import static example.callbacks.Logger.beforeAllMethod;
import static example.callbacks.Logger.beforeEachMethod;
import static example.callbacks.Logger.testMethod;
```

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

/**
 * Extension of {@link AbstractDatabaseTests} that inserts test data
 * into the database (after the database connection has been opened)
 * and deletes test data (before the database connection is closed).
 */
@ExtendWith({ Extension1.class, Extension2.class })
class DatabaseTestsDemo extends AbstractDatabaseTests {

    @BeforeAll
    static void beforeAll() {
        beforeAllMethod(DatabaseTestsDemo.class.getSimpleName() + ".beforeAll()");
    }

    @BeforeEach
    void insertTestDataIntoDatabase() {
        beforeEachMethod(getClass().getSimpleName() +
".insertTestDataIntoDatabase()");
    }

    @Test
    void testDatabaseFunctionality() {
        testMethod(getClass().getSimpleName() + ".testDatabaseFunctionality()");
    }

    @AfterEach
    void deleteTestDataFromDatabase() {
        afterEachMethod(getClass().getSimpleName() + ".deleteTestDataFromDatabase()");
    }

    @AfterAll
    static void afterAll() {
        beforeAllMethod(DatabaseTestsDemo.class.getSimpleName() + ".afterAll()");
    }

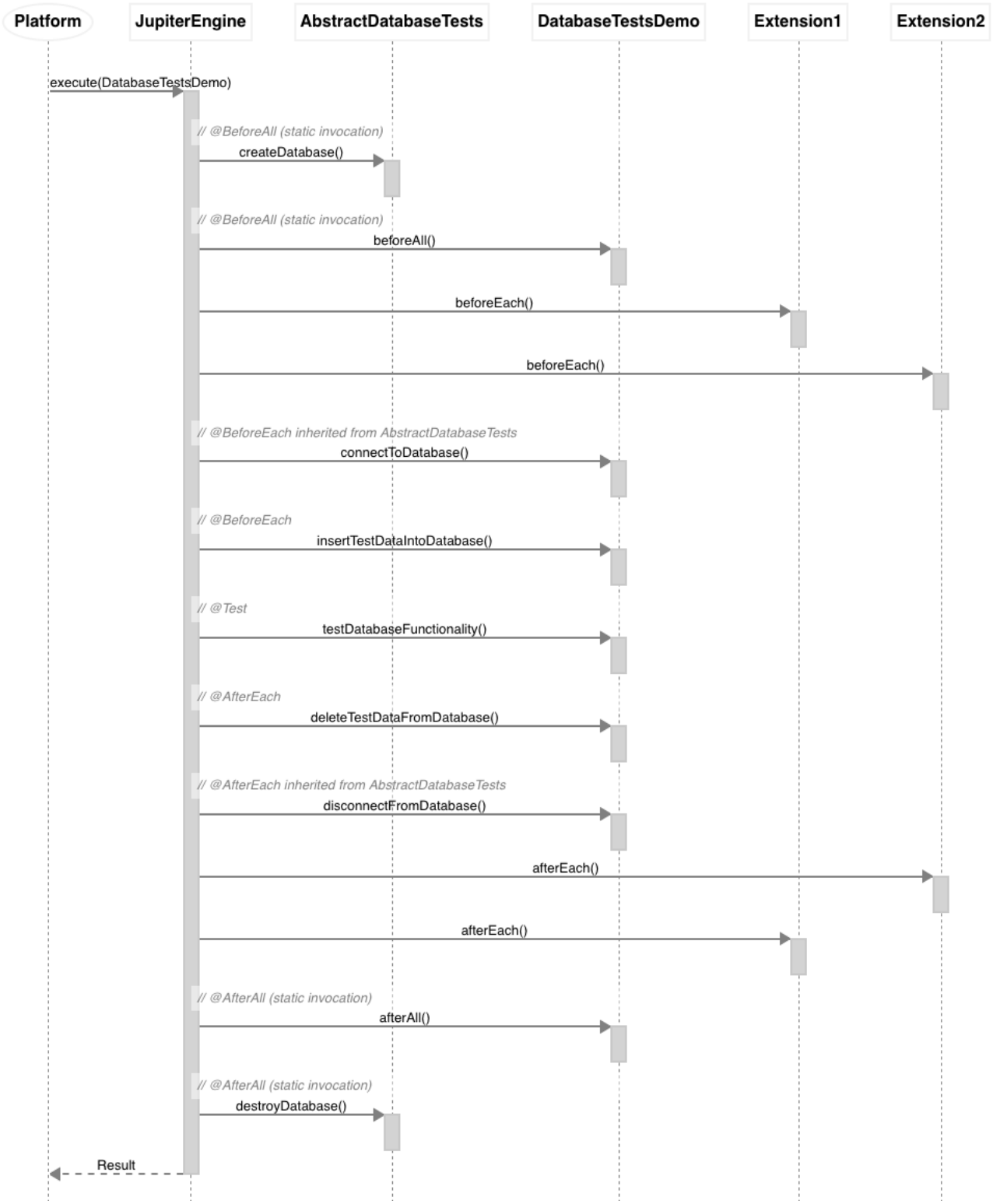
}
```

When the `DatabaseTestsDemo` test class is executed, the following is logged.

```
@BeforeAll AbstractDatabaseTests.createDatabase()
@BeforeAll DatabaseTestsDemo.beforeAll()
  Extension1.beforeEach()
  Extension2.beforeEach()
    @BeforeEach AbstractDatabaseTests.connectToDatabase()
```

```
    @BeforeEach DatabaseTestsDemo.insertTestDataIntoDatabase()
      @Test DatabaseTestsDemo.testDatabaseFunctionality()
    @AfterEach DatabaseTestsDemo.deleteTestDataFromDatabase()
    @AfterEach AbstractDatabaseTests.disconnectFromDatabase()
  Extension2.afterEach()
  Extension1.afterEach()
@BeforeAll DatabaseTestsDemo.afterAll()
@AfterAll AbstractDatabaseTests.destroyDatabase()
```

The following sequence diagram helps to shed further light on what actually goes on within the `JupiterTestEngine` when the `DatabaseTestsDemo` test class is executed.



DatabaseTestsDemo

JUnit Jupiter does **not** guarantee the execution order of multiple lifecycle methods that are declared within a *single* test class or test interface. It may at times appear that JUnit Jupiter invokes such methods in alphabetical order. However, that is not precisely true. The ordering is analogous to the ordering for `@Test` methods within a single test class.



Lifecycle methods that are declared within a *single* test class or test interface will be ordered using an algorithm that is deterministic but intentionally non-obvious. This ensures that subsequent runs of a test suite execute lifecycle methods in the

> same order, thereby allowing for repeatable builds.

In addition, JUnit Jupiter does **not** support *wrapping* behavior for multiple lifecycle methods declared within a single test class or test interface.

The following example demonstrates this behavior. Specifically, the lifecycle method configuration is *broken* due to the order in which the locally declared lifecycle methods are executed.

- Test data is inserted *before* the database connection has been opened, which results in a failure to connect to the database.
- The database connection is closed *before* deleting the test data, which results in a failure to connect to the database.

BrokenLifecycleMethodConfigDemo

```
import static example.callbacks.Logger.afterEachMethod;
import static example.callbacks.Logger.beforeEachMethod;
import static example.callbacks.Logger.testMethod;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

/**
 * Example of "broken" lifecycle method configuration.
 *
 * <p>Test data is inserted before the database connection has been opened.
 *
 * <p>Database connection is closed before deleting test data.
 */
@ExtendWith({ Extension1.class, Extension2.class })
class BrokenLifecycleMethodConfigDemo {

    @BeforeEach
    void connectToDatabase() {
        beforeEachMethod(getClass().getSimpleName() + ".connectToDatabase()");
    }

    @BeforeEach
    void insertTestDataIntoDatabase() {
        beforeEachMethod(getClass().getSimpleName() +
".insertTestDataIntoDatabase()");
    }

    @Test
    void testDatabaseFunctionality() {
        testMethod(getClass().getSimpleName() + ".testDatabaseFunctionality()");
    }
```

```
    @AfterEach
    void deleteTestDataFromDatabase() {
        afterEachMethod(getClass().getSimpleName() + ".deleteTestDataFromDatabase()");
    }

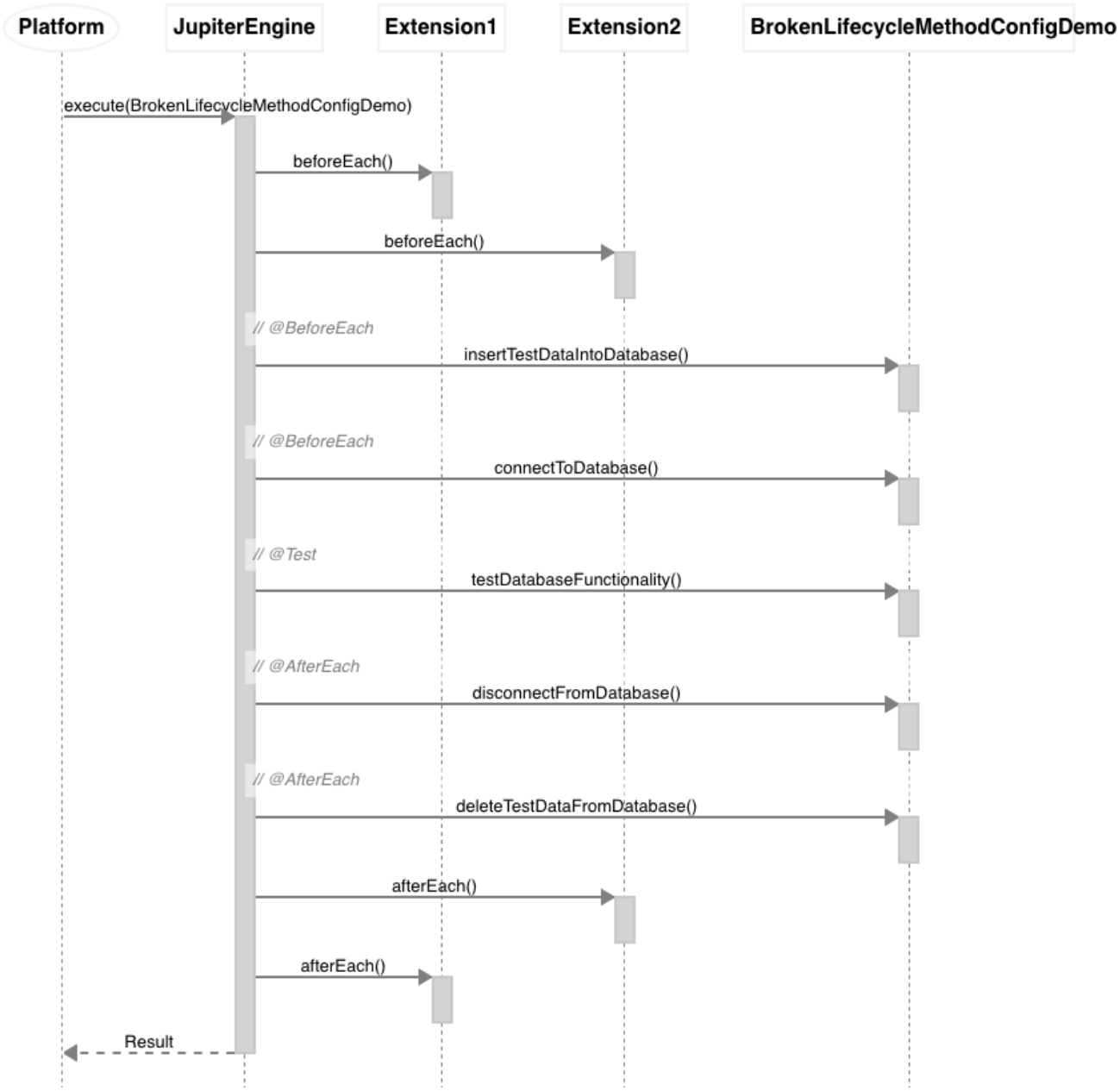
    @AfterEach
    void disconnectFromDatabase() {
        afterEachMethod(getClass().getSimpleName() + ".disconnectFromDatabase()");
    }

}
```

When the `BrokenLifecycleMethodConfigDemo` test class is executed, the following is logged.

```
Extension1.beforeEach()
Extension2.beforeEach()
  @BeforeEach BrokenLifecycleMethodConfigDemo.insertTestDataIntoDatabase()
  @BeforeEach BrokenLifecycleMethodConfigDemo.connectToDatabase()
    @Test BrokenLifecycleMethodConfigDemo.testDatabaseFunctionality()
  @AfterEach BrokenLifecycleMethodConfigDemo.disconnectFromDatabase()
  @AfterEach BrokenLifecycleMethodConfigDemo.deleteTestDataFromDatabase()
Extension2.afterEach()
Extension1.afterEach()
```

The following sequence diagram helps to shed further light on what actually goes on within the `JupiterTestEngine` when the `BrokenLifecycleMethodConfigDemo` test class is executed.



BrokenLifecycleMethodConfigDemo



Due to the aforementioned behavior, the JUnit Team recommends that developers declare at most one of each type of *lifecycle method* (see Definitions) per test class or test interface unless there are no dependencies between such lifecycle methods.

Advanced Topics

JUnit Platform Reporting

The `junit-platform-reporting` artifact contains `TestExecutionListener` implementations that generate XML test reports in two flavors: Open Test Reporting and legacy.



The module also contains other `TestExecutionListener` implementations that can be used to build custom reporting. See Using Listeners and Interceptors for details.

Output Directory

The JUnit Platform provides an `OutputDirectoryCreator` via `EngineDiscoveryRequest` and `TestPlan` to registered test engines and listeners, respectively. Its root directory can be configured via the following configuration parameter:

`junit.platform.reporting.output.dir=<path>`

Configure the output directory for reporting. By default, `build` is used if a Gradle build script is found, and `target` if a Maven POM is found; otherwise, the current working directory is used.

To create a unique output directory per test run, you can use the `{uniqueNumber}` placeholder in the path. For example, `reports/junit-{uniqueNumber}` will create directories like `reports/junit-8803697269315188212`. This can be useful when using Gradle's or Maven's parallel execution capabilities which create multiple JVM forks that run concurrently.

Open Test Reporting

`OpenTestReportGeneratingListener` writes an XML report for the entire execution in the event-based format specified by Open Test Reporting which supports all features of the JUnit Platform such as hierarchical test structures, display names, tags, etc.

The listener is auto-registered and can be configured via the following configuration parameters:

`junit.platform.reporting.open.xml.enabled=true|false`

Enable/disable writing the report; defaults to `false`.

`junit.platform.reporting.open.xml.git.enabled=true|false`

Enable/disable including information about the Git repository (see Git extension schema of open-test-reporting); defaults to `false`.

If enabled, the listener creates an XML report file named `open-test-report.xml` in the configured output directory.

If output capturing is enabled, the captured output written to `System.out` and `System.err` will be included in the report as well.



The Open Test Reporting CLI Tool can be used to convert from the event-based

format to the hierarchical format which is more human-readable.

Gradle

For Gradle, writing Open Test Reporting compatible XML reports can be enabled and configured via system properties. The following samples configure its output directory to be the same directory Gradle uses for its own XML reports. A `CommandLineArgumentProvider` is used to keep the tasks relocatable across different machines which is important when using Gradle's Build Cache.

Groovy DSL

```
dependencies {
    testRuntimeOnly("org.junit.platform:junit-platform-reporting:6.0.1")
}
tasks.withType(Test).configureEach {
    def outputDir = reports.junitXml.outputLocation
    jvmArgumentProviders << ({
        [
            "-Djunit.platform.reporting.open.xml.enabled=true",
            "-
Djunit.platform.reporting.output.dir=${outputDir.get().asFile.absolutePath}"
        ]
    } as CommandLineArgumentProvider)
}
```

Kotlin DSL

```
dependencies {
    testRuntimeOnly("org.junit.platform:junit-platform-reporting:6.0.1")
}
tasks.withType<Test>().configureEach {
    val outputDir = reports.junitXml.outputLocation
    jvmArgumentProviders += CommandLineArgumentProvider {
        listOf(
            "-Djunit.platform.reporting.open.xml.enabled=true",
            "-
Djunit.platform.reporting.output.dir=${outputDir.get().asFile.absolutePath}"
        )
    }
}
```

Maven

For Maven Surefire/Failsafe, you can enable Open Test Reporting output and configure the resulting XML files to be written to the same directory Surefire/Failsafe uses for its own XML reports as follows:

```
<project>
    <!-- ... -->
```

```
    <dependencies>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-reporting</artifactId>
            <version>6.0.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.5.4</version>
                <configuration>
                    <properties>
                        <configurationParameters>
                            junit.platform.reporting.open.xml.enabled = true
                            junit.platform.reporting.output.dir = target/surefire-
reports
                        </configurationParameters>
                    </properties>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <!-- ... -->
</project>
```

Console Launcher

When using the Console Launcher, you can enable Open Test Reporting output by setting the configuration parameters via `--config`:

```
$ java -jar junit-platform-console-standalone-6.0.1.jar <OPTIONS> \
  --config=junit.platform.reporting.open.xml.enabled=true \
  --config=junit.platform.reporting.output.dir=reports
```

Configuration parameters can also be set in a custom properties file supplied as a classpath resource via the `--config-resource` option:

```
$ java -jar junit-platform-console-standalone-6.0.1.jar <OPTIONS> \
  --config-resource=configuration.properties
```

Legacy XML format

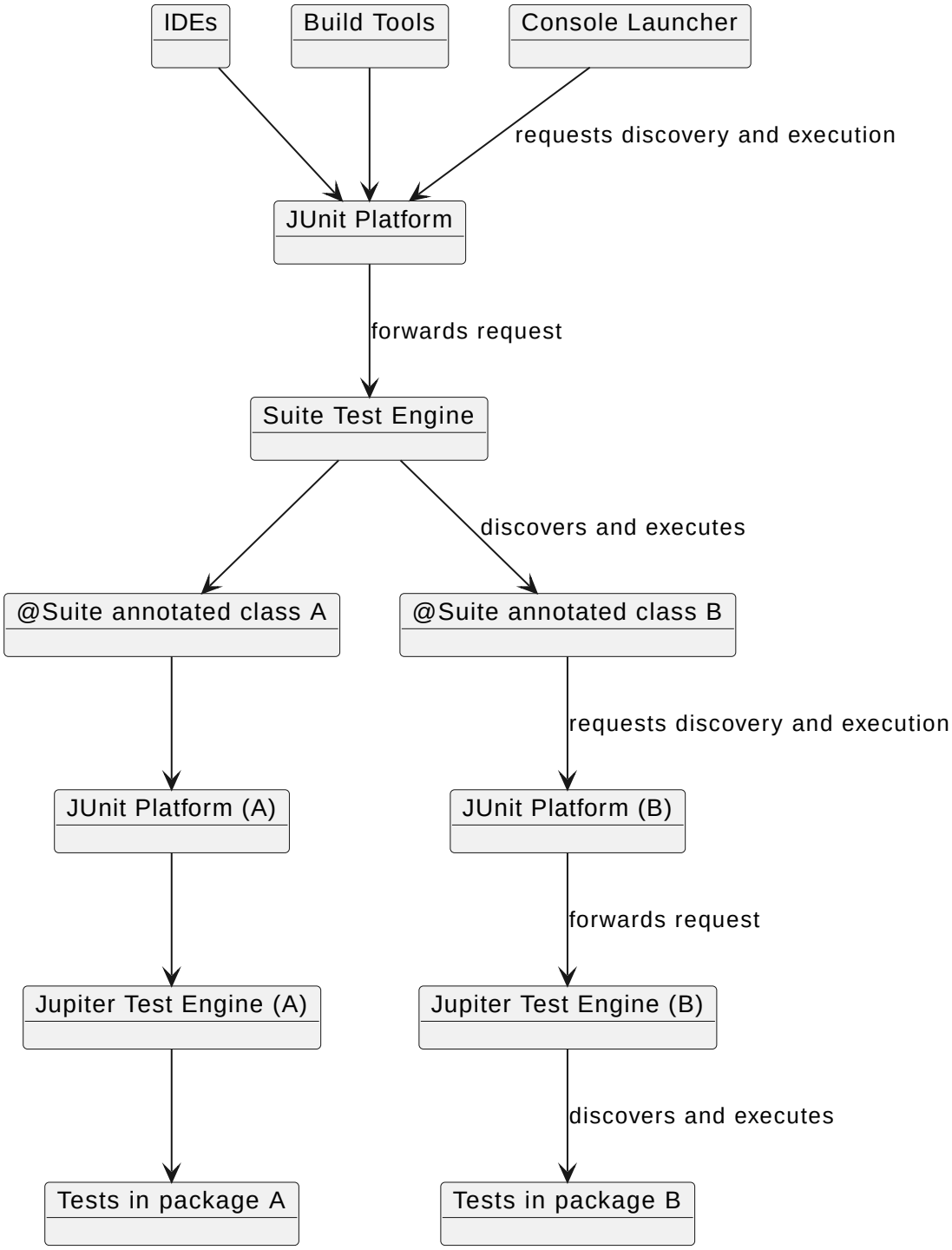
`LegacyXmlReportGeneratingListener` generates a separate XML report for each root in the `TestPlan`. Note that the generated XML format is compatible with the de facto standard for JUnit 4 based test

reports that was made popular by the Ant build system.

The `LegacyXmlReportGeneratingListener` is used by the Console Launcher as well.

JUnit Platform Suite Engine

The Suite Engine supports the declarative selection and execution of tests from *any* test engine on the JUnit Platform using the JUnit Platform Launcher API.



Setup

In addition to the `junit-platform-suite-api` and `junit-platform-suite-engine` artifacts, you need *at*

least one other test engine and its dependencies on the classpath. See Dependency Metadata for details regarding group IDs, artifact IDs, and versions.

Required Dependencies

- `junit-platform-suite-api` in *test* scope: artifact containing annotations needed to configure a test suite
- `junit-platform-suite-engine` in *test runtime* scope: implementation of the `TestEngine` API for declarative test suites



Both of the required dependencies are aggregated in the `junit-platform-suite` artifact which can be declared in *test* scope instead of declaring explicit dependencies on `junit-platform-suite-api` and `junit-platform-suite-engine`.

Transitive Dependencies

- `junit-platform-launcher` in *test* scope
- `junit-platform-engine` in *test* scope
- `junit-platform-commons` in *test* scope
- `opentest4j` in *test* scope

@Suite Example

Annotate a class with `@Suite` to have it marked as a test suite on the JUnit Platform. As seen in the following example, selector and filter annotations can be used to control the contents of the suite.

```
import org.junit.platform.suite.api.IncludeClassNamePatterns;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.platform.suite.api.Suite;
import org.junit.platform.suite.api.SuiteDisplayName;

@Suite
@SuiteDisplayName("JUnit Platform Suite Demo")
@SelectPackages("example")
@IncludeClassNamePatterns(".*Tests")
class SuiteDemo {
}
```



Additional Configuration Options

There are numerous configuration options for discovering and filtering tests in a test suite. Please consult the Javadoc of the `org.junit.platform.suite.api` package for a full list of supported annotations and further details.

@BeforeSuite and @AfterSuite

`@BeforeSuite` and `@AfterSuite` annotations can be used on methods inside a `@Suite`-annotated class.

They will be executed before and after all tests of the test suite, respectively.

```
@Suite
@SelectPackages("example")
class BeforeAndAfterSuiteDemo {

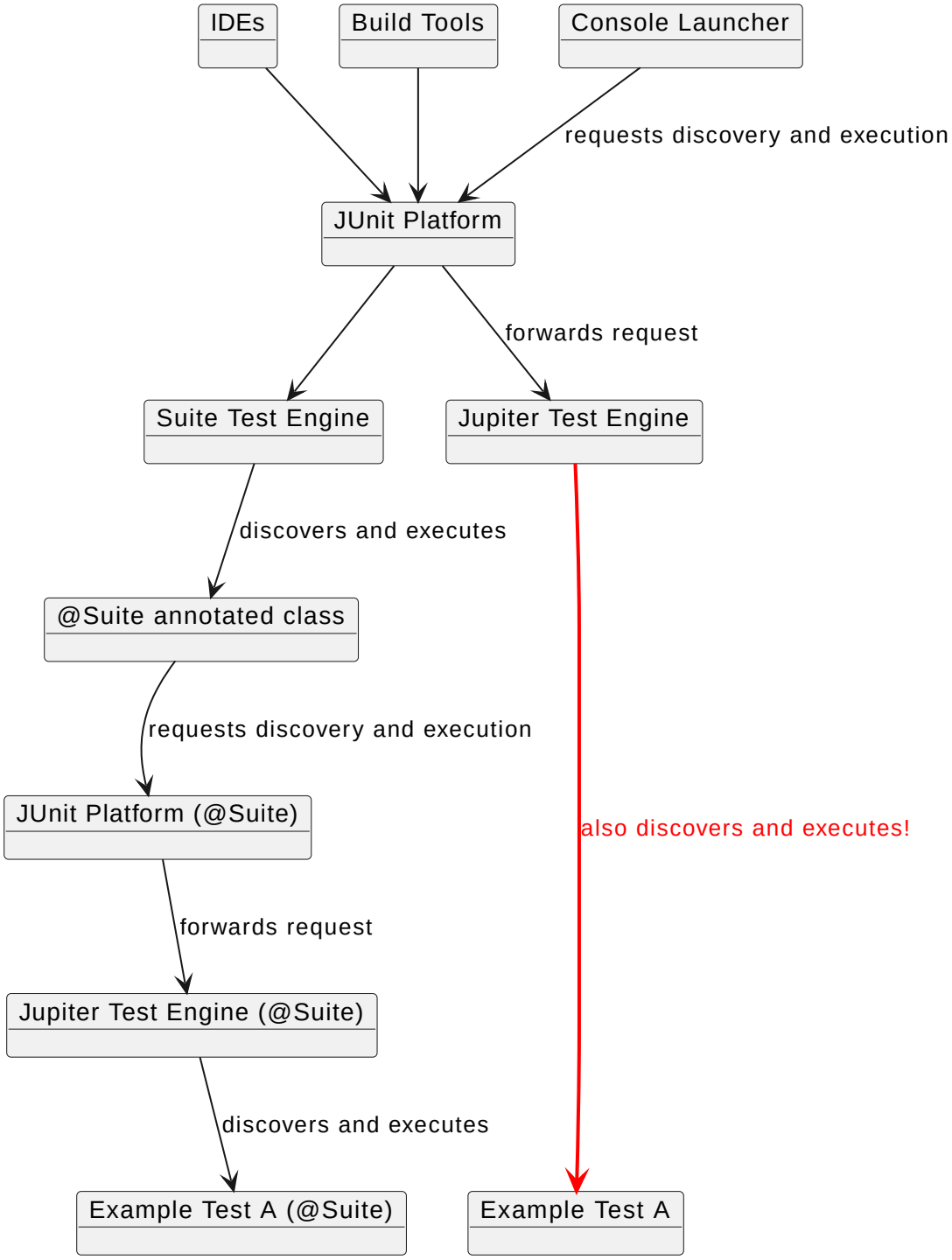
    @BeforeSuite
    static void beforeSuite() {
        // executes before the test suite
    }

    @AfterSuite
    static void afterSuite() {
        // executes after the test suite
    }

}
```

Duplicate Test Execution

Depending on the declared selectors, different suites may contain the same tests, potentially with different configurations. Moreover, tests in a suite are executed in addition to the tests executed by every other test engine, which can result in the same tests being executed twice.



To prevent duplicate execution of tests within a suite, configure your build tool to include only the `junit-platform-suite` engine, or use a custom naming pattern. For example, name all suites `*Suite` and all tests `*Test`, and configure your build tool to include only the former.

Alternatively, consider using tags to select specific groups of tests.

JUnit Platform Test Kit

The `junit-platform-testkit` artifact provides support for executing a test plan on the JUnit Platform and then verifying the expected results. This support is currently limited to the execution of a single `TestEngine` (see Engine Test Kit).

Engine Test Kit

The `org.junit.platform.testkit.engine` package provides support for discovering and executing a `TestPlan` for a given `TestEngine` running on the JUnit Platform and then accessing the results via convenient result objects. For execution, a fluent API may be used to verify the expected execution events were received. The key entry point into this API is the `EngineTestKit` which provides static factory methods named `engine()`, `discover()`, and `execute()`. It is recommended that you select one of the `engine()` variants to benefit from the fluent API for building a `LauncherDiscoveryRequest`.



If you prefer to use the `LauncherDiscoveryRequestBuilder` from the `Launcher` API to build your `LauncherDiscoveryRequest`, you must use one of the `discover()` or `execute()` variants in `EngineTestKit`.

The following test class written using JUnit Jupiter will be used in subsequent examples.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assumptions.assumeTrue;

import example.util.Calculator;

import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;

@TestMethodOrder(OrderAnnotation.class)
public class ExampleTestCase {

    private final Calculator calculator = new Calculator();

    @Test
    @Disabled("for demonstration purposes")
    @Order(1)
    void skippedTest() {
        // skipped ...
    }

    @Test
    @Order(2)
    void succeedingTest() {
        assertEquals(42, calculator.multiply(6, 7));
    }

    @Test
    @Order(3)
    void abortedTest() {
        assumeTrue("abc".contains("Z"), "abc does not contain Z");
        // aborted ...
    }
```

```
    @Test
    @Order(4)
    void failingTest() {
        // The following throws an ArithmeticException: "/ by zero"
        calculator.divide(1, 0);
    }

}
```

For the sake of brevity, the following sections demonstrate how to test JUnit's own `JupiterTestEngine` whose unique engine ID is `"junit-jupiter"`. If you want to test your own `TestEngine` implementation, you need to use its unique engine ID. Alternatively, you may test your own `TestEngine` by supplying an instance of it to the `EngineTestKit.engine(TestEngine)` static factory method.

Verifying Test Discovery

The following test demonstrates how to verify that a `TestPlan` was discovered as expected by the JUnit Jupiter `TestEngine`.

```
import static java.util.Collections.emptyList;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.platform.engine.discovery.DiscoverySelectors.selectClass;

import example.ExampleTestCase;

import org.junit.jupiter.api.Test;
import org.junit.platform.testkit.engine.EngineDiscoveryResults;
import org.junit.platform.testkit.engine.EngineTestKit;

class EngineTestKitDiscoveryDemo {

    @Test
    void verifyJupiterDiscovery() {
        EngineDiscoveryResults results = EngineTestKit.engine("junit-jupiter") ①
                .selectors(selectClass(ExampleTestCase.class)) ②
                .discover(); ③

        assertEquals("JUnit Jupiter", results.getEngineDescriptor().getDisplayName());
④
        assertEquals(emptyList(), results.getDiscoveryIssues()); ⑤
    }

}
```

① Select the JUnit Jupiter `TestEngine`.

② Select the `ExampleTestCase` test class.

③ Discover the `TestPlan`.

④ Assert engine root descriptor has expected display name.

⑤ Assert no discovery issues were encountered.

Asserting Execution Statistics

One of the most common features of the Test Kit is the ability to assert statistics against events fired during the execution of a `TestPlan`. The following tests demonstrate how to assert statistics for *containers* and *tests* in the JUnit Jupiter `TestEngine`. For details on what statistics are available, consult the Javadoc for `EventStatistics`.

```
import static org.junit.platform.engine.discovery.DiscoverySelectors.selectClass;

import example.ExampleTestCase;

import org.junit.jupiter.api.Test;
import org.junit.platform.testkit.engine.EngineTestKit;

class EngineTestKitStatisticsDemo {

    @Test
    void verifyJupiterContainerStats() {
        EngineTestKit
            .engine("junit-jupiter") ①
            .selectors(selectClass(ExampleTestCase.class)) ②
            .execute() ③
            .containerEvents() ④
            .assertStatistics(stats -> stats.started(2).succeeded(2)); ⑤
    }

    @Test
    void verifyJupiterTestStats() {
        EngineTestKit
            .engine("junit-jupiter") ①
            .selectors(selectClass(ExampleTestCase.class)) ②
            .execute() ③
            .testEvents() ⑥
            .assertStatistics(stats ->
                stats.skipped(1).started(3).succeeded(1).aborted(1).failed(1)); ⑦
    }

}
```

① Select the JUnit Jupiter `TestEngine`.

② Select the `ExampleTestCase` test class.

③ Execute the `TestPlan`.

④ Filter by *container* events.

⑤ Assert statistics for *container* events.

⑥ Filter by *test* events.

⑦ Assert statistics for *test* events.



In the `verifyJupiterContainerStats()` test method, the counts for the `started` and `succeeded` statistics are `2` since the `JupiterTestEngine` and the `ExampleTestCase` class are both considered containers.

Asserting Events

If you find that asserting statistics alone is insufficient for verifying the expected behavior of test execution, you can work directly with the recorded `Event` elements and perform assertions against them.

For example, if you want to verify the reason that the `skippedTest()` method in `ExampleTestCase` was skipped, you can do that as follows.



The `assertThatEvents()` method in the following example is a shortcut for `org.assertj.core.api.Assertions.assertThat(events.list())` from the AssertJ assertion library.

For details on what *conditions* are available for use with AssertJ assertions against events, consult the Javadoc for `EventConditions`.

```
import static org.junit.platform.engine.discovery.DiscoverySelectors.selectMethod;
import static org.junit.platform.testkit.engine.EventConditions.event;
import static org.junit.platform.testkit.engine.EventConditions.skippedWithReason;
import static org.junit.platform.testkit.engine.EventConditions.test;

import example.ExampleTestCase;

import org.junit.jupiter.api.Test;
import org.junit.platform.testkit.engine.EngineTestKit;
import org.junit.platform.testkit.engine.Events;

class EngineTestKitSkippedMethodDemo {

    @Test
    void verifyJupiterMethodWasSkipped() {
        String methodName = "skippedTest";

        Events testEvents = EngineTestKit ⑤
            .engine("junit-jupiter") ①
            .selectors(selectMethod(ExampleTestCase.class, methodName)) ②
            .execute() ③
            .testEvents(); ④

        testEvents.assertStatistics(stats -> stats.skipped(1)); ⑥
```

```
        testEvents.assertThatEvents() ⑦
            .haveExactly(1, event(test(methodName),
                skippedWithReason("for demonstration purposes")));
    }

}
```

① Select the JUnit Jupiter `TestEngine`.

② Select the `skippedTest()` method in the `ExampleTestCase` test class.

③ Execute the `TestPlan`.

④ Filter by *test* events.

⑤ Save the *test* `Events` to a local variable.

⑥ Optionally assert the expected statistics.

⑦ Assert that the recorded *test* events contain exactly one skipped test named `skippedTest` with `"for demonstration purposes"` as the *reason*.

If you want to verify the type of exception thrown from the `failingTest()` method in `ExampleTestCase`, you can do that as follows.



For details on what *conditions* are available for use with AssertJ assertions against events and execution results, consult the Javadoc for `EventConditions` and `TestExecutionResultConditions`, respectively.

```
import static org.junit.platform.engine.discovery.DiscoverySelectors.selectClass;
import static org.junit.platform.testkit.engine.EventConditions.event;
import static org.junit.platform.testkit.engine.EventConditions.finishedWithFailure;
import static org.junit.platform.testkit.engine.EventConditions.test;
import static
org.junit.platform.testkit.engine.TestExecutionResultConditions.instanceOf;
import static org.junit.platform.testkit.engine.TestExecutionResultConditions.message;

import example.ExampleTestCase;

import org.junit.jupiter.api.Test;
import org.junit.platform.testkit.engine.EngineTestKit;

class EngineTestKitFailedMethodDemo {

    @Test
    void verifyJupiterMethodFailed() {
        EngineTestKit.engine("junit-jupiter") ①
            .selectors(selectClass(ExampleTestCase.class)) ②
            .execute() ③
            .testEvents() ④
            .assertThatEvents().haveExactly(1, ⑤
                event(test("failingTest"),
```

```
                    finishedWithFailure(
                        instanceOf(ArithmeticException.class), message(it -> it
.endsWith("by zero"))));
    }

}
```

① Select the JUnit Jupiter `TestEngine`.

② Select the `ExampleTestCase` test class.

③ Execute the `TestPlan`.

④ Filter by *test* events.

⑤ Assert that the recorded *test* events contain exactly one failing test named `failingTest` with an exception of type `ArithmeticException` and an error message that ends with `"/ by zero"`.

Although typically unnecessary, there are times when you need to verify **all** of the events fired during the execution of a `TestPlan`. The following test demonstrates how to achieve this via the `assertEventsMatchExactly()` method in the `EngineTestKit` API.



Since `assertEventsMatchExactly()` matches conditions exactly in the order in which the events were fired, `ExampleTestCase` has been annotated with `@TestMethodOrder(OrderAnnotation.class)` and each test method has been annotated with `@Order(…)`. This allows us to enforce the order in which the test methods are executed, which in turn allows our `verifyAllJupiterEvents()` test to be reliable.

If you want to do a *partial* match *with* or *without* ordering requirements, you can use the methods `assertEventsMatchLooselyInOrder()` and `assertEventsMatchLoosely()`, respectively.

```
import static org.junit.platform.engine.discovery.DiscoverySelectors.selectClass;
import static org.junit.platform.testkit.engine.EventConditions.abortedWithReason;
import static org.junit.platform.testkit.engine.EventConditions.container;
import static org.junit.platform.testkit.engine.EventConditions.engine;
import static org.junit.platform.testkit.engine.EventConditions.event;
import static org.junit.platform.testkit.engine.EventConditions.finishedSuccessfully;
import static org.junit.platform.testkit.engine.EventConditions.finishedWithFailure;
import static org.junit.platform.testkit.engine.EventConditions.skippedWithReason;
import static org.junit.platform.testkit.engine.EventConditions.started;
import static org.junit.platform.testkit.engine.EventConditions.test;
import static
org.junit.platform.testkit.engine.TestExecutionResultConditions.instanceOf;
import static org.junit.platform.testkit.engine.TestExecutionResultConditions.message;

import java.io.StringWriter;
import java.io.Writer;

import example.ExampleTestCase;
```

```
import org.junit.jupiter.api.Test;
import org.junit.platform.testkit.engine.EngineTestKit;
import org.opentest4j.TestAbortedException;

class EngineTestKitAllEventsDemo {

    @Test
    void verifyAllJupiterEvents() {
        Writer writer = // create a java.io.Writer for debug output

        EngineTestKit.engine("junit-jupiter") ①
            .selectors(selectClass(ExampleTestCase.class)) ②
            .execute() ③
            .allEvents() ④
            .debug(writer) ⑤
            .assertEventsMatchExactly( ⑥
                event(engine(), started()),
                event(container(ExampleTestCase.class), started()),
                event(test("skippedTest"), skippedWithReason("for demonstration
purposes")),
                event(test("succeedingTest"), started()),
                event(test("succeedingTest"), finishedSuccessfully()),
                event(test("abortedTest"), started()),
                event(test("abortedTest"),
                    abortedWithReason(instanceOf(TestAbortedException.class),
                        message(m -> m.contains("abc does not contain Z")))),
                event(test("failingTest"), started()),
                event(test("failingTest"), finishedWithFailure(
                    instanceOf(ArithmeticException.class), message(it -> it.endsWith
("by zero")))),
                event(container(ExampleTestCase.class), finishedSuccessfully()),
                event(engine(), finishedSuccessfully()));
    }

}
```

① Select the JUnit Jupiter `TestEngine`.

② Select the `ExampleTestCase` test class.

③ Execute the `TestPlan`.

④ Filter by *all* events.

⑤ Print all events to the supplied `writer` for debugging purposes. Debug information can also be written to an `OutputStream` such as `System.out` or `System.err`.

⑥ Assert *all* events in exactly the order in which they were fired by the test engine.

The `debug()` invocation from the preceding example results in output similar to the following.

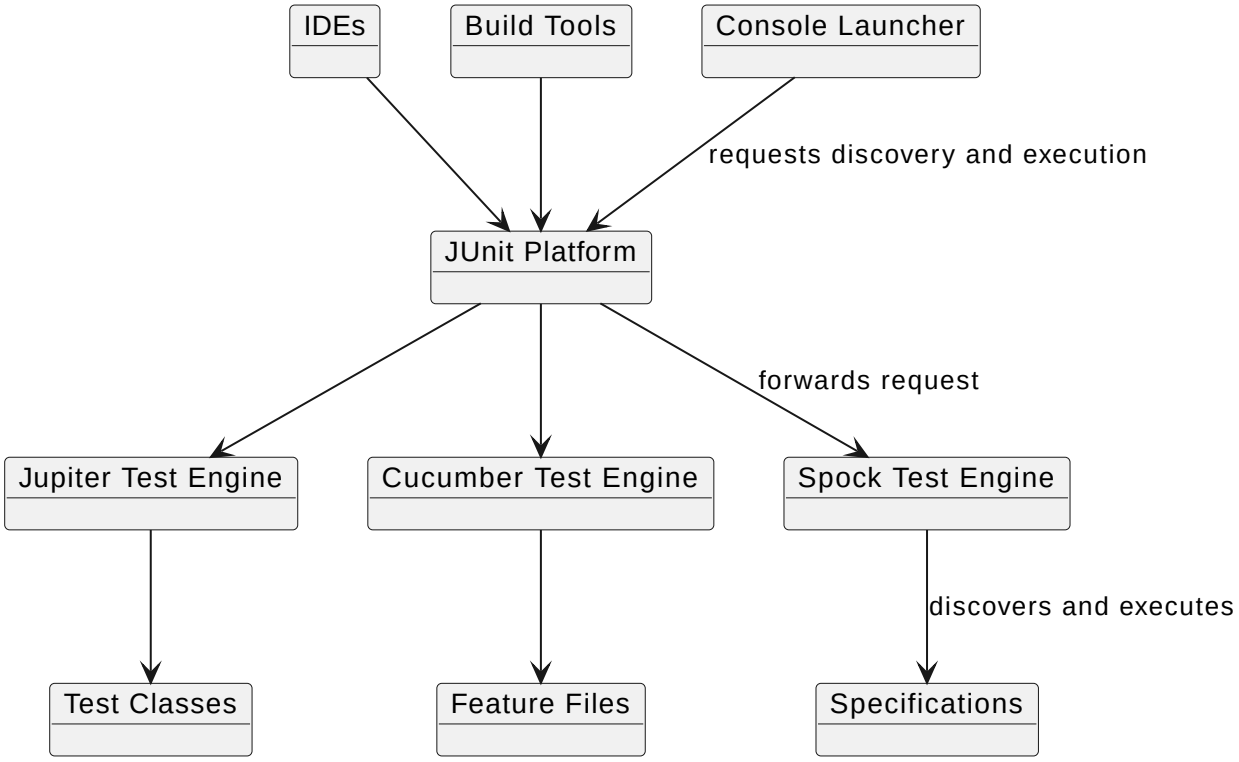
```
All Events:
    Event [type = STARTED, testDescriptor = JupiterEngineDescriptor: [engine:junit-
```

```
jupiter], timestamp = 2018-12-14T12:45:14.082280Z, payload = null]
    Event [type = STARTED, testDescriptor = ClassTestDescriptor: [engine:junit-
jupiter]/[class:example.ExampleTestCase], timestamp = 2018-12-14T12:45:14.089339Z,
payload = null]
    Event [type = SKIPPED, testDescriptor = TestMethodTestDescriptor: [engine:junit-
jupiter]/[class:example.ExampleTestCase]/[method:skippedTest()], timestamp = 2018-12-
14T12:45:14.094314Z, payload = 'for demonstration purposes']
    Event [type = STARTED, testDescriptor = TestMethodTestDescriptor: [engine:junit-
jupiter]/[class:example.ExampleTestCase]/[method:succeedingTest()], timestamp = 2018-
12-14T12:45:14.095182Z, payload = null]
    Event [type = FINISHED, testDescriptor = TestMethodTestDescriptor: [engine:junit-
jupiter]/[class:example.ExampleTestCase]/[method:succeedingTest()], timestamp = 2018-
12-14T12:45:14.104922Z, payload = TestExecutionResult [status = SUCCESSFUL, throwable
= null]]
    Event [type = STARTED, testDescriptor = TestMethodTestDescriptor: [engine:junit-
jupiter]/[class:example.ExampleTestCase]/[method:abortedTest()], timestamp = 2018-12-
14T12:45:14.106121Z, payload = null]
    Event [type = FINISHED, testDescriptor = TestMethodTestDescriptor: [engine:junit-
jupiter]/[class:example.ExampleTestCase]/[method:abortedTest()], timestamp = 2018-12-
14T12:45:14.109956Z, payload = TestExecutionResult [status = ABORTED, throwable =
org.opentest4j.TestAbortedException: Assumption failed: abc does not contain Z]]
    Event [type = STARTED, testDescriptor = TestMethodTestDescriptor: [engine:junit-
jupiter]/[class:example.ExampleTestCase]/[method:failingTest()], timestamp = 2018-12-
14T12:45:14.110680Z, payload = null]
    Event [type = FINISHED, testDescriptor = TestMethodTestDescriptor: [engine:junit-
jupiter]/[class:example.ExampleTestCase]/[method:failingTest()], timestamp = 2018-12-
14T12:45:14.111217Z, payload = TestExecutionResult [status = FAILED, throwable =
java.lang.ArithmeticException: / by zero]]
    Event [type = FINISHED, testDescriptor = ClassTestDescriptor: [engine:junit-
jupiter]/[class:example.ExampleTestCase], timestamp = 2018-12-14T12:45:14.113731Z,
payload = TestExecutionResult [status = SUCCESSFUL, throwable = null]]
    Event [type = FINISHED, testDescriptor = JupiterEngineDescriptor: [engine:junit-
jupiter], timestamp = 2018-12-14T12:45:14.113806Z, payload = TestExecutionResult
[status = SUCCESSFUL, throwable = null]]
```

JUnit Platform Launcher API

One of the prominent goals of the JUnit Platform is to make the interface between JUnit and its programmatic clients – build tools and IDEs – more powerful and stable. The purpose is to decouple the internals of discovering and executing tests from all the filtering and configuration that is necessary from the outside.

JUnit Platform provides a `Launcher` API that can be used to discover, filter, and execute tests. Moreover, third party test libraries – like Spock or Cucumber – can plug into the JUnit Platform's launching infrastructure by providing a custom TestEngine.



The launcher API is in the `junit-platform-launcher` module.

An example consumer of the launcher API is the `ConsoleLauncher` in the `junit-platform-console` project.

Discovering Tests

Having *test discovery* as a dedicated feature of the platform frees IDEs and build tools from most of the difficulties they had to go through to identify test classes and test methods in previous versions of JUnit.

Usage Example:

```
import static
org.junit.platform.engine.discovery.ClassNameFilter.includeClassNamePatterns;
import static org.junit.platform.engine.discovery.DiscoverySelectors.selectClass;
import static org.junit.platform.engine.discovery.DiscoverySelectors.selectPackage;
import static
org.junit.platform.launcher.core.LauncherDiscoveryRequestBuilder.discoveryRequest;

import org.junit.platform.launcher.LauncherDiscoveryRequest;
import org.junit.platform.launcher.LauncherSession;
import org.junit.platform.launcher.TestPlan;
import org.junit.platform.launcher.core.LauncherFactory;
```

```
LauncherDiscoveryRequest discoveryRequest = discoveryRequest()
    .selectors(
        selectPackage("com.example.mytests"),
```

```
        selectClass(MyTestClass.class)
    )
    .filters(
        includeClassNamePatterns(".*Tests")
    )
    .build();

try (LauncherSession session = LauncherFactory.openSession()) {
    TestPlan testPlan = session.getLauncher().discover(discoveryRequest);

    // ... discover additional test plans or execute tests
}
```

You can select classes, methods, and all classes in a package or even search for all tests in the class-path or module-path. Discovery takes place across all participating test engines.

The resulting `TestPlan` is a hierarchical (and read-only) description of all engines, classes, and test methods that fit the `LauncherDiscoveryRequest`. The client can traverse the tree, retrieve details about a node, and get a link to the original source (like class, method, or file position). Every node in the test plan has a *unique ID* that can be used to invoke a particular test or group of tests.

Clients can register one or more `LauncherDiscoveryListener` implementations via the `LauncherDiscoveryRequestBuilder` to gain insight into events that occur during test discovery. By default, the builder registers an "abort on failure" listener that aborts test discovery after the first discovery failure is encountered. The default `LauncherDiscoveryListener` can be changed via the `junit.platform.discovery.listener.default` configuration parameter.

Executing Tests

To execute tests, clients can use the same `LauncherDiscoveryRequest` as in the discovery phase or create a new request. Test progress and reporting can be achieved by registering one or more `TestExecutionListener` implementations with the `Launcher` as in the following example.

```
LauncherDiscoveryRequest discoveryRequest = LauncherDiscoveryRequestBuilder.request()
    .selectors(
        selectPackage("com.example.mytests"),
        selectClass(MyTestClass.class)
    )
    .filters(
        includeClassNamePatterns(".*Tests")
    )
    .build();

SummaryGeneratingListener listener = new SummaryGeneratingListener();

try (LauncherSession session = LauncherFactory.openSession()) {
    Launcher launcher = session.getLauncher();
    // Register one ore more listeners of your choice.
    launcher.registerTestExecutionListeners(listener);
```

```
    // Discover tests and build a test plan.
    TestPlan testPlan = launcher.discover(discoveryRequest);
    // Execute the test plan.
    launcher.execute(testPlan);
    // Alternatively, execute the discovery request directly.
    launcher.execute(discoveryRequest);
}

TestExecutionSummary summary = listener.getSummary();
// Do something with the summary...
```

There is no return value for the `execute()` method, but you can use a `TestExecutionListener` to aggregate the results. For examples see the `SummaryGeneratingListener`, `LegacyXmlReportGeneratingListener`, and `UniqueIdTrackingListener`.



All `TestExecutionListener` methods are called sequentially. Methods for start events are called in registration order while methods for finish events are called in reverse order. Test case execution won't start before all `executionStarted` calls have returned.

Registering a TestEngine

See the dedicated section on TestEngine registration for details.

Registering a PostDiscoveryFilter

In addition to specifying post-discovery filters as part of a `LauncherDiscoveryRequest` passed to the `Launcher` API, `PostDiscoveryFilter` implementations will be discovered at runtime via Java's `ServiceLoader` mechanism and automatically applied by the `Launcher` in addition to those that are part of the request.

For example, an `example.CustomTagFilter` class implementing `PostDiscoveryFilter` and declared within the `/META-INF/services/org.junit.platform.launcher.PostDiscoveryFilter` file is loaded and applied automatically.

Registering a LauncherSessionListener

Registered implementations of `LauncherSessionListener` are notified when a `LauncherSession` is opened (before a `Launcher` first discovers and executes tests) and closed (when no more tests will be discovered or executed). They can be registered programmatically via the `LauncherConfig` that is passed to the `LauncherFactory`, or they can be discovered at runtime via Java's `ServiceLoader` mechanism and automatically registered with `LauncherSession` (unless automatic registration is disabled.)

Tool Support

The following build tools and IDEs are known to provide full support for `LauncherSession`:

- Gradle 4.6 and later

- Maven Surefire/Failsafe 3.0.0-M6 and later
- IntelliJ IDEA 2017.3 and later

Other tools might also work but have not been tested explicitly.

Example Usage

A `LauncherSessionListener` is well suited for implementing once-per-JVM setup/teardown behavior since it's called before the first and after the last test in a launcher session, respectively. The scope of a launcher session depends on the used IDE or build tool but usually corresponds to the lifecycle of the test JVM. A custom listener that starts an HTTP server before executing the first test and stops it after the last test has been executed, could look like this:

src/test/java/example/session/GlobalSetupTeardownListener.java

```
package example.session;

import static java.net.InetAddress.getLoopbackAddress;

import java.io.IOException;
import java.io.UncheckedIOException;
import java.net.InetSocketAddress;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import com.sun.net.httpserver.HttpServer;

import org.junit.platform.engine.support.store.Namespace;
import org.junit.platform.engine.support.store.NamespacedHierarchicalStore;
import org.junit.platform.launcher.LauncherSession;
import org.junit.platform.launcher.LauncherSessionListener;
import org.junit.platform.launcher.TestExecutionListener;
import org.junit.platform.launcher.TestPlan;

public class GlobalSetupTeardownListener implements LauncherSessionListener {

    @Override
    public void launcherSessionOpened(LauncherSession session) {
        // Avoid setup for test discovery by delaying it until tests are about to be
executed
        session.getLauncher().registerTestExecutionListeners(new
TestExecutionListener() {
            @Override
            public void testPlanExecutionStarted(TestPlan testPlan) {
                NamespacedHierarchicalStore<Namespace> store = session.getStore(); ①
                store.computeIfAbsent(Namespace.GLOBAL, "httpServer", key -> { ②
                    InetSocketAddress address = new InetSocketAddress
(getLoopbackAddress(), 0);
                    HttpServer server;
                    try {
                        server = HttpServer.create(address, 0);
```

```
                    }
                    catch (IOException e) {
                        throw new UncheckedIOException("Failed to start HTTP server",
e);
                    }
                    server.createContext("/test", exchange -> {
                        exchange.sendResponseHeaders(204, -1);
                        exchange.close();
                    });
                    ExecutorService executorService = Executors.newCachedThreadPool();
                    server.setExecutor(executorService);
                    server.start(); ③

                    return new CloseableHttpServer(server, executorService);
                });
            }
        });
    }

}
```

① Get the store from the launcher session

② Lazily create the HTTP server and put it into the store

③ Start the HTTP server

It uses a wrapper class to ensure the server is stopped when the launcher session is closed:

src/test/java/example/session/CloseableHttpServer.java

```
package example.session;

import java.util.concurrent.ExecutorService;

import com.sun.net.httpserver.HttpServer;

public class CloseableHttpServer implements AutoCloseable {

    private final HttpServer server;
    private final ExecutorService executorService;

    CloseableHttpServer(HttpServer server, ExecutorService executorService) {
        this.server = server;
        this.executorService = executorService;
    }

    public HttpServer getServer() {
        return server;
    }

    @Override
```

```
    public void close() { ①
        server.stop(0); ②
        executorService.shutdownNow();
    }
}
```

① The `close()` method is called when the launcher session is closed

② Stop the HTTP server

This sample uses the HTTP server implementation from the jdk.httpserver module that comes with the JDK but would work similarly with any other server or resource. In order for the listener to be picked up by JUnit Platform, you need to register it as a service by adding a resource file with the following name and contents to your test runtime classpath (e.g. by adding the file to `src/test/resources`):

src/test/resources/META-INF/services/org.junit.platform.launcher.LauncherSessionListener

```
example.session.GlobalSetupTeardownListener
```

You can now use the resource from your test:

src/test/java/example/session/HttpTests.java

```
package example.session;

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

import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URI;
import java.net.URL;

import com.sun.net.httpserver.HttpServer;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.api.extension.ParameterContext;
import org.junit.jupiter.api.extension.ParameterResolver;

@ExtendWith(HttpServerParameterResolver.class)
class HttpTests {

    @Test
    void respondsWith204(HttpServer server) throws IOException {
        String host = server.getAddress().getHostString(); ②
        int port = server.getAddress().getPort(); ③
        URL url = URI.create("http://" + host + ":" + port + "/test").toURL();

        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
```

```
        connection.setRequestMethod("GET");
        int responseCode = connection.getResponseCode(); ④

        assertEquals(204, responseCode); ⑤
    }
}

class HttpServerParameterResolver implements ParameterResolver {
    @Override
    public boolean supportsParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
        return HttpServer.class.equals(parameterContext.getParameter().getType());
    }

    @Override
    public Object resolveParameter(ParameterContext parameterContext, ExtensionContext
extensionContext) {
        return extensionContext
                .getStore(ExtensionContext.Namespace.GLOBAL)
                .get("httpServer", CloseableHttpServer.class) ①
                .getServer();
    }
}
```

① Retrieve the HTTP server instance from the store

② Get the host string directly from the injected HTTP server instance

③ Get the port number directly from the injected HTTP server instance

④ Send a request to the server

⑤ Check the status code of the response

Registering a LauncherInterceptor

In order to intercept the creation of instances of `Launcher` and `LauncherSessionListener` and calls to the `discover` and `execute` methods of the former, clients can register custom implementations of `LauncherInterceptor` via Java's `ServiceLoader` mechanism by setting the `junit.platform.launcher.interceptors.enabled` configuration parameter to `true`.



Since interceptors are registered before the test run starts, the `junit.platform.launcher.interceptors.enabled` *configuration parameter* can only be supplied as a JVM system property or via the JUnit Platform configuration file (see Configuration Parameters for details). This *configuration parameter* cannot be supplied in the `LauncherDiscoveryRequest` that is passed to the `Launcher`.

A typical use case is to create a custom interceptor to replace the `ClassLoader` used by the JUnit Platform to load test classes and engine implementations.

```
import java.io.IOException;
```

```
import java.io.UncheckedIOException;
import java.net.URI;
import java.net.URL;
import java.net.URLClassLoader;

import org.junit.platform.launcher.LauncherInterceptor;

public class CustomLauncherInterceptor implements LauncherInterceptor {

    private final URLClassLoader customClassLoader;

    public CustomLauncherInterceptor() throws Exception {
        ClassLoader parent = Thread.currentThread().getContextClassLoader();
        customClassLoader = new URLClassLoader(new URL[] { URI.create("some.jar"
).toURL() }, parent);
    }

    @Override
    public <T> T intercept(Invocation<T> invocation) {
        Thread currentThread = Thread.currentThread();
        ClassLoader originalClassLoader = currentThread.getContextClassLoader();
        currentThread.setContextClassLoader(customClassLoader);
        try {
            return invocation.proceed();
        }
        finally {
            currentThread.setContextClassLoader(originalClassLoader);
        }
    }

    @Override
    public void close() {
        try {
            customClassLoader.close();
        }
        catch (IOException e) {
            throw new UncheckedIOException("Failed to close custom class loader", e);
        }
    }
}
```

Registering a LauncherDiscoveryListener

In addition to specifying discovery listeners as part of a `LauncherDiscoveryRequest` or registering them programmatically via the `Launcher` API, custom `LauncherDiscoveryListener` implementations can be discovered at runtime via Java's `ServiceLoader` mechanism and automatically registered with the `Launcher` created via the `LauncherFactory`.

For example, an `example.CustomLauncherDiscoveryListener` class implementing `LauncherDiscoveryListener` and declared within the `/META-`

INF/services/org.junit.platform.launcher.LauncherDiscoveryListener file is loaded and registered automatically.

Registering a TestExecutionListener

In addition to the public Launcher API method for registering test execution listeners programmatically, custom TestExecutionListener implementations will be discovered at runtime via Java's ServiceLoader mechanism and automatically registered with the Launcher created via the LauncherFactory.

For example, an example.CustomTestExecutionListener class implementing TestExecutionListener and declared within the /META-INF/services/org.junit.platform.launcher.TestExecutionListener file is loaded and registered automatically.

Configuring a TestExecutionListener

When a TestExecutionListener is registered programmatically via the Launcher API, the listener may provide programmatic ways for it to be configured — for example, via its constructor, setter methods, etc. However, when a TestExecutionListener is registered automatically via Java's ServiceLoader mechanism (see Registering a TestExecutionListener), there is no way for the user to directly configure the listener. In such cases, the author of a TestExecutionListener may choose to make the listener configurable via configuration parameters. The listener can then access the configuration parameters via the TestPlan supplied to the testPlanExecutionStarted(TestPlan) and testPlanExecutionFinished(TestPlan) callback methods. See the UniqueIdTrackingListener for an example.

Deactivating a TestExecutionListener

Sometimes it can be useful to run a test suite *without* certain execution listeners being active. For example, you might have custom a TestExecutionListener that sends the test results to an external system for reporting purposes, and while debugging you might not want these *debug* results to be reported. To do this, provide a pattern for the junit.platform.execution.listeners.deactivate *configuration parameter* to specify which execution listeners should be deactivated (i.e. not registered) for the current test run.



Only listeners registered via the ServiceLoader mechanism within the /META-INF/services/org.junit.platform.launcher.TestExecutionListener file can be deactivated. In other words, any TestExecutionListener registered explicitly via the LauncherDiscoveryRequest cannot be deactivated via the junit.platform.execution.listeners.deactivate *configuration parameter*.

In addition, since execution listeners are registered before the test run starts, the junit.platform.execution.listeners.deactivate *configuration parameter* can only be supplied as a JVM system property or via the JUnit Platform configuration file (see Configuration Parameters for details). This *configuration parameter* cannot be supplied in the LauncherDiscoveryRequest that is passed to the Launcher.

Pattern Matching Syntax

Refer to Pattern Matching Syntax for details.

Configuring the Launcher

If you require fine-grained control over automatic detection and registration of test engines and listeners, you may create an instance of `LauncherConfig` and supply that to the `LauncherFactory`. Typically, an instance of `LauncherConfig` is created via the built-in fluent *builder* API, as demonstrated in the following example.

```
LauncherConfig launcherConfig = LauncherConfig.builder()
    .enableTestEngineAutoRegistration(false)
    .enableLauncherSessionListenerAutoRegistration(false)
    .enableLauncherDiscoveryListenerAutoRegistration(false)
    .enablePostDiscoveryFilterAutoRegistration(false)
    .enableTestExecutionListenerAutoRegistration(false)
    .addTestEngines(new CustomTestEngine())
    .addLauncherSessionListeners(new CustomLauncherSessionListener())
    .addLauncherDiscoveryListeners(new CustomLauncherDiscoveryListener())
    .addPostDiscoveryFilters(new CustomPostDiscoveryFilter())
    .addTestExecutionListeners(new LegacyXmlReportGeneratingListener(reportsDir, out))
    .addTestExecutionListeners(new CustomTestExecutionListener())
    .build();

LauncherDiscoveryRequest discoveryRequest = LauncherDiscoveryRequestBuilder.request()
    .selectors(selectPackage("com.example.mytests"))
    .build();

try (LauncherSession session = LauncherFactory.openSession(launcherConfig)) {
    session.getLauncher().execute(discoveryRequest);
}
```

Dry-Run Mode

When running tests via the `Launcher` API, you can enable *dry-run mode* by setting the `junit.platform.execution.dryRun.enabled` configuration parameter to `true`. In this mode, the `Launcher` will not actually execute any tests but will notify registered `TestExecutionListener` instances as if all tests had been skipped and their containers had been successful. This can be useful to test changes in the configuration of a build or to verify a listener is called as expected without having to wait for all tests to be executed.

Managing State Across Test Engines

When running tests on the JUnit Platform, multiple test engines may need to access shared resources. Rather than initializing these resources multiple times, JUnit Platform provides mechanisms to share state across test engines efficiently. Test engines can use the Platform's `NamespacedHierarchicalStore` API to lazily initialize and share resources, ensuring they are created only once regardless of execution order. Any resource that is put into the store and implements

`AutoCloseable` will be closed automatically when the execution is finished.



The Jupiter engine allows read and write access to such resources via its `Store` API.

The following example demonstrates two custom test engines sharing a `ServerSocket` resource. `FirstCustomEngine` attempts to retrieve an existing `ServerSocket` from the global store or creates a new one if it doesn't exist:

```
import static java.net.InetAddress.getLoopbackAddress;
import static org.junit.platform.engine.TestExecutionResult.successful;

import java.io.IOException;
import java.io.UncheckedIOException;
import java.net.ServerSocket;

import org.jspecify.annotations.Nullable;
import org.junit.platform.engine.EngineDiscoveryRequest;
import org.junit.platform.engine.ExecutionRequest;
import org.junit.platform.engine.TestDescriptor;
import org.junit.platform.engine.TestEngine;
import org.junit.platform.engine.UniqueId;
import org.junit.platform.engine.support.descriptor.EngineDescriptor;
import org.junit.platform.engine.support.store.Namespace;
import org.junit.platform.engine.support.store.NamespacedHierarchicalStore;

/**
 * First custom test engine implementation.
 */
public class FirstCustomEngine implements TestEngine {

    public ServerSocket socket;

    @Override
    public String getId() {
        return "first-custom-test-engine";
    }

    public ServerSocket getSocket() {
        return this.socket;
    }

    @Override
    public TestDescriptor discover(EngineDiscoveryRequest discoveryRequest, UniqueId
uniqueId) {
        return new EngineDescriptor(uniqueId, "First Custom Test Engine");
    }

    @Override
    public void execute(ExecutionRequest request) {
        request.getEngineExecutionListener()
```

```
                .executionStarted(request.getRootTestDescriptor());

        NamespacedHierarchicalStore<Namespace> store = request.getStore();
        socket = store.computeIfAbsent(Namespace.GLOBAL, "serverSocket", key -> {
            try {
                return new ServerSocket(0, 50, getLoopbackAddress());
            }
            catch (IOException e) {
                throw new UncheckedIOException("Failed to start ServerSocket", e);
            }
        }, ServerSocket.class);

        request.getEngineExecutionListener()
                .executionFinished(request.getRootTestDescriptor(), successful());
    }

}
```

`SecondCustomEngine` follows the same pattern, ensuring that regardless whether it runs before or after `FirstCustomEngine`, it will use the same socket instance:

```
/**
 * Second custom test engine implementation.
 */
public class SecondCustomEngine implements TestEngine {

    public ServerSocket socket;

    @Override
    public String getId() {
        return "second-custom-test-engine";
    }

    public ServerSocket getSocket() {
        return this.socket;
    }

    @Override
    public TestDescriptor discover(EngineDiscoveryRequest discoveryRequest, UniqueId
uniqueId) {
        return new EngineDescriptor(uniqueId, "Second Custom Test Engine");
    }

    @Override
    public void execute(ExecutionRequest request) {
        request.getEngineExecutionListener()
                .executionStarted(request.getRootTestDescriptor());

        NamespacedHierarchicalStore<Namespace> store = request.getStore();
        socket = store.computeIfAbsent(Namespace.GLOBAL, "serverSocket", key -> {
```

```
            try {
                return new ServerSocket(0, 50, getLoopbackAddress());
            }
            catch (IOException e) {
                throw new UncheckedIOException("Failed to start ServerSocket", e);
            }
        }, ServerSocket.class);

        request.getEngineExecutionListener()
                .executionFinished(request.getRootTestDescriptor(), successful());
    }

}
```



In this case, the `ServerSocket` can be stored directly in the global store while ensuring since it gets closed because it implements `AutoCloseable`. If you need to use a type that does not do so, you can wrap it in a custom class that implements `AutoCloseable` and delegates to the original type. This is important to ensure that the resource is closed properly when the test run is finished.

For illustration, the following test verifies that both engines are sharing the same `ServerSocket` instance and that it's closed after `Launcher.execute()` returns:

```
@Test
void runBothCustomEnginesTest() {
    FirstCustomEngine firstCustomEngine = new FirstCustomEngine();
    SecondCustomEngine secondCustomEngine = new SecondCustomEngine();

    LauncherConfig launcherConfig = LauncherConfig.builder()
            .addTestEngines(firstCustomEngine, secondCustomEngine)
            .enableTestEngineAutoRegistration(false)
            .build();

    LauncherDiscoveryRequest discoveryRequest = discoveryRequest()
            .selectors(selectPackage("com.example.mytests"))
            .build();

    Launcher launcher = LauncherFactory.create(launcherConfig);
    launcher.execute(discoveryRequest);

    assertSame(firstCustomEngine.getSocket(), secondCustomEngine.getSocket());
    assertTrue(firstCustomEngine.getSocket().isClosed(), "socket should be closed");
}
```

By using the Platform's `NamespacedHierarchicalStore` API with shared namespaces in this way, test engines can coordinate resource creation and sharing without direct dependencies between them.

Alternatively, it's possible to inject resources into test engines by registering a

`LauncherSessionListener`.

Cancelling a Running Test Execution

The launcher API provides the ability to cancel a running test execution mid-flight while allowing engines to clean up resources. To request an execution to be cancelled, you need to call `cancel()` on the `CancellationToken` that is passed to `Launcher.execute` as part of the `LauncherExecutionRequest`.

For example, implementing a listener that cancels test execution after the first test failed can be achieved as follows.

```
CancellationToken cancellationToken = CancellationToken.create(); ①

TestExecutionListener failFastListener = new TestExecutionListener() {
    @Override
    public void executionFinished(TestIdentifier identifier, TestExecutionResult
result) {
        if (result.getStatus() == FAILED) {
            cancellationToken.cancel(); ②
        }
    }
};

LauncherExecutionRequest executionRequest = LauncherDiscoveryRequestBuilder.request()
        .selectors(selectClass(MyTestClass.class))
        .forExecution()
        .cancellationToken(cancellationToken) ③
        .listeners(failFastListener) ④
        .build();

try (LauncherSession session = LauncherFactory.openSession()) {
    session.getLauncher().execute(executionRequest); ⑤
}
```

① Create a `CancellationToken`

② Implement a `TestExecutionListener` that calls `cancel()` when a test fails

③ Register the cancellation token

④ Register the listener

⑤ Pass the `LauncherExecutionRequest` to `Launcher.execute`



Test Engine Support for Cancellation

Cancelling tests relies on Test Engines checking and responding to the `CancellationToken` appropriately (see Test Engine Requirements for details). The `Launcher` will also check the token and cancel test execution when multiple test engines are present at runtime.

At the time of writing, the following test engines support cancellation:

- `junit-jupiter-engine`
- `junit-vintage-engine`
- `junit-platform-suite-engine`
- Any `TestEngine` extending `HierarchicalTestEngine` such as Spock and Cucumber

Test Engines

A `TestEngine` facilitates *discovery* and *execution* of tests for a particular programming model.

For example, JUnit provides a `TestEngine` that discovers and executes tests written using the JUnit Jupiter programming model (see Writing Tests and Extension Model).

JUnit Test Engines

JUnit provides three `TestEngine` implementations.

- `junit-jupiter-engine`: The core of JUnit Jupiter.
- `junit-vintage-engine`: A thin layer on top of JUnit 4 to allow running *vintage* tests (based on JUnit 3.8 and JUnit 4) with the JUnit Platform launcher infrastructure.
- `junit-platform-suite-engine`: Executes declarative suites of tests with the JUnit Platform launcher infrastructure.

Custom Test Engines

You can contribute your own custom `TestEngine` by implementing the interfaces in the junit-platform-engine module and *registering* your engine.

Every `TestEngine` must provide its own *unique ID*, *discover* tests from an `EngineDiscoveryRequest`, and *execute* those tests according to an `ExecutionRequest`.

The `junit-` unique ID prefix is reserved for TestEngines from the JUnit Team

The JUnit Platform `Launcher` enforces that only `TestEngine` implementations published by the JUnit Team may use the `junit-` prefix for their `TestEngine` IDs.



- If any third-party `TestEngine` claims to be `junit-jupiter` or `junit-vintage`, an exception will be thrown, immediately halting execution of the JUnit Platform.
- If any third-party `TestEngine` uses the `junit-` prefix for its ID, a warning message will be logged. Later releases of the JUnit Platform will throw an exception for such violations.

In order to facilitate test discovery within IDEs and tools prior to launching the JUnit Platform, `TestEngine` implementations are encouraged to make use of the `@Testable` annotation. For example, the `@Test` and `@TestFactory` annotations in JUnit Jupiter are meta-annotated with `@Testable`. Consult the Javadoc for `@Testable` for further details.

If your custom `TestEngine` needs to be configured, consider allowing users to supply configuration

via configuration parameters. Please note, however, that you are strongly encouraged to use a unique prefix for all configuration parameters supported by your test engine. Doing so will ensure that there are no conflicts between the names of your configuration parameters and those from other test engines. In addition, since configuration parameters may be supplied as JVM system properties, it is wise to avoid conflicts with the names of other system properties. For example, JUnit Jupiter uses `junit.jupiter.` as a prefix of all of its supported configuration parameters. Furthermore, as with the warning above regarding the `junit-` prefix for `TestEngine` IDs, you should not use `junit.` as a prefix for the names of your own configuration parameters.

Although there is currently no official guide on how to implement a custom `TestEngine`, you can consult the implementation of JUnit Test Engines or the implementation of third-party test engines listed in the JUnit wiki. You will also find various tutorials and blogs on the Internet that demonstrate how to write a custom `TestEngine`.



`HierarchicalTestEngine` is a convenient abstract base implementation of the `TestEngine` SPI (used by the `junit-jupiter-engine`) that only requires implementors to provide the logic for test discovery. It implements execution of `TestDescriptors` that implement the `Node` interface, including support for parallel execution.

Registering a TestEngine

`TestEngine` registration is supported via Java's `ServiceLoader` mechanism.

For example, the `junit-jupiter-engine` module registers its `org.junit.jupiter.engine.JupiterTestEngine` in a file named `org.junit.platform.engine.TestEngine` within the `/META-INF/services` folder in the `junit-jupiter-engine` JAR.

Requirements



The words "must", "must not", "required", "shall", "shall not", "should", "should not", "recommended", "may", and "optional" in this section are to be interpreted as described in RFC 2119.

Mandatory requirements

For interoperability with build tools and IDEs, `TestEngine` implementations must adhere to the following requirements:

- The `TestDescriptor` returned from `TestEngine.discover()` *must* be the root of a tree of `TestDescriptor` instances. This implies that there *must not* be any cycles between a node and its descendants.
- A `TestEngine` *must* be able to discover `UniqueIdSelectors` for any unique ID that it previously generated and returned from `TestEngine.discover()`. This enables selecting a subset of tests to execute or rerun.
- The `executionSkipped`, `executionStarted`, and `executionFinished` methods of the `EngineExecutionListener` passed to `TestEngine.execute()` *must* be called for every `TestDescriptor` node in the tree returned from `TestEngine.discover()` at most once. Parent nodes *must* be

reported as started before their children and as finished after their children. If a node is reported as skipped, there *must not* be any events reported for its descendants.

Enhanced compatibility

Adhering to the following requirements is optional but recommended for enhanced compatibility with build tools and IDEs:

- Unless to indicate an empty discovery result, the `TestDescriptor` returned from `TestEngine.discover()` *should* have children rather than being completely dynamic. This allows tools to display the structure of the tests and to select a subset of tests to execute.
- When resolving `UniqueIdSelectors`, a `TestEngine` *should* only return `TestDescriptor` instances with matching unique IDs including their ancestors but *may* return additional siblings or other nodes that are required for the execution of the selected tests.
- `TestEngines` *should* support tagging tests and containers so that tag filters can be applied when discovering tests.
- A `TestEngine` *should* cancel its execution when the `CancellationToken` it is passed as part of the `ExecutionRequest` indicates that cancellation has been requested. In this case, it *should* report any remaining `TestDescriptors` as skipped but not report any events for their descendants. It *may* report already started `TestDescriptors` as aborted in case they have not been executed completely. If a `TestEngine` supports cancellation, it should clean up any resources that it has created just like if execution had finished regularly.

Reporting Discovery Issues

Test engines should report discovery issues if they encounter any problems or potential misconfigurations during test discovery. This is especially important if the issue could lead to tests not being executed at all or only partially.

In order to report a `DiscoveryIssue`, a test engine should call the `issueEncountered()` method on the `EngineDiscoveryListener` available via the `EngineDiscoveryRequest` passed to its `discover()` method. Rather than passing the listener around, the `DiscoveryIssueReporter` interface should be used. It also provides a way to create a `Condition` that reports a discovery issue if its check fails and may be used as a `Predicate` or `Consumer`. Please refer to the implementations of the test engines provided by JUnit for examples.

Moreover, Engine Test Kit provides a way to write tests for reported discovery issues.

API Evolution

One of the major goals of the JUnit Platform architecture is to improve maintainers' capabilities to evolve JUnit despite its being used in many projects. With JUnit 4 a lot of stuff that was originally added as an internal construct only got used by external extension writers and tool builders. That made changing JUnit 4 especially difficult and sometimes impossible.

That's why JUnit now uses a defined lifecycle for all publicly available interfaces, classes, and methods.

API Version and Status

Every published artifact has a version number `<major>.<minor>.<patch>`, and all publicly available interfaces, classes, and methods are annotated with @API from the @API Guardian project. The annotation's `status` attribute can be assigned one of the following values.

Status	Description
`INTERNAL`	Must not be used by any code other than JUnit itself. Might be removed without prior notice.
`DEPRECATED`	Should no longer be used; might disappear in the next minor release.
`EXPERIMENTAL`	Intended for new, experimental features where we are looking for feedback. Use this element with caution; it might be promoted to `MAINTAINED` or `STABLE` in the future, but might also be removed without prior notice, even in a patch.
`MAINTAINED`	Intended for features that will not be changed in a backwards- incompatible way for **at least** the next minor release of the current major version. If scheduled for removal, it will be demoted to `DEPRECATED` first.
`STABLE`	Intended for features that will not be changed in a backwards- incompatible way in the current major version (`6.*`).

If the `@API` annotation is present on a type, it is considered to be applicable for all public members of that type as well. A member is allowed to declare a different `status` value of lower stability.

Experimental APIs

The following tables list which APIs are currently designated as *experimental* via `@API(status = EXPERIMENTAL)`. Caution should be taken when relying on such APIs.

Module org.junit.jupiter.api

Package org.junit.jupiter.api

Package org.junit.jupiter.api.extension

Module org.junit.jupiter.params

Package org.junit.jupiter.params

Package org.junit.jupiter.params.aggregator

Package org.junit.jupiter.params.converter

Package org.junit.jupiter.params.provider

Package org.junit.jupiter.params.support

Module org.junit.platform.commons

Package org.junit.platform.commons.function

Package org.junit.platform.commons.support

Module org.junit.platform.engine

Package org.junit.platform.engine

Package org.junit.platform.engine.discovery

Package org.junit.platform.engine.support.descriptor

Package org.junit.platform.engine.support.discovery

Package org.junit.platform.engine.support.hierarchical

Package org.junit.platform.engine.support.store

Module org.junit.platform.launcher

Package org.junit.platform.launcher.core

Module org.junit.platform.testkit

Package org.junit.platform.testkit.engine

Deprecated APIs

The following tables list which APIs are currently designated as *deprecated* via `@API(status = DEPRECATED)`. You should avoid using deprecated APIs whenever possible, since such APIs will likely be removed in an upcoming release.

Module org.junit.jupiter.api

Package org.junit.jupiter.api

Package org.junit.jupiter.api.condition

Package org.junit.jupiter.api.extension

Module org.junit.jupiter.migrationsupport

Package org.junit.jupiter.migrationsupport

Package org.junit.jupiter.migrationsupport.conditions

Package org.junit.jupiter.migrationsupport.rules

Module org.junit.jupiter.params

Package org.junit.jupiter.params.provider

Package org.junit.jupiter.params.support

Module org.junit.platform.commons

Package org.junit.platform.commons.support

Package org.junit.platform.commons.support.scanning

Module org.junit.platform.engine

Package org.junit.platform.engine

Package org.junit.platform.engine.discovery

Package org.junit.platform.engine.reporting

Package org.junit.platform.engine.support.discovery

Package org.junit.platform.engine.support.store

Module org.junit.platform.launcher

Package org.junit.platform.launcher

Package org.junit.platform.launcher.core

Module org.junit.platform.testkit

Package org.junit.platform.testkit.engine

Module org.junit.vintage.engine

Package org.junit.vintage.engine

@API Tooling Support

The @API Guardian project plans to provide tooling support for publishers and consumers of APIs annotated with @API. For example, the tooling support will likely provide a means to check if JUnit APIs are being used in accordance with `@API` annotation declarations.

Release Notes

This document contains the change log for all JUnit releases since 5.14 GA.

Please refer to the User Guide for comprehensive reference documentation for programmers writing tests, extension authors, and engine authors as well as build tool and IDE vendors.

6.0.1

Date of Release: October 31, 2025

Scope: Bug fixes and enhancements since 6.0.0

For a complete list of all *closed* issues and pull requests for this release, consult the 6.0.1 milestone page in the JUnit repository on GitHub.

JUnit Platform

Bug Fixes

- The `jdk.jfr` package is now an optional import when using the `junit-platform-launcher` as an OSGi bundle.

New Features and Improvements

- Legacy documentation regarding Java 8 compatibility has been removed from the User Guide.

JUnit Jupiter

Bug Fixes

- A regression introduced in version 6.0.0 caused an exception when using `@CsvSource` or `@CsvFileSource` if the `delimiter` or `delimiterString` attribute was set to `#`. This occurred because `#` was used as the default comment character without an option to change it. To resolve this, a new `commentCharacter` attribute has been added to both annotations. Its default value remains `#`, but it can now be customized to avoid conflicts with other control characters.
- Fix `IllegalAccessError` thrown when using the Kotlin-specific `assertDoesNotThrow` assertion.
- Stop reporting discovery issues for synthetic methods, particularly in conjunction with Kotlin suspend functions.
- Fix support for test methods with the same signature as package-private methods declared in super classes in different packages.

Deprecations and Breaking Changes

- The `org.junit.jupiter.migrationsupport` module descriptor has been marked as deprecated for removal.

New Features and Improvements

- The `@CsvSource` and `@CsvFileSource` annotations now allow specifying a custom comment character using the new `commentCharacter` attribute.
- Improve error message when `@ParameterizedClass` is used with field injection without providing enough arguments.
- Allow calling the `TypedArgumentConverter` constructor for `@Nullable T` target types without having to cast class literals to `Class<@Nullable T>`.

JUnit Vintage

New Features and Improvements

- Allow disabling the reporting of discovery issues by the JUnit Vintage engine (including the one reported for its deprecation) by setting the new `junit.vintage.discovery.issue.reporting.enabled` configuration parameter to `false`.

6.0.0

Date of Release: September 30, 2025

Scope:

- Java 17 and Kotlin 2.1 baseline
- Single version number for Platform, Jupiter, and Vintage
- Use of JSpecify annotations to express nullability
- Integration of JFR functionality in `junit-platform-launcher`
- Removal of `junit-platform-runner` and `junit-platform-jfr`
- Deterministic order of `@Nested` classes
- `MethodOrderer.Default` and `ClassOrderer.Default` for `@Nested` classes
- Inheritance of `@TestMethodOrder` by enclosed `@Nested` classes
- Switch to FastCSV library for `@CsvSource` and `@CsvFileSource`
- Support for using Kotlin `suspend` functions as test methods
- New `--fail-fast` mode for ConsoleLauncher
- Support for cancelling test execution via `CancellationToken`
- Removal of various deprecated behaviors and APIs

For a complete list of all *closed* issues and pull requests for this release, consult the 6.0.0-M1, 6.0.0-M2, 6.0.0-RC1, 6.0.0-RC2, 6.0.0-RC3, and 6.0.0 milestone pages in the JUnit repository on GitHub.



Migration Guide

Please refer to the wiki for guidance on upgrading from JUnit 5.x.y to 6.0.0.

Overall Improvements

Deprecations and Breaking Changes

- Minimum required Java version is now 17.
- Minimum required Kotlin version is now 2.2.
- Platform artifacts now use the same version number as Jupiter and Vintage artifacts.

New Features and Improvements

- All JUnit modules now use JSpecify nullability annotations to indicate which method parameters, return types, etc. can be `null`.

JUnit Platform

Deprecations and Breaking Changes

- Support for running the `ConsoleLauncher` without specifying a subcommand has been removed.
- Support for non-conventional `ConsoleLauncher` options such as `--h` (rather than `-h`) or `-help` (rather than `--help`) has been removed.
- The `junit-platform-runner` module which provided the JUnit 4 based `JUnitPlatform` runner has been removed.
- The `junit-platform-jfr` module which provided custom Java Flight Recorder (JFR) events for test discovery and execution has been removed. The functionality is now available directly in the `junit-platform-launcher` module.
- Support for Maven Surefire/Failsafe versions less than 3.0.0 has been removed.
- The following deprecated APIs have been removed:
  - `ReflectionSupport.loadClass(String)` method
  - `ReflectionUtils.readFieldValue(…)` methods
  - `ReflectionUtils.getMethod(…)` method
  - `BlacklistedExceptions` class
  - `PreconditionViolationException` class (from `org.junit.platform.commons.util`)
  - `ClasspathScanningSupport` class
  - `ConfigurationParameters.size()` method
  - `MethodSelector.getMethodParameterTypes()` method
  - `NestedMethodSelector.getMethodParameterTypes()` method
  - `ReportEntry()` constructor
  - `SingleTestExecutor` class
  - `LauncherDiscoveryRequestBuilder()` constructor
  - `LegacyReportingUtils` class

    - `TestPlan.add(TestIdentifier)`, `TestPlan.getChildren(String)`, and `TestPlan.getTestIdentifier(String)` methods
    - `@UseTechnicalNames` annotation for suites
    - `EngineTestKit.execute(String, EngineDiscoveryRequest)`, `EngineTestKit.execute(TestEngine, EngineDiscoveryRequest)`, and `EngineTestKit.Builder.filters(DiscoveryFilter…)` methods
- Support for "legacy semantics" for field and method searches that used to be configurable via the `junit.platform.reflection.search.useLegacySemantics` system property has been removed. JUnit now always adheres to standard Java semantics regarding whether a given field or method is visible or overridden according to the rules of the Java language.
- The type bounds of the following methods have been changed to be more flexible and allow nullable and non-nullable types:
    - `ConfigurationParameters.get(String, Function)`
    - `NamespacedHierarchicalStore.getOrComputeIfAbsent(N, K, Function)`
    - `NamespacedHierarchicalStore.getOrComputeIfAbsent(N, K, Function, Class)`
- The `junit-platform-suite-commons` module has been removed. Its functionality is now integrated directly in the `junit-platform-suite` module.
- `ConversionSupport` now converts `String` to `Locale` using the IETF BCP 47 language tag format supported by the `Locale.forLanguageTag(String)` factory method instead of the format used by the deprecated `Locale(String)` constructor.
- All `getOrComputeIfAbsent(…)` methods in `NamespacedHierarchicalStore` have been deprecated in favor of the new `computeIfAbsent(…)` methods.
- Setting an invalid value for one of the following enum-based configuration parameters now causes test discovery to fail:
    - `junit.platform.discovery.issue.failure.phase`
    - `junit.platform.discovery.issue.severity.critical`
- The methods `findNestedClasses(…)` and `streamNestedClasses(…)` in `ReflectionSupport` now return nested classes, which are declared in the same enclosing class or interface, ordered in a deterministic but intentionally nonobvious way.
- Serialization support for `TestIdentifier` has been changed in a backwards-incompatible way in order to simplify the implementation.

New Features and Improvements

- New `Launcher.execute(LauncherExecutionRequest)` API with corresponding `LauncherExecutionRequestBuilder` to enable the addition of parameters to test executions in the future without additional overloads of `execute(…)`.
- New `discoveryRequest()` method in `LauncherDiscoveryRequestBuilder` (analogous to `executionRequest(…)` in `LauncherExecutionRequestBuilder`), which is an alias for the existing `request()` method and is intended to be used via a `static` import.
- New `LauncherDiscoveryRequestBuilder.forExecution()` method which serves as a convenience for constructing a `LauncherExecutionRequest` that contains a `LauncherDiscoveryRequest`.

- New `selectClasses(…)` and `selectClassesByName(…)` factory methods have been introduced in `DiscoverySelectors` to simplify use cases where one needs to select multiple individual test classes for the `Launcher` or `EngineTestKit`.
- New `selectors(List)` builder method for `EngineTestKit` which can be used in conjunction with the new `selectClasses(…)` and `selectClassesByName(…)` factory methods in `DiscoverySelectors`.
- New support for cancelling a running test execution via a `CancellationToken` passed to the `Launcher` as part of a `LauncherExecutionRequest` and from there to all registered test engines. Please refer to the User Guide for details and a usage example.
- Passing the `--fail-fast` option to the `execute` subcommand of the `ConsoleLauncher` now causes test execution to be cancelled after the first failed test.
- Cancellation support for implementations of `HierarchicalTestEngine` such as JUnit Jupiter, Spock, and Cucumber.
- Cancellation support for the `@Suite` test engine.
- New `TestTask.getTestDescriptor()` method for use in `HierarchicalTestExecutorService` implementations.
- New `computeIfAbsent(…)` methods in `NamespacedHierarchicalStore` to simplify working with non-nullable types.
- Convention-based conversion in `ConversionSupport` now supports factory methods and factory constructors that accept a single `CharSequence` argument in addition to the existing support for factories that accept a single `String` argument.
- Non-printable control characters in display names are now replaced with alternative representations. For example, `\n` is replaced with `<LF>`. This applies to all display names in JUnit Jupiter, `@SuiteDisplayName`, and any other test engines that subclass `AbstractTestDescriptor`. Please refer to the User Guide for details.
- Stack traces are now pruned up to the test method or lifecycle method.

JUnit Jupiter

Bug Fixes

- CSV headers are now properly supported with the default display name pattern and the explicit `{argumentsWithNames}` display name pattern for parameterized tests that utilize the `useHeadersInDisplayName` flag in `@CsvSource` and `@CsvFileSource`. Specifically, the parameter name is no longer duplicated in the display name when a CSV header is desired instead.

Deprecations and Breaking Changes

- The following deprecated APIs have been removed:
  - `MethodOrderer.Alphanumeric` class
  - `InvocationInterceptor.interceptDynamicTest(Invocation, ExtensionContext)` method
- The deprecated `junit.jupiter.tempdir.scope` configuration parameter is no longer supported.
- The `JRE` enum constants for `JAVA_8` to `JAVA_16` have been deprecated because they can no longer be used at runtime since `JAVA_17` is the new baseline. Please also manually update any values

used with the `minVersion` and `maxVersion` attributes in `@EnabledForJreRange` and `@DisabledForJreRange` or the `versions` attributes in `@EnabledOnJre` and `@DisabledOnJre` to ensure that you are no longer declaring version values less than 17.

- `@EnabledForJreRange` and `@DisabledForJreRange` now use `JAVA_17` as their default `min` value.
- For consistency with test methods, `@Nested` classes declared in the same enclosing class or interface are now ordered in a deterministic but intentionally nonobvious way.
- The `junit-jupiter-migrationsupport` module and its contained classes are now deprecated and will be removed in the next major version.
- As a result of migrating from univocity-parsers to FastCSV for `@CsvSource` and `@CsvFileSource`, root causes and messages of exceptions thrown for malformed CSV input may differ in some cases. While the overall parsing behavior remains consistent, this may affect custom error handling that relies on specific exception types or messages.
- The `lineSeparator` attribute in `@CsvFileSource` has been removed. The line separator is now automatically detected, meaning that any of `\r`, `\n`, or `\r\n` is treated as a line separator.
- Attributes such as `ignoreLeadingAndTrailingWhitespace`, `nullValues`, and others in `@CsvSource` and `@CsvFileSource` now apply to header fields as well as to regular fields.
- Extra characters after a closing quote are no longer allowed in `@CsvSource` and `@CsvFileSource`. For example, if a single quote is used as the quote character, the following CSV value `'foo□INVALID,'bar'` will now cause an exception to be thrown. This helps ensure that malformed input is not silently accepted or misinterpreted.
- Support for the `junit.jupiter.params.arguments.conversion.locale.format` configuration parameter has been removed. `Locale` conversions are now always performed using the IETF BCP 47 language tag format supported by the `Locale.forLanguageTag(String)` factory method.
- The return type of the `provideTestTemplateInvocationContexts(ExtensionContext)` method in the `TestTemplateInvocationContextProvider` interface has been changed from `Stream<TestTemplateInvocationContext>` to `Stream<? extends TestTemplateInvocationContext>`.
- The type bounds of the following methods have been changed to be more flexible and allow nullable and non-nullable types:
  - `ExtensionContext.getConfigurationParameter(String, Function)`
  - `ExtensionContext.getOrComputeIfAbsent(K, Function)`
  - `ExtensionContext.getOrComputeIfAbsent(K, Function, Class)`
- All `getOrComputeIfAbsent(…)` methods in `ExtensionContext.Store` are now deprecated in favor of the new `computeIfAbsent(…)` methods.
- Setting an invalid value for one of the following enum-based configuration parameters now causes test discovery or execution to fail:
  - `junit.jupiter.execution.parallel.mode.default`
  - `junit.jupiter.execution.parallel.mode.classes.default`
  - `junit.jupiter.execution.timeout.mode`
  - `junit.jupiter.execution.timeout.thread.mode.default`
  - `junit.jupiter.extensions.testinstantiation.extensioncontextscope.default`

    - `junit.jupiter.tempdir.cleanup.mode.default`
    - `junit.jupiter.testinstance.lifecycle.default`
- The contracts for `Executable` parameters of Kotlin-specific `assertTimeout` functions have been changed from `callsInPlace(executable, EXACTLY_ONCE)` to `callsInPlace(executable, AT_MOST_ONCE)` which might result in compilation errors.

New Features and Improvements

- Non-printable control characters in display names are now replaced with alternative representations. Please refer to the User Guide for details.
- For consistency with `@TestClassOrder`, `@TestMethodOrder` annotations specified on a test class are now inherited by its `@Nested` inner classes, recursively.
- New `MethodOrderer.Default` and `ClassOrderer.Default` types for reverting back to default ordering on a `@Nested` class and its `@Nested` inner classes when an enclosing class specifies a different orderer via `@TestMethodOrder` or `@TestClassOrder`, respectively.
- The implementation of `@CsvSource` and `@CsvFileSource` has been migrated from the no longer maintained univocity-parsers to FastCSV. This improves the consistency of CSV input handling, especially for malformed entries, and provides better error reporting and overall performance.
- Display names for `@ParameterizedClass` and `@ParameterizedTest` now consistently style name-value pairs for arguments using `name = value` formatting – for example, `fruit = apple` instead of `fruit=apple`.
- Text-based arguments in display names for parameterized tests are now quoted by default. In addition, special characters are escaped within quoted text. Please refer to the User Guide for details.
- Fallback String-to-Object Conversion for parameterized tests now supports factory methods and factory constructors that accept a single `CharSequence` argument in addition to the existing support for factories that accept a single `String` argument.
- The `Arguments` interface for parameterized tests is now officially a `@FunctionalInterface`.
- Kotlin's `suspend` modifier may now be applied to test and lifecycle methods.
- New `computeIfAbsent(…)` methods in `ExtensionContext.Store` to simplify working with non-nullable types.
- Reason strings supplied to `ConditionEvaluationResult` APIs are now officially declared as `@Nullable`.

JUnit Vintage

Deprecations and Breaking Changes

- The JUnit Vintage test engine is now deprecated and will report an INFO level discovery issue if it finds at least one JUnit 4 test class. The deprecation warning is intended to clarify the purpose of the engine: it should only be used temporarily while migrating tests to JUnit Jupiter or another testing framework with native JUnit Platform support.

New Features and Improvements

- Cancellation support for the Vintage test engine.

5.14.1

Date of Release: October 31, 2025

Scope: Bug fixes and enhancements since 5.14.0

For a complete list of all *closed* issues and pull requests for this release, consult the 5.14.1 milestone page in the JUnit repository on GitHub.

JUnit Platform

No changes.

JUnit Jupiter

Bug Fixes

- Fix support for test methods with the same signature as a package-private methods declared in super classes in different packages.

New Features and Improvements

- Improve error message when using `@ParameterizedClass` with field injection and not providing enough arguments.

JUnit Vintage

No changes.

5.14.0

Date of Release: September 30, 2025

Scope:

- Bug fixes and enhancements since 5.13.x
- Deprecations along with new APIs to ease migration to JUnit 6

For a complete list of all *closed* issues and pull requests for this release, consult the 5.14.0-RC1 and 5.14.0 milestone pages in the JUnit repository on GitHub.

For complete details consult the 5.14.0 Release Notes online.

Appendix

Reproducible Builds

JUnit aims for its non-javadoc JARs to be reproducible.

Under identical build conditions, such as Java version, repeated builds should provide the same output byte-for-byte.

This means that anyone can reproduce the build conditions of the artifacts on Maven Central/Sonatype and produce the same output artifact locally, confirming that the artifacts in the repositories were actually generated from this source code.

Dependency Metadata

Artifacts for final releases and milestones are deployed to Maven Central. Consult Sonatype's documentation for how to consume those artifacts with a build tool of your choice.

Snapshot artifacts are deployed to Sonatype's snapshots repository under /org/junit. Please refer to Sonatype's documentation for instructions on how to consume them with a build tool of your choice.

The sections below list all artifacts with their versions for the three groups: Platform, Jupiter, and Vintage. The Bill of Materials (BOM) contains a list of all of the above artifacts and their versions.



Aligning dependency versions

To ensure that all JUnit artifacts are compatible with each other, their versions should be aligned. If you rely on Spring Boot for dependency management, please see the corresponding section. Otherwise, instead of managing individual versions of the JUnit artifacts, it is recommended to apply the BOM to your project. Please refer to the corresponding sections for Maven or Gradle.

JUnit Platform

- **Group ID**: `org.junit.platform`
- **Version**: `6.0.1`
- **Artifact IDs**:

`junit-platform-commons`

Common APIs and support utilities for the JUnit Platform. Any API annotated with `@API(status = INTERNAL)` is intended solely for usage within the JUnit framework itself. *Any usage of internal APIs by external parties is not supported!*

`junit-platform-console`

Support for discovering and executing tests on the JUnit Platform from the console. See Console Launcher for details.

`junit-platform-console-standalone`

An executable *Fat JAR* that contains all dependencies is provided in Maven Central under the junit-platform-console-standalone directory. See Console Launcher for details.

`junit-platform-engine`

Public API for test engines. See Registering a TestEngine for details.

`junit-platform-launcher`

Public API for configuring and launching test plans — typically used by IDEs and build tools. See JUnit Platform Launcher API for details.

`junit-platform-reporting`

`TestExecutionListener` implementations that generate test reports — typically used by IDEs and build tools. See JUnit Platform Reporting for details.

`junit-platform-suite`

JUnit Platform Suite artifact that transitively pulls in dependencies on `junit-platform-suite-api` and `junit-platform-suite-engine` for simplified dependency management in build tools such as Gradle and Maven.

`junit-platform-suite-api`

Annotations for configuring test suites on the JUnit Platform. Supported by the JUnit Platform Suite Engine.

`junit-platform-suite-engine`

Engine that executes test suites on the JUnit Platform; only required at runtime. See JUnit Platform Suite Engine for details.

`junit-platform-testkit`

Provides support for executing a test plan for a given `TestEngine` and then accessing the results via a fluent API to verify the expected results.

JUnit Jupiter

- **Group ID**: `org.junit.jupiter`
- **Version**: `6.0.1`
- **Artifact IDs**:

`junit-jupiter`

JUnit Jupiter aggregator artifact that transitively pulls in dependencies on `junit-jupiter-api`, `junit-jupiter-params`, and `junit-jupiter-engine` for simplified dependency management in build tools such as Gradle and Maven.

`junit-jupiter-api`

JUnit Jupiter API for writing tests and extensions.

`junit-jupiter-engine`

JUnit Jupiter test engine implementation; only required at runtime.

`junit-jupiter-params`

Support for Parameterized Classes and Tests in JUnit Jupiter.

`junit-jupiter-migrationsupport`

Deprecated support for migrating from JUnit 4 to JUnit Jupiter; only required for support for JUnit 4's `@Ignore` annotation and for running selected JUnit 4 rules.

JUnit Vintage

- **Group ID**: `org.junit.vintage`
- **Version**: `6.0.1`
- **Artifact ID**:

`junit-vintage-engine`

JUnit Vintage test engine implementation that allows one to run *vintage* JUnit tests on the JUnit Platform. *Vintage* tests include those written using JUnit 3 or JUnit 4 APIs or tests written using testing frameworks built on those APIs.

Bill of Materials (BOM)

The *Bill of Materials* POM provided under the following Maven coordinates can be used to ease dependency management when referencing multiple of the above artifacts using Maven or Gradle.

- **Group ID**: `org.junit`
- **Artifact ID**: `junit-bom`
- **Version**: `6.0.1`

Dependencies

Most of the above artifacts have a dependency in their published Maven POMs on the following *@API Guardian* JAR.

- **Group ID**: `org.apiguardian`
- **Artifact ID**: `apiguardian-api`
- **Version**: `1.1.2`

In addition, most of the above artifacts have a direct or transitive dependency on the following *OpenTest4J* JAR.

- **Group ID**: `org.opentest4j`
- **Artifact ID**: `opentest4j`
- **Version**: `1.3.0`

Dependency Diagram

