Семинар 4 – JUNIT5 (Часть 2)

# Задача 11(15)

1. Каковы этапы жизненного цикла теста в JUnit 5?

1. Фазы жизненного цикла теста

Обычно тестовый класс содержит несколько тестовых методов. JUnit управляет выполнением каждого метода тестирования в форме жизненного цикла.

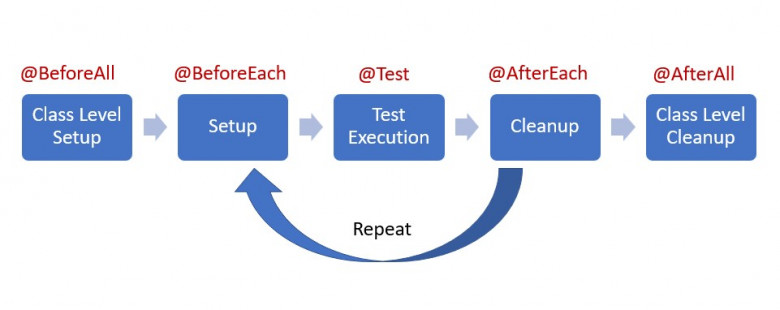
Полный жизненный цикл тестового сценария можно разделить на три фазы с помощью аннотаций.

Setup (настройка): на этом этапе создается тестовая инфраструктура. JUnit обеспечивает настройку уровня класса (@BeforeAll) и настройку уровня метода (@BeforeEach). Как правило, тяжелые объекты, такие как подключения к базам данных, создаются при настройке уровня класса, в то время как легкие объекты, такие как тестовые объекты, перезапускаются при настройке уровня метода.

Test Execution (выполнение теста): на этом этапе происходит выполнение теста и assertion. Результат выполнения будет означать успех или неудачу теста.

Cleanup (очистка): этот этап используется для очистки настройки тестовой инфраструктуры, настроенной на первом этапе. Как на этане настройки, очистка также выполняется на уровне класса (@AfterAll) и уровне метода (@AfterEach).

Фазы жизненного цикла теста



1. Что такое «повторяющийся тест (англ., repeated test)» в Junit5? Какие аннотации можно использовать при повторяющихся тестах и для чего они нужны?

Аннотация @RepeatedTest используется для обозначения метода тестирования, который должен повторяться указанное количество раз с настраиваемым отображаемым именем.

Обратите внимание, что каждый вызов повторного теста ведет себя как выполнение обычного теста с полной поддержкой тех же обратных вызовов и расширений жизненного цикла.

Это означает, что аннотированные методы жизненного цикла @BeforeEach и @AfterEach будут вызываться при каждом вызове теста.

Помимо указания количества повторений, мы можем дать каждому повторению собственное отображаемое имя. Это настраиваемое отображаемое имя может быть комбинацией {статический текст + динамические заполнители}.

В настоящее время поддерживаются три заполнителя (placeholder):

{displayName}: отображаемое имя метода @RepeatedTest.

{currentRepetition}: текущее количество повторений.

{totalRepetitions}: общее количество повторов.

**public** **class** **JUnit5AnnotationsExample**

{

@DisplayName("Add operation test")

@RepeatedTest(value = 5, name = "{displayName} - repetition {currentRepetition} of {totalRepetitions}")

**void** **addNumber**(TestInfo testInfo) {

Assertions.assertEquals(2, Calculator.add(1, 1), "1 + 1 should equal 2");

}

}

1. Что такое «параметризованный тест (англ., parametrized test)» в JUnit 5? Какие аннотации можно использовать с параметризованными тестами и для чего они нужны?

Используйте аннотацию @ParameterizedTest, чтобы выполнить тест несколько раз, но с разными аргументами. Нам не нужно использовать аннотацию @Test, вместо этого в таких тестах используется только аннотация @ParameterizedTest.

Мы должны объявить по крайней мере один источник аргументов, предоставляющий аргументы для каждого вызова, которые будут использоваться в тестовом методе.

В данном примере testPalindrome будет вызываться 2 раза для каждой строки, указанной в аннотации @ValueSource. Мы получаем доступ к аргументу, используя параметр word метода.

Используйте аргумент name в аннотации @ParameterizedTest, чтобы настроить отображаемое сообщение.

* Используйте @ValueSource для простых буквальных значений, таких как примитивы и строки.

3.2. Аннотация @NullSource

Она предоставляет единственный null аргумент методу, аннотированному @ParameterizedTest.

3.3. Аннотация @EmptySource

Она предоставляет метод, аннотированный @ParameterizedTest, с единственным пустым аргументом следующих типов:

* java.lang.String
* java.util.List
* java.util.Set
* java.util.Map
* примитивные массивы (например, int [])
* массивы объектов (например, String [])

3.4. Аннотация @NullAndEmptySource

Она сочетает в себе функциональность @NullSource и @EmptySource. В данном примере тестовый метод будет вызываться два раза - сначала со значением null, а затем со значением empty.

3.5. Аннотация @EnumSource

Это удобный способ использования Enum констант. Метод тестирования будет вызываться для каждой константы перечисления за раз.

3.6. Аннотация @MethodSource

Она используется для ссылки на один или несколько фабричных методов тестового класса или внешних классов. Фабричный метод должен генерировать поток аргументов, где каждый аргумент в потоке будет использоваться методом, аннотированным @ParameterizedTest.

3.7. Аннотация @CsvSource

Эта аннотация позволяет нам задавать списки аргументов как значения, разделенные запятыми. Каждый CSV токен представляет собой строку CSV и приводит к одному вызову параметризованного теста.

3.8. Аннотация @CsvFileSource

Эта аннотация очень похожа на @CsvSource за исключением того, что мы читаем токены CSV из файла вместо чтения токенов в исходном тексте. CSV файл можно прочитать по classpath или из локальной файловой системы.

3.9. Аннотация @ArgumentsSource

Аннотацию @ArgumentsSource можно использовать для указания настраиваемого многоразового поставщика аргументов ArgumentsProvider.

4.2. Интерфейс ArgumentsProvider

Чтобы предоставить несколько тестовых аргументов сложных или настраиваемых типов, мы должны использовать аннотацию @ArgumentsSource с аннотацией ArgumentsProvider.

1. Что такое «динамический тест (англ., dynamic test)»? Чем он отличается от обычного теста? Какие аннотации необходимы для работы с динамическими тестами?

Стандартные тесты, снабженные аннотацией @Test, являются статическими тестами, которые полностью задаются во время компиляции. Динамический тест - это тест, сгенерированный во время выполнения. Эти тесты генерируются с помощью фабричного метода, снабженного аннотацией @TestFactory .

Метод @TestFactory должен возвращать поток, коллекцию, итерацию или итератор экземпляров DynamicTest. Возврат чего-либо еще приведет к исключению JUnitException, поскольку недопустимые возвращаемые типы не могут быть обнаружены во время компиляции. Помимо этого, метод @TestFactory не может быть статическим или частным.

Динамические тесты выполняются иначе, чем стандартные @Test и не поддерживают обратные вызовы жизненного цикла. Это означает, что методы @beforeEach и @afterEach не будут вызываться для динамических тестов.

@TestFactory Collection<DynamicTest> **dynamicTestsWithCollection**() { **return** Arrays.asList( DynamicTest.dynamicTest("Add test", () -> assertEquals(2, Math.addExact(1, 1))), DynamicTest.dynamicTest("Multiply Test", () -> assertEquals(4, Math.multiplyExact(2, 2)))); }

# Задача 12 (20)

Создать набор классов и пакетов и протестировать их с помощью JUnit5. Надо использовать повторяющиеся, параметризованные и динамические тесты.