JUnit 개발자 가이드

이 문서는 Junit을 통해 단위테스트를 수행하는 개발자 가이드를 공유하기 위해 작성되었다.

- 유닛 테스트의 이점
- JUnit 5 개요
  - JUnit Platform
  - JUnit Jupiter
  - JUnit Vintage
- 테스트 작성
  - 테스트 대상 코드 예
  - 테스트 케이스 예
  - 지원하는 주석 (Annotations)
  - 용어 정의
  - Maven 의존성
  - 테스트 클리서와 메소드
  - 표시 이름
  - Assertions
  - Assumption
  - 테스트 비활성화
  - 조건부 테스트 실행
  - 태그
  - 태그 정의
  - 필터링 방법
  - 테스트 실행 순서
  - 테스트 인스턴스 생명주기
  - 중첩 테스트
  - 의존성 주입
  - 테스트 인터페이스 지원
  - 반복 시험
  - 매개변수 테스트
    - 지원하는 인자 종류
      - @EnumSource
      - @MethodSource
      - @CsvSource
      - @CsvFileSource
      - @ArgumentsSource
  - Timeouts
  - 병렬 실행
- Spring Boot Example
  - pom.xml
  - 테스트 대상 코드
    - HelloController.java
    - MessageService.java
  - 테스트 코드
    - Spring_boot_junit5_api_test.java
    - Spring_boot_Junit5_service_test.java
  - 테스트 결과
- Mock Example
  - pom.xml
  - 테스트 대상 코드
    - StartApplication.java
    - service
  - 테스트 결과

유닛 테스트의 이점

개발 프로세스를 민첩하게 한다

- 새로운 기능을 추가할 때 기존 기능을 테스트 하여 보다 안정된 소프트웨어의 리팩터링이 가능하다

코드 품질 향상

- 유닛 테스트는 코드 품질을 개선한다.
- 코딩하기 전에 테스트를 작성하면 문제에 대해 더 고민하게 되어 예외 케이스를 발견하게되어 더 나은 코드를 작성하게 한다.

조기에 결함 발견

- 기능의 통합 전에 개발자가 개별 코드를 테스트 하므로 문제를 이른 시점에서 발견할 수 있으며, 다른 코드에 문제를 전파하기 전에 문제를 해결할 수 있다.

변경을 촉진하고 통합을 쉽게 한다

- 유닛 테스트를 통해 개발자는 나중에 코드를 리팩터링 할 때 코드가 올바르게 동작 하는지 확신을 가질 수 있으므로 코드를 변경하는데 도움을 제공한다.
- 유닛 테스트는 새로운 기능의 결함을 줄이고 기존 기능을 변경할 때 버그를 줄여준다.
- 유닛 테스트는 각 단위 개발 기능의 정확성을 검증하여 나중에 전체 프로그램의 테스트를 쉽게 한다.

문서 제공

- 다른 개발자에게 단위에서 제공하는 기능과 이를 사용하는 방법을 제공하고 구현 단위의 인터페이스 (API)에 대한 기본적인 이해를 제공한다.

디버깅 프로세스 개선

- 테스트가 실패하면 코드의 최신 변경 사항만 디버깅하면 되므로 디버깅 프로세스가 간단해진다.

설계 개선

- 테스트를 먼저 작성하면 코드를 작성하기 전에 설계와 수행해야 하는 작업을 충분히 고려하게 되어 더 나은 디자인을 만들 수 있다.
- 일부 코드를 테스트하려면 해당 코드가 담당하는 역할을 정의해야 하므로 코드의 역할 정의가 명확져 코드의 응집력이 높아진다.

개발 비용 절감

버그가 조기에 발견되기 때문에 유닛 테스트는 버그 수정 비용을 줄이는 데 도움이 된다.

JUnit 5 개요

JUnit 5는 이전 버전과 달리 JUnit 5는 세 가지 하위 프로젝트의 여러 모듈로 구성된다.

JUnit 5 = JUnit 플랫폼 + JUnit Jupiter + JUnit Vintage

JUnit 5는 Java 8 또는 이상의 런타임을 요구한다. 그러나 이전 버전의 JDK로 컴파된 코드도 테스트 가능하다.

JUnit Platform

JUnit 플랫폼은 JVM에서 테스트 프레임워크를 시작하기 위한 기반 역할을 한다. 또한 플랫폼에서 실행되는 테스트 프레임워크를 개발하기 위한 TestEngine API를 정의한다. 또한 플랫폼은 플랫폼에서 하나 이상의 테스트 엔진을 사용하여 사용자 지정 테스트 스위트를 실행하기 위한 명령줄 및 JUnit 플랫폼 스위트 엔진에서 플랫폼을 시작하기 위한 Console Launcher를 제공한다.

JUnit Jupiter

JUnit Jupiter는 JUnit 5에서 테스트 및 확장을 작성하기 위한 프로그래밍 모델과 확장 모델의 조합입니다. Jupiter 하위 프로젝트는 플랫폼에서 Jupiter 기반 테스트를 실행하기 위한 TestEngine을 제공한다.

JUnit Vintage

JUnit Vintage는 플랫폼에서 JUnit 3 및 JUnit 4 기반 테스트를 실행하기 위한 TestEngine을 제공합니다. 클래스 경로 또는 모듈 경로에 JUnit 4.12 이상이 있어야 한다.

테스트 작성

Jupiter를 이용해 테스트 케이스를 작성한다.

테스트 대상 코드 예

```
package com.example.project;

public class Calculator {

        public int add(int a, int b) {
                return a + b;
        }

}
```

테스트 케이스 예

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

import example.util.Calculator;

import org.junit.jupiter.api.Test;

class MyFirstJUnitJupiterTests {

    private final Calculator calculator = new Calculator();

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

}
```

지원하는 주석 (Annotations)

JUnit 5가 지원하는 주석에 추가로 사용자 정의 주석 (meta-annotation) 추가를 지원한다.

Annotation	Description
`@Test`	Denotes that a method is a test method. Unlike JUnit 4's `@Test` annotation, this annotation does not declare any attributes, since test extensions in JUnit Jupiter operate based on their own dedicated annotations. Such methods are *inherited* unless they are *overridden*.
`@ParameterizedTest`	Denotes that a method is a parameterized test. Such methods are *inherited* unless they are *overridden*.
`@RepeatedTest`	Denotes that a method is a test template for a repeated test. Such methods are *inherited* unless they are *overridden*.
`@TestFactory`	Denotes that a method is a test factory for dynamic tests. Such methods are *inherited* unless they are *overridden*.
`@TestTemplate`	Denotes that a method is a template for test cases designed to be invoked multiple times depending on the number of invocation contexts returned by the registered providers. Such methods are *inherited* unless they are *overridden*.
`@TestClassOrder`	Used to configure the test class execution order for `@Nested` test classes in the annotated test class. Such annotations are *inherited*.
`@TestMethodOrder`	Used to configure the test method execution order for the annotated test class; similar to JUnit 4's `@FixMethodOrder`. Such annotations are *inherited*.
`@TestInstance`	Used to configure the test instance lifecycle for the annotated test class. Such annotations are *inherited*.
`@DisplayName`	Declares a custom display name for the test class or test method. Such annotations are not *inherited*.
`@DisplayNameGeneration`	Declares a custom display name generator for the test class. Such annotations are *inherited*.
`@BeforeEach`	Denotes that the annotated method should be executed *before* **each** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, or `@TestFactory` method in the current class; analogous to JUnit 4's `@Before`. Such methods are *inherited* - unless they are *overridden* or *superseded* (i.e., replaced based on signature only, irrespective of Java's visibility rules).
`@AfterEach`	Denotes that the annotated method should be executed *after* **each** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, or `@TestFactory` method in the current class; analogous to JUnit 4's `@After`. Such methods are *inherited* - unless they are *overridden* or *superseded* (i.e., replaced based on signature only, irrespective of Java's visibility rules).
`@BeforeAll`	Denotes that the annotated method should be executed *before* **all** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, and `@TestFactory` methods in the current class; analogous to JUnit 4's `@BeforeClass`. Such methods are *inherited* - unless they are *hidden*, *overridden*, or *superseded*, (i.e., replaced based on signature only, irrespective of Java's visibility rules) - and must be `static` unless the "per-class" test instance lifecycle is used.
`@AfterAll`	Denotes that the annotated method should be executed *after* **all** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, and `@TestFactory` methods in the current class; analogous to JUnit 4's `@AfterClass`. Such methods are *inherited* - unless they are *hidden*, *overridden*, or *superseded*, (i.e., replaced based on signature only, irrespective of Java's visibility rules) - and must be `static` unless the "per-class" test instance lifecycle is used.

`@Nested`	Denotes that the annotated class is a non-static nested test class. On Java 8 through Java 15, `@BeforeAll` and `@AfterAll` methods cannot be used directly in a `@Nested` test class unless the "per-class" test instance lifecycle is used. Beginning with Java 16, `@BeforeAll` and `@AfterAll`methods can be declared as `static` in a `@Nested` test class with either test instance lifecycle mode. Such annotations are not *inherited*.
`@Tag`	Used to declare tags for filtering tests, either at the class or method level; analogous to test groups in TestNG or Categories in JUnit 4. Such annotations are *inherited* at the class level but not at the method level.
`@Disabled`	Used to disable a test class or test method; analogous to JUnit 4's `@Ignore`. Such annotations are not *inherited*.
`@Timeout`	Used to fail a test, test factory, test template, or lifecycle method if its execution exceeds a given duration. Such annotations are *inherited*.
`@ExtendWith`	Used to register extensions declaratively. Such annotations are *inherited*.
`@RegisterExtension`	Used to register extensions programmatically via fields. Such fields are *inherited* unless they are *shadowed*.
`@TempDir`	Used to supply a temporary directory via field injection or parameter injection in a lifecycle method or test method; located in the `org.junit.jupiter.api.io` package.

용어 정의

플랫폼

- Container: 다른 컨테이너나 테스트를 자식으로 포함하는 테스트 트리의 노드(예: 테스트 class)
- Test: 실행될 때 예상되는 동작을 확인하는 테스트 트리의 노드(예: @Test 메소드)

Jupiter

- Lifecyle Method: @BeforeAll, @AfterAll, @BeforeEach 또는 @AfterEach로 직접 주석을 달거나 메타 주석을 단 모든 메소드
- Test Class: 최상위 클래스, 정적 멤버 클래스 또는 하나 이상의 테스트 메소드(예: 컨테이너)를 포함하는 @Nested 클래스. 테스트 클래스는 추상 클래스가 아니어야 하며 단일 생성자가 있어야 한다.
- Test Method: @Test, @RepeatedTest, @ParameterizedTest, @TestFactory 또는 @TestTemplate으로 직접 주석을 달거나 메타 주석을 단 모든 인스턴스 메소드
  @Test를 제외하고 이들은 테스트 또는 잠재적으로(@TestFactory의 경우) 다른 컨테이너를 그룹화하는 테스트 트리에 컨테이너를 생성한다.

Maven 의존성

Maven 프로젝트의 경우 pom.xml에 다음 종속성을 추가한다.

JUnit 5 의존성 예

```
...
<properties>
        <junit.jupiter.version>5.9.1</junit.jupiter.version>
        <junit.platform.version>1.9.1</junit.platform.version>
</properties>
...
<dependencies>
    <!-- Only needed to run tests in a version of IntelliJ IDEA that bundles older versions -->
    <dependency>
        <groupId>org.junit.platform</groupId>
        <artifactId>junit-platform-launcher</artifactId>
        <scope>test</scope>
    </dependency>
         <dependency>
                <groupId>org.junit.platform</groupId>
                <artifactId>junit-platform-suite</artifactId>
                <version>${junit.platform.version}</version>
                <scope>test</scope>
        </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
                <version>${junit.jupiter.version}</version>
        <scope>test</scope>
    </dependency>
        <dependency>
                <groupId>org.junit.jupiter</groupId>
                <artifactId>junit-jupiter-api</artifactId>
                <version>${junit.jupiter.version}</version>
                <scope>test</scope>
        </dependency>
    <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-params</artifactId>
             <version>${junit.jupiter.version}</version>
             <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.junit</groupId>
            <artifactId>junit-bom</artifactId>
            <version>5.9.1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

테스트 클리서와 메소드

테스트 메소드 및 라이프사이클 메소드는 현재 테스트 클래스 내에서 로컬로 선언될 수 있습니다.

- 테스트 클래스, 테스트 메소드 및 라이프사이클 메소드는 public 일 필요는 없지만 private 으로 정의하면 안된다.

테스트 클래스 예

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

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

class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

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

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

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

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }

}
```

표시 이름

테스트 클래스 및 테스트 메소드는 @DisplayName을 통해(공백, 특수 문자, 이모티콘까지 포함) 테스트 보고서와 테스트 runner 및 IDE에 표시될 맞춤 표시 이름을 선언할 수 있다.

- Jupiter는 커스텀 표시 이름 생성자를 추가할 수 있다
- Jupiter는 몇 가지 생성자를 제공한다 (Standard, Simple, ReplaceUnderscores, IndicativeSentences)

```
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() {
    }

}
```

Assertions

JUnit Jupiter는 JUnit 4의 여러 어설션 메소드와 함께 제공되며 Java 8 lambdas와 함께 사용하기에 적합한 몇 가지를 추가한다. 모든 JUnit Jupiter assertion은 org.junit.jupiter.api.Assertions 클래스의 정적 메소드이다.

- JUnit Jupiter에는 Kotlin에서 사용하기에 적합한 몇 가지 어설션 메소드도 함께 제공한다

Assertion 사용 예

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

import java.util.concurrent.CountDownLatch;

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

import org.junit.jupiter.api.Test;

class AssertionsDemo {

    private final Calculator calculator = new Calculator();

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

    @Test
    void standardAssertions() {
        assertEquals(2, calculator.add(1, 1));
        assertEquals(4, calculator.multiply(2, 2),
                "The optional failure message is now the last parameter");
        assertTrue('a' < 'b', () -> "Assertion messages can be lazily evaluated -- "
                + "to avoid constructing complex messages unnecessarily.");
    }

    @Test
    void groupedAssertions() {
        // In a grouped assertion all assertions are executed, and all
        // failures will be reported together.
```

```
    assertAll("person",
        () -> assertEquals("Jane", person.getFirstName()),
        () -> assertEquals("Doe", person.getLastName())
    );
}

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

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

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

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

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

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

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

@Test
void timeoutExceeded() {
    // The following assertion fails with an error message similar to:
    // execution exceeded timeout of 10 ms by 91 ms
```

```
        assertTimeout(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms.
            Thread.sleep(100);
        });
    }

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

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

}
```

Assumption

특정 조건이 만족할 때 테스트를 수행하는 기능을 제공한다. 만족하지 않는 경우 테스트 실패가 아닌 abort 상태로 테스트가 종료된다. JUnit Jupiter는 JUnit 4가 제공하는 assumption 메소드의 하위 집합과 함께 제공되며 Java 8 람다 식 및 메소드 참조와 함께 사용하기에 적합한 몇 가지를 추가한다. 모든 JUnit Jupiter assumption은 org.junit.jupiter.api.Assumptions 클래스의 정적 메소드이다.

Assumption 사용 예

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

import example.util.Calculator;

import org.junit.jupiter.api.Test;

class AssumptionsDemo {

    private final Calculator calculator = new Calculator();

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

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

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

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

}
```

테스트 비활성화

전체 테스트 클래스 또는 개별 테스트 메소드는 @Disabled 주석, 조건부 테스트 실행에 설명된 주석 중 하나 또는 사용자 지정 ExecutionCondition을 통해 비활성화할 수 있다.

테스트 클래스 비활성화

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

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

    @Test
    void testWillBeSkipped() {
    }

}
```

특정 테스트 메서드 비활성화

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

class DisabledTestsDemo {

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

    @Test
    void testWillBeExecuted() {
    }

}
```

조건부 테스트 실행

조건에 따라 container (테스트 클래스) 또는 test를 활성화 또는 비활성화 하는 기능을 제공한다.

운영체제 조건

- @EnabledOnOs(MAC)
- @EnabledOnOs({ LINUX, MAC })
- @DisabledOnOs(WINDOWS)

예)

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

아키텍처 조건

- @EnabledOnOs(architectures = "aarch64")
- @DisabledOnOs(architectures = "x86_64")
- @EnabledOnOs(value = MAC, architectures = "aarch64")
- @DisabledOnOs(value = MAC, architectures = "aarch64")

예)

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

Java 런타임 조건

- @EnabledOnJre(JAVA_8)
- @EnabledOnJre({ JAVA_9, JAVA_10 })
- @EnabledForJreRange(min = JAVA_9, max = JAVA_11)
- @EnabledForJreRange(min = JAVA_9)
- @EnabledForJreRange(max = JAVA_11)
- @DisabledOnJre(JAVA_9)
- @DisabledForJreRange(min = JAVA_9, max = JAVA_11)
- @DisabledForJreRange(max = JAVA_11)

예)

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

네이티브 이미지 조건

GraalVM native image 환경에 따라 container 와 test 실행을 제어할 수 있다. 일반적으로 GraalVM 네이티브 빌드 도구에서 제공하는 Gradle과 Maven 플러그인으로 테스트할 때 사용한다.

- @EnabledInNativeImage
- @DisabledInNativeImage

예)

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

시스템 속성

JVM 시스템 속성 (properties)에 따라 container 와 test 실행을 제어할 수 있다.

- @EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
- @DisabledIfSystemProperty(named = "ci-server", matches = "true")

예)

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

환경 변수 조건

시스템 변수에 따라 container 와 test 실행을 제어할 수 있다.

- @EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server")
- @DisabledIfEnvironmentVariable(named = "ENV", matches = ".*development.*")

예)

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

사용자 정의 조건

사용자가 정의한 함수의 결과에 따라 container 와 test 실행을 제어할 수 있다.

- @EnabledIf("customCondition")
- @DisabledIf("customCondition")

예)

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

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

태그

태그는 테스트 표시 및 필터링을 위한 JUnit 플랫폼 개념이다.

태그 정의

- and (&), or (|), not (!) 오퍼레이션 지원
  예) (micro | integration) & (product | shipping)

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

@Tag("TagName")
@Tag("TagName2")
class TaggingDemo {

    @Test
    @Tag("TagName1")
    void testingTaxCalculation() {
    }

}
```

필터링 방법

- Test Suite: @IncludeTags("TagName")

예)

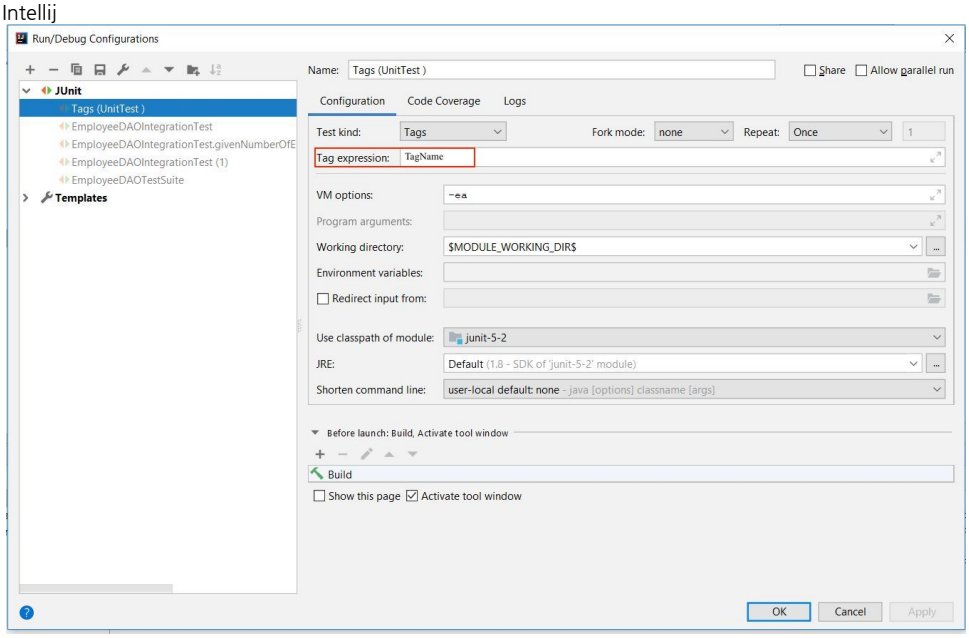
```
@SelectPackages("com.curvc.tags")
@IncludeTags("TagName")
public class EmployeeDAOUnitTestSuite {
}
```

- Maven Surefire Plugin

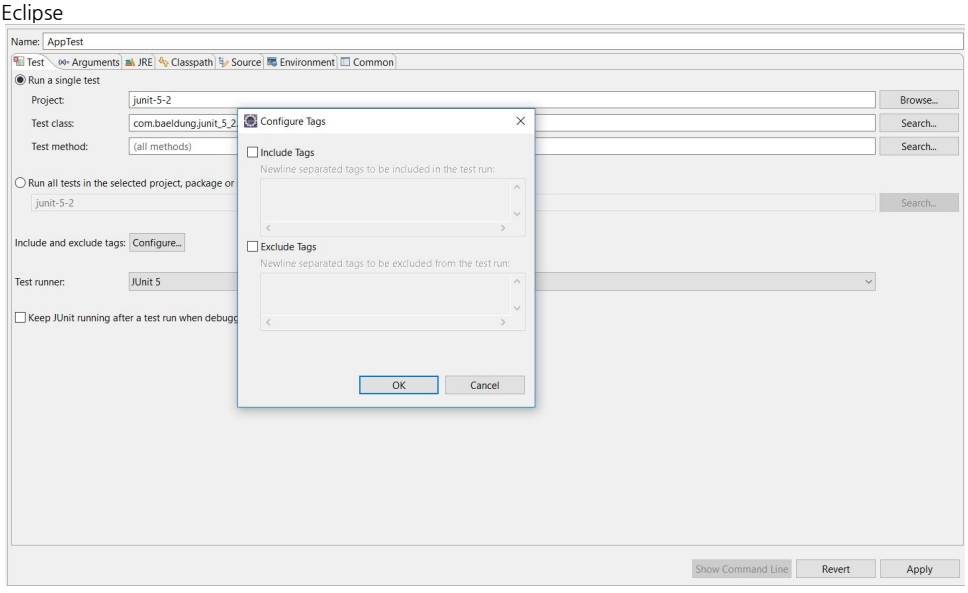
pom.xml

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
    <configuration>
        <groups>TagName</groups>
    </configuration>
</plugin>
```

- Intellij



- Eclipse



테스트 실행 순서

기본적으로 테스트 클래스와 메소드는 결정적이지만 의도적으로 명백하지 않은 알고리즘을 사용하여 정렬된다. 테스트 suite에 포함된 테스트 클래스와 테스트 메소드는 반복 수행해도 동일한 순서로 실행된다.

메소드 순서

유닛 테스트는 일반적으로 실행되는 순서에 의존해서는 안 되지만, 특정 테스트 방법 실행 순서를 실행해야 할 때가 있다. @TestMethodOrder (MethodOrderer) 주석을 이용해 메소드 실행 순서를 지정할 수 있다.

또한 사용자 정의 MethodOrderer를 구현할 수 있다.

- `MethodOrderer.DisplayName`: 메소드의 DisplayName을 기준으로 알파벳 순서로 실행
- `MethodOrderer.MethodName`: 메소드 이름을 기준으로 알파벳 순서로 실행
- `MethodOrderer.OrderAnnotation`: `@Order`
- `MethodOrderer.Random`: 테스트 메소드를 pseudo-random 방식으로 정렬하고 사용자 정의 시드의 구성 지원
- `MethodOrderer.Alphanumeric`: 메소드 이름과 파라메터 목록을 기준으로 정렬

예)

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

@TestMethodOrder(OrderAnnotation.class)
class OrderedTestsDemo {

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

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

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

}
```

디폴트 메소드 실행 순서 지정:

Jupiter 프라퍼티에 기본 메소드 정렬 방식을 지정할 수 있다.

- src/test/resources/junit-platform.properties

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

클래스 실행 순서

유닛 테스트는 일반적으로 실행되는 순서에 의존해서는 안 되지만, 특정 테스트 방법 실행 순서를 실행해야 할 때가 있다. @TestClassOrder(ClassOrderer) 주석을 이용해 클래스 실행 순서를 지정할 수 있다.

- `ClassOrderer.ClassName`: 클래스 이름을 기준으로 알파벳 순서로 실행
- `ClassOrderer.DisplayName`: 클래서의 display name을 기준으로 알파벳 순서로 실행
- `ClassOrderer.OrderAnnotation`: `@Order`
- `ClassOrderer.Random`: 테스트 클래스를 pseudo-random 방식으로 정렬하고 사용자 정의 시드의 구성 지원

예)

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

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

    @Nested
    @Order(1)
    class PrimaryTests {

        @Test
        void test1() {
        }
    }

    @Nested
    @Order(2)
    class SecondaryTests {

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

테스트 인스턴스 생명주기

개별 테스트 메소드를 격리하여 실행할 수 있도록 하고 변경 가능한 테스트 인스턴스 상태로 인한 예상치 못한 부작용을 피하기 위해 JUnit은 각 테스트 메소드를 실행하기 전에 각 테스트 클래스의 새 인스턴스를 생성한다.

그리고 @TestInstance(Lifecycle.PER_CLASS) 주석을 이용해 테스트 클래스 단위로 테스트 인스턴스를 생성하도록 설정 할 수 있다.

Annotation 활용

- @TestInstance(TestInstance.Lifecycle.PER_CLASS)

예)

```
package org.wesome.junit5;

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

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

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class AppleCalculatorTest {
    public AppleCalculatorTest() {
        System.out.println("AppleCalculatorTest instantiated");
    }

    @BeforeAll
    static void beforeAll() {
        System.out.println("AppleCalculatorTest BeforeAll");
    }

    @AfterAll
    static void afterAll() {
        System.out.println("AppleCalculatorTest AfterAll");
    }

    @BeforeEach
    void beforeEach() {
        System.out.println("AppleCalculatorTest BeforeEach");
    }

    @AfterEach
    void afterEach() {
        System.out.println("AppleCalculatorTest AfterEach");
    }

    private int instanceVar = 0;

    @Test
    void addAppleTest1() {
        instanceVar += 1;
        System.out.println("AppleCalculatorTest.addAppleTest1#instanceVar = " + instanceVar);
        AppleCalculator appleCalculator = new AppleCalculator();
        assertEquals(2, appleCalculator.addApple(1, 1), "1 apple + 1 apple is 2 apple");
    }

    @Test
    void addAppleTest2() {
        instanceVar += 1;
        System.out.println("AppleCalculatorTest.addAppleTest2#instanceVar = " + instanceVar);
        AppleCalculator appleCalculator = new AppleCalculator();
        assertEquals(2, appleCalculator.addApple(1, 1), "1 apple + 1 apple is 2 apple");
    }
}
```

테스트 결과

```
AppleCalculatorTest instantiated
AppleCalculatorTest BeforeAll
AppleCalculatorTest BeforeEach
AppleCalculatorTest.addAppleTest1#instanceVar = 1
AppleCalculatorTest AfterEach
AppleCalculatorTest BeforeEach
AppleCalculatorTest.addAppleTest2#instanceVar = 2
AppleCalculatorTest AfterEach
AppleCalculatorTest AfterAll
```

프라퍼티 활용

- src/test/resources/junit-platform.properties

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

- JVM 옵션

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

중첩 테스트

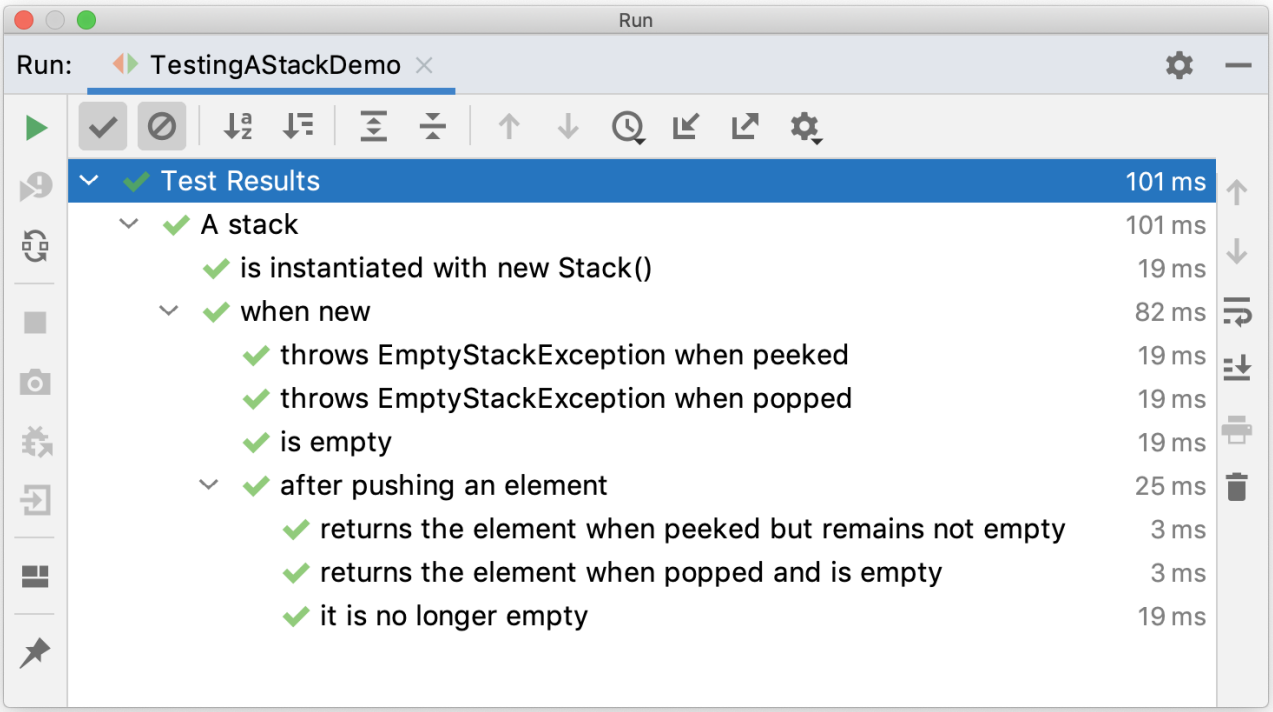
@Nested 테스트는 테스트 작성자에게 여러 테스트 그룹 간의 관계를 표현할 수 있는 기능을 제공한다. 이러한 중첩된 테스트는 자바의 중첩된 클래스를 사용하여 계층적인 테스트를 구성할 수 있다.

예)

```
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());
            }
        }
    }
}
```

실행 결과



의존성 주입

이전의 JUnit 버전에서 테스트 생성자나 메소드는 매개 변수를 가질 수 없었다(표준 Runner를 구현하는 경우 제외). JUnit Jupiter의 주요 변화 중 하나로서, 테스트 생성자와 메소드 모두 이제 매개 변수를 가질 수 있게되어 유연성이 높아졌고 생성자와 메소드에 대한 종속성 주입이 가능하다.

의존성 주입 예

- TestInfoDemo 클래스에 TestInfo 주입
- test1() 메소드에 TestInfo 주입
- 테스트 메소드 test1(), test2() 실행 전 호출되는 init() 함수에 TestInfo 주입

```
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() {
    }

}
```

테스트 인터페이스 지원

JUnit Jupiter는 @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, @TestTemplate, @BeforeEach 및 @AfterEach를 인터페이스 기본 메소드에 선언할 수 있다. 테스트 인터페이스의 정적 메소드 또는 테스트 클래스에 @TestInstance(Lifecycle.PER_CLASS)로 주석이 달린 경우 @BeforeAll 및 @AfterAll을 테스트 인터페이스의 정적 메소드 또는 인터페이스 기본 메소드에 선언할 수 있다.

테스트 코드 예

- TestLifecycleLogger, TestInterfaceDynamicTestsDemo, TimeExecutionLogger 인터페이스를 TestInterfaceDemo 테스트에 구현하는 예

TestLifecycleLogger

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

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

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

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

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

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

}
```

TestInterfaceDynamicTestsDemo

```
interface TestInterfaceDynamicTestsDemo {

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

}
```

TimeExecutionLogger

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

TestInterfaceDemo

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

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

}
```

실행 결과

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

반복 시험

JUnit Jupiter는 @RepeatedTest로 방법에 주석을 달고 원하는 총 반복 횟수를 지정하여 지정된 횟수를 반복할 수 있는 기능을 제공한다.

테스트 코드 예시

예)

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

import java.util.logging.Logger;

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

class RepeatedTestsDemo {

    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("Repeat! 1/1", testInfo.getDisplayName());
    }

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

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

}
```

테스트 결과

```
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
```

매개변수 테스트

매개 변수화된 테스트를 사용하면 다른 인수로 테스트를 여러 번 실행할 수 있습니다. 일반 @Test 방법과 마찬가지로 선언되지만 대신 @ParameterizedTest 주석을 사용한다. 또한, 각 호출에 대한 인수를 제공하는 적어도 하나의 소스를 선언한 다음 테스트 메소드의 인수로 사용한다.

- junit-jupiter-params dependency 추가 필요

테스트 코드 예

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

실행 결과 (ConsoleLauncher)

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

지원하는 인자 종류

JUnit Jupiter는 매개변수 소스를 지정하는 몇 가지 주석을 제공한다.

리터럴 값으로 구성된 단일 배열을 지정할 수 있으며 매개 변수가 있는 테스트 호출당 단일 인수를 제공하는 데만 사용할 수 있다.

지원하는 리터럴 유형:

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

예)

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

@EnumSource

Enum 상수를 제공한다.

속성 미지정

단순 예)

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

names 속성 지정

- names 에서 지정한 값 포함

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

mode 속성 지정

모드에 따라 테스트 메소드에 전달되는 인수를 제어한다.

mode:

- INCLUDE: names 항목 포함
- EXCLUDE: names 항목 제외
- MATCH_ALL: names에 지정한 정규식과 일치하는 항목 포함

@MethodSource

테스트 클래스 또는 외부 클래스의 하나 이상의 factory method를 참조할 수 있다.

테스트 클래스 내의 factory method는 테스트 클래스에 @TestInstance(Lifecycle.PER_CLASS)로 주석이 달지 않는 한 정적이어야 하는 반면, 외부 클래스의 factory method는 항상 정적이어야 한다.

Factory 이름 지정

- MethodSource 인자로 factory 이름을 지정한다

예)

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

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

Factory 이름 미정의

- Method 이름을 지정하지 않으면 @ParameterizedTest 주석이 지정된 메소드 이름과 일치하는 factory method를 찾는다.

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

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

다중 매개변수 할당

매개 변수화된 테스트 메서드가 여러 매개 변수를 선언하는 경우, 인수 인스턴스 또는 개체 배열의 컬렉션, 스트림 또는 배열을 반환해야 한다.

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

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

외부 정적 factory 메소드 참조

정규화된 메소드 이름을 지정하여 외부의 정적 factory 메소드를 참조 할 수 있다.

```
package example;

import java.util.stream.Stream;

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

class ExternalMethodSourceDemo {

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

class StringsProviders {

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

@CsvSource

인수 목록을 쉼표로 구분된 값으로 표현할 수 있다.

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

표현 예시

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

텍스트 블럭 표현 방법

텍스트 블럭을 이용해 표현 가능하다.

- #로 시작하는 라인은 주석으로 간주한다

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

실행 결과

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

@CsvFileSource

클래스 경로 또는 로컬 파일 시스템에서 쉼표로 구분된 값(CSV) 파일을 지원한다.

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

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

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

데이터 파일 내용

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

@ArgumentsSource

사용자 지정, 재사용 가능한 ArgumentsProvider를 지정하는 데 사용한다.

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

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

Timeouts

실행 시간이 주어진 기간을 초과하면 테스트, 테스트 팩토리, 테스트 템플릿 또는 수명 주기 방법이 실패해야 한다고 선언할 수 있다.

```
class TimeoutDemo {

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

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

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

}
```

병렬 실행

⚠️ 병렬 테스트 실행은 실험적인 기능이다.

기본적으로 JUnit Jupiter는 단일 쓰래드에 의해 순차적으로 수행된다. 버전 5.3 부터 빠른 테스트 수행을 위해 병렬 테스트 수행 옵션이 추가되었다. junit.jupiter.execution.parallel.enabled 옵션을 true로 설정하여 기능을 활성화 한다.

기능 활성화 후 junit.jupiter.execution.parallel.mode.default 옵션 설정에 따라 두 가지 실행 모드 중 하나를 선택할 수 있다.

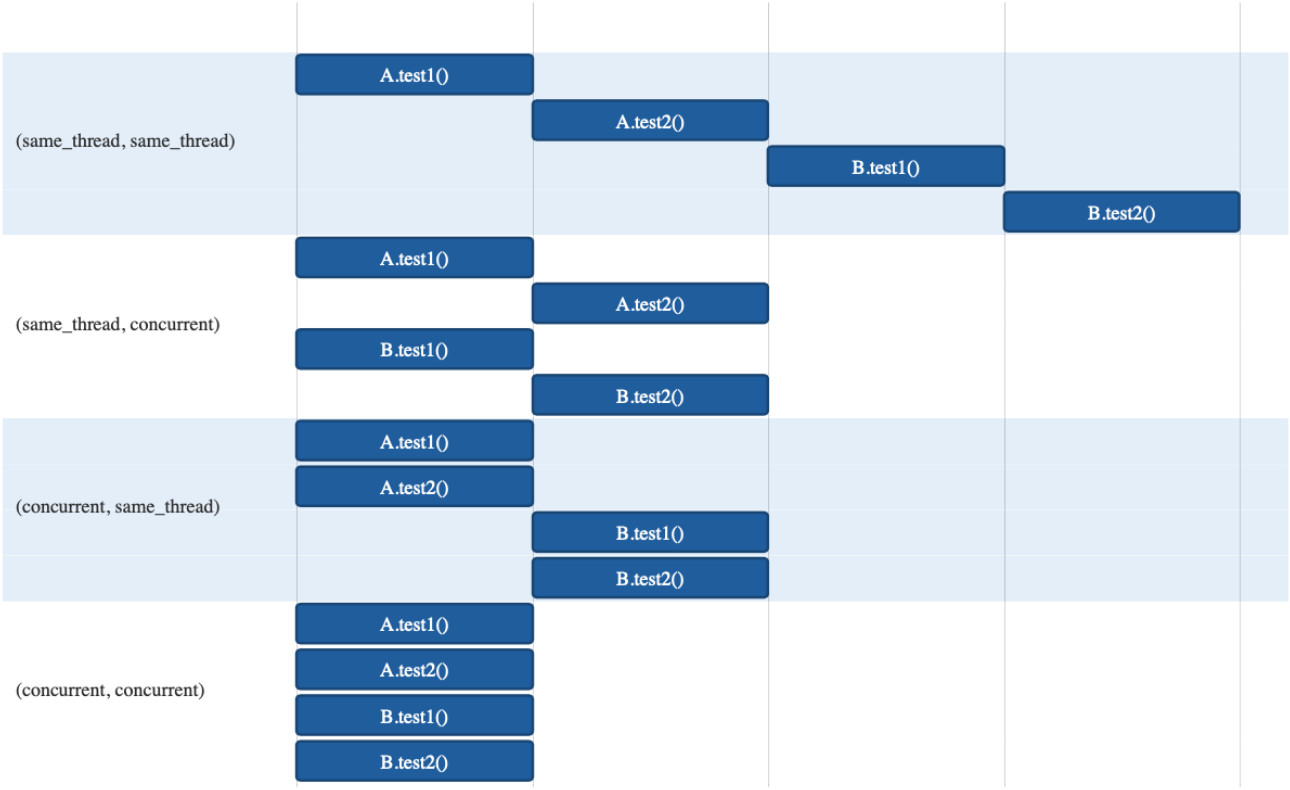
- same_thread: 단일 쓰래드로 실행
- concurrent: 동일한 스레드에서 lock이 걸리지 않는 한 동시에 실행

병렬 실행 설정 예

junit-platform.properties

```
junit.jupiter.execution.parallel.enabled = true
junit.jupiter.execution.parallel.mode.default = concurrent
```

`junit.jupiter.execution.parallel.mode.default` and`junit.jupiter.execution.parallel.mode.classes.default`



설정 가능한 속성

Property	Description	Supported Values	Default Value
`junit.jupiter.execution.parallel.enabled`	병렬 실행 활성	• `true` • `false`	`false`
`junit.jupiter.execution.parallel.mode.default`	테스트 트리의 기본 실행 모드	• `concurrent` • `same_thread`	`same_thread`
`junit.jupiter.execution.parallel.mode.classes.default`	최상위 테스트 클래스의 기본 실행 모드	• `concurrent` • `same_thread`	`same_thread`
`junit.jupiter.execution.parallel.config.strategy`	원하는 병렬 처리와 최대 풀 크기를 위한 실행 전략	• `dynamic` • `fixed` • `custom`	`dynamic`
`junit.jupiter.execution.parallel.config.dynamic.factor`	동적 구성 전략에 원하는 병렬 처리를 결정하기 위해 사용 가능한 프로세서/코어 수를 곱한 요소	a positive decimal number	`1.0`
`junit.jupiter.execution.parallel.config.fixed.parallelism`	고정 구성 전략에 대한 원하는 병렬 처리	a positive integer	no default value
`junit.jupiter.execution.parallel.config.custom.class`	사용자 지정 구성 전략에 사용할 ParallelExecutionConfigurationStrategy의 정규화된 클래스 이름	for example, *org.example.CustomStrategy*	no default value

동기화

실행 모드를 컨트롤 하기 위해서 @Execution 어노테이션을 이용한다. Junit은 또 다른 어노테이션 기반 선언적 동기화 메카니즘을 제공한다. @ResourceLock 어노테이션은 테스트 클래스나 메서드에 선언할 수 있으며, 안정적인 테스트 실행 보장하기 위해 동기화된 접근이 필요한 특정 공유 자원에 사용한다.

공유 자원은 String 타입으로 유일한 이름을 갖도록하여 식별한다. 이름은 사용자가 정의하거나, Resources 상수 안에 미리 선언된 SYSTEM_PROPERTIES, SYSTEM_OUT, SYSTEMERR, LOCALE, TIME_ZONE을 사용할 수 있다.

`@ResourceLock` 어노테이션이 붙은 공유 자원에 접근하려고 할 때 Junit은 병렬적으로 실행되는 테스트에 충돌이 없게 보장한다.

격리된 테스트 실행

대부분의 테스트가 병렬적으로 실행되는데, 어떠한 동기화도 없이 실행되는 클래스라면,@Isolated 어노테이션을 이용하여 격리된 상태로 테스트를 실행할 수 있다. 이런 테스트 클래스는 다른 테스트와 동시에 실행되지 않고, 순차적으로 실행 된다.

공유 자원을 고유하게 식별하는 String 타입외에도 접근 모드를 지정해줄 수 있다. 공유 자원에 대한 READ 접근이 필요한 두 테스트는 서로 병렬로 실행 될 수 있지만, 공유 자원에 대한 READ_WRITE 접근이 필요한 다른 테스트가 실행되는 동안에는 실행되지 않는다. 즉 READ_WRTIE의 테스트가 전부 끝날 때 까지 대기한다.

```
@Execution(CONCURRENT)
class SharedResourcesDemo {

    private Properties backup;

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

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

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

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

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

}
```

Spring Boot Example

Spring-Boot-Junit5-example.zip

```
pom.xml
src
 main
    java
       com
           junit5example
               controller
                  HelloController.java
               demo
                  MySpringBootApplication.java
               service
                   MessageService.java
    resources
        application.properties
 test
     java
         com
             junit5example
                 demo
                     Spring_boot_Junit5_service_test.java
                     Spring_boot_junit5_api_test.java
```

pom.xml

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.2.2.RELEASE</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.junt5example</groupId>
	<artifactId>Spring-Boot-Junit5-example</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>Spring-Boot-Junit5-example</name>
	<description>Spring boot2 Junit 5 example</description>

	<properties>
		<!-- Dependency versions -->
		<junit.jupiter.version>5.5.2</junit.jupiter.version>
		<maven-surefire-plugin.version>2.22.2</maven-surefire-plugin.version>

		<maven-jar-plugin.version>3.1.1</maven-jar-plugin.version>
	</properties>

	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>

		<!-- Jupiter API for writing tests -->
		<dependency>
			<groupId>org.junit.jupiter</groupId>
			<artifactId>junit-jupiter-engine</artifactId>
			<version>${junit.jupiter.version}</version>
			<scope>test</scope>
		</dependency>
	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
			<!-- Maven plugin to use perticular java version to compile code -->
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-compiler-plugin</artifactId>
				<version>3.8.1</version>
				<configuration>
					<source>1.8</source>
					<target>1.8</target>
				</configuration>
			</plugin>
		</plugins>
	</build>
</project>
```

테스트 대상 코드

MySpringBootApplication.java

```
package com.junit5example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication(scanBasePackages =
{"com.javabydeveloper.controller", "com.javabydeveloper.service"})
public class MySpringBootApplication {

	public static void main(String[] args) {
		SpringApplication.run(MySpringBootApplication.class, args);
	}

}
```

HelloController.java

```
package com.junit5example.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import com.junit5example.service.MessageService;

@RestController
public class HelloController {

	@Autowired
	private MessageService messageService;

	@GetMapping("/hello")
	public String sayHello(@RequestParam String user) {
		return messageService.getSubscriptionMessage(user);
	}

}
```

MessageService.java

```
package com.junit5example.service;

import org.springframework.stereotype.Component;

@Component
public class MessageService {

	public String getSubscriptionMessage(String user) {

		return "Hello "+user+", Thanks for the subscription!";
	}
}
```

테스트 코드

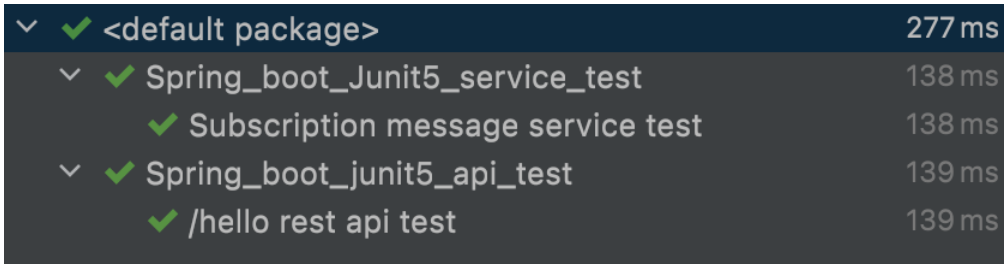
Spring_boot_junit5_api_test.java

```
package com.junit5example.demo;

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

import java.net.URI;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.web.util.UriComponentsBuilder;

/*
 * JUnit 5 test for Spring Boot Rest API Conroller
 */

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class Spring_boot_junit5_api_test {

        @Autowired
        private TestRestTemplate restTemplate;

         @Test
         @DisplayName("/hello rest api test ")
         void testMessage() {

                 String user = "Peter";
                 URI targetUrl= UriComponentsBuilder.fromUriString("/hello")
                                     .queryParam("user", user)
                                     .build()
                                     .encode()
                                     .toUri();

          String message = this.restTemplate.getForObject(targetUrl, String.class);
          assertEquals("Hello "+user+", Thanks for the subscription!", message);
         }

}
```

Spring_boot_Junit5_service_test.java

```
package com.junit5example.demo;

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

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import com.junit5example.service.MessageService;

/*
 * JUnit 5 test for Spring Boot Component
 */

@SpringBootTest
public class Spring_boot_Junit5_service_test {

        @Autowired
        private MessageService messageService;

         @Test
         @DisplayName("Subscription message service test ")
         void testSubscriptionMessage() {

                String user = "Peter";

            String message = messageService.getSubscriptionMessage(user);
            assertEquals("Hello "+user+", Thanks for the subscription!", message);
         }
}
```

테스트 결과



Mock Example

Mock 객체란 개발한 프로그램을 테스트할 때 테스트를 수행할 모듈과 연결되는 외부의 다른 모듈을 흉내 내는 가짜 모듈을 생성하여 테스트의 효율성을 높이는 데 사용하는 객체이다.

Mockito는 단위 테스트를 위한 Java Mocking Framework이다. JUnit에서 가짜 객체인 Mock을 생성해주고 관리하고 검증할 수 있도록 지원해주는 Framework이다. 구현체가 아직 없는 경우나 구현체가 있더라도 특정 단위만 테스트하고 싶을 경우 사용할 수 있도록 적절한 환경을 제공한다.

다음은 HelloServiceImpl를 Mock 개체로 대체하는 예이다.

```
pom.xml
src
    main
       java
           com
               junit5mockito
                   core
                       StartApplication.java
                       services
                           HelloService.java
                           HelloServiceImpl.java
    test
        java
            com
                junit5mockito
                    core
                        services
                            HelloServiceMockTest.java
```

testing-junit5-mockito.zip

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.junit5mockito.spring</groupId>
    <artifactId>testing-junit5-mockito</artifactId>
    <version>1.0</version>

    <properties>
        <java.version>1.8</java.version>
        <junit-jupiter.version>5.3.2</junit-jupiter.version>
        <mockito.version>2.24.0</mockito.version>
    </properties>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.2.RELEASE</version>
    </parent>

    <dependencies>

        <!-- mvc -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- exclude junit 4 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>junit</groupId>
                    <artifactId>junit</artifactId>
                </exclusion>
            </exclusions>
```

```
        </dependency>

        <!-- junit 5 -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>${junit-jupiter.version}</version>
            <scope>test</scope>
        </dependency>

        <!-- mockito + junit 5 -->
        <!-- exclude this, mockito still ok with junit 5, why need this? -->
        <!--
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-junit-jupiter</artifactId>
            <version>${mockito.version}</version>
            <scope>test</scope>
        </dependency>
        -->

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <optional>true</optional>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.0</version>
            </plugin>

        </plugins>
    </build>

</project>
```

테스트 대상 코드

StartApplication.java

```
package com.junit5mockito.core;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class StartApplication {

    public static void main(String[] args) {
        SpringApplication.run(StartApplication.class, args);
    }

}
```

service

HelloService.java

```
package com.junit5mockito.core.services;

public interface HelloService {

    String get();

}
```

HelloServiceImpl.java

```
package com.junit5mockito.core.services;

import com.junit5mockito.core.repository.HelloRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class HelloServiceImpl implements HelloService {

    @Autowired
    HelloRepository helloRepository;

    @Override
    public String get() {
        return helloRepository.get();
    }

}
```

테스트 결과