JUnit 5 User Guide

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

Version 5.0.0-SNAPSHOT

Table of Contents

1. Overview . . . 1
  1.1. What is JUnit 5? . . . 1
  1.2. Supported Java Versions . . . 1
2. Installation . . . 1
  2.1. Dependency Metadata . . . 1
  2.2. Dependency Diagram . . . 3
  2.3. JUnit Jupiter Sample Projects . . . 3
3. Writing Tests . . . 3
  3.1. Annotations . . . 3
  3.2. Standard Test Class . . . 5
  3.3. Display Names . . . 7
  3.4. Assertions . . . 7
  3.5. Assumptions . . . 10
  3.6. Disabling Tests . . . 11
  3.7. Tagging and Filtering . . . 12
  3.8. Nested Tests . . . 13
  3.9. Dependency Injection for Constructors and Methods . . . 15
  3.10. Test Interfaces and Default Methods . . . 18
  3.11. Repeated Tests . . . 24
  3.12. Parameterized Tests . . . 27
  3.13. Dynamic Tests . . . 33
4. Running Tests . . . 37
  4.1. IDE Support . . . 37
  4.2. Build Support . . . 37
  4.3. Console Launcher . . . 45
  4.4. Using JUnit 4 to Run the JUnit Platform . . . 48
5. Extension Model . . . 50
  5.1. Overview . . . 50
  5.2. Registering Extensions . . . 50
  5.3. Conditional Test Execution . . . 52
  5.4. Test Instance Post-processing . . . 53
  5.5. Parameter Resolution . . . 53
  5.6. Test Lifecycle Callbacks . . . 53
  5.7. Exception Handling . . . 56
  5.8. Keeping State in Extensions . . . 57
  5.9. Supported Utilities in Extensions . . . 57
  5.10. Relative Execution Order of User Code and Extensions . . . 57
6. Migrating from JUnit 4 . . . 59

6.1. Running JUnit 4 Tests on the JUnit Platform . . . 59
6.2. Migration Tips . . . 59
6.3. Limited JUnit 4 Rule Support . . . 60
7. Advanced Topics . . . 60
7.1. JUnit Platform Launcher API . . . 60
8. API Evolution . . . 63
8.1. API Annotations . . . 63
8.2. Tooling Support . . . 63
9. Contributors . . . 63
10. Release Notes . . . 64
5.0.0-ALPHA . . . 64
5.0.0-M1 . . . 64
5.0.0-M2 . . . 68
5.0.0-M3 . . . 69
5.0.0-M4 . . . 73

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.

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 at runtime. However, you can still test code that has been compiled with previous versions of the JDK.

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**: `1.0.0-SNAPSHOT`
- **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-gradle-plugin`
Support for discovering and executing tests on the JUnit Platform using Gradle.

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

`junit-platform-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**: `5.0.0-SNAPSHOT`
- **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-migration-support`
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**: `4.12.0-SNAPSHOT`
- **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.

2.2. Dependency Diagram

[component diagram] | *component-diagram.svg*

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.gradle` and `pom.xml` in the projects below.

- For Gradle, check out the `junit5-gradle-consumer` project.
- For Maven, check out the `junit5-maven-consumer` project.

3. Writing Tests

A first test case

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

import org.junit.jupiter.api.Test;

class FirstJUnit5Tests {

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

}
```

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.
`@RepeatedTest`	Denotes that a method is a test template for a repeated test
`@TestFactory`	Denotes that a method is a test factory for dynamic tests
`@DisplayName`	Declares a custom display name for the test class or test method
`@BeforeEach`	Denotes that the annotated method should be executed *before* **each** `@Test` method in the current class; analogous to JUnit 4's `@Before`. Such methods are *inherited*.
`@AfterEach`	Denotes that the annotated method should be executed *after* **each** `@Test` method in the current class; analogous to JUnit 4's `@After`. Such methods are *inherited*.
`@BeforeAll`	Denotes that the annotated method should be executed *before* **all** `@Test` methods in the current class; analogous to JUnit 4's `@BeforeClass`. Such methods must be `static` and are *inherited*.
`@AfterAll`	Denotes that the annotated method should be executed *after* **all** `@Test` methods in the current class; analogous to JUnit 4's `@AfterClass`. Such methods must be `static` and are *inherited*.
`@Nested`	Denotes that the annotated class is a nested, non-static test class. Due to restrictions of the Java language, `@BeforeAll` and `@AfterAll` methods cannot be used in a `@Nested` test class.
`@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
`@Disabled`	Used to *disable* a test class or test method; analogous to JUnit 4's `@Ignore`
`@ExtendWith`	Used to register custom extensions

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. Standard Test Class

A standard test case

```java
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 {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

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

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

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }

}
```

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

3.3. Display Names

Test classes and test methods can declare custom display names — with spaces, special characters, and even emojis — 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 {

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

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

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

}
```

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.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.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 {

    @Test
```

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

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

    @Test
    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
```

```java
    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() {
        return "hello world!";
    }

}
```

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.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.

```java
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 {

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

}
```

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.Assumptions` class.

```java
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;

public class AssumptionsDemo {

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

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

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

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

}
```

3.6. Disabling Tests

Here’s a disabled test case.

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

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

And here's a test case with a disabled test method.

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

class DisabledTestsDemo {

    @Disabled
    @Test
    void testWillBeSkipped() {
    }

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

3.7. Tagging and Filtering

Test classes and methods can be tagged. Those tags can later be used to filter test discovery and execution.

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

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

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

}
```

3.8. 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 {

    Stack<Object> stack;

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

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

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

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

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

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

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

            String anElement = "an element";

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

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

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

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

Only non-static nested classes (i.e. *inner classes*) can serve as `@Nested` tests. 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` do not work, because Java does not allow `static` members in inner classes.

3.9. 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.

```java
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 {

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

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

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

    @Test
    void test2() {
    }

}
```

- `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`.

```java
import java.util.HashMap;

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

class TestReporterDemo {

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

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

        testReporter.publishEntry(values);
    }

}
```

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

Check out the `MockitoExtension` 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. `MyMockitoTest` demonstrates how to inject Mockito mocks into `@BeforeEach` and `@Test` methods.

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

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import com.example.Person;
import com.example.mockito.MockitoExtension;

@ExtendWith(MockitoExtension.class)
class MyMockitoTest {

    @BeforeEach
    void init(@Mock Person person) {
        when(person.getName()).thenReturn("Dilbert");
    }

    @Test
    void simpleTestWithInjectedMock(@Mock Person person) {
        assertEquals("Dilbert", person.getName());
    }

}
```

3.10. Test Interfaces and Default Methods

JUnit Jupiter allows `@Test`, `@TestFactory`, `@BeforeEach`, and `@AfterEach` to be declared on interface default methods. In addition, `@BeforeAll` and `@AfterAll` can be declared on static methods in a test interface, while `@ExtendWith` and `@Tag` can be declared on test interfaces to configure extensions and tags. Here are some examples.

```java
public interface TestLifecycleLogger {

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

    @BeforeAll
    static void beforeAllTests() {
        LOG.info("beforeAllTests");
    }

    @AfterAll
    static void afterAllTests() {
        LOG.info("afterAllTests");
    }

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

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

}
```

```java
interface TestInterfaceDynamicTestsDemo {

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

}
```

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

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

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

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

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

}
```

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

```
:junitPlatformTest
18:28:13.967 [main] INFO  example.testinterface.TestLifecycleLogger - beforeAllTests
18:28:13.982 [main] INFO  example.testinterface.TestLifecycleLogger - About to execute
[dynamicTestsFromCollection()]
18:28:14.000 [main] INFO  example.testinterface.TimingExtension - Method
[dynamicTestsFromCollection] took 13 ms.
18:28:14.004 [main] INFO  example.testinterface.TestLifecycleLogger - Finished
executing [dynamicTestsFromCollection()]
18:28:14.007 [main] INFO  example.testinterface.TestLifecycleLogger - About to execute
[isEqualValue()]
18:28:14.008 [main] INFO  example.testinterface.TimingExtension - Method
[isEqualValue] took 1 ms.
18:28:14.009 [main] INFO  example.testinterface.TestLifecycleLogger - Finished
executing [isEqualValue()]
18:28:14.011 [main] INFO  example.testinterface.TestLifecycleLogger - afterAllTests

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> {

    T createValue();

}
```

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

    T createNotEqualValue();

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

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

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

}
```

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

    T createSmallerValue();

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

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

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

}
```

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

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

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

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

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

}
```

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

3.11. 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() {
    // ...
}
```

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.

3.11.1. 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 — in this case German, resulting in names for individual repetitions such as: `Wiederholung 1 von 5`, `Wiederholung 2 von 5`, etc.

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

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

```java
class RepeatedTestsDemo {

    private Logger logger = // ...

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

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

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

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

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

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

}
```

When using the `ConsoleLauncher` or the `junitPlatformTest` Gradle plugin 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
```

3.12. 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.

```
@ParameterizedTest
@ValueSource(strings = { "Hello", "World" })
void testWithStringParameter(String argument) {
    assertNotNull(argument);
}
```

This parameterized test uses the `@ValueSource` annotation to specify a `String` array as the source of arguments. When executing this method, each invocation will be reported separately. For instance, the `ConsoleLauncher` will print output similar to the following.

```
testWithStringParameter(String)
├── [1] Hello
└── [2] World
```

3.12.1. 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.

3.12.2. 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 an array of literals of primitive types (either `String`, `int`, `long`, or `double`) and can only be used for providing a single parameter per invocation.

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertNotNull(argument);
}
```

@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.name());
}
```

@MethodSource

`@MethodSource` allows to refer to one or multiple methods of the test class. Each method must return a `Stream`, an `Iterable`, an `Iterator`, or an array of arguments. In addition, each method must be `static` and must not accept any arguments.

If you only need a single parameter, you can return instances of the parameter type directly as demonstrated by the following example.

```
@ParameterizedTest
@MethodSource(names = "stringProvider")
void testWithSimpleMethodSource(String argument) {
    assertNotNull(argument);
}

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

In case you need multiple parameters, you need to return `Argument` instances as shown below. Note that `arguments(Object…)` is a static factory method defined in `org.junit.jupiter.params.provider.ObjectArrayArguments`.

```
@ParameterizedTest
@MethodSource(names = "stringAndIntProvider")
void testWithMultiArgMethodSource(String first, int second) {
    assertNotNull(first);
    assertNotEquals(0, second);
}

static Stream<Arguments> stringAndIntProvider() {
    return Stream.of(arguments("foo", 1), arguments("bar", 2));
}
```

@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) {
    assertNotNull(first);
    assertNotEquals(0, second);
}
```

@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")
void testWithCsvFileSource(String first, int second) {
    assertNotNull(first);
    assertNotEquals(0, second);
}
```

two-column.csv

```
foo, 1
bar, 2
"baz, qux", 3
```

@ArgumentsSource

`@ArgumentsSource` can be used to specify a custom, reusable `ArgumentsProvider`.

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

static class MyArgumentsProvider implements ArgumentsProvider {

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

3.12.3. Argument Conversion

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 automatically be converted into the corresponding `TimeUnit` enum constant.

```java
@ParameterizedTest
@ValueSource(strings = "SECONDS")
void testWithImplicitArgumentConversion(TimeUnit argument) {
    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`
`long/Long`	`"1" → 1L`
`float/Float`	`"1.0" → 1.0f`
`double/Double`	`"1.0" → 1.0d`
`Enum` subclass	`"SECONDS" → TimeUnit.SECONDS`
`java.time.Instant`	`"1970-01-01T00:00:00Z" → Instant.ofEpochMilli(0)`
`java.time.LocalDate`	`"2017-03-14" → LocalDate.of(2017, 3, 14)`
`java.time.LocalDateTime`	`"2017-03-14T12:34:56.789" → LocalDateTime.of(2017, 3, 14, 12, 34, 56, 789_000_000)`
`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)`
`java.time.Year`	`"2017" → Year.of(2017)`

Target Type	Example
`java.time.YearMonth`	`"2017-03"` → `YearMonth.of(2017, 3)`
`java.time.ZonedDateTime`	`"2017-03-14T12:34:56.789Z"` → `ZonedDateTime.of(2017, 3, 14, 12, 34, 56, 789_000_000, ZoneOffset.UTC)`

Explicit Conversion

Instead of using implicit argument conversion you may explicitly specify an `ArgumentConverter` to use for a certain parameter using the `@ConvertWith` annotation like in the following example.

```
@ParameterizedTest
@EnumSource(TimeUnit.class)
void testWithExplicitArgumentConversion(@ConvertWith(ToStringArgumentConverter.class)
String argument) {
    assertNotNull(TimeUnit.valueOf(argument));
}

static class ToStringArgumentConverter extends SimpleArgumentConverter {

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

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

```
@ParameterizedTest
@ValueSource(strings = { "01.01.2017", "31.12.2017" })
void testWithExplicitJavaTimeConverter(@JavaTimeConversionPattern("dd.MM.yyyy")
LocalDate argument) {
    assertEquals(2017, argument.getYear());
}
```

3.12.4. 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.

```java
@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}`, …	an individual argument

3.12.5. 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.

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

3.13. 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`, or `Iterator` of `DynamicTest` instances. These `DynamicTest` instances will then 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 not any lifecycle callbacks for dynamic tests. This means that `@BeforeEach` and `@AfterEach` methods and their corresponding extension callbacks are not executed for dynamic tests. 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 dynamic tests generated by the same `@TestFactory` method.

As of JUnit Jupiter 5.0.0-SNAPSHOT, dynamic tests must always be created by factory methods; however, this might be complemented by a registration facility in a later release.

3.13.1. 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 last 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.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;
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.DynamicTest;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.TestFactory;
import org.junit.jupiter.api.function.ThrowingConsumer;

class DynamicTestsDemo {

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

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

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

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

```
        ).iterator();
    }

    @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<Integer>() {

            Random random = new Random();
            int current;

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

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

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

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

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

}
```

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.

4.1.2. Eclipse Beta Support

Eclipse 4.7 (*Oxygen*) has beta support for the JUnit Platform and JUnit Jupiter. For details on how to set it up, please consult the Eclipse JDT UI/JUnit 5 wiki page.

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 with IntelliJ IDEA or the beta support in 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

The JUnit team has developed a very basic Gradle plugin that allows you to run any kind of test that is supported by a `TestEngine` (e.g., JUnit 3, JUnit 4, JUnit Jupiter, Specsy, etc.). See `build.gradle` in the `junit5-gradle-consumer` project for an example of the plugin in action.

Enabling the JUnit Gradle Plugin

To use the JUnit Gradle plugin, you first need to make sure that you are running Gradle 2.5 or higher. Once you've done that, you can configure `build.gradle` as follows.

```
buildscript {
    repositories {
        mavenCentral()
        // The following is only necessary if you want to use SNAPSHOT releases.
        // maven { url 'https://oss.sonatype.org/content/repositories/snapshots' }
    }
    dependencies {
        classpath 'org.junit.platform:junit-platform-gradle-plugin:1.0.0-SNAPSHOT'
    }
}

apply plugin: 'org.junit.platform.gradle.plugin'
```

Configuring the JUnit Gradle Plugin

Once the JUnit Gradle plugin has been applied, you can configure it as follows.

> ⚠ These options are very likely to change as we continue to work towards the final release.

```
junitPlatform {
    platformVersion 1.0
    logManager 'org.apache.logging.log4j.jul.LogManager'
    reportsDir file('build/test-results/junit-platform') // this is the default
    // enableStandardTestTask true
    // selectors (optional)
    // filters (optional)
}
```

Setting `logManager` instructs the JUnit Gradle plugin to set the `java.util.logging.manager` system property to the supplied *fully qualified class* name of the `java.util.logging.LogManager` implementation to use. The above example demonstrates how to configure log4j as the `LogManager`.

By default, the JUnit Gradle plugin disables the standard Gradle `test` task, but this can be overridden via the `enableStandardTestTask` flag.

Configuring Selectors

By default, the plugin will scan your project's output directories for tests. However, you can specify which tests to execute explicitly using the `selectors` extension element.

```
junitPlatform {
    // ...
    selectors {
        uris 'file:///foo.txt', 'http://example.com/'
        uri 'foo:resource' ①
        files 'foo.txt', 'bar.csv'
        file 'qux.json' ②
        directories 'foo/bar', 'bar/qux'
        directory 'qux/bar' ③
        packages 'com.acme.foo', 'com.acme.bar'
        aPackage 'com.example.app' ④
        classes 'com.acme.Foo', 'com.acme.Bar'
        aClass 'com.example.app.Application' ⑤
        methods 'com.acme.Foo#a', 'com.acme.Foo#b'
        method 'com.example.app.Application#run(java.lang.String[])' ⑥
        resources '/bar.csv', '/foo/input.json'
        resource '/com/acme/my.properties' ⑦
    }
    // ...
}
```

① URIs

② Local files

③ Local directories

④ Packages

⑤ Classes, fully qualified class names

⑥ Methods, fully qualified method names (see selectMethod(String) in DiscoverySelectors)

⑦ Classpath resources

Configuring Filters

You can configure filters for the test plan by using the `filters` extension. By default, all engines and tags are included in the test plan. Only the default `includeClassNamePattern` (`^.*Tests?$`) is applied. You can override the default pattern as in the following example. When you specify multiple patterns, they are combined using OR semantics.

```
junitPlatform {
    // ...
    filters {
        engines {
            include 'junit-jupiter'
            // exclude 'junit-vintage'
        }
        tags {
            include 'fast', 'smoke'
            // exclude 'slow', 'ci'
        }
        packages {
            include 'com.sample.included1', 'com.sample.included2'
            // exclude 'com.sample.excluded1', 'com.sample.excluded2'
        }
        includeClassNamePattern '.*Spec'
        includeClassNamePatterns '.*Test', '.*Tests'
    }
    // ...
}
```

If you supply a *Test Engine ID* via `engines {include …}` or `engines {exclude …}`, the JUnit Gradle plugin will only run tests for the desired test engines. Similarly, if you supply a *tag* via `tags {include …}` or `tags {exclude …}`, the JUnit Gradle plugin will only run tests that are *tagged* accordingly (e.g., via the `@Tag` annotation for JUnit Jupiter based tests). The same applies to package names that can be included or excluded using `packages {include …}` or `packages {exclude …}`.

Configuring Test Engines

In order to have the JUnit Gradle plugin 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 {
    testCompile("org.junit.jupiter:junit-jupiter-api:5.0.0-SNAPSHOT")
    testRuntime("org.junit.jupiter:junit-jupiter-engine:5.0.0-SNAPSHOT")
}
```

The JUnit Gradle plugin 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 {
    testCompile("junit:junit:4.12")
    testRuntime("org.junit.vintage:junit-vintage-engine:4.12.0-SNAPSHOT")
}
```

Using the JUnit Gradle Plugin

Once the JUnit Gradle plugin has been applied and configured, you have a new `junitPlatformTest` task at your disposal.

Invoking `gradlew junitPlatformTest` (or `gradlew test`) from the command line will execute all tests within the project whose class names match the regular expression supplied via the `includeClassNamePattern` (which defaults to `^.*Tests?$`).

Executing the `junitPlatformTest` task in the `junit5-gradle-consumer` project results in output similar to the following:

```
:junitPlatformTest

Test run finished after 93 ms
[         3 containers found      ]
[         0 containers skipped    ]
[         3 containers started    ]
[         0 containers aborted    ]
[         3 containers successful ]
[         0 containers failed     ]
[         3 tests found           ]
[         1 tests skipped         ]
[         2 tests started         ]
[         0 tests aborted         ]
[         2 tests successful      ]
[         0 tests failed          ]

BUILD SUCCESSFUL
```

If a test fails, the build will fail with output similar to the following:

```
:junitPlatformTest

Test failures (1):
  JUnit Jupiter:SecondTest:mySecondTest()
    MethodSource [className = 'com.example.project.SecondTest', methodName =
'mySecondTest', methodParameterTypes = '']
    => Exception: 2 is not equal to 1 ==> expected: <2> but was: <1>

Test run finished after 99 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         ]
[         2 tests successful      ]
[         1 tests failed          ]

:junitPlatformTest FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':junitPlatformTest'.
> Process 'command
'/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java'' finished
with non-zero exit value 1
```

The *exit value* is `1` if any containers or tests failed; otherwise, it is `0`.

Current Limitations of the JUnit Gradle Plugin

The results of any tests run via the JUnit Gradle plugin will not be included in the standard test report generated by Gradle; however, the test results can typically be aggregated on a CI server. See the `reportsDir` property of the plugin.

4.2.2. Maven

The JUnit team has developed a very basic provider for Maven Surefire that lets you run JUnit 4 and JUnit Jupiter tests via `mvn test`. The `pom.xml` file in the `junit5-maven-consumer` project demonstrates how to use it and can serve as a starting point.

```
...
<build>
    <plugins>
        ...
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.19</version>
            <dependencies>
                <dependency>
                    <groupId>org.junit.platform</groupId>
                    <artifactId>junit-platform-surefire-provider</artifactId>
                    <version>1.0.0-SNAPSHOT</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
...
```

Configuring Test Engines

In order to have Maven Surefire run any tests at all, a `TestEngine` implementation must be added to the runtime classpath.

To configure support for JUnit Jupiter based tests, configure a `test` dependency on the JUnit Jupiter API, and add the JUnit Jupiter `TestEngine` implementation to the dependencies of the `maven-surefire-plugin` similar to the following.

```
...
<build>
    <plugins>
        ...
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.19</version>
            <dependencies>
                <dependency>
                    <groupId>org.junit.platform</groupId>
                    <artifactId>junit-platform-surefire-provider</artifactId>
                    <version>1.0.0-SNAPSHOT</version>
                </dependency>
                <dependency>
                    <groupId>org.junit.jupiter</groupId>
                    <artifactId>junit-jupiter-engine</artifactId>
                    <version>5.0.0-SNAPSHOT</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
...
<dependencies>
    ...
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.0.0-SNAPSHOT</version>
        <scope>test</scope>
    </dependency>
</dependencies>
...
```

The JUnit Platform Surefire Provider can run JUnit 4 based tests as long as you configure a `test` dependency on JUnit 4 and add the JUnit Vintage `TestEngine` implementation to the dependencies of the `maven-surefire-plugin` similar to the following.

```
...
<build>
    <plugins>
        ...
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.19</version>
            <dependencies>
                <dependency>
                    <groupId>org.junit.platform</groupId>
                    <artifactId>junit-platform-surefire-provider</artifactId>
                    <version>1.0.0-SNAPSHOT</version>
                </dependency>
                ...
                <dependency>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                    <version>4.12.0-SNAPSHOT</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
...
<dependencies>
    ...
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
...
```

Filtering by tags

You can filter tests by tags using the following configuration properties.

- to include a tag, use either `groups` or `includeTags`
- to exclude a tag, use either `excludedGroups` or `excludeTags`

```
...
<build>
    <plugins>
        ...
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.19</version>
            <configuration>
                <properties>
                    <includeTags>acceptance</includeTags>
                    <excludeTags>integration, regression</excludeTags>
                </properties>
            </configuration>
            <dependencies>
                ...
            </dependencies>
        </plugin>
    </plugins>
</build>
...
```

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-1.0.0-SNAPSHOT.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-1.0.0-SNAPSHOT.jar <Options>
```

Here's an example of its output:

```
├── JUnit Vintage
│   └── example.JUnit4Tests
│      └── standardJUnit4Test
└── JUnit Jupiter
   ├── StandardTests
   │   ├── succeedingTest()
   │   └── skippedTest()    for demonstration purposes
   └── A special test case
      ├── Custom test name containing spaces
      ├──   °□°)
      └──


Test run finished after 64 ms
[         5 containers found      ]
[         0 containers skipped    ]
[         5 containers started    ]
[         0 containers aborted    ]
[         5 containers successful ]
[         0 containers failed     ]
[         6 tests found           ]
[         1 tests skipped         ]
[         5 tests started         ]
[         0 tests aborted         ]
[         5 tests successful      ]
[         0 tests failed          ]
```

Exit Code

The `ConsoleLauncher` exits with a status code of `1` if any containers or tests failed. Otherwise the exit code is `0`.

4.3.1. Options

These options are very likely to change as we continue to work towards the final release.

```
Option                                        Description
------                                        -----------
-h, --help                                    Display help information.
--disable-ansi-colors                         Disable ANSI colors in output (not
                                                supported by all terminals).
--details <[none,flat,tree,verbose]>          Select an output details mode for when
                                                tests are executed. Use one of: [none,
                                                flat, tree, verbose]. If 'none' is
                                                selected, then only the summary and
test
                                                failures are shown. (default: tree)
--details-theme <[ascii,unicode]>             Select an output details tree theme for
```

```
                                             when tests are executed. Use one of:
                                             [ascii, unicode] (default: unicode)
--class-path, --classpath, --cp <Path:     Provide additional classpath entries --
  path1:path2:...>                           for example, for adding engines and
                                             their dependencies. This option can be
                                             repeated.
--reports-dir <Path>                       Enable report output into a specified
                                             local directory (will be created if it
                                             does not exist).
--scan-class-path, --scan-classpath [Path: Scan all directories on the classpath or
  path1:path2:...]                           explicit classpath roots. Without
                                             arguments, only directories on the
                                             system classpath as well as additional
                                             classpath entries supplied via -cp
                                             (directories and JAR files) are
scanned.
                                             Explicit classpath roots that are not
on
                                             the classpath will be silently
ignored.
                                             This option can be repeated.
-u, --select-uri <URI>                     Select a URI for test discovery. This
                                             option can be repeated.
-f, --select-file <String>                 Select a file for test discovery. This
                                             option can be repeated.
-d, --select-directory <String>            Select a directory for test discovery.
                                             This option can be repeated.
-p, --select-package <String>              Select a package for test discovery.
This
                                             option can be repeated.
-c, --select-class <String>                Select a class for test discovery. This
                                             option can be repeated.
-m, --select-method <String>               Select a method for test discovery. This
                                             option can be repeated.
-r, --select-resource <String>             Select a classpath resource for test
                                             discovery. This option can be
repeated.
-n, --include-classname <String>           Provide a regular expression to include
                                             only classes whose fully qualified
names
                                             match. To avoid loading classes
                                             unnecessarily, the default pattern
only
                                             includes class names that end with
                                             "Test" or "Tests". When this option is
                                             repeated, all patterns will be
combined
                                             using OR semantics. (default:
^.*Tests?$)
-N, --exclude-classname <String>           Provide a regular expression to exclude
                                             those classes whose fully qualified
```

```
                                                names match. When this option is
                                                repeated, all patterns will be
combined
                                                using OR semantics.
--include-package <String>                    Provide a package to be included in the
                                                test run. This option can be repeated.
--exclude-package <String>                    Provide a package to be excluded from
the
                                                test run. This option can be repeated.
-t, --include-tag <String>                    Provide a tag to be included in the test
                                                run. This option can be repeated.
-T, --exclude-tag <String>                    Provide a tag to be excluded from the
test
                                                run. This option can be repeated.
-e, --include-engine <String>                 Provide the ID of an engine to be
included
                                                in the test run. This option can be
                                                repeated.
-E, --exclude-engine <String>                 Provide the ID of an engine to be
excluded
                                                from the test run. This option can be
                                                repeated.
```

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 — 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-4.12.jar` in *test* scope: to run tests using JUnit 4.
- `junit-platform-runner` in *test* scope: location of the `JUnitPlatform` runner
- `junit-jupiter-api` in *test* scope: API for writing tests, including `@Test`, etc.

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

Transitive Dependencies

- `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

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 — for example, fully qualified class names — 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)`.

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, IDEs won't 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 {

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

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

}
```

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.

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

@RunWith(JUnitPlatform.class)
@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 match the pattern `^.*Tests?$`.

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

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 explicitly via `@ExtendWith` 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(…)` and supplying class references for the extensions to register.

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

```java
@ExtendWith(MockitoExtension.class)
@Test
void mockTest() {
    // ...
}
```

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

```java
@ExtendWith(MockitoExtension.class)
class MockTests {
    // ...
}
```

Multiple extensions can be registered together like this:

```java
@ExtendWith({ FooExtension.class, BarExtension.class })
class MyTestsV1 {
    // ...
}
```

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

```java
@ExtendWith(FooExtension.class)
@ExtendWith(BarExtension.class)
class MyTestsV2 {
    // ...
}
```

The execution of tests in both `MyTestsV1` and `MyTestsV2` will be extended by the `FooExtension` and `BarExtension`, in exactly that order.

5.2.2. Automatic Extension Registration

In addition to declarative 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.extensions.autodetection.enabled` configuration key to `true`. This can be supplied as a JVM system property or as a *configuration parameter* in the `LauncherDiscoveryRequest` that is passed to the `Launcher`.

For example, to enable auto-detection of extensions, you can start your JVM with the following system property.

`-Djunit.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.).

5.2.3. 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

`ContainerExecutionCondition` and `TestExecutionCondition` define the `Extension` APIs for programmatic, *conditional test execution*.

A `ContainerExecutionCondition` is *evaluated* to determine if all tests in a given container (e.g., a test class) should be executed based on the supplied `ContainerExtensionContext`. Similarly, a `TestExecutionCondition` is *evaluated* to determine if a given test method should be executed based on the supplied `TestExtensionContext`.

When multiple `ContainerExecutionCondition` or `TestExecutionCondition` extensions are registered, a container or test, respectively, 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.conditions.deactivate` configuration key 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 or as a *configuration parameter* in the `LauncherDiscoveryRequest` that is passed to the `Launcher`.

For example, to deactivate JUnit's `@Disabled` condition, you can start your JVM with the following system property.

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

Pattern Matching Syntax

If the `junit.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 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 concrete examples, consult the source code for the `MockitoExtension` and the `SpringExtension`.

5.5. Parameter Resolution

`ParameterResolver` defines the `Extension` API for dynamically resolving 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 `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`.

5.6. 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.

5.6.1. 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.Namespace;
import org.junit.jupiter.api.extension.ExtensionContext.Store;
import org.junit.jupiter.api.extension.TestExtensionContext;

public class TimingExtension implements BeforeTestExecutionCallback,
AfterTestExecutionCallback {

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

    @Override
    public void beforeTestExecution(TestExtensionContext context) throws Exception {
        getStore(context).put(context.getTestMethod().get(), System.currentTimeMillis
());
    }

    @Override
    public void afterTestExecution(TestExtensionContext context) throws Exception {
        Method testMethod = context.getTestMethod().get();
        long start = getStore(context).remove(testMethod, long.class);
        long duration = System.currentTimeMillis() - start;

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

    private Store getStore(TestExtensionContext context) {
        return context.getStore(Namespace.create(getClass(), context));
    }

}
```

Since the `TimingExtensionTests` class registers the `TimingExtension` via `@ExtendWith`, its tests will have this timing applied when they execute.

A test class that uses the example TimingExtension

```
@ExtendWith(TimingExtension.class)
class TimingExtensionTests {

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

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

}
```

The following is an example of the logging produced when `TimingExtensionTests` is run.

```
INFO: Method [sleep20ms] took 24 ms.
INFO: Method [sleep50ms] took 53 ms.
```

5.7. 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 {

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

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

5.8. 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 a `TestExtensionContext` during test execution will not be available in the surrounding `ContainerExtensionContext`. Since `ContainerExtensionContexts` 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`.

5.9. 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, 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.

5.10. 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.

Figure 2: 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 Figure 2.

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

Step	Interface/Annotation	Description
11	annotation `org.junit.jupiter.api.AfterAll`	user code executed after all tests of the container are executed
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.

6.2. 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 codebases including 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.

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 codebases 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.

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 – build tools and IDEs – more powerful and stable. The purpose is to decouple the internals of discovering and executing tests from all the filtering and configuration that's 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 – like Spock, Cucumber, and FitNesse – can plug into the JUnit Platform's launching infrastructure by providing a custom `TestEngine`.

The launching API is in the `junit-platform-launcher` module.

An example consumer of the launching 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.jupiter.api.Test;
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.LauncherDiscoveryRequestBuilder;
import org.junit.platform.launcher.core.LauncherFactory;
import org.junit.platform.launcher.listeners.SummaryGeneratingListener;
```

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

TestPlan plan = LauncherFactory.create().discover(request);
```

There's currently the possibility to search for classes, methods, all classes in a package, or even all tests in the classpath. Discovery takes place across all participating test engines.

The resulting test plan is basically a hierarchical (and read-only) description of all engines, classes, and test methods that fit the `specification` object. 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 tree has a *unique ID* that can be used to invoke a particular test or group of tests.

7.1.2. Executing Tests

There are two ways to execute tests. Clients can either use the same test specification object as in the discovery phase, or – to speed things up a bit – pass in the prepared `TestPlan` object from a previous discovery step. Test progress and result reporting can be achieved through a `TestExecutionListener`:

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

Launcher launcher = LauncherFactory.create();

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

launcher.execute(request);
```

There's currently no result object, 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 out of the box:

- `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. Engine registration is currently 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.

7.1.4. Plugging in Your Own Test Execution Listeners

In addition to the public `Launcher` API method for registering test execution listeners programmatically, custom `TestExecutionListener` implementations discovered at runtime via Java's `java.util.ServiceLoader` facility are automatically registered with the `DefaultLauncher`. 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.

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 Annotations

Every published artifact has a version number `<major>.<minor>.<patch>` and all publicly available interfaces, classes, and methods are annotated with @API. The annotation's `Usage` value can be assigned one of the following five values:

Usage	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 `Usage` value of lower stability.

8.2. Tooling Support

The JUnit team plans to provide native tooling support for all JUnit users, extenders, and tool builders. The tooling support will provide a means to check if the 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

5.0.0-ALPHA

Date of Release: February 1, 2016

Scope: Alpha release of JUnit 5

5.0.0-M1

Date of Release: July 7, 2016

Scope: First milestone release of JUnit 5

Summary of Changes

The following is a list of global changes. For details regarding changes specific to the Platform, Jupiter, and Vintage, consult the dedicated subsections below. For a complete list of all *closed* issues and pull requests for this release, consult the 5.0 M1 milestone page in the JUnit repository on GitHub.

- JAR manifests in published artifacts now contain additional metadata such as `Created-By`, `Built-By`, `Build-Date`, `Build-Time`, `Build-Revision`, `Implementation-Title`, `Implementation-Version`, `Implementation-Vendor`, etc.
- Published artifacts now contain `LICENSE.md` in `META-INF`.
- JUnit now participates in the Up For Grabs movement for open source contributions.
  - See the up-for-grabs label on GitHub.
- Group IDs, artifact IDs, and versions have changed for all published artifacts.
  - See Artifact Migration and Dependency Metadata.
- All base packages have been renamed.
  - See Package Migration.

Table 1. Artifact Migration

Old Group ID	Old Artifact ID	New Group ID	New Artifact ID	New Base Version
`org.junit`	`junit-commons`	`org.junit.platform`	`junit-platform-commons`	`1.0.0`
`org.junit`	`junit-console`	`org.junit.platform`	`junit-platform-console`	`1.0.0`
`org.junit`	`junit-engine-api`	`org.junit.platform`	`junit-platform-engine`	`1.0.0`
`org.junit`	`junit-gradle`	`org.junit.platform`	`junit-platform-gradle-plugin`	`1.0.0`
`org.junit`	`junit-launcher`	`org.junit.platform`	`junit-platform-launcher`	`1.0.0`
`org.junit`	`junit4-runner`	`org.junit.platform`	`junit-platform-runner`	`1.0.0`
`org.junit`	`surefire-junit5`	`org.junit.platform`	`junit-platform-surefire-provider`	`1.0.0`
`org.junit`	`junit5-api`	`org.junit.jupiter`	`junit-jupiter-api`	`5.0.0`

Old Group ID	Old Artifact ID	New Group ID	New Artifact ID	New Base Version
org.junit	junit5-engine	org.junit.jupiter	junit-jupiter-engine	5.0.0
org.junit	junit4-engine	org.junit.vintage	junit-vintage-engine	4.12.0

Table 2. Package Migration

Old Base Package	New Base Package
org.junit.gen5.api	org.junit.jupiter.api
org.junit.gen5.commons	org.junit.platform.commons
org.junit.gen5.console	org.junit.platform.console
org.junit.gen5.engine.junit4	org.junit.vintage.engine
org.junit.gen5.engine.junit5	org.junit.jupiter.engine
org.junit.gen5.engine	org.junit.platform.engine
org.junit.gen5.gradle	org.junit.platform.gradle.plugin
org.junit.gen5.junit4.runner	org.junit.platform.runner
org.junit.gen5.launcher	org.junit.platform.launcher
org.junit.gen5.launcher.main	org.junit.platform.launcher.core
org.junit.gen5.surefire	org.junit.platform.surefire.provider

JUnit Platform

- The `ConsoleRunner` has been renamed to `ConsoleLauncher`.
- `ConsoleLauncher` now always returns the status code on exit, and the *enable exit code* flags have been removed.
- The `junit-platform-console` artifact no longer defines transitive dependencies on `junit-platform-runner`, `junit-jupiter-engine`, or `junit-vintage-engine`.
- The `JUnit5` Runner has been renamed to `JUnitPlatform`.
  - `@Packages` has been renamed to `@SelectPackages`.
  - `@Classes` has been renamed to `@SelectClasses`.
  - `@UniqueIds` has been removed.
  - `@UseTechnicalNames` has been introduced.
    - See Display Names vs. Technical Names.
- The Gradle plugin for the JUnit Platform has been completely overhauled.
  - The JUnit Platform Gradle plugin now requires Gradle 2.5 or higher.

- The `junit5Test` Gradle task has been renamed to `junitPlatformTest`.
- The `junit5` Gradle plugin configuration has been renamed to `junitPlatform`.
  - `runJunit4` has been replaced by `enableStandardTestTask`.
  - `version` has been replaced by `platformVersion`.
- See Gradle for further details.

- XML test report generation has been overhauled.
  - XML reports now contain newlines.
  - Attributes specific to the JUnit Platform that do not align with standard attributes in the de facto standard XML schema are now contained in `CDATA` blocks within the `<system-out>` element.
  - XML reports now use real method names and fully qualified class names instead of display names.
- Unique ID in `TestIdentifier` is now a `String`.
- `TestSource` is now an interface with a dedicated hierarchy consisting of `CompositeTestSource`, `JavaSource`, `JavaPackageSource`, `JavaClassSource`, `JavaMethodSource`, `UriSource`, `FileSystemSource`, `DirectorySource`, and `FileSource`.
- All `DiscoverySelector` factory methods have been moved to a new `DiscoverySelectors` class that serves as a centralized collection of all *select* methods.
- `Filter.filter()` has been renamed to `Filter.apply()`.
- `TestTag.of()` has been renamed to `TestTag.create()`.
- `TestDiscoveryRequest` has been renamed to `LauncherDiscoveryRequest`.
- `TestDiscoveryRequestBuilder` has been renamed to `LauncherDiscoveryRequestBuilder`.
- `LauncherDiscoveryRequest` is now immutable.
- `TestDescriptor.allDescendants()` has been renamed to `TestDescriptor.getAllDescendants()`.
- `TestEngine#discover(EngineDiscoveryRequest)` has been replaced by `TestEngine#discover(EngineDiscoveryRequest, UniqueId)`.
- Introduced `ConfigurationParameters` which the `Launcher` supplies to engines via the `EngineDiscoveryRequest` and `ExecutionRequest`
- The `Container` and `Leaf` abstractions have been removed from the `HierarchicalTestEngine`.
- The `getName()` method has been removed from `TestIdentifier` and `TestDescriptor` in favor of retrieving an implementation specific name via the `TestSource`.
- Test engines are now permitted to be completely dynamic in nature. In other words, a `TestEngine` is no longer required to create `TestDescriptor` entries during the *discovery phase*; a `TestEngine` may now optionally register containers and tests dynamically during the *execution phase*.
- Include and exclude support for engines and tags has been completely revised.
  - Engines and tags can no longer be *required* but rather *included*.
  - `ConsoleLauncher` now supports the following options: `t/include-tag`, `T/exclude-tag`, `e/include-`

engine, E/exclude-engine.

  - The Gradle plugin now supports engines and tags configuration blocks with nested include and exclude entries.
  - EngineFilter now supports includeEngines() and excludeEngines() factory methods.
  - The JUnitPlatform runner now supports @IncludeTags, @ExcludeTags, @IncludeEngines, and @ExcludeEngines.

JUnit Jupiter

- The junit5 engine ID has been renamed to junit-jupiter.
- JUnit5TestEngine has been renamed to JupiterTestEngine.
- Assertions now provides the following support:
  - assertEquals() for primitive types
  - assertEquals() for doubles and floats with deltas
  - assertArrayEquals()
  - Expected and actual values are now supplied to the AssertionFailedError.
- Dynamic Tests: tests can now be registered dynamically at runtime via lambda expressions.
- TestInfo now provides access to tags via getTags().
- @AfterEach methods and *after* callbacks are now invoked if an exception is thrown by a @Test method, a @BeforeEach method, or a *before* callback.
- @AfterAll methods and *after all* callbacks are now guaranteed to be invoked.
- Repeatable annotations such as @ExtendWith and @Tag are now discovered in superclasses within a test class hierarchy as well as on interfaces.
- Extensions are now registered *top-down* within a test class or interface hierarchy.
- Test and container execution *conditions* can now be deactivated.
- InstancePostProcessor has been renamed to TestInstancePostProcessor.
  - TestInstancePostProcessor implementations are now properly applied within @Nested test class hierarchies.
- MethodParameterResolver has been renamed to ParameterResolver.
  - The ParameterResolver API is now based on java.lang.reflect.Executable and can therefore be used to resolve parameters for methods *and* constructors.
  - New ParameterContext which is passed to the supports() and resolve() methods of ParameterResolver extensions.
  - Resolution of primitive types is now supported for ParameterResolver extensions.
- The ExtensionPointRegistry and ExtensionRegistrar have been removed in favor of declarative registration via @ExtendWith.
- BeforeAllExtensionPoint has been renamed to BeforeAllCallback.
- AfterAllExtensionPoint has been renamed to AfterAllCallback.

- `BeforeEachExtensionPoint` has been renamed to `BeforeEachCallback`.
- `BeforeAllExtensionPoint` has been renamed to `BeforeAllCallback`.
- New `BeforeTestExecutionCallback` and `AfterTestExecutionCallback` extension APIs.
- `ExceptionHandlerExtensionPoint` has been renamed to `TestExecutionExceptionHandler`.
- Test exceptions are now supplied to extensions via the `TestExtensionContext`.
- `ExtensionContext.Store` now supports type-safe variants of many of its methods.
- `ExtensionContext.getElement()` now returns an `Optional`.
- `Namespace.of()` has been renamed to `Namespace.create()`.
- `TestInfo` and `ExtensionContext` have new `getTestClass()` and `getTestMethod()` methods.
- The `getName()` method has been removed from `TestInfo` and `ExtensionContext` in favor of retrieving a context specific name via the current test class or test method.

JUnit Vintage

- The `junit4` engine ID has been renamed to `junit-vintage`.
- `JUnit4TestEngine` has been renamed to `VintageTestEngine`.

5.0.0-M2

Date of Release: July 23, 2016

Scope: Second milestone release of JUnit 5

Summary of Changes

This release is primarily a bugfix release for bugs discovered since `5.0.0-M1`.

The following is a list of global changes. For details regarding changes specific to the Platform, Jupiter, and Vintage, consult the dedicated subsections below. For a complete list of all *closed* issues and pull requests for this release, consult the 5.0 M2 milestone page in the JUnit repository on GitHub.

- The JUnit 5 Gradle build now runs properly on Microsoft Windows.
- A continuous integration build against Microsoft Windows has been set up on AppVeyor.

JUnit Platform

Bug Fixes

- Failures in containers — for example, `@BeforeAll` methods that throw exceptions — now fail the build when using the `ConsoleLauncher` or the JUnit Platform Gradle plugin.
- The JUnit Platform Surefire Provider no longer silently ignores purely dynamic test classes — for example, test classes that only declare `@TestFactory` methods.
- The `junit-platform-console` and `junit-platform-console.bat` shell scripts included in the `junit-`

`platform-console-<release version>` TAR and ZIP distributions now properly refer to the `ConsoleLauncher` instead of the `ConsoleRunner`.

- The `TestExecutionSummary` used by the `ConsoleLauncher` and the JUnit Platform Gradle plugin now includes the actual exception type for failures.
- Classpath scanning is now safeguarded against exceptions encountered during class loading and processing — for example, when processing classes with malformed names. The underlying exception is swallowed and logged along with the offending file path. If the exception is a *blacklisted exception* such as an `OutOfMemoryError`, however, it will be rethrown.

Deprecations

- Generic name-based discovery selectors (i.e., `selectName()` and `selectNames()`) in `DiscoverySelectors` have been deprecated in favor of the dedicated `selectPackage(String)`, `selectClass(String)`, and `selectMethod(String)` methods.

New Features

- New `selectMethod(String)` method in `DiscoverySelectors` that supports selection of a *fully qualified method name*.

JUnit Jupiter

Bug Fixes

- Extension implementations declared via `@ExtendWith` at the class-level and at the method-level will no longer be registered multiple times.

JUnit Vintage

No changes since 5.0.0-M1

5.0.0-M3

Date of Release: November 30, 2016

Scope: Third milestone release of JUnit 5 with a focus on JUnit 4 interoperability, additional discovery selectors, and documentation.

For a complete list of all *closed* issues and pull requests for this release, consult the 5.0 M3 milestone page in the JUnit repository on GitHub.

JUnit Platform

Bug Fixes

- `ColoredPrintingTestListener`, which is used by the `ConsoleLauncher`, now outputs the actual exception type and its stack trace when printing an exception message.
- Test classes in the *default* package are now picked up via classpath scanning when scanning

classpath roots — for example, in conjunction with the JUnit Platform Gradle plugin when no explicit packages have been selected.

- Classpath scanning no longer loads classes that are excluded by a class name filter.
- Classpath scanning no longer attempts to load Java 9 `module-info.class` files.

Deprecations and Breaking Changes

- `ClasspathSelector` has been renamed to `ClasspathRootSelector` to avoid confusion with `ClasspathResourceSelector`.
- `JavaPackageSource` has been renamed to `PackageSource` to align with `PackageSelector`.
- `JavaClassSource` has been renamed to `ClassSource` to align with `ClassSelector`.
- `JavaMethodSource` has been renamed to `MethodSource` to align with `MethodSelector`.
- `PackageSource`, `ClassSource`, and `MethodSource` now directly implement the `TestSource` interface instead of the `JavaSource` interface which has been removed.
- Generic name-based discovery selectors (i.e., `selectName()` and `selectNames()`) in `DiscoverySelectors` have been deprecated in favor of the dedicated `selectPackage(String)`, `selectClass(String)`, and `selectMethod(String)` methods.
- `ClassFilter` has been renamed to `ClassNameFilter` and now implements `DiscoveryFilter<String>` instead of `DiscoveryFilter<Class<?>>` so it can be applied before loading classes during classpath scanning.
- `ClassNameFilter.includeClassNamePattern` is now deprecated in favor of `ClassNameFilter.includeClassNamePatterns`.
- `@IncludeClassNamePattern` is now deprecated in favor of `@IncludeClassNamePatterns`.
- The `-p` command-line option for configuring additional classpath entries for the `ConsoleLauncher` has been renamed to `-cp` in order to align with the option names for the standard `java` executable. In addition, a new `--class-path` alias has been introduced, while the existing `--classpath` command-line option remains unchanged.
- The `-a` and `--all` command-line options for the `ConsoleLauncher` have been renamed to `--scan-class-path`.
- The `--xml-reports-dir` command-line option for the `ConsoleLauncher` has been renamed to `--reports-dir`.
- The `-C`, `-D`, and `-r` short command-line options for the `ConsoleLauncher` have been removed in favor of using their *long* equivalents `--disable-ansi-colors`, `--hide-details`, and `--reports-dir` respectively.
- The `ConsoleLauncher` no longer supports passing non-option arguments. Please use the new explicit selector options to specify package, class, or method names instead. Alternatively, `--scan-class-path` now accepts an optional argument that can be used to select classpath entries to be scanned. Multiple classpath entries may be used by separating them using the system's path separator (`;` on Windows, `:` on Unix).
- `LauncherDiscoveryRequestBuilder` no longer accepts `null` values for `selectors`, `filters`, or maps of configuration parameters.

- Filters for class name, engine IDs, and tags in the Gradle plugin configuration now need to be wrapped in a `filters` extension element (see Configuring Filters).

New Features

- New `selectUri(…)` methods in `DiscoverySelectors` for selecting URIs. A `TestEngine` can retrieve such values by querying registered instances of `UriSelector`.
- New `selectFile(…)` and `selectDirectory(…)` methods in `DiscoverySelectors` for selecting files and directories in the file system. A `TestEngine` can retrieve such values by querying registered instances of `FileSelector` and `DirectorySelector`.
- New `selectClasspathResource(String)` method in `DiscoverySelectors` for selecting *classpath resources* such as XML or JSON files by name, where the name is a `/`-separated path name for the resource within the current classpath. A `TestEngine` can retrieve such values by querying registered instances of `ClasspathResourceSelector`. Furthermore, a classpath resource can be made available as the `TestSource` of a `TestIdentifier` via the new `ClasspathResourceSource`.
- The `selectMethod(String)` method in `DiscoverySelectors` now supports selection of a *fully qualified method name* for a method that accepts parameters — for example, `"org.example.TestClass#testMethod(org.junit.jupiter.api.TestInfo)"`.
- The `TestExecutionSummary` used by the `ConsoleLauncher` and the JUnit Platform Gradle plugin now includes statistics for all container events in addition to test events.
- The `TestExecutionSummary` can now be used to get the list of all failures.
- The `ConsoleLauncher`, the Gradle plugin, and the `JUnitPlatform` runner now use `^.*Tests?$` as the default pattern for class names to be included in the test run.
- The Gradle plugin now allows to explicitly select which tests should be executed (see Configuring Selectors).
- New `@Testable` annotation that `TestEngine` implementations can use to signal to IDEs and tooling vendors that the annotated or meta-annotated element is *testable* (i.e. that it can be executed as a test on the JUnit Platform).
- Multiple regular expressions that are combined using OR semantics can now be passed to `ClassNameFilter.includeClassNamePatterns`.
- Multiple regular expressions that are combined using OR semantics can now be passed via `@IncludeClassNamePatterns` to the `JUnitPlatform` runner.
- Multiple regular expressions that are combined using OR semantics can now be passed to the JUnit Platform Gradle plugin (see Configuring Filters) and the `ConsoleLauncher` (see Options).
- Package names can now be included or excluded using `PackageNameFilter.includePackageNames` or `PackageNameFilter.excludePackageNames`.
- Package names can now be included or excluded using `@IncludePackages` and `@ExcludePackages` in combination with the `JUnitPlatform` runner.
- Package names can now be included or excluded via a filter configuration to the JUnit Platform Gradle plugin (see Configuring Filters) and the `ConsoleLauncher` (see Options).
- The `junit-platform-console` no longer has a mandatory dependency on JOpt Simple. Thus, it's now possible to test custom code that uses a different version of that library.

- Resolution of package selectors now also scans JAR files.
- The Surefire provider now supports forking.
- The Surefire provider now supports filtering by tags using one of the following parameters:
  - include: `groups`/`includeTags`
  - exclude: `excludedGroups`/`excludeTags`
- The Surefire provider is now licensed under the Apache License v2.0.

JUnit Jupiter

Bug Fixes

- `@AfterEach` methods are now executed with *bottom-up* semantics within a test class hierarchy.
- `DynamicTest.stream()` now accepts a `ThrowingConsumer` instead of a conventional `Consumer` for its *test executor*, thereby allowing for custom streams of dynamic tests that may potentially throw checked exceptions.
- Extensions registered at the test method level are now used when invoking `@BeforeEach` and `@AfterEach` methods for the corresponding test method.
- The `JupiterTestEngine` now supports selection of test methods via their unique ID for methods that accept arrays or primitive types as parameters.
- `ExtensionContext.Store` is now thread-safe.

Deprecations and Breaking Changes

- The `Executable` functional interface has been relocated to a new dedicated `org.junit.jupiter.api.function` package.
- `Assertions.expectThrows()` has been deprecated in favor of `Assertions.assertThrows()`.

New Features and Improvements

- Support for lazy and preemptive *timeouts* with lambda expressions in `Assertions`. See examples in `AssertionsDemo` and consult the `org.junit.jupiter.Assertions` Javadoc for further details.
- New `assertIterableEquals()` assertion that checks that two Iterables are deeply equal (see Javadoc for details).
- New variants of `Assertions.assertAll()` that accept streams of executables (i.e., `Stream<Executable>`).
- `Assertions.assertThrows()` now returns the thrown exception.
- `@BeforeAll` and `@AfterAll` may now be declared on static methods in interfaces.
- The JUnit 4 Rules `org.junit.rules.ExternalResource`, `org.junit.rules.Verifier`, `org.junit.rules.ExpectedException` including subclasses are now supported in JUnit Jupiter to facilitate migration of JUnit 4 codebases.

JUnit Vintage

No changes since 5.0.0-M2

5.0.0-M4

Date of Release: April 1, 2017

Scope: Fourth milestone release of JUnit 5 with a focus on test templates, repeated tests, and parameterized tests.

For a complete list of all *closed* issues and pull requests for this release, consult the 5.0 M4 milestone page in the JUnit repository on GitHub.

JUnit Platform

Bug Fixes

- The JUnit Platform Gradle plugin now adds its dependencies with a fixed version (same as the plugin version) instead of a dynamic versioning scheme (was `1.+`) by default to ensure reproducible builds.
- The JUnit Platform Gradle plugin now explicitly applies the Gradle built-in `java` plugin as it has an implicit dependency on it being applied.
- All `findMethods()` implementations in `ReflectionUtils` no longer return synthetic methods. Shadowed override-equal methods are also no longer included in the results.
- Introduced `getLegacyReportingName()` in `TestIdentifier` and `TestDescriptor`. This allows the JUnit Platform Gradle Plugin and the Surefire Provider to resolve the class and method names through `getLegacyReportingName()` instead of using the source location.
- JAR files that contain spaces in their paths (`%20`) are now properly decoded before being used in classpath scanning routines.
- Updated `findNestedClasses()` in `ReflectionUtils` so that searches for nested classes also return inherited nested classes (regardless of whether they are static or not).

Deprecations and Breaking Changes

- All *test suite* annotations from the `org.junit.platform.runner` package have been moved to the `org.junit.platform.suite.api` package in the new `junit-platform-suite-api` module. This includes annotations such as `@SelectClasses`, `@SelectPackages`, etc.
- Removed deprecated `selectNames()` name-based discovery selector from `DiscoverySelectors` in favor of the dedicated `selectPackage(String)`, `selectClass(String)`, and `selectMethod(String)` methods.
- `ClassNameFilter.includeClassNamePattern` was removed; please use `ClassNameFilter.includeClassNamePatterns` instead.
- `@IncludeClassNamePattern` has been removed; please use `@IncludeClassNamePatterns` instead.
- The `--hide-details` option of the `ConsoleLauncher` is deprecated; use `--details none` instead.

- The ZIP distribution containing the console launcher is no longer provided. It has been replaced by an executable standalone JAR distribution. For details, see the "New Features" section below.
- The `MethodSortOrder` enum in `ReflectionUtils` has been renamed to `HierarchyTraversalMode`. Those affected should now use `ReflectionSupport` instead of `ReflectionUtils`.
- The method `execute(LauncherDiscoveryRequest launcherDiscoveryRequest)` in `Launcher` has been deprecated and will be removed in milestone M5. Instead use the following new method that registers supplied `TestExecutionListeners` in addition to already registered listeners but only for the supplied `LauncherDiscoveryRequest`: `execute(LauncherDiscoveryRequest launcherDiscoveryRequest, TestExecutionListener… listeners)`

New Features and Improvements

- Custom `TestExecutionListener` implementations can now be registered automatically via Java's `ServiceLoader` mechanism.
- New `getGroupId()`, `getArtifactId()`, and `getVersion()` default methods in the `TestEngine` API for debugging and reporting purposes. By default, package attributes (i.e., typically from JAR manifest attributes) are used to determine the artifact ID and version; whereas, the group ID is empty by default. Consult the Javadoc for `TestEngine` for details.
- Logging information for discovered test engines has been enhanced to include the group ID, artifact ID, and version of each test engine if available via the `getGroupId()`, `getArtifactId()`, and `getVersion()` methods.
- The `--scan-classpath` option of the `ConsoleLauncher` now allows to scan JAR files for tests when they are supplied as explicit arguments (see Options).
- The new `--details <Details>` option of the `ConsoleLauncher` now allows to select an output details mode for when tests are executed. Use one of: `none`, `flat`, `tree` or `verbose`. If `none` is selected, then only the summary and test failures are shown (see Options).
- The new `--details-theme <Theme>` option of the `ConsoleLauncher` now allows to select a theme for when tests are executed and printed as a tree. Use one of: `ascii` or `unicode` (see Options).
- An executable `junit-platform-console-standalone-1.0.0-SNAPSHOT.jar` artifact containing the Jupiter and Vintage test engines with all dependencies is generated by the default build process, stored in `junit-platform-console-standalone/build/libs`, and published to Maven Central. It provides hassle-free usage of JUnit 5 in projects that manually manage their dependencies similar to the plain-old JAR known from JUnit 4.
- New `--exclude-classname` (`--N`) option added to the `ConsoleLauncher` accepting a regular expression to exclude those classes whose fully qualified names match. When this option is repeated, all patterns will be combined using OR semantics.
- New JUnit Platform support package `org.junit.platform.commons.support` that contains *maintained* utility methods for annotation, 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.
- Squashed logic behind `TestDescriptor.isTest()` and `TestDescriptor.isContainer()` from two independent `boolean` properties into a single enumeration, namely `TestDescriptor.Type`. See next bullet point for details.
- Introduced `TestDescriptor.Type` enumeration with `TestDescriptor.getType()` accessor defining

all possible descriptor types. `TestDescriptor.isTest()` and `TestDescriptor.isContainer()` now delegate to `TestDescriptor.Type` constants.

- Introduced `TestDescriptor.prune()` and `TestDescriptor.pruneTree()` which allow engine authors to customize what happens when pruning is triggered by the JUnit Platform.
- `TestIdentifier` now uses the new `TestDescriptor.Type` enumeration to store the underlying type. It can be retrieved via the new `TestIdentifier.getType()` method. Furthermore, `TestIdentifier.isTest()` and `TestIdentifier.isContainer()` now delegate to `TestDescriptor.Type` constants.

JUnit Jupiter

Bug Fixes

- Fixed bug that prevented discovery of two or more methods in the same class when selected via a method selector.
- `@Nested` non-static test classes are now detected when declared in super classes.
- The correct execution order of overridden `@BeforeEach` and `@AfterEach` methods is now enforced when declared at multiple levels within a class hierarchy. It's now always `super.before`, `this.before`, `this.test`, `this.after`, and `super.after`, even if the compiler adds synthetic methods.
- `TestExecutionExceptionHandler`s are now invoked in the opposite order in which they were registered, analogous to all other "*after*" extensions.

Deprecations and Breaking Changes

- Removed deprecated `Assertions.expectThrows()` method in favor of `Assertions.assertThrows()`.
- `ExtensionContext.Namespace`s composed of the same parts but in a different order are no longer considered equal to each other.

New Features and Improvements

- First-class support for *parameterized tests* via the new `@ParameterizedTest` annotation. See Parameterized Tests for details.
- First-class support for *repeated tests* via the new `@RepeatedTest` annotation and `RepetitionInfo` API. See Repeated Tests for details.
- Introduce `@TestTemplate` annotation and accompanying extension point `TestTemplateInvocationContextProvider`.
- `Assertions.assertThrows()` now uses canonical names for exception types when generating assertion failure messages.
- `TestInstancePostProcessor`s registered on test methods are now invoked.
- New variants of `Assertions.fail`: `Assertions.fail(Throwable cause)` and `Assertions.fail(String message, Throwable cause)`.
- New `Assertions.assertLinesMatch()` comparing lists of strings, featuring `Object::equals` and regular expression checks. `assertLinesMatch()` also provides a fast-forward mechanism to skip

lines that are expected to change in each invocation — for example, duration, timestamps, stack traces, etc. Consult the JavaDoc for `org.junit.jupiter.Assertions` for details.

* Extensions can now be registered automatically via Java's `ServiceLoader` mechanism. See Automatic Extension Registration.

JUnit Vintage

Bug Fixes

* Fixed bug that caused only the last failure of a test to be reported. For example, when using the `ErrorCollector` rule, only the last failed check was reported. Now, all failures are reported using an `org.opentest4j.MultipleFailuresError`.