JUnit 5 User Guide

Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Christian Stein

Version 5.3.2

Table of Contents

1. 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.

> ! *Translations*
>
> This document is also available in Simplified Chinese and Japanese.

1.1. What is JUnit 5?

Unlike previous versions of JUnit, JUnit 5 is composed of several different modules from three different sub-projects.

JUnit 5 = *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 build plugins for Gradle and Maven as well as a JUnit 4 based Runner for running any `TestEngine` on the platform.

JUnit Jupiter is the combination of the new programming model and extension model for writing tests and extensions in JUnit 5. 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.

1.2. Supported Java Versions

JUnit 5 requires Java 8 (or higher) at runtime. However, you can still test code that has been compiled with previous versions of the JDK.

1.3. Getting Help

Ask JUnit 5 related questions on Stack Overflow or chat with us on Gitter.

2. Installation

Artifacts for final releases and milestones are deployed to Maven Central.

Snapshot artifacts are deployed to Sonatype's snapshots repository under /org/junit.

2.1. Dependency Metadata

2.1.1. JUnit Platform

¥ Group ID: `org.junit.platform`

¥ Version: `1.3.2`

¥ Artifact IDs:

`junit-platform-commons`
: Internal common library/utilities of JUnit. These utilities are intended solely for usage within the JUnit framework itself. *Any usage by external parties is not supported.* Use at your own risk!

`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 JAR with all dependencies included is provided at Maven Central under the junit-platform-console-standalone directory. See Console Launcher for details.

`junit-platform-engine`
: Public API for test engines. See Plugging in your own Test Engine for details.

`junit-platform-launcher`
: Public API for configuring and launching test plans!Ñ!typically used by IDEs and build tools. See JUnit Platform Launcher API for details.

`junit-platform-runner`
: Runner for executing tests and test suites on the JUnit Platform in a JUnit 4 environment. See Using JUnit 4 to run the JUnit Platform for details.

`junit-platform-suite-api`
: Annotations for configuring test suites on the JUnit Platform. Supported by the JUnitPlatform runner and possibly by third-party `TestEngine` implementations.

`junit-platform-surefire-provider`
: Support for discovering and executing tests on the JUnit Platform using Maven Surefire.

2.1.2. JUnit Jupiter

¥ Group ID: `org.junit.jupiter`

¥ Version: `5.3.2`

¥ Artifact IDs:

`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 tests in JUnit Jupiter.

`junit-jupiter-migrationsupport`
: Migration support from JUnit 4 to JUnit Jupiter, only required for running selected JUnit 4 rules.

2.1.3. JUnit Vintage

- Group ID: `org.junit.vintage`
- Version: `5.3.2`
- Artifact ID:

`junit-vintage-engine`
: JUnit Vintage test engine implementation that allows to run vintage JUnit tests, i.e. tests written in the JUnit 3 or JUnit 4 style, on the new JUnit Platform.

2.1.4. 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: `5.3.2`

2.1.5. Dependencies

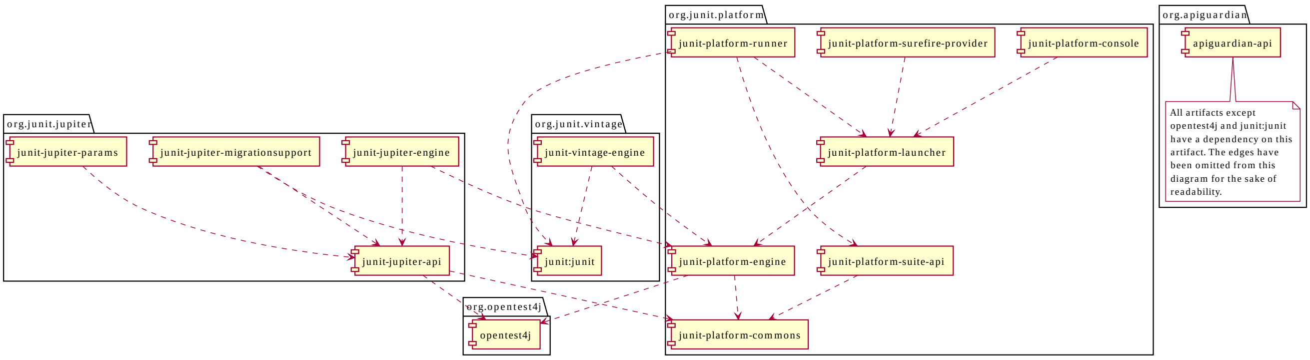
All of the above artifacts have a dependency in their published Maven POMs on the following *@API Guardian* JAR.

- Group ID: `org.apiguardian`
- Artifact ID: `apiguardian-api`
- Version: `1.0.0`

In addition, most of the above artifacts have a direct or transitive dependency to the following *OpenTest4J* JAR.

- Group ID: `org.opentest4j`
- Artifact ID: `opentest4j`
- Version: `1.1.1`

2.2. Dependency Diagram



2.3. JUnit Jupiter Sample Projects

The `junit5-samples` repository hosts a collection of sample projects based on JUnit Jupiter and JUnit Vintage. You’ll find the respective build scripts (e.g., `build.gradle`, `pom.xml`, etc.) in the projects below.

¥ For Gradle and Java, check out the `junit5-jupiter-starter-gradle` project.

¥ For Gradle and Kotlin, check out the `junit5-jupiter-starter-gradle-kotlin` project.

¥ For Gradle and Groovy, check out the `junit5-jupiter-starter-gradle-groovy` project.

¥ For Maven, check out the `junit5-jupiter-starter-maven` project.

¥ For Ant, check out the `junit5-jupiter-starter-ant` project.

3. Writing Tests

A first test case

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

import org.junit.jupiter.api.Test;

class FirstJUnit5Tests {

Ê   @Test
Ê   void myFirstTest() {
Ê       assertEquals(2, 1 + 1);
Ê   }

}
```

3.1. Annotations

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

All core annotations are located in the `org.junit.jupiter.api` package in the `junit-jupiter-api` module.

Annotation	Description
`@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*.
`@TestInstance`	Used to configure the test instance lifecycle for the annotated test class. Such annotations are *inherited*.
`@TestTemplate`	Denotes that a method is a template for test cases 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*.
`@DisplayName`	Declares a custom display name for the test class or test method. Such annotations are not *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 class; analogous to JUnit 4’s `@BeforeClass`. Such methods are *inherited* (unless they are *hidden* or *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 class; analogous to JUnit 4’s `@AfterClass`. Such methods are *inherited* (unless they are *hidden* or *overridden*) and must be `static` (unless the "per-class" test instance lifecycle is used).
`@Nested`	Denotes that the annotated class is a nested, non-static test class. `@BeforeAll` and `@AfterAll` methods cannot be used directly in a `@Nested` test class unless the "per-class" test instance lifecycle is used. 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*.

Annotation	Description
`@ExtendWith`	Used to register custom extensions. Such annotations are *inherited*.

Methods annotated with `@Test`, `@TestTemplate`, `@RepeatedTest`, `@BeforeAll`, `@AfterAll`, `@BeforeEach`, or `@AfterEach` annotations must not return a value.

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

3.1.1. 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")`.

```java
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 {
}
```

3.2. Test Classes and Methods

A *test method* is any instance method that is directly or meta-annotated with `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, or `@TestTemplate`. A *test class* is any top level or static member class that contains at least one test method.

A standard test class

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

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 {

Ê   @BeforeAll
Ê   static void initAll() {
Ê   }

Ê   @BeforeEach
Ê   void init() {
Ê   }

Ê   @Test
Ê   void succeedingTest() {
Ê   }

Ê   @Test
Ê   void failingTest() {
Ê       fail("a failing test");
Ê   }

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

Ê   @AfterEach
Ê   void tearDown() {
Ê   }

Ê   @AfterAll
Ê   static void tearDownAll() {
Ê   }

}
```

Neither test classes nor test methods need to be `public`.

3.3. Display Names

Test classes and test methods can declare custom display names!Ñ!with spaces, special characters, and even emojis!Ñ!that will be displayed by test runners and test reporting.

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

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

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

Ê   @Test
Ê   @DisplayName(" ! ¡ ! ¡ " ! ")
Ê   void testWithDisplayNameContainingSpecialCharacters() {
Ê   }

Ê   @Test
Ê   @DisplayName(" " ")
Ê   void testWithDisplayNameContainingEmoji() {
Ê   }

}
```

3.4. 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 8 lambdas. All JUnit Jupiter assertions are `static` methods in the `org.junit.jupiter.api.Assertions` class.

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

class AssertionsDemo {
```

```
Ê   @Test
Ê   void standardAssertions() {
Ê       assertEquals(2, 2);
Ê       assertEquals(4, 4, "The optional assertion message is now the last parameter.
");
Ê       assertTrue('a' < 'b', () -> "Assertion messages can be lazily evaluated -- "
Ê               + "to avoid constructing complex messages unnecessarily.");
Ê   }

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

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

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

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

Ê   @Test
Ê   void exceptionTesting() {
```

```
        Throwable exception = assertThrows(IllegalArgumentException.class, () -> {
            throw new IllegalArgumentException("a message");
        });
        assertEquals("a message", 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.
            Thread.sleep(100);
        });
    }

    private static String greeting() {
```

```
Ê        return "Hello, World!";
Ê    }

}
```

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.

```kotlin
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.assertAll
import org.junit.jupiter.api.Assertions.assertEquals
import org.junit.jupiter.api.Assertions.assertTrue
import org.junit.jupiter.api.assertThrows

class AssertionsKotlinDemo {

Ê   @Test
Ê   fun `grouped assertions`() {
Ê       assertAll("person",
Ê           { assertEquals("John", person.firstName) },
Ê           { assertEquals("Doe", person.lastName) }
Ê       )
Ê   }

Ê   @Test
Ê   fun `exception testing`() {
Ê       val exception = assertThrows<IllegalArgumentException> ("Should throw an
exception") {
Ê           throw IllegalArgumentException("a message")
Ê       }
Ê       assertEquals("a message", exception.message)
Ê   }

Ê   @Test
Ê   fun `assertions from a stream`() {
Ê       assertAll(
Ê           "people with name starting with J",
Ê           people
Ê               .stream()
Ê               .map {
Ê                   // This mapping returns Stream<() -> Unit>
Ê                   { assertTrue(it.firstName.startsWith("J")) }
Ê               }
Ê       )
Ê   }

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

3.4.1. 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 such as *matchers* 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 combination of *matchers* and a fluent API can be used to make assertions more descriptive and readable. However, JUnit Jupiter's `org.junit.jupiter.api.Assertions` class does not provide an `assertThat()` method like the one found in JUnit 4's `org.junit.Assert` class which accepts a Hamcrest `Matcher`. Instead, developers are encouraged to use the built-in support for matchers provided by third-party assertion libraries.

The following example demonstrates how to use the `assertThat()` support from Hamcrest in a JUnit Jupiter test. As long as the Hamcrest library has been added to the classpath, you can statically import methods such as `assertThat()`, `is()`, and `equalTo()` and then use them in tests like in the `assertWithHamcrestMatcher()` method below.

```
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;

class HamcrestAssertionDemo {

Ê   @Test
Ê   void assertWithHamcrestMatcher() {
Ê       assertThat(2 + 1, is(equalTo(3)));
Ê   }

}
```

Naturally, legacy tests based on the JUnit 4 programming model can continue using `org.junit.Assert#assertThat`.

3.5. Assumptions

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 8 lambdas. 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 org.junit.jupiter.api.Test;

class AssumptionsDemo {

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

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

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

Ê       // perform these assertions in all environments
Ê       assertEquals("a string", "a string");
Ê   }

}
```

3.6. 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`.

Here’s a `@Disabled` test class.

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

@Disabled
class DisabledClassDemo {
Ê   @Test
Ê   void testWillBeSkipped() {
Ê   }
}
```

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 {

Ê   @Disabled
Ê   @Test
Ê   void testWillBeSkipped() {
Ê   }

Ê   @Test
Ê   void testWillBeExecuted() {
Ê   }
}
```

3.7. Conditional Test Execution

The `ExecutionCondition` extension API in JUnit Jupiter allows developers to either *enable* or *disable* a container or test 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 containers and tests *declaratively*. See 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.

> 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.

3.7.1. Operating System Conditions

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

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

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

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

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

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

3.7.2. Java Runtime Environment Conditions

A container or test may be enabled or disabled on a particular version of the Java Runtime Environment (JRE) via the `@EnabledOnJre` and `@DisabledOnJre` annotations.

```
@Test
@EnabledOnJre(JAVA_8)
void onlyOnJava8() {
Ê   // ...
}

@Test
@EnabledOnJre({ JAVA_9, JAVA_10 })
void onJava9Or10() {
Ê   // ...
}

@Test
@DisabledOnJre(JAVA_9)
void notOnJava9() {
Ê   // ...
}
```

3.7.3. 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() {
Ê   // ...
}

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

3.7.4. 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.

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

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

3.7.5. Script-based Conditions

JUnit Jupiter provides the ability to either *enable* or *disable* a container or test depending on the evaluation of a script configured via the `@EnabledIf` or `@DisabledIf` annotation. Scripts can be written in JavaScript, Groovy, or any other scripting language for which there is support for the Java Scripting API, defined by JSR 223.

> " Conditional test execution via `@EnabledIf` and `@DisabledIf` is currently an *experimental* feature. Consult the table in Experimental APIs for details.

> ! If the logic of your script depends only on the current operating system, the current Java Runtime Environment version, a particular JVM system property, or a particular environment variable, you should consider using one of the built-in annotations dedicated to that purpose. See the previous sections of this chapter for further details.

> # If you find yourself using the same script-based condition many times, consider writing a dedicated ExecutionCondition extension in order to implement the condition in a faster, type-safe, and more maintainable manner.

```
@Test // Static JavaScript expression.
@EnabledIf("2 * 3 == 6")
void willBeExecuted() {
Ê   // ...
}

@RepeatedTest(10) // Dynamic JavaScript expression.
@DisabledIf("Math.random() < 0.314159")
void mightNotBeExecuted() {
Ê   // ...
}

@Test // Regular expression testing bound system property.
@DisabledIf("/32/.test(systemProperty.get('os.arch'))")
void disabledOn32BitArchitectures() {
Ê   assertFalse(System.getProperty("os.arch").contains("32"));
}

@Test
@EnabledIf("'CI' == systemEnvironment.get('ENV')")
void onlyOnCiServer() {
Ê   assertTrue("CI".equals(System.getenv("ENV")));
}

@Test // Multi-line script, custom engine name and custom reason.
@EnabledIf(value = {
Ê               "load('nashorn:mozilla_compat.js')",
Ê               "importPackage(java.time)",
Ê               "",
Ê               "var today = LocalDate.now()",
Ê               "var tomorrow = today.plusDays(1)",
Ê               "tomorrow.isAfter(today)"
Ê           },
Ê           engine = "nashorn",
Ê           reason = "Self-fulfilling: {result}")
void theDayAfterTomorrow() {
Ê   LocalDate today = LocalDate.now();
Ê   LocalDate tomorrow = today.plusDays(1);
Ê   assertTrue(tomorrow.isAfter(today));
}
```

Script Bindings

The following names are bound to each script context and therefore usable within the script. An *accessor* provides access to a map-like structure via a simple `String get(String name)` method.

Name	Type	Description
`systemEnvironment`	*accessor*	Operating system environment variable accessor.

Name	Type	Description
`systemProperty`	*accessor*	JVM system property accessor.
`junitConfigurationParameter`	*accessor*	Configuration parameter accessor.
`junitDisplayName`	`String`	Display name of the test or container.
`junitTags`	`Set<String>`	All tags assigned to the test or container.
`junitUniqueId`	`String`	Unique ID of the test or container.

3.8. 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.

3.8.1. Syntax Rules for Tags

- A tag must not be `null` or *blank*.
- A *trimmed* tag must not contain whitespace.
- A *trimmed* tag must not contain ISO control characters.
- A *trimmed* tag must not contain any of the following *reserved characters*.
  - `,`: *comma*
  - `(`: *left parenthesis*
  - `)`: *right parenthesis*
  - `&`: *ampersand*
  - `|`: *vertical bar*
  - `!`: *exclamation point*

> In the above context, "trimmed" means that leading and trailing whitespace characters have been removed.

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

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

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

}
```

3.9. 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 Test Classes and Methods). 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, simply 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. The "per-class" mode therefore also makes it possible to use `@BeforeAll` and `@AfterAll` methods in `@Nested` test classes.

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

3.9.1. 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, simply 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.

3.10. Nested Tests

Nested tests give the test writer more capabilities to express the relationship among several group of tests. Here's an elaborate example.

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 {

Ê   Stack<Object> stack;

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

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

Ê       @BeforeEach
Ê       void createNewStack() {
Ê           stack = new Stack<>();
Ê       }

Ê       @Test
Ê       @DisplayName("is empty")
Ê       void isEmpty() {
```

```
Ê            assertTrue(stack.isEmpty());
Ê        }

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

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

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

Ê            String anElement = "an element";

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

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

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

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

#	*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 considered to be full members of the test class family with one exception: @BeforeAll and @AfterAll methods do not work *by default*. The reason is that Java does not allow static members in inner classes. However, this restriction can be circumvented by annotating a @Nested test class with @TestInstance(Lifecycle.PER_CLASS) (see Test Instance Lifecycle).

3.11. 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 constructor or a @Test, @TestFactory, @BeforeEach, @AfterEach, @BeforeAll, or @AfterAll method 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 method parameter is of type TestInfo, the TestInfoParameterResolver will supply an instance of TestInfo corresponding to the current test as the value for the parameter. The TestInfo can then be used to retrieve information about the current test such as the test's display name, the test class, the test method, or 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 test 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.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 {

Ê   TestInfoDemo(TestInfo testInfo) {
Ê       assertEquals("TestInfo Demo", testInfo.getDisplayName());
Ê   }

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

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

Ê   @Test
Ê   void test2() {
Ê   }

}
```

¥ `RepetitionInfoParameterResolver`: if a method parameter in a `@RepeatedTest`, `@BeforeEach`, or `@AfterEach` method is of type `RepetitionInfo`, the `RepetitionInfoParameterResolver` will supply an instance of `RepetitionInfo`. `RepetitionInfo` can then be used to retrieve information about the current repetition and the total number of repetitions for the corresponding `@RepeatedTest`. Note, however, that `RepetitionInfoParameterResolver` is not registered outside the context of a `@RepeatedTest`. See Repeated Test Examples.

¥ `TestReporterParameterResolver`: if a 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. The data can be consumed through `TestExecutionListener.reportingEntryPublished()` and thus be viewed by 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. Using @RunWith(JUnitPlatform.class) will even output all reported entries to stdout.

```
class TestReporterDemo {

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

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

Ê   @Test
Ê   void reportMultipleKeyValuePairs(TestReporter testReporter) {
Ê       testReporter.publishEntry(
Ê           Map.of(
Ê               "user name", "dk38",
Ê               "award year", "1974"
Ê           ));
Ê   }

}
```

> 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 {

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

Ê   @Test
Ê   void injectsDouble(@Random double d) {
Ê       assertEquals(0.0, d, 1.0);
Ê   }

}
```

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

3.12. 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 {

Ê   static final Logger LOG = Logger.getLogger(TestLifecycleLogger.class.getName());

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

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

Ê   @BeforeEach
Ê   default void beforeEachTest(TestInfo testInfo) {
Ê       LOG.info(() -> String.format("About to execute [%s]",
Ê           testInfo.getDisplayName()));
Ê   }

Ê   @AfterEach
Ê   default void afterEachTest(TestInfo testInfo) {
Ê       LOG.info(() -> String.format("Finished executing [%s]",
Ê           testInfo.getDisplayName()));
Ê   }

}
```

```
interface TestInterfaceDynamicTestsDemo {

Ê   @TestFactory
Ê   default Collection<DynamicTest> dynamicTestsFromCollection() {
Ê       return Arrays.asList(
Ê           dynamicTest("1st dynamic test in test interface", () -> assertTrue(true)),
Ê           dynamicTest("2nd dynamic test in test interface", () -> assertEquals(4, 2
* 2))
Ê       );
Ê   }

}
```

`@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,
Ê       TimeExecutionLogger, TestInterfaceDynamicTestsDemo {

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

}
```

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

```
:junitPlatformTest
INFO  example.TestLifecycleLogger - Before all tests
INFO  example.TestLifecycleLogger - About to execute [dynamicTestsFromCollection()]
INFO  example.TimingExtension - Method [dynamicTestsFromCollection] took 13 ms.
INFO  example.TestLifecycleLogger - Finished executing [dynamicTestsFromCollection()]
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

Test run finished after 190 ms
[         3 containers found      ]
[         0 containers skipped    ]
[         3 containers started    ]
[         0 containers aborted    ]
[         3 containers successful ]
[         0 containers failed     ]
[         3 tests found           ]
[         0 tests skipped         ]
[         3 tests started         ]
[         0 tests aborted         ]
[         3 tests successful      ]
[         0 tests failed          ]

BUILD SUCCESSFUL
```

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> {

Ê   T createValue();

}
```

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

Ê   T createNotEqualValue();

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

Ê   @Test
Ê   default void valueDoesNotEqualNull() {
Ê       T value = createValue();
Ê       assertFalse(value.equals(null));
Ê   }

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

}
```

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

Ê   T createSmallerValue();

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

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

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

}
```

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.

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

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

Ê   @Override
Ê   public String createSmallerValue() {
Ê       return "bar"; // 'b' < 'f' in "foo"
Ê   }

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

}
```

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

3.13. Repeated Tests

JUnit Jupiter provides the ability to repeat a test a specified number of times simply 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() {
Ê   // ...
}
```

In addition to specifying the number of repetitions, 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 and the total number of repetitions programmatically, a developer can choose to have an instance of `RepetitionInfo` injected into a `@RepeatedTest`, `@BeforeEach`, or `@AfterEach` method.

3.13.1. Repeated Test Examples

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

The `repeatedTest()` method is identical to 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.

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!Ñ!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 itÕs 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 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
```

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

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 {

Ê   private Logger logger = // ...

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

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

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

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

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

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

}
```

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 %
&   # $ 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 %
```

3.14. Parameterized Tests

Parameterized tests make it possible to run a test multiple times with different arguments. They are declared just like regular `@Test` methods but use the `@ParameterizedTest` annotation instead. In addition, you must declare at least one *source* that will provide the arguments for each invocation and then *consume* the arguments in the test method.

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) {
Ê   assertTrue(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] racecar %
 # $ [2] radar %
 ' $ [3] able was I ere I saw elba %
```

> Parameterized tests are currently an *experimental* feature. Consult the table in Experimental APIs for details.

3.14.1. Required Setup

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

3.14.2. Consuming Arguments

Parameterized test methods typically *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 arguments* 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 argument* 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`.

3.14.3. 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.

@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
- 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);
}
```

@EnumSource

@EnumSource provides a convenient way to use Enum constants. The annotation provides an optional names parameter that lets you specify which constants shall be used. If omitted, all constants will be used like in the following example.

```
@ParameterizedTest
@EnumSource(TimeUnit.class)
void testWithEnumSource(TimeUnit timeUnit) {
    assertNotNull(timeUnit);
}
```

```
@ParameterizedTest
@EnumSource(value = TimeUnit.class, names = { "DAYS", "HOURS" })
void testWithEnumSourceInclude(TimeUnit timeUnit) {
    assertTrue(EnumSet.of(TimeUnit.DAYS, TimeUnit.HOURS).contains(timeUnit));
}
```

The @EnumSource annotation also provides an optional mode parameter 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(value = TimeUnit.class, mode = EXCLUDE, names = { "DAYS", "HOURS" })
void testWithEnumSourceExclude(TimeUnit timeUnit) {
Ê   assertFalse(EnumSet.of(TimeUnit.DAYS, TimeUnit.HOURS).contains(timeUnit));
Ê   assertTrue(timeUnit.name().length() > 5);
}
```

```
@ParameterizedTest
@EnumSource(value = TimeUnit.class, mode = MATCH_ALL, names = "^(M|N).+SECONDS$")
void testWithEnumSourceRegex(TimeUnit timeUnit) {
Ê   String name = timeUnit.name();
Ê   assertTrue(name.startsWith("M") || name.startsWith("N"));
Ê   assertTrue(name.endsWith("SECONDS"));
}
```

@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`. In addition, such factory methods must not accept any arguments.

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 `@ParameterizedTest` method. 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 an array of primitives. 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 test method accepts a single argument.

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 testWithSimpleMethodSource(String argument) {
Ê   assertNotNull(argument);
}

static Stream<String> stringProvider() {
Ê   return Stream.of("foo", "bar");
}
```

If you do not explicitly provide a factory method name via `@MethodSource`, JUnit Jupiter will search for a *factory* method that has the same name as the current `@ParameterizedTest` method by convention. This is demonstrated in the following example.

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

static Stream<String> testWithSimpleMethodSourceHavingNoValue() {
Ê   return Stream.of("foo", "bar");
}
```

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

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

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

If a parameterized 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(ObjectÉ)` is a static factory method defined in the `Arguments` interface. In addition, `Arguments.of(ObjectÉ)` may be used as an alternative to `arguments(ObjectÉ)`.

```
@ParameterizedTest
@MethodSource("stringIntAndListProvider")
void testWithMultiArgMethodSource(String str, int num, List<String> list) {
Ê   assertEquals(3, str.length());
Ê   assertTrue(num >=1 && num <=2);
Ê   assertEquals(2, list.size());
}

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

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 {

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

class StringsProviders {

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

@CsvSource

`@CsvSource` allows you to express argument lists as comma-separated values (i.e., `String` literals).

```
@ParameterizedTest
@CsvSource({ "foo, 1", "bar, 2", "'baz, qux', 3" })
void testWithCsvSource(String first, int second) {
Ê   assertNotNull(first);
Ê   assertNotEquals(0, second);
}
```

`@CsvSource` uses a single quote `'` as its quote character. See the `'baz, qux'` value in the example above and in the table below. An empty, quoted value `''` results in an empty `String`; whereas, an entirely *empty* value is interpreted as a `null` reference. An `ArgumentConversionException` is raised if the target type of a `null` reference is a primitive type.

Example Input	Resulting Argument List
`@CsvSource({ "foo, bar" })`	`"foo"`, `"bar"`
`@CsvSource({ "foo, 'baz, qux'" })`	`"foo"`, `"baz, qux"`
`@CsvSource({ "foo, ''" })`	`"foo"`, `""`

Example Input	Resulting Argument List
`@CsvSource({ "foo, " })`	`"foo"`, `null`

@CsvFileSource

`@CsvFileSource` lets you use CSV files from the classpath. Each line from a CSV file results in one invocation of the parameterized test.

```
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSource(String first, int second) {
Ê   assertNotNull(first);
Ê   assertNotEquals(0, second);
}
```

two-column.csv

```
Country, reference
Sweden, 1
Poland, 2
"United States of America", 3
```

> # In contrast to the syntax used in `@CsvSource`, `@CsvFileSource` uses a double quote `"` as the quote character. See the `"United States of America"` value in the example above. An empty, quoted value `""` results in an empty `String`; whereas, an entirely *empty* value is interpreted as a `null` reference. An `ArgumentConversionException` is raised if the target type of a `null` reference is a primitive type.

@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) {
Ê   assertNotNull(argument);
}
```

```
public class MyArgumentsProvider implements ArgumentsProvider {

Ê   @Override
Ê   public Stream<? extends Arguments> provideArguments(ExtensionContext context) {
Ê       return Stream.of("foo", "bar").map(Arguments::of);
Ê   }
}
```

3.14.4. Argument Conversion

Widening Conversion

JUnit Jupiter supports Widening Primitive Conversion for arguments supplied to a `@ParameterizedTest`. For example, a parameterized test 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 `@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(TimeUnit argument) {
Ê   assertNotNull(argument.name());
}
```

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

Target Type	Example
`boolean`/`Boolean`	`"true" # true`
`byte`/`Byte`	`"1" # (byte) 1`
`char`/`Character`	`"o" # 'o'`
`short`/`Short`	`"1" # (short) 1`
`int`/`Integer`	`"1" # 1`

Target Type	Example
`long`/`Long`	`"1"` → `1L`
`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`	`"http://junit.org/"` → `URI.create("http://junit.org/")`
`java.net.URL`	`"http://junit.org/"` → `new URL("http://junit.org/")`
`java.nio.charset.Charset`	`"UTF-8"` → `Charset.forName("UTF-8")`
`java.nio.file.Path`	`"/path/to/file"` → `Paths.get("/path/to/file")`
`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.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)`

Target Type	Example
`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.util.Currency`	`"JPY"` # `Currency.getInstance("JPY")`
`java.util.Locale`	`"en"` # `new Locale("en")`
`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 a single `String` 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 single `String` 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 {

Ê   private final String title;

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

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

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

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(TimeUnit.class)
void testWithExplicitArgumentConversion(
Ê       @ConvertWith(ToStringArgumentConverter.class) String argument) {

Ê   assertNotNull(TimeUnit.valueOf(argument));
}
```

```
public class ToStringArgumentConverter extends SimpleArgumentConverter {

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

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(
Ê       @JavaTimeConversionPattern("dd.MM.yyyy") LocalDate argument) {

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

3.14.5. Argument Aggregation

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

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.

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

Ê   if (person.getFirstName().equals("Jane")) {
Ê       assertEquals(Gender.F, person.getGender());
Ê   }
Ê   else {
Ê       assertEquals(Gender.M, person.getGender());
Ê   }
Ê   assertEquals("Doe", person.getLastName());
Ê   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 a `@ParameterizedTest` method's arguments using an `ArgumentsAccessor`, JUnit Jupiter also supports the usage of custom, reusable *aggregators*.

To use a custom aggregator simply implement the `ArgumentsAggregator` interface and register it via the `@AggregateWith` annotation on a compatible parameter in the `@ParameterizedTest` method. 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({
Ê   "Jane, Doe, F, 1990-05-20",
Ê   "John, Doe, M, 1990-10-22"
})
void testWithArgumentsAggregator(@AggregateWith(PersonAggregator.class) Person person)
{
Ê   // perform assertions against person
}
```

```
public class PersonAggregator implements ArgumentsAggregator {
Ê   @Override
Ê   public Person aggregateArguments(ArgumentsAccessor arguments, ParameterContext
context) {
Ê       return new Person(arguments.getString(0),
Ê                         arguments.getString(1),
Ê                         arguments.get(2, Gender.class),
Ê                         arguments.get(3, LocalDate.class));
Ê   }
}
```

If you find yourself repeatedly declaring `@AggregateWith(MyTypeAggregator.class)` for multiple parameterized test 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({
Ê   "Jane, Doe, F, 1990-05-20",
Ê   "John, Doe, M, 1990-10-22"
})
void testWithCustomAggregatorAnnotation(@CsvToPerson Person person) {
Ê   // perform assertions against person
}
```

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

3.14.6. Customizing Display Names

By default, the display name of a parameterized test invocation contains the invocation index and the `String` representation of all arguments for that specific invocation. However, you can customize invocation display names via the `name` attribute of the `@ParameterizedTest` annotation like in the following example.

```
@DisplayName("Display name of container")
@ParameterizedTest(name = "{index} ==> first=''{0}'', second={1}")
@CsvSource({ "foo, 1", "bar, 2", "'baz, qux', 3" })
void testWithCustomDisplayNames(String first, int second) {
}
```

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

```
Display name of container %
 # $  1 ==> first='foo', second=1 %
 # $  2 ==> first='bar', second=2 %
 ' $  3 ==> first='baz, qux', second=3 %
```

The following placeholders are supported within custom display names.

Placeholder	Description
`{index}`	the current invocation index (1-based)
`{arguments}`	the complete, comma-separated arguments list
`{0}`, `{1}`, É	an individual argument

3.14.7. Lifecycle and Interoperability

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 argument 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) {
Ê   // ...
}

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

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

3.15. Test Templates

A `@TestTemplate` method is not a regular test case but rather a template for test cases. 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.

3.16. 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 `Stream`, `Collection`, `Iterable`, `Iterator`, or array of `DynamicNode` instances. 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.

As of JUnit Jupiter 5.3.2, dynamic tests must always be created by factory methods; however, this might be complemented by a registration facility in a later release.

> Dynamic tests are currently an *experimental* feature. Consult the table in Experimental APIs for details.

3.16.1. Dynamic Test Examples

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

The first method returns an invalid return type. Since an invalid return type cannot be detected at compile time, a `JUnitException` is thrown when it is detected at runtime.

The next five methods are very simple examples that demonstrate the generation of a `Collection`, `Iterable`, `Iterator`, 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 easy it is 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 last method generates a nested hierarchy of dynamic tests utilizing `DynamicContainer`.

```
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 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 {

Ê   // This will result in a JUnitException!
Ê   @TestFactory
Ê   List<String> dynamicTestsWithInvalidReturnType() {
Ê       return Arrays.asList("Hello");
Ê   }

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

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

Ê   @TestFactory
Ê   Iterator<DynamicTest> dynamicTestsFromIterator() {
Ê       return Arrays.asList(
Ê           dynamicTest("5th dynamic test", () -> assertTrue(true)),
Ê           dynamicTest("6th dynamic test", () -> assertEquals(4, 2 * 2))
```

```
        ).iterator();
    }

    @TestFactory
    DynamicTest[] dynamicTestsFromArray() {
        return new DynamicTest[] {
            dynamicTest("7th dynamic test", () -> assertTrue(true)),
            dynamicTest("8th dynamic test", () -> assertEquals(4, 2 * 2))
        };
    }

    @TestFactory
    Stream<DynamicTest> dynamicTestsFromStream() {
        return Stream.of("A", "B", "C")
            .map(str -> dynamicTest("test" + str, () -> { /* ... */ }));
    }

    @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, () -> assertTrue(n % 2 == 0)));
    }

    @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
```

```
);

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

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

}
```

3.17. Parallel Execution

By default, JUnit Jupiter tests are run sequentially in a single thread. Running tests in parallel, e.g. to speed up execution, is available as an opt-in feature since version 5.3. To enable parallel execution, simply set the `junit.jupiter.execution.parallel.enabled` configuration parameter to `true`, e.g. in `junit-platform.properties` (see Configuration Parameters for other options).

Once enabled, the JUnit Jupiter engine will execute tests on all levels of the test tree fully in parallel according to the provided configuration while observing the declarative synchronization mechanisms. Please note that the Capturing Standard Output/Error feature needs to enabled separately.

> Parallel test execution is 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.

3.17.1. Configuration

Properties like 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, simply 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`).

`fixed`
: Uses the mandatory `junit.jupiter.execution.parallel.config.fixed.parallelism` configuration parameter as desired parallelism.

`custom`
: Allows 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, i.e. the desired parallelism will equal the number of available processors/cores.

3.17.2. Synchronization

In the `org.junit.jupiter.api.parallel` package, JUnit Jupiter provides two annotation-based declarative mechanisms to change the execution mode and allow for synchronization when using shared resources in different tests.

If parallel execution is enabled, all classes and methods are executed concurrently by default. You can change the execution mode for the annotated element and its subelements (if any) by using the `@Execution` annotation. 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 constraint forces execution in the same thread.

In addition, the `@ResourceLock` annotation allows to declare that a test class or method uses a specific shared resource that requires synchronized access to ensure reliable test execution.

If the tests in the following example were run in parallel they would be flaky, i.e. sometimes pass and other times fail, because of the inherent race condition of writing and then reading the same system property.

```java
@Execution(CONCURRENT)
class SharedResourcesDemo {

Ê   private Properties backup;

Ê   @BeforeEach
Ê   void backup() {
Ê       backup = new Properties();
Ê       backup.putAll(System.getProperties());
Ê   }

Ê   @AfterEach
Ê   void restore() {
Ê       System.setProperties(backup);
Ê   }

Ê   @Test
Ê   @ResourceLock(value = SYSTEM_PROPERTIES, mode = READ)
Ê   void customPropertyIsNotSetByDefault() {
Ê       assertNull(System.getProperty("my.prop"));
Ê   }

Ê   @Test
Ê   @ResourceLock(value = SYSTEM_PROPERTIES, mode = READ_WRITE)
Ê   void canSetCustomPropertyToFoo() {
Ê       System.setProperty("my.prop", "foo");
Ê       assertEquals("foo", System.getProperty("my.prop"));
Ê   }

Ê   @Test
Ê   @ResourceLock(value = SYSTEM_PROPERTIES, mode = READ_WRITE)
Ê   void canSetCustomPropertyToBar() {
Ê       System.setProperty("my.prop", "bar");
Ê       assertEquals("bar", System.getProperty("my.prop"));
Ê   }
}
```

When access to shared resources is declared using this annotation, the JUnit Jupiter engine uses this information to ensure that no conflicting tests are run in parallel.

In addition to the string that uniquely identifies the used resource, you may specify an access mode. Two tests that require `READ` access to a resource may run in parallel with each other but not while any other test that requires `READ_WRITE` access is running.

4. Running Tests

4.1. IDE Support

4.1.1. IntelliJ IDEA

IntelliJ IDEA supports running tests on the JUnit Platform since version 2016.2. For details please see the post on the IntelliJ IDEA blog. Note, however, that it is recommended to use IDEA 2017.3 or newer since these newer versions of IDEA will download the following JARs automatically based on the API version used in the project: `junit-platform-launcher`, `junit-jupiter-engine`, and `junit-vintage-engine`.

> IntelliJ IDEA releases prior to IDEA 2017.3 bundle specific versions of JUnit 5. Thus, if you want to use a newer version of JUnit Jupiter, execution of tests within the IDE might fail due to version conflicts. In such cases, please follow the instructions below to use a newer version of JUnit 5 than the one bundled with IntelliJ IDEA.

In order to use a different JUnit 5 version (e.g., 5.3.2), 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

```
// Only needed to run tests in a version of IntelliJ IDEA that bundles older versions
testRuntime("org.junit.platform:junit-platform-launcher:1.3.2")
testRuntime("org.junit.jupiter:junit-jupiter-engine:5.3.2")
testRuntime("org.junit.vintage:junit-vintage-engine:5.3.2")
```

Additional Maven Dependencies

```
<!-- Only needed to run tests in a version of IntelliJ IDEA that bundles older
versions -->
<dependency>
Ê    <groupId>org.junit.platform</groupId>
Ê    <artifactId>junit-platform-launcher</artifactId>
Ê    <version>1.3.2</version>
Ê    <scope>test</scope>
</dependency>
<dependency>
Ê    <groupId>org.junit.jupiter</groupId>
Ê    <artifactId>junit-jupiter-engine</artifactId>
Ê    <version>5.3.2</version>
Ê    <scope>test</scope>
</dependency>
<dependency>
Ê    <groupId>org.junit.vintage</groupId>
Ê    <artifactId>junit-vintage-engine</artifactId>
Ê    <version>5.3.2</version>
Ê    <scope>test</scope>
</dependency>
```

4.1.2. Eclipse

Eclipse IDE offers support for the JUnit Platform since the Eclipse Oxygen.1a (4.7.1a) release.

For more information on using JUnit 5 in Eclipse consult the official *Eclipse support for JUnit 5* section of the Eclipse Project Oxygen.1a (4.7.1a) - New and Noteworthy documentation.

4.1.3. Other IDEs

At the time of this writing, there is no direct support for running tests on the JUnit Platform within IDEs other than IntelliJ IDEA and Eclipse. However, the JUnit team provides two intermediate solutions so that you can go ahead and try out JUnit 5 within your IDE today. You can use the Console Launcher manually or execute tests with a JUnit 4 based Runner.

4.2. Build Support

4.2.1. Gradle

Starting with version 4.6, Gradle provides native support for executing tests on the JUnit Platform. To enable it, you just need to specify `useJUnitPlatform()` within a `test` task declaration in `build.gradle`:

```
test {
Ê   useJUnitPlatform()
}
```

Filtering by tags or engines is also supported:

```
test {
Ê   useJUnitPlatform {
Ê       includeTags 'fast', 'smoke & feature-a'
Ê       // excludeTags 'slow', 'ci'
Ê       includeEngines 'junit-jupiter'
Ê       // excludeEngines 'junit-vintage'
Ê   }
}
```

Please refer to the official Gradle documentation for a comprehensive list of options.

> *The JUnit Platform Gradle Plugin has been discontinued*
>
> The very basic `junit-platform-gradle-plugin` developed by the JUnit team was deprecated in JUnit Platform 1.2 and discontinued in 1.3. Please switch to Gradle’s standard `test` task.

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 {
Ê   // ...
Ê   systemProperty 'junit.jupiter.conditions.deactivate', '*'
Ê   systemProperties = [
Ê       'junit.jupiter.extensions.autodetection.enabled': 'true',
Ê       'junit.jupiter.testinstance.lifecycle.default': 'per_class'
Ê   ]
Ê   // ...
}
```

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 `testCompile` dependency on the JUnit Jupiter API and a `testRuntime` dependency on the JUnit Jupiter `TestEngine` implementation similar to the following.

```
dependencies {
Ê   testCompile("org.junit.jupiter:junit-jupiter-api:5.3.2")
Ê   testRuntime("org.junit.jupiter:junit-jupiter-engine:5.3.2")
}
```

The JUnit Platform can run JUnit 4 based tests as long as you configure a `testCompile` dependency on JUnit 4 and a `testRuntime` dependency on the JUnit Vintage `TestEngine` implementation similar to the following.

```
dependencies {
Ê   testCompile("junit:junit:4.12")
Ê   testRuntime("org.junit.vintage:junit-vintage-engine:5.3.2")
}
```

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'
}
```

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 an additional dependency to the runtime classpath.

4.2.2. Maven

> # The custom `junit-platform-surefire-provider`, which was originally developed by the JUnit team, has been deprecated and is scheduled to be removed in JUnit Platform 1.4. Please use Maven Surefire's native support instead.

Starting with version 2.22.0, Maven Surefire provides native support for executing tests on the JUnit Platform. The `pom.xml` file in the `junit5-jupiter-starter-maven` project demonstrates how to use it and can serve as a starting point for configuring your Maven build.

Configuring Test Engines

In order to have Maven Surefire 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.

```
<build>
Ê   <plugins>
Ê       <plugin>
Ê           <artifactId>maven-surefire-plugin</artifactId>
Ê           <version>2.22.0</version>
Ê       </plugin>
Ê   </plugins>
</build>
...
<dependencies>
Ê   ...
Ê   <dependency>
Ê       <groupId>org.junit.jupiter</groupId>
Ê       <artifactId>junit-jupiter-api</artifactId>
Ê       <version>5.3.2</version>
Ê       <scope>test</scope>
Ê   </dependency>
Ê   <dependency>
Ê       <groupId>org.junit.jupiter</groupId>
Ê       <artifactId>junit-jupiter-engine</artifactId>
Ê       <version>5.3.2</version>
Ê       <scope>test</scope>
Ê   </dependency>
Ê   ...
</dependencies>
...
```

Maven Surefire 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.

```
...
<build>
Ê   <plugins>
Ê       <plugin>
Ê           <artifactId>maven-surefire-plugin</artifactId>
Ê           <version>2.22.0</version>
Ê       </plugin>
Ê   </plugins>
</build>
...
<dependencies>
Ê   ...
Ê   <dependency>
Ê       <groupId>junit</groupId>
Ê       <artifactId>junit</artifactId>
Ê       <version>4.12</version>
Ê       <scope>test</scope>
Ê   </dependency>
Ê   <dependency>
Ê       <groupId>org.junit.vintage</groupId>
Ê       <artifactId>junit-vintage-engine</artifactId>
Ê       <version>5.3.2</version>
Ê       <scope>test</scope>
Ê   </dependency>
Ê   ...
</dependencies>
...
```

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>
Ê   <plugins>
Ê       <plugin>
Ê           <artifactId>maven-surefire-plugin</artifactId>
Ê           <version>2.22.0</version>
Ê           <configuration>
Ê               <excludes>
Ê                   <exclude/>
Ê               </excludes>
Ê           </configuration>
Ê       </plugin>
Ê   </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>
Ê   <plugins>
Ê       <plugin>
Ê           <artifactId>maven-surefire-plugin</artifactId>
Ê           <version>2.22.0</version>
Ê           <configuration>
Ê               <groups>acceptance | !feature-a</groups>
Ê               <excludedGroups>integration, regression</excludedGroups>
Ê           </configuration>
Ê       </plugin>
Ê   </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>
Ê   <plugins>
Ê       <plugin>
Ê           <artifactId>maven-surefire-plugin</artifactId>
Ê           <version>2.22.0</version>
Ê           <configuration>
Ê               <properties>
Ê                   <configurationParameters>
Ê                       junit.jupiter.conditions.deactivate = *
Ê                       junit.jupiter.extensions.autodetection.enabled = true
Ê                       junit.jupiter.testinstance.lifecycle.default = per_class
Ê                   </configurationParameters>
Ê               </properties>
Ê           </configuration>
Ê       </plugin>
Ê   </plugins>
</build>
...
```

4.2.3. Ant

Starting with version `1.10.3` of Ant, a new `junitlauncher` task has been introduced to provide 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 close 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.

> # The version of the `junitlauncher` task shipped in Ant 1.10.3 provides basic, minimal support for launching the JUnit Platform. Additional enhancements (including support for forking the tests in a separate JVM) will be available in subsequent releases of Ant.

The `build.xml` file in the `junit5-jupiter-starter-ant` project demonstrates how to use it 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., `org.myapp.test.MyFirstJUnit5Test`).

```
<path id="test.classpath">
Ê   <!-- The location where you have your compiled classes -->
Ê   <pathelement location="${build.classes.dir}" />
</path>

<!-- ... -->

<junitlauncher>
Ê   <classpath refid="test.classpath" />
Ê   <test name="org.myapp.test.MyFirstJUnit5Test" />
</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">
Ê   <!-- The location where you have your compiled classes -->
Ê   <pathelement location="${build.classes.dir}" />
</path>
....
<junitlauncher>
Ê   <classpath refid="test.classpath" />
Ê   <testclasses outputdir="${output.dir}">
Ê       <fileset dir="${build.classes.dir}">
Ê           <include name="org/example/**/demo/**/" />
Ê       </fileset>
Ê       <fileset dir="${some.other.dir}">
Ê           <include name="org/myapp/**/" />
Ê       </fileset>
Ê   </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.

4.3. 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 `junit-platform-console-standalone-1.3.2.jar` with all dependencies included is published in the central Maven repository under the junit-platform-console-standalone directory. You can run the standalone `ConsoleLauncher` as shown below.

`java -jar junit-platform-console-standalone-1.3.2.jar <Options>`

Here's an example of its output:

```
 # $ JUnit Vintage
 &   ' $ example.JUnit4Tests
 &      ' $ standardJUnit4Test %
 ' $ JUnit Jupiter
Ê   # $ StandardTests
Ê   &   # $ succeedingTest() %
Ê   &   ' $ skippedTest() ( for demonstration purposes
Ê   ' $ A special test case
Ê      # $ Custom test name containing spaces %
Ê      # $  ! ¡ ! ¡ " !  %
Ê      ' $  "  %

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          ]
```

#

Exit Code

The `ConsoleLauncher` exits with a status code of `1` 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`. Otherwise the exit code is `0`.

4.3.1. Options

```
Usage: ConsoleLauncher [-h] [--disable-ansi-colors] [--fail-if-no-tests] [--scan-
modules]
Ê                      [--scan-classpath[=PATH[;|:PATH...]]]... [--details=MODE]
Ê                      [--details-theme=THEME] [--reports-dir=DIR]
Ê                      [--config=KEY=VALUE]... [--exclude-package=PKG]...
Ê                      [--include-package=PKG]... [-c=CLASS]... [-
```

```
cp=PATH[;|:PATH...]]...
Ê                       [-d=DIR]... [-e=ID]... [-E=ID]... [-f=FILE]... [-m=NAME]...
Ê                       [-n=PATTERN]... [-N=PATTERN]... [-o=NAME]... [-p=PKG]...
Ê                       [-r=RESOURCE]... [-t=TAG]... [-T=TAG]... [-u=URI]...
Launches the JUnit Platform from the console.
Ê -h, --help                Display help information.
Ê     --disable-ansi-colors  Disable ANSI colors in output (not supported by all
terminals).
Ê     --details=MODE         Select an output details mode for when tests are
executed. Use
Ê                              one of: none, summary, flat, tree, verbose. If 'none'
is
Ê                              selected, then only the summary and test failures are
shown.
Ê                              Default: tree.
Ê     --details-theme=THEME  Select an output details tree theme for when tests are
executed.
Ê                              Use one of: ascii, unicode. Default: unicode.
Ê     -cp, --classpath, --class-path=PATH[;|:PATH...]
Ê                            Provide additional classpath entries -- for example, for
adding
Ê                              engines and their dependencies. This option can be
repeated.
Ê     --fail-if-no-tests     Fail and return exit status code 2 if no tests are found.
Ê     --reports-dir=DIR      Enable report output into a specified local directory
(will be
Ê                              created if it does not exist).
Ê     --scan-modules         EXPERIMENTAL: Scan all resolved modules for test
discovery.
Ê -o, --select-module=NAME   EXPERIMENTAL: Select single module for test discovery.
This
Ê                              option can be repeated.
Ê     --scan-classpath, --scan-class-path[=PATH[;|:PATH...]]
Ê                            Scan all directories on the classpath or explicit
classpath
Ê                              roots. Without arguments, only directories on the
system
Ê                              classpath as well as additional classpath entries
supplied via
Ê                              -cp (directories and JAR files) are scanned. Explicit
classpath
Ê                              roots that are not on the classpath will be silently
ignored.
Ê                              This option can be repeated.
Ê -u, --select-uri=URI       Select a URI for test discovery. This option can be
repeated.
Ê -f, --select-file=FILE     Select a file for test discovery. This option can be
repeated.
Ê -d, --select-directory=DIR Select a directory for test discovery. This option can be
Ê                              repeated.
Ê -p, --select-package=PKG   Select a package for test discovery. This option can be
```

```
repeated.
Ê -c, --select-class=CLASS   Select a class for test discovery. This option can be
repeated.
Ê -m, --select-method=NAME   Select a method for test discovery. This option can be
repeated.
Ê -r, --select-resource=RESOURCE
Ê                            Select a classpath resource for test discovery. This
option can
Ê                              be repeated.
Ê -n, --include-classname=PATTERN
Ê                            Provide a regular expression to include only classes
whose fully
Ê                              qualified names match. To avoid loading classes
unnecessarily,
Ê                              the default pattern only includes class names that
begin with
Ê                              "Test" or end with "Test" or "Tests". When this option
is
Ê                              repeated, all patterns will be combined using OR
semantics.
Ê                              Default: [^(Test.*|.+[.$]Test.*|.*Tests?)$]
Ê -N, --exclude-classname=PATTERN
Ê                            Provide a regular expression to exclude those classes
whose fully
Ê                              qualified names match. When this option is repeated,
all
Ê                              patterns will be combined using OR semantics.
Ê     --include-package=PKG  Provide a package to be included in the test run. This
option can
Ê                              be repeated.
Ê     --exclude-package=PKG  Provide a package to be excluded from the test run. This
option
Ê                              can be repeated.
Ê -t, --include-tag=TAG      Provide a tag or tag expression to include only tests
whose tags
Ê                              match. When this option is repeated, all patterns will
be
Ê                              combined using OR semantics.
Ê -T, --exclude-tag=TAG      Provide a tag or tag expression to exclude those tests
whose tags
Ê                              match. When this option is repeated, all patterns will
be
Ê                              combined using OR semantics.
Ê -e, --include-engine=ID    Provide the ID of an engine to be included in the test
run. This
Ê                              option can be repeated.
Ê -E, --exclude-engine=ID    Provide the ID of an engine to be excluded from the test
run.
Ê                              This option can be repeated.
Ê     --config=KEY=VALUE     Set a configuration parameter for test discovery and
execution.
```

Ê This option can be repeated.

4.3.2. 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.

Since version 1.3, 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!Ñ!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.

4.4. Using JUnit 4 to run the JUnit Platform

The `JUnitPlatform` runner is a JUnit 4 based `Runner` which enables you to run any test whose programming model is supported on the JUnit Platform in a JUnit 4 environment!Ñ!for example, a JUnit Jupiter test class.

Annotating a class with `@RunWith(JUnitPlatform.class)` allows it to be run with IDEs and build systems that support JUnit 4 but do not yet support the JUnit Platform directly.

Since the JUnit Platform has features that JUnit 4 does not have, the runner is only able to support a subset of the JUnit Platform functionality, especially with regard to reporting (see Display Names vs. Technical Names). But for the time being the `JUnitPlatform` runner is an easy way to get started.

4.4.1. Setup

You need the following artifacts and their dependencies on the classpath. See Dependency Metadata for details regarding group IDs, artifact IDs, and versions.

Explicit Dependencies

¥ `junit-platform-runner` in *test* scope: location of the `JUnitPlatform` runner

¥ `junit-4.12.jar` in *test* scope: to run tests using JUnit 4

¥ `junit-jupiter-api` in *test* scope: API for writing tests using JUnit Jupiter, including `@Test`, etc.

¥ `junit-jupiter-engine` in *test runtime* scope: implementation of the `TestEngine` API for JUnit Jupiter

Transitive Dependencies

¥ `junit-platform-suite-api` in *test* scope

¥ `junit-platform-launcher` in *test* scope

¥ `junit-platform-engine` in *test* scope

¥ `junit-platform-commons` in *test* scope

¥ `opentest4j` in *test* scope

4.4.2. Display Names vs. Technical Names

To define a custom *display name* for the class run via `@RunWith(JUnitPlatform.class)` simply annotate the class with `@SuiteDisplayName` and provide a custom value.

By default, *display names* will be used for test artifacts; however, when the `JUnitPlatform` runner is used to execute tests with a build tool such as Gradle or Maven, the generated test report often needs to include the *technical names* of test artifacts Ñ for example, fully qualified class names Ñ instead of shorter display names like the simple name of a test class or a custom display name containing special characters. To enable technical names for reporting purposes, simply declare the `@UseTechnicalNames` annotation alongside `@RunWith(JUnitPlatform.class)`.

Note that the presence of `@UseTechnicalNames` overrides any custom display name configured via `@SuiteDisplayName`.

4.4.3. Single Test Class

One way to use the `JUnitPlatform` runner is to annotate a test class with `@RunWith(JUnitPlatform.class)` directly. Please note that the test methods in the following example are annotated with `org.junit.jupiter.api.Test` (JUnit Jupiter), not `org.junit.Test` (JUnit Vintage). Moreover, in this case the test class must be `public`; otherwise, some IDEs and build tools might not recognize it as a JUnit 4 test class.

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

import org.junit.jupiter.api.Test;
import org.junit.platform.runner.JUnitPlatform;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
public class JUnit4ClassDemo {

Ê   @Test
Ê   void succeedingTest() {
Ê       /* no-op */
Ê   }

Ê   @Test
Ê   void failingTest() {
Ê       fail("Failing for failing's sake.");
Ê   }

}
```

4.4.4. Test Suite

If you have multiple test classes you can create a test suite as can be seen in the following example.

```java
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.platform.suite.api.SuiteDisplayName;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SuiteDisplayName("JUnit 4 Suite Demo")
@SelectPackages("example")
public class JUnit4SuiteDemo {
}
```

The `JUnit4SuiteDemo` will discover and run all tests in the `example` package and its subpackages. By default, it will only include test classes whose names either begin with `Test` or end with `Test` or `Tests`.

> *Additional Configuration Options*
>
> There are more configuration options for discovering and filtering tests than just `@SelectPackages`. Please consult the Javadoc for further details.

4.5. 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 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

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 the `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. JVM system properties.
3. The JUnit Platform configuration file: a file named `junit-platform.properties` in the root of the class path that follows the syntax rules for a Java `Properties` file.

> # 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 system properties and the configuration file. Similarly, configuration parameters supplied via system properties take precedence over those supplied via the configuration file.

4.6. Tag Expressions

Tag expressions are boolean expressions with the operators `!`, `&` and `|`. In addition, `(` and `)` can be used to adjust for operator precedence.

Table 1. 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. Tagging by test type (e.g. *micro*, *integration*, *end-to-end*) and feature (e.g. foo, bar, baz) the following tag expressions can be useful.

Tag Expression	Selection
foo	all tests for foo
bar | baz	all tests for bar plus all tests for baz
bar & baz	all tests for the intersection between bar and baz
foo & !end-to-end	all tests for foo, but not the *end-to-end* tests
(micro | integration) & (foo | baz)	all *micro* or *integration* tests for foo or baz

4.7. Capturing Standard Output/Error

Since version 1.3, the JUnit Platform provides opt-in support for capturing output printed to `System.out` and `System.err`. To enable it, simply 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.

> Capturing output is 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.

5. Extension Model

5.1. Overview

In contrast to the competing `Runner`, `@Rule`, and `@ClassRule` 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.

5.2. Registering Extensions

Extensions can be registered *declaratively* via `@ExtendWith`, *programmatically* via `@RegisterExtension`, or *automatically* via Java's `ServiceLoader` mechanism.

5.2.1. 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(É)` and supplying class references for the extensions to register.

For example, to register a custom `RandomParametersExtension` for a particular test method, you would annotate the test method as follows.

```
@ExtendWith(RandomParametersExtension.class)
@Test
void test(@Random int i) {
Ê   // ...
}
```

To register a custom `RandomParametersExtension` for all tests in a particular class and its subclasses, you would annotate the test class as follows.

```
@ExtendWith(RandomParametersExtension.class)
class MyTests {
Ê   // ...
}
```

Multiple extensions can be registered together like this:

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

As an alternative, multiple extensions can be registered separately like this:

```
@ExtendWith(FooExtension.class)
@ExtendWith(BarExtension.class)
class MySecondTests {
Ê   // ...
}
```

#

Extension Registration Order

Extensions registered declaratively via `@ExtendWith` 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 `FooExtension` and `BarExtension`, in exactly that order.

5.2.2. 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*!Ñ!for example, in order to pass arguments to the extensionÕs constructor, a static factory method, or a builder API.

> `@RegisterExtension` fields must not be `private` or `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`, and `TestInstancePostProcessor` 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!Ñ!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.

An extension registered via a static field

```
class WebServerDemo {

Ê   @RegisterExtension
Ê   static WebServerExtension server = WebServerExtension.builder()
Ê       .enableSecurity(false)
Ê       .build();

Ê   @Test
Ê   void getProductList() {
Ê       WebClient webClient = new WebClient();
Ê       String serverUrl = server.getServerUrl();
Ê       // Use WebClient to connect to web server using serverUrl and verify response
Ê       assertEquals(200, webClient.get(serverUrl + "/products").getResponseStatus());
Ê   }

}
```

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 static fields using annotations. Since, as stated earlier, `@RegisterExtension` fields must not be `private` nor `null`, one cannot use the `@JvmStatic` annotation in Kotlin as it generates `private` fields. Rather, the `@JvmField` annotation must be used.

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 {

Ê   companion object {
Ê       @JvmField
Ê       @RegisterExtension
Ê       val server = WebServerExtension.builder()
Ê               .enableSecurity(false)
Ê               .build()
Ê   }

Ê   @Test
Ê   fun getProductList() {
Ê       // Use WebClient to connect to web server using serverUrl and verify response
Ê       val webClient = WebClient()
Ê       val serverUrl = server.serverUrl
Ê       assertEquals(200, webClient.get("$serverUrl/products").responseStatus)
Ê   }
}
```

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. By default, an instance extension will be registered *after* extensions that are registered at the method level via `@ExtendWith`; however, if the test class is configured with `@TestInstance(Lifecycle.PER_CLASS)` semantics, an instance extension 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 {

Ê   static Path lookUpDocsDir() {
Ê       // return path to docs dir
Ê   }

Ê   @RegisterExtension
Ê   DocumentationExtension docs = DocumentationExtension.forPath(lookUpDocsDir());

Ê   @Test
Ê   void generateDocumentation() {
Ê       // use this.docs ...
Ê   }
}
```

5.2.3. 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 `java.util.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, simply 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.).

5.2.4. 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. Furthermore, 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.

5.3. 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.

5.3.1. 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, simply 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 JUnitʼs `@Disabled` condition, you can start your JVM with the following system property.

```
-Djunit.jupiter.conditions.deactivate=org.junit.*DisabledCondition
```

Pattern Matching Syntax

If the `junit.jupiter.conditions.deactivate` pattern consists solely of an asterisk (`*`), all conditions will be deactivated. Otherwise, the pattern will be used to match against the fully qualified class name (*FQCN*) of each registered condition. Any dot (`.`) in the pattern will match against a dot (`.`) or a dollar sign (`$`) in the FQCN. Any asterisk (`*`) will match against one or more characters in the FQCN. All other characters in the pattern will be matched one-to-one against the FQCN.

Examples:

- `*`: deactivates all conditions.
- `org.junit.*`: deactivates every condition under the `org.junit` base package and any of its subpackages.
- `*.MyCondition`: deactivates every condition whose simple class name is exactly `MyCondition`.
- `*System*`: deactivates every condition whose simple class name contains `System`.

¥ `org.example.MyCondition`: deactivates the condition whose FQCN is exactly `org.example.MyCondition`.

5.4. 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 simply 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.

5.5. 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`.

5.6. Parameter Resolution

`ParameterResolver` defines the `Extension` API for dynamically resolving parameters at runtime.

If a test constructor or a `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, `@BeforeEach`, `@AfterEach`, `@BeforeAll`, or `@AfterAll` method accepts 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. For concrete examples, consult the source code for `CustomTypeParameterResolver` and `CustomAnnotationParameterResolver`.

" 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)`

5.7. 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`
  - `BeforeEachCallback`
    - `BeforeTestExecutionCallback`
    - `AfterTestExecutionCallback`
  - `AfterEachCallback`
- `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.

5.7.1. 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 {

Ê   private static final Logger logger = Logger.getLogger(TimingExtension.class
.getName());

Ê   private static final String START_TIME = "start time";

Ê   @Override
Ê   public void beforeTestExecution(ExtensionContext context) throws Exception {
Ê       getStore(context).put(START_TIME, System.currentTimeMillis());
Ê   }

Ê   @Override
Ê   public void afterTestExecution(ExtensionContext context) throws Exception {
Ê       Method testMethod = context.getRequiredTestMethod();
Ê       long startTime = getStore(context).remove(START_TIME, long.class);
Ê       long duration = System.currentTimeMillis() - startTime;

Ê       logger.info(() -> String.format("Method [%s] took %s ms.", testMethod.getName
(), duration));
Ê   }

Ê   private Store getStore(ExtensionContext context) {
Ê       return context.getStore(Namespace.create(getClass(), context
.getRequiredTestMethod()));
Ê   }

}
```

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 {

Ê   @Test
Ê   void sleep20ms() throws Exception {
Ê       Thread.sleep(20);
Ê   }

Ê   @Test
Ê   void sleep50ms() throws Exception {
Ê       Thread.sleep(50);
Ê   }

}
```

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.
```

5.8. Exception Handling

`TestExecutionExceptionHandler` defines the API for `Extensions` that wish to handle exceptions thrown during test execution.

The following example shows an extension which will swallow all instances of `IOException` but rethrow any other type of exception.

An exception handling extension

```
public class IgnoreIOExceptionExtension implements TestExecutionExceptionHandler {

Ê   @Override
Ê   public void handleTestExecutionException(ExtensionContext context, Throwable
throwable)
Ê           throws Throwable {

Ê       if (throwable instanceof IOException) {
Ê           return;
Ê       }
Ê       throw throwable;
Ê   }
}
```

5.9. 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

```
@TestTemplate
@ExtendWith(MyTestTemplateInvocationContextProvider.class)
void testTemplate(String parameter) {
Ê   assertEquals(3, parameter.length());
}

public class MyTestTemplateInvocationContextProvider implements
TestTemplateInvocationContextProvider {
Ê   @Override
Ê   public boolean supportsTestTemplate(ExtensionContext context) {
Ê       return true;
Ê   }

Ê   @Override
Ê   public Stream<TestTemplateInvocationContext>
provideTestTemplateInvocationContexts(ExtensionContext context) {
Ê       return Stream.of(invocationContext("foo"), invocationContext("bar"));
Ê   }

Ê   private TestTemplateInvocationContext invocationContext(String parameter) {
Ê       return new TestTemplateInvocationContext() {
Ê           @Override
Ê           public String getDisplayName(int invocationIndex) {
Ê               return parameter;
Ê           }

Ê           @Override
Ê           public List<Extension> getAdditionalExtensions() {
Ê               return Collections.singletonList(new ParameterResolver() {
Ê                   @Override
Ê                   public boolean supportsParameter(ParameterContext
parameterContext,
Ê                           ExtensionContext extensionContext) {
Ê                       return parameterContext.getParameter().getType().equals(
String.class);
Ê                   }

Ê                   @Override
Ê                   public Object resolveParameter(ParameterContext parameterContext,
Ê                           ExtensionContext extensionContext) {
Ê                       return parameter;
Ê                   }
Ê               });
Ê           }
Ê       };
Ê   }
}
```

In this example, the test template will be invoked twice. The display names of the invocations will be Òfoo0 and Òbar0 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) %
Ê  # $ foo %
Ê  ' $ bar %
```

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 Ñ 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.

5.10. 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. It is important to remember that values stored in an `ExtensionContext` during test execution will not be available in the surrounding `ExtensionContext`. Since `ExtensionContexts` may be nested, the scope of inner contexts may also be limited. Consult the corresponding JavaDoc for details on the methods available for storing and retrieving values via the `Store`.

> #
>
> `ExtensionContext.Store.CloseableResource`
>
> 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 `CloseableResource` are notified by an invocation of their `close()` method.

5.11. Supported Utilities in Extensions

The `junit-platform-commons` artifact exposes a package named `org.junit.platform.commons.support` that contains *maintained* utility methods for working with annotations, classes, reflection, and classpath scanning tasks. `TestEngine` and `Extension` authors are encouraged to use these supported methods in order to align with the behavior of the JUnit Platform.

5.11.1. 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.

5.11.2. 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.

5.11.3. 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.

5.12. 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-provided test and lifecycle methods. The following diagram illustrates the relative order of user-provided code and extension code.

User code and extension code

User-provided test and lifecycle methods are shown in orange, with callback code provided 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.

The following table further explains the twelve steps in the User code and extension code diagram.

Step	Interface/Annotation	Description
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.BeforeEachCallback`	extension code executed before each test is executed
4	annotation `org.junit.jupiter.api.BeforeEach`	user code executed before each test is executed
5	interface `org.junit.jupiter.api.extension.BeforeTestExecutionCallback`	extension code executed immediately before a test is executed
6	annotation `org.junit.jupiter.api.Test`	user code of the actual test method
7	interface `org.junit.jupiter.api.extension.TestExecutionExceptionHandler`	extension code for handling exceptions thrown during a test
8	interface `org.junit.jupiter.api.extension.AfterTestExecutionCallback`	extension code executed immediately after test execution and its corresponding exception handlers
9	annotation `org.junit.jupiter.api.AfterEach`	user code executed after each test is executed
10	interface `org.junit.jupiter.api.extension.AfterEachCallback`	extension code executed after each test is executed
11	annotation `org.junit.jupiter.api.AfterAll`	user code executed after all tests of the container are executed

Step	Interface/Annotation	Description
12	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 6); 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.

6. Migrating from JUnit 4

Although the JUnit Jupiter programming model and extension model will 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 a new `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. Furthermore, since the JUnit team will continue to provide maintenance and bug fix releases for the JUnit 4.x baseline, developers have plenty of time to migrate to JUnit Jupiter on their own schedule.

6.1. Running JUnit 4 Tests on the JUnit Platform

Just 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 `junit5-samples` repository to find out how this is done with Gradle and Maven.

6.1.1. 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 of the corresponding test identifier. 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).

6.2. Migration Tips

The following are things you have to watch out for 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`.
- Assumptions reside in `org.junit.jupiter.api.Assumptions`.
- `@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` instead.
- `@Category` no longer exists; use `@Tag` instead.
- `@RunWith` no longer exists; superseded by `@ExtendWith`.
- `@Rule` and `@ClassRule` no longer exist; superseded by `@ExtendWith`; see the following section for partial rule support.

6.3. Limited JUnit 4 Rule Support

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 those 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 `org.junit.jupiter.migrationsupport.rules.EnableRuleMigrationSupport`. This annotation is a *composed annotation* which enables all migration support extensions: `VerifierSupport`, `ExternalResourceSupport`, and `ExpectedExceptionSupport`.

However, if you intend to develop a new extension for JUnit 5 please use the new extension model of JUnit Jupiter instead of the rule-based model of JUnit 4.

> JUnit 4 `Rule` support in JUnit Jupiter is currently an *experimental* feature. Consult the table in Experimental APIs for detail.

7. Advanced Topics

7.1. JUnit Platform Launcher API

One of the prominent goals of JUnit 5 is to make the interface between JUnit and its programmatic clients Ð build tools and IDEs Ð more powerful and stable. The purpose is to decouple the internals of discovering and executing tests from all the filtering and configuration thatÕs necessary from the outside.

JUnit 5 introduces the concept of a `Launcher` that can be used to discover, filter, and execute tests. Moreover, third party test libraries Ð like Spock, Cucumber, and FitNesse Ð can plug into the JUnit PlatformÕs 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.

7.1.1. Discovering Tests

Introducing *test discovery* as a dedicated feature of the platform itself will (hopefully) free IDEs and build tools from most of the difficulties they had to go through to identify test classes and test methods in the past.

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 org.junit.platform.launcher.Launcher;
import org.junit.platform.launcher.LauncherDiscoveryRequest;
import org.junit.platform.launcher.TestExecutionListener;
import org.junit.platform.launcher.TestPlan;
import org.junit.platform.launcher.core.LauncherConfig;
import org.junit.platform.launcher.core.LauncherDiscoveryRequestBuilder;
import org.junit.platform.launcher.core.LauncherFactory;
import org.junit.platform.launcher.listeners.SummaryGeneratingListener;
```

```
LauncherDiscoveryRequest request = LauncherDiscoveryRequestBuilder.request()
Ê   .selectors(
Ê       selectPackage("com.example.mytests"),
Ê       selectClass(MyTestClass.class)
Ê   )
Ê   .filters(
Ê       includeClassNamePatterns(".*Tests")
Ê   )
Ê   .build();

Launcher launcher = LauncherFactory.create();

TestPlan testPlan = launcher.discover(request);
```

There's currently the possibility to select classes, methods, and all classes in a package or even search for all tests in the classpath. 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.

7.1.2. 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 request = LauncherDiscoveryRequestBuilder.request()
Ê   .selectors(
Ê       selectPackage("com.example.mytests"),
Ê       selectClass(MyTestClass.class)
Ê   )
Ê   .filters(
Ê       includeClassNamePatterns(".*Tests")
Ê   )
Ê   .build();

Launcher launcher = LauncherFactory.create();

// Register a listener of your choice
TestExecutionListener listener = new SummaryGeneratingListener();
launcher.registerTestExecutionListeners(listener);

launcher.execute(request);
```

There is no return value for the `execute()` method, but you can easily use a listener to aggregate the final results in an object of your own. For an example see the `SummaryGeneratingListener`.

7.1.3. Plugging in your own Test Engine

JUnit currently provides two `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 with the launcher infrastructure.

Third parties may also contribute their own `TestEngine` by implementing the interfaces in the junit-platform-engine module and *registering* their engine. By default, engine registration is supported via Java's `java.util.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` in the `junit-jupiter-engine` JAR.

> `HierarchicalTestEngine` is a convenient abstract base implementation (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.

> *The `junit-` 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.

7.1.4. Plugging in your own Test Execution Listener

In addition to the public `Launcher` API method for registering test execution listeners programmatically, by default custom `TestExecutionListener` implementations will be discovered at runtime via Java's `java.util.ServiceLoader` mechanism and automatically registered with the `Launcher` created via the `LauncherFactory`. For example, an `example.TestInfoPrinter` class implementing `TestExecutionListener` and declared within the `/META-INF/services/org.junit.platform.launcher.TestExecutionListener` file is loaded and registered automatically.

7.1.5. Configuring the Launcher

If you require fine-grained control over automatic detection and registration of test engines and test execution listeners, you may create an instance of `LauncherConfig` and supply that to the `LauncherFactory.create(LauncherConfig)` method. Typically an instance of `LauncherConfig` is created via the built-in fluent *builder* API, as demonstrated in the following example.

```
LauncherConfig launcherConfig = LauncherConfig.builder()
Ê   .enableTestEngineAutoRegistration(false)
Ê   .enableTestExecutionListenerAutoRegistration(false)
Ê   .addTestEngines(new CustomTestEngine())
Ê   .addTestExecutionListeners(new CustomTestExecutionListener())
Ê   .build();

Launcher launcher = LauncherFactory.create(launcherConfig);

LauncherDiscoveryRequest request = LauncherDiscoveryRequestBuilder.request()
Ê   .selectors(selectPackage("com.example.mytests"))
Ê   .build();

launcher.execute(request);
```

8. API Evolution

One of the major goals of JUnit 5 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 5 introduces a defined lifecycle for all publicly available interfaces, classes, and methods.

8.1. 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 (`5.*`).

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.

8.2. Experimental APIs

The following table lists which APIs are currently designated as *experimental* via `@API(status = EXPERIMENTAL)`. Caution should be taken when relying on such APIs.

Package Name	Type Name	Since
`org.junit.jupiter.api`	`AssertionsKt` *(class)*	`5.1`
`org.junit.jupiter.api.condition`	`DisabledIf` *(annotation)*	`5.1`
`org.junit.jupiter.api.condition`	`EnabledIf` *(annotation)*	`5.1`
`org.junit.jupiter.api.extension`	`ScriptEvaluationException` *(class)*	`5.1`
`org.junit.jupiter.api.extension`	`TestInstanceFactory` *(interface)*	`5.3`
`org.junit.jupiter.api.extension`	`TestInstanceFactoryContext` *(interface)*	`5.3`
`org.junit.jupiter.api.extension`	`TestInstantiationException` *(class)*	`5.3`
`org.junit.jupiter.api.parallel`	`Execution` *(annotation)*	`5.3`
`org.junit.jupiter.api.parallel`	`ExecutionMode` *(enum)*	`5.3`
`org.junit.jupiter.api.parallel`	`ResourceAccessMode` *(enum)*	`5.3`
`org.junit.jupiter.api.parallel`	`ResourceLock` *(annotation)*	`5.3`
`org.junit.jupiter.api.parallel`	`ResourceLocks` *(annotation)*	`5.3`
`org.junit.jupiter.api.parallel`	`Resources` *(class)*	`5.3`
`org.junit.jupiter.params`	`ParameterizedTest` *(annotation)*	`5.0`
`org.junit.jupiter.params.aggregator`	`AggregateWith` *(annotation)*	`5.2`
`org.junit.jupiter.params.aggregator`	`ArgumentAccessException` *(class)*	`5.2`
`org.junit.jupiter.params.aggregator`	`ArgumentsAccessor` *(interface)*	`5.2`
`org.junit.jupiter.params.aggregator`	`ArgumentsAggregationException` *(class)*	`5.2`
`org.junit.jupiter.params.aggregator`	`ArgumentsAggregator` *(interface)*	`5.2`
`org.junit.jupiter.params.converter`	`ArgumentConversionException` *(class)*	`5.0`
`org.junit.jupiter.params.converter`	`ArgumentConverter` *(interface)*	`5.0`
`org.junit.jupiter.params.converter`	`ConvertWith` *(annotation)*	`5.0`
`org.junit.jupiter.params.converter`	`JavaTimeConversionPattern` *(annotation)*	`5.0`
`org.junit.jupiter.params.converter`	`SimpleArgumentConverter` *(class)*	`5.0`

Package Name	Type Name	Since
org.junit.jupiter.params.provider	Arguments *(interface)*	5.0
org.junit.jupiter.params.provider	ArgumentsProvider *(interface)*	5.0
org.junit.jupiter.params.provider	ArgumentsSource *(annotation)*	5.0
org.junit.jupiter.params.provider	ArgumentsSources *(annotation)*	5.0
org.junit.jupiter.params.provider	CsvFileSource *(annotation)*	5.0
org.junit.jupiter.params.provider	CsvParsingException *(class)*	5.3
org.junit.jupiter.params.provider	CsvSource *(annotation)*	5.0
org.junit.jupiter.params.provider	EnumSource *(annotation)*	5.0
org.junit.jupiter.params.provider	MethodSource *(annotation)*	5.0
org.junit.jupiter.params.provider	ValueSource *(annotation)*	5.0
org.junit.jupiter.params.support	AnnotationConsumer *(interface)*	5.0
org.junit.platform.engine.support.config	PrefixedConfigurationParameters *(class)*	1.3
org.junit.platform.engine.support.hierarchical	DefaultParallelExecutionConfigurationStrategy *(enum)*	1.3
org.junit.platform.engine.support.hierarchical	ExclusiveResource *(class)*	1.3
org.junit.platform.engine.support.hierarchical	ForkJoinPoolHierarchicalTestExecutorService *(class)*	1.3
org.junit.platform.engine.support.hierarchical	HierarchicalTestExecutorService *(interface)*	1.3
org.junit.platform.engine.support.hierarchical	ExecutionMode *(enum)*	1.3
org.junit.platform.engine.support.hierarchical	ParallelExecutionConfiguration *(interface)*	1.3
org.junit.platform.engine.support.hierarchical	ParallelExecutionConfigurationStrategy *(interface)*	1.3
org.junit.platform.engine.support.hierarchical	ResourceLock *(interface)*	1.3
org.junit.platform.engine.support.hierarchical	SameThreadHierarchicalTestExecutorService *(class)*	1.3
org.junit.platform.launcher	LauncherConstants *(class)*	1.3
org.junit.platform.launcher.core	LauncherConfig *(interface)*	1.3

8.3. Deprecated APIs

The following table lists 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.

Package Name	Type Name	Since
`org.junit.platform.engine.support.hierarchical`	`SingleTestExecutor` *(class)*	`1.2`
`org.junit.platform.surefire.provider`	`JUnitPlatformProvider` *(class)*	`1.3`

8.4. @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.

9. Contributors

Browse the current list of contributors directly on GitHub.

10. Release Notes

The release notes are available here.