Написание тестов с JUnit 5

[Триша Джи](https://blog.jetbrains.com/author/trishagee) 16 сентября 2020 г.

## **Настройка Gradle для JUnit 5**

**Alt + Insert -** добавить новую зависимость. Ввод «junit»

**Settings/Preferences | Build, Execution, Deployment | Build Tools | Maven | Repositories >** Update вообще то-сама индексирует

**Ctrl + Shift + O или слоник sync**

можем увидеть зависимости junit-jupiter в разделе «Внешние библиотеки» окна нашего проекта

Нам нужно [указать Gradle использовать платформу JUnit](https://junit.org/junit5/docs/current/user-guide/#running-tests-build-gradle) при запуске тестов, добавив useJUnitPlatform()в testраздел

## **Создание и запуск теста**

**Alt + Insert** в проекте - создать java class в каталоге test

**Alt + Insert**  в классе – сгенерировать пустой тест

 Единственное отличие от JUnit 5 состоит в том, что он использует аннотацию Test из пакета jupiter

void shouldShowSimpleAssertion() {  
 Assertions.*assertEquals*(1,1);

запустить – работает

для [тестов JUnit 5](https://junit.org/junit5/docs/current/user-guide/#writing-tests-classes-and-methods) метод тестирования не обязательно должен быть общедоступным, чтобы он работал

## **Конфигурация: Подсказки по параметрам**

## **Конфигурация: Test Runner**

 По умолчанию IntelliJ IDEA использует Gradle для сборки и запуска кода и тестов в проектах Gradle. Это гарантирует, что когда мы запускаем приложение или тесты в среде IDE, оно работает так же, как и в других средах, таких как командная строка или среда непрерывной интеграции. Это также гарантирует выполнение любой сложной логики сборки или настройки или генерации кода. Однако мы можем использовать средство запуска IntelliJ IDEA для запуска наших тестов. В некоторых случаях это может быть быстрее, чем использование Gradle, и обеспечивать более быстрый цикл обратной связи.

## **Отключение или игнорирование тестов**

Довольно часто мы хотим сказать, что не хотим запускать тест. Это обычное дело для [разработки, основанной](https://martinfowler.com/bliki/TestDrivenDevelopment.html) на тестировании, поскольку тесты по определению не работают, когда мы их впервые напишем. [JUnit 5 поддерживает это](https://junit.org/junit5/docs/current/user-guide/" \l "writing-tests-disabling) с помощью @Disabledаннотации. Мы можем добавить описательный текст, чтобы указать, почему тест не следует запускать.

@Тест

@ Disabled ( "Еще не реализовано" )

ПРИМЕЧАНИЕ: тесты обычно следует отключать только на короткий период времени, пока код, который они тестируют, не заработает. Если тест отключен на долгое время, возможно, из-за того, что мы не знаем, почему он не работает или каково его ожидаемое поведение, он не добавляет никакой ценности набору тестов. Подобный тест следует удалить.

## **Полезные названия тестов для отображения**

@ DisplayName ( «Должно продемонстрировать простое утверждение» )

Это не только побуждает нас быть описательными, поскольку это текстовая строка, а не имя метода, но и поддерживаются специальные символы, которые могут улучшить читаемость.

## **Совет IDE: живые шаблоны**

Editor -> Live Templates .

Если у нас есть стандартный шаблон для новых методов тестирования, которым мы хотели бы следовать, мы могли бы изменить [шаблон метода тестирования](https://www.jetbrains.com/help/idea/using-file-and-code-templates.html) по [умолчанию в IntelliJ IDEA](https://www.jetbrains.com/help/idea/using-file-and-code-templates.html) или написать [Live Template,](https://www.jetbrains.com/help/idea/using-live-templates.html) который поможет нам создавать новые методы тестирования, которые выглядят именно так. мы хотим.

Давайте создадим живой шаблон для создания нового метода тестирования с, DisplayNameкоторый изначально преобразуется в CamelCase и применяется к имени метода. Это побуждает нас использовать DisplayNameаннотации для написания удобочитаемых описаний тестов и использовать их для создания допустимых имен методов, поэтому имя метода также полезно. Код, который должен сгенерировать наш Live Template, будет выглядеть примерно так:

@Тест

@ DisplayName ( «Следует проверить все элементы в списке» )

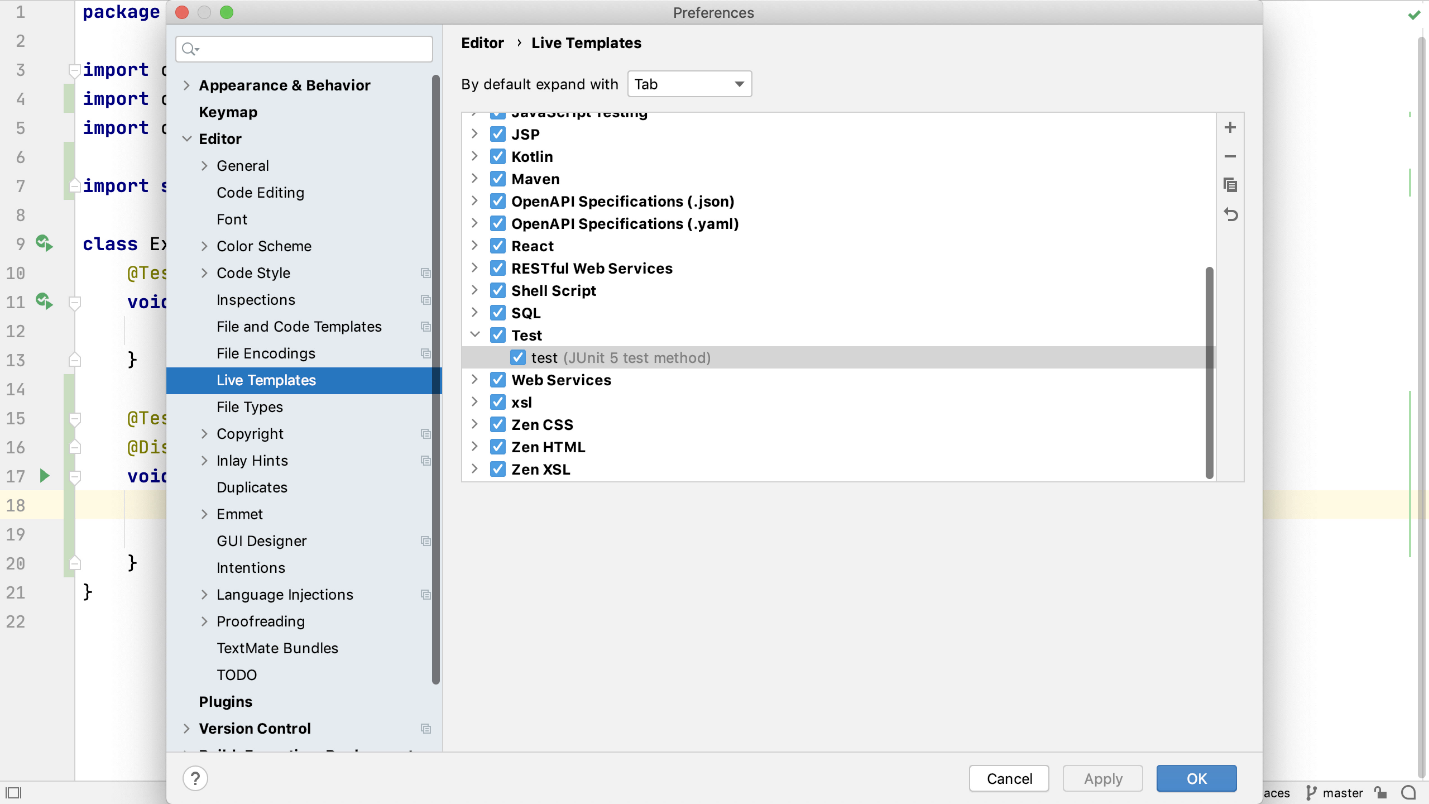
**void** shouldCheckAllItemsInTheList () {

fail ( «Не реализовано» ) ;

}

Хорошей практикой является автоматическая вставка сгенерированных тестов failв сгенерированный метод - любой тест должен сначала завершиться неудачно, даже если мы еще не закончили его писать.

Чтобы [создать этот живой шаблон](https://www.jetbrains.com/help/idea/creating-and-editing-live-templates.html) , откройте настройки и перейдите в Editor -> Live Templates .



Используя «+» в правом верхнем углу панели прокрутки, создайте новую группу живых шаблонов под названием «Test». Выбрав эту группу, снова используйте «+», чтобы создать новый живой шаблон.

В подробностях живого шаблона внизу экрана:

* Дайте шаблону аббревиатуру "тест".
* Дайте ему полезное описание, например "Метод тестирования JUnit 5".

Ключ к живым шаблонам - это создание текста шаблона. Это довольно сложный шаблон, поэтому текст довольно сложный:

@org.junit.jupiter.api.Test

@org.junit.jupiter.api.DisplayName("$TEST\_NAME$")

void $METHOD\_NAME$() {

$END$

$BODY$

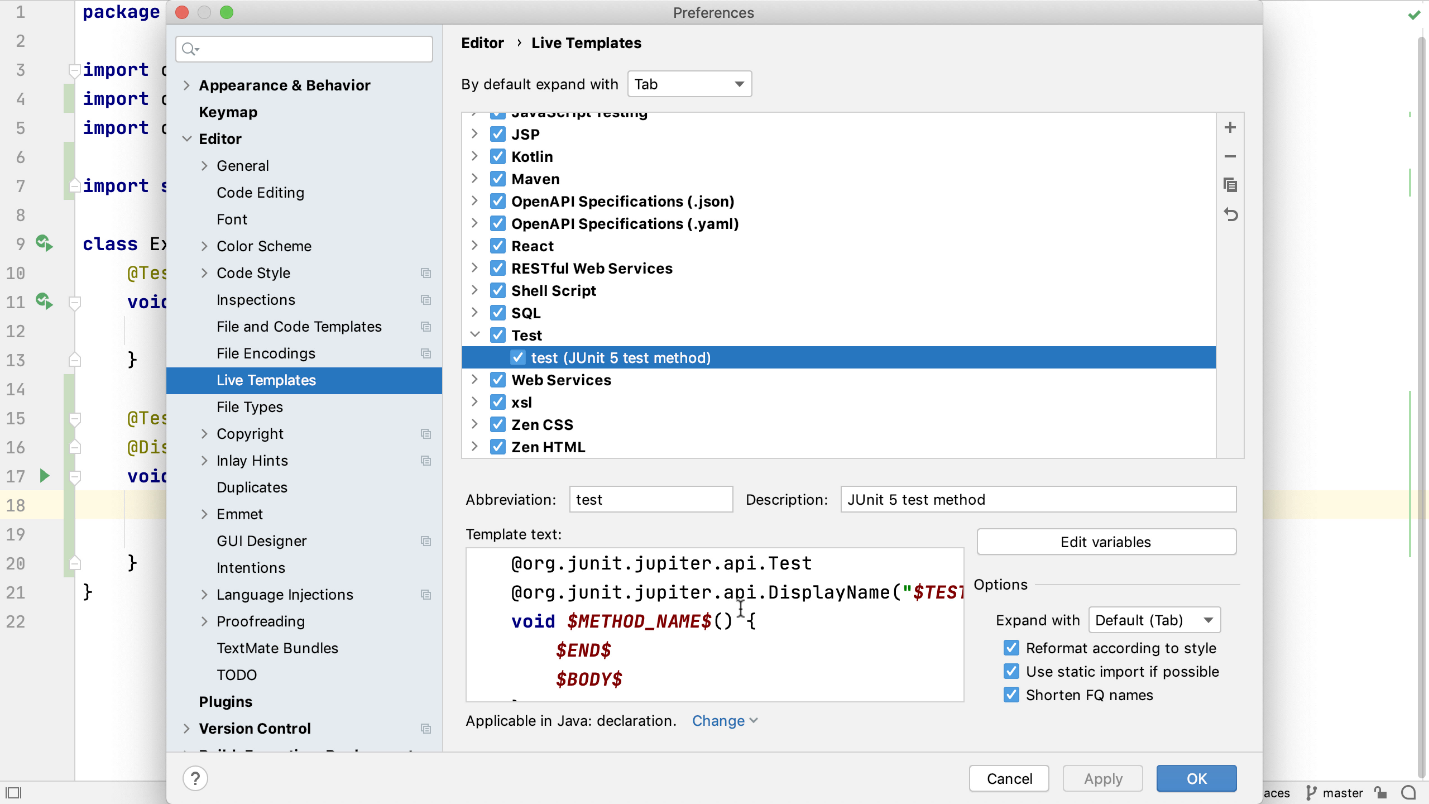
}

ПРИМЕЧАНИЕ. Используйте полностью определенные имена (имя пакета плюс имя класса) для аннотаций, чтобы IntelliJ IDEA точно знала, какой класс вам нужен. Установите флажок «Сократить имена FQ», чтобы IntelliJ IDEA автоматически добавлял правильный импорт и использовал только имя класса в аннотации.

Я люблю отмечать:

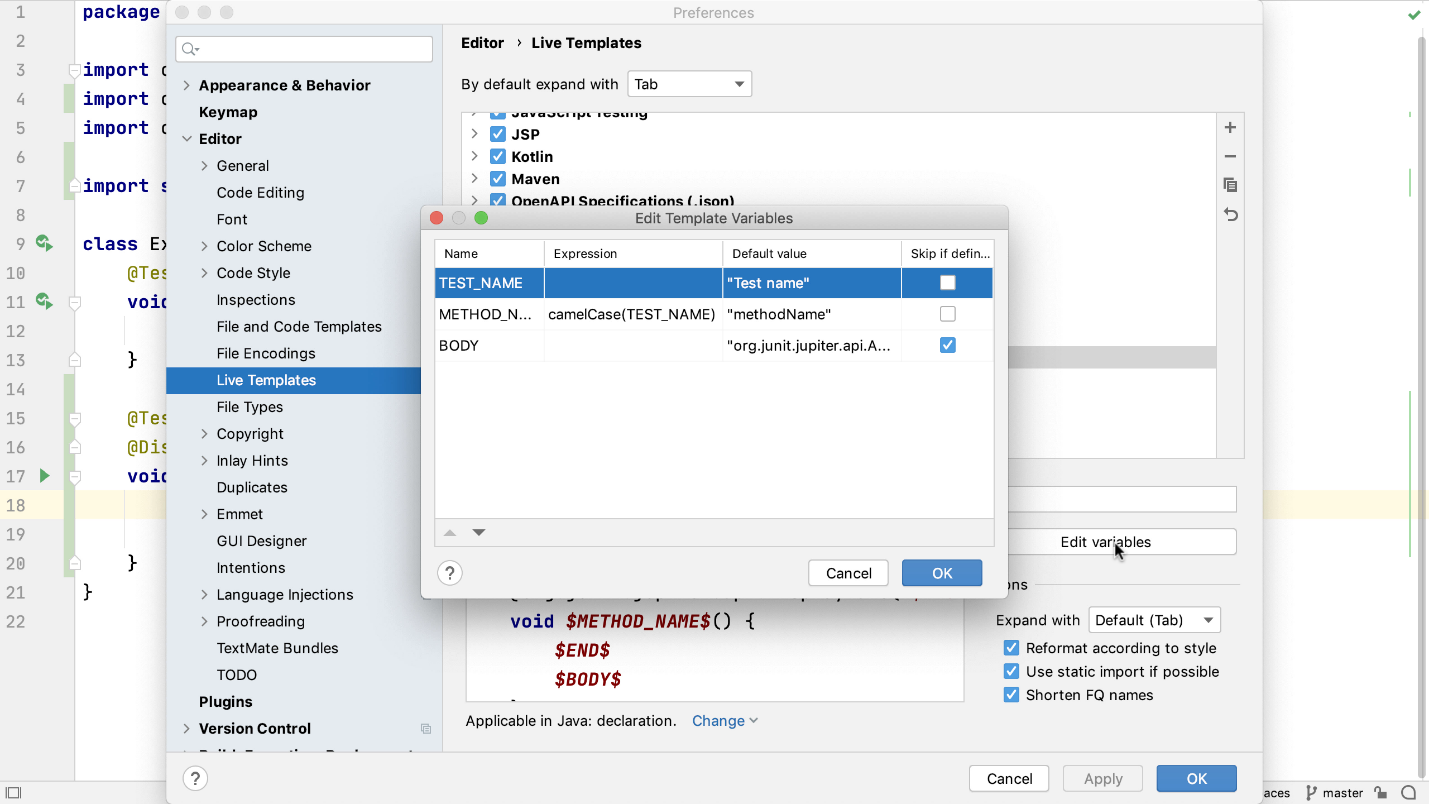
* Переформатировать в соответствии со стилем
* Если возможно, используйте статический импорт
* Сократить имена FQ

в моих живых шаблонах, то, когда код вставляется в файл класса, он обычно следует тем же стандартам, что и остальная часть приложения.



Вам необходимо определить область, к которой применяется действующий шаблон, иначе IDE не будет знать, в каких типах файлов и в какое время она должна предлагать этот шаблон. Щелкните ссылку «определить» рядом с предупреждением «Нет применимых контекстов» и выберите Java -> Объявление . IntelliJ IDEA теперь добавит это в список предложений, когда мы находимся в файле класса Java.

Обратите внимание на переменные в шаблоне. Некоторые из них встроены в IDE, например $END, там, где курсор окажется, когда живой шаблон завершит вставку всего кода. Некоторые ценности вам придется определить. Давайте теперь определим их. Нажмите кнопку «Редактировать переменные», чтобы открыть окно переменных.

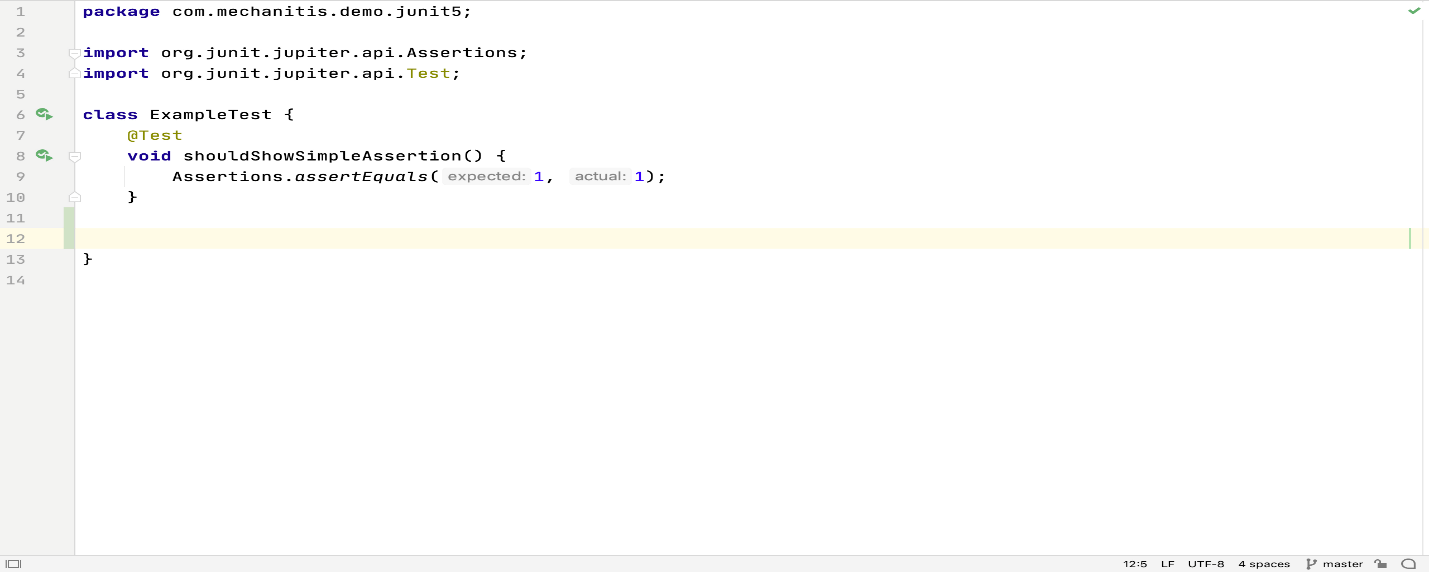


Установите следующие значения для переменных:

* TEST\_NAME:
  + Выражение = <оставить пустым>
  + Значение по умолчанию ="Test name"
* METHOD\_NAME:
  + Выражение =camelCase(TEST\_NAME)
  + Значение по умолчанию ="methodName"
* BODY:
  + Выражение = <оставить пустым>
  + Значение по умолчанию ="org.junit.jupiter.api.Assertions.fail(\"Not implemented\");"
  + Отметьте «Пропустить, если определено».

Нажмите OK в окне переменных и OK в окне настроек.

Проверьте живой шаблон в редакторе. Убедитесь, что курсор находится внутри тестового класса Java, но за пределами существующего тестового метода. Введите testи нажмите вкладку. IntelliJ IDEA должна сгенерировать скелет метода тестирования, а курсор должен находиться в значении DisplayNameаннотации. Введите здесь удобочитаемое описание теста, и вы должны увидеть, что текстовое описание также преобразовано в допустимое имя метода Java camelCase. Нажмите Enter, когда вы закончите значение для DisplayName, и курсор должен переместиться, чтобы выбрать имя метода, если вы хотите его отредактировать. Повторное нажатие Enter должно поместить курсор над failвызовом.



@Test  
@DisplayName("Test name")  
void testName() {  
   
 *fail*("Not implemented");  
}

**Множественные утверждения**

Как мы уже видели, [JUnit 5 поддерживает стандартные утверждения,](https://junit.org/junit5/docs/current/user-guide/" \l "writing-tests-assertions) которые могут быть вам знакомы, если мы использовали другие среды тестирования. В реальном мире нам часто приходится проверять несколько вещей, чтобы доказать, что что-то работает так, как мы ожидали. Возьмем, к примеру, список. Если мы хотим проверить правильность каждого элемента в нем, мы можем написать несколько утверждений для проверки каждого значения.

@Тест

@ DisplayName ( «Следует проверить все элементы в списке» )

**void** shouldCheckAllItemsInTheList () {

Список < Целое число > чисел = Список. из ( 2 , 3 , 5 , 7 ) ;

Утверждения. assertEquals ( 2 , числа. получить ( 0 )) ;

Утверждения. assertEquals ( 3 , числа. получить ( 1 )) ;

Утверждения. assertEquals ( 5 , числа. получить ( 2 )) ;

Утверждения. assertEquals ( 7 , числа. получить ( 3 )) ;

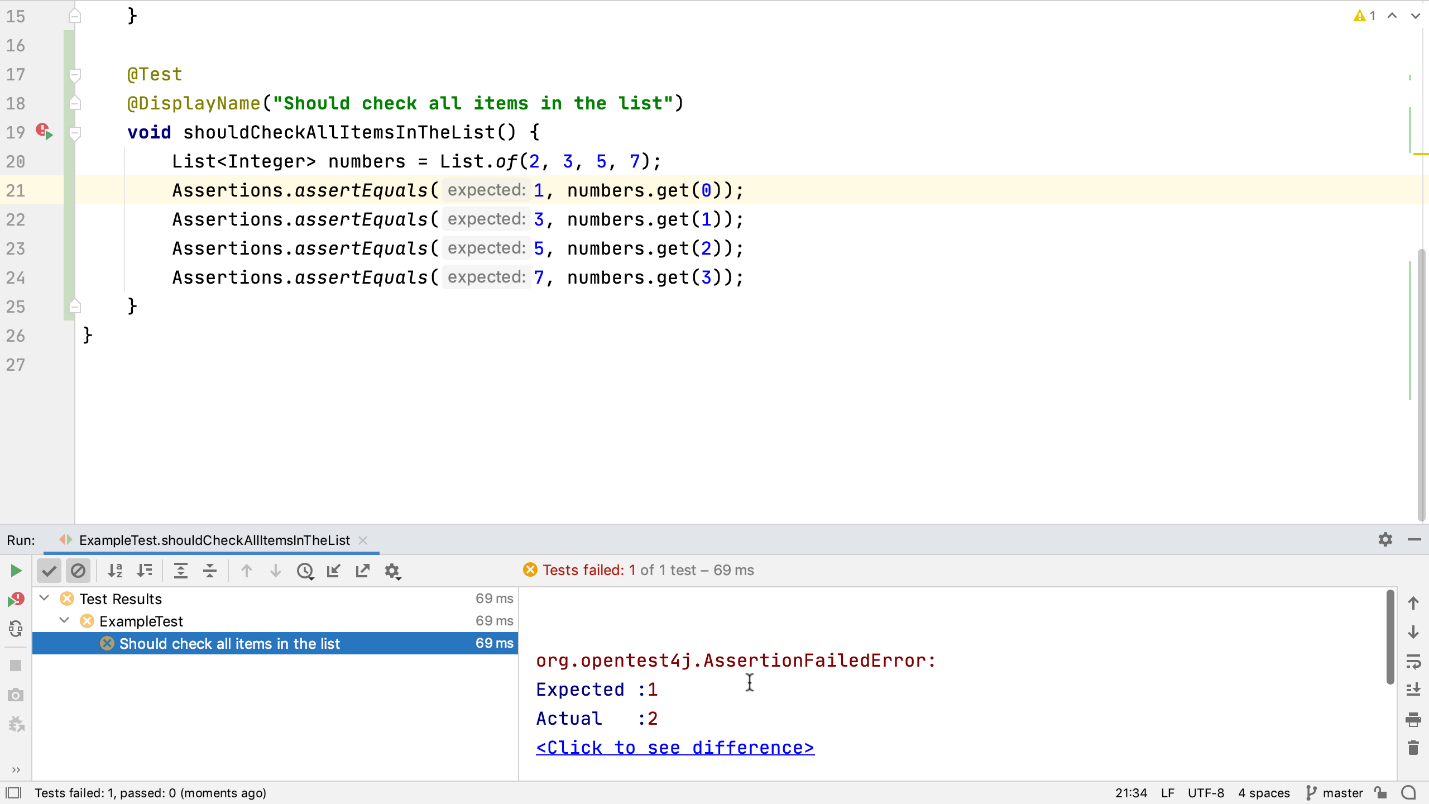
}

Это работает, оно обязательно пройдет, если все элементы в списке будут такими, как ожидалось. Проблема возникает, когда одно из утверждений не выполняется. Измените первое утверждение так, чтобы оно не сработало:

Список < Целое число > чисел = Список. из ( 2 , 3 , 5 , 7 ) ;

Утверждения. assertEquals ( 1 , числа. получить ( 0 )) ;

Выходные данные показывают, что тест не прошел и почему.



Однако мы не знаем, прошли ли другие утверждения или нет, потому что JUnit не будет запускать утверждения после первого сбоя. Вы можете видеть, что если вы измените все остальные утверждения на неудачные:

@Тест

@ DisplayName ( «Следует проверить все элементы в списке» )

**void** shouldCheckAllItemsInTheList () {

Список < Целое число > чисел = Список. из ( 2 , 3 , 5 , 7 ) ;

Утверждения. assertEquals ( 1 , числа. получить ( 0 )) ;

Утверждения. assertEquals ( 1 , числа. получить ( 1 )) ;

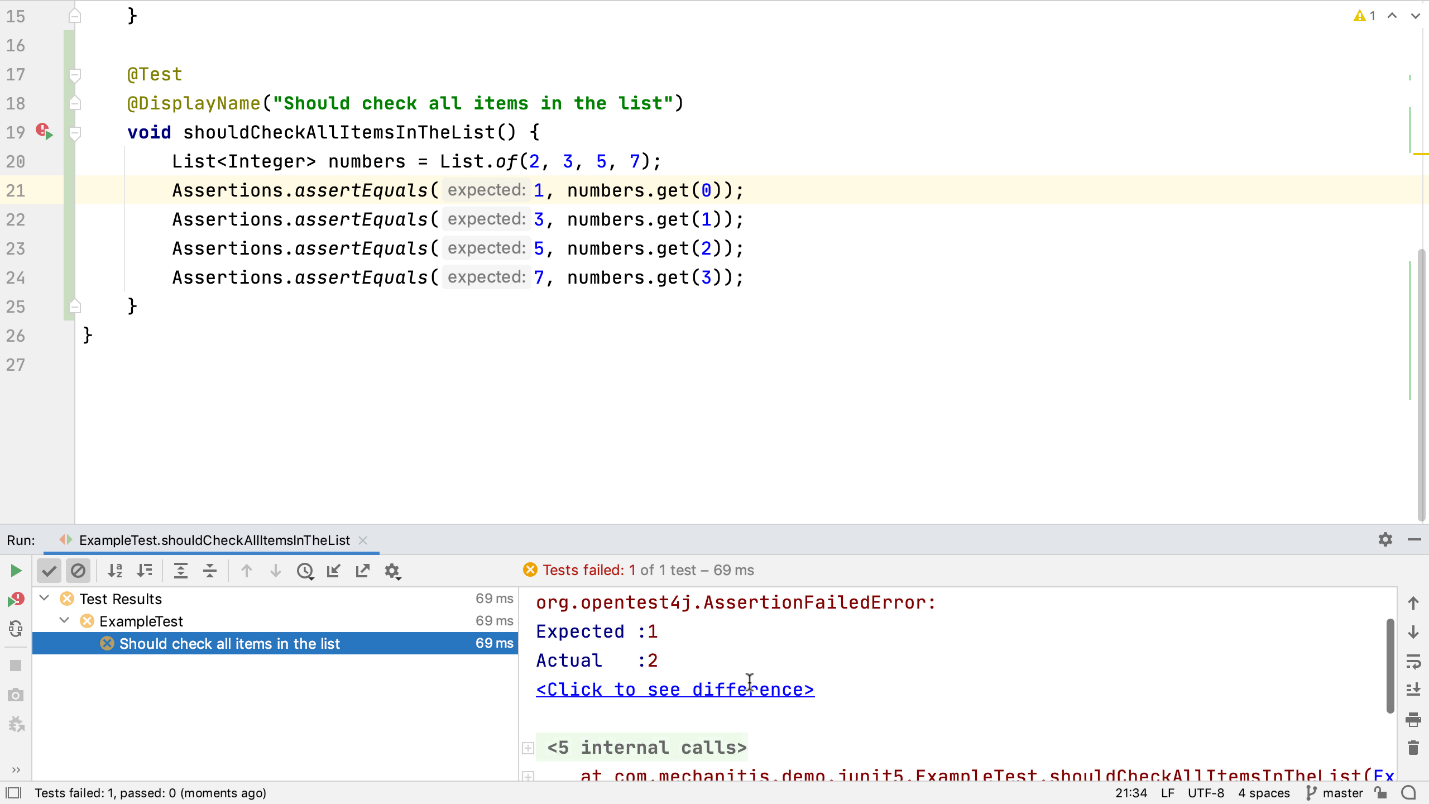
Утверждения. assertEquals ( 1 , числа. получить ( 2 )) ;

Утверждения. assertEquals ( 1 , числа. получить ( 3 )) ;

}

ПРИМЕЧАНИЕ: вы можете использовать [режим выбора](https://www.jetbrains.com/help/idea/working-with-source-code.html#multiple_cursor) столбца [или несколько знаков вставки,](https://www.jetbrains.com/help/idea/working-with-source-code.html#multiple_cursor) чтобы легко редактировать все «ожидаемые» значения сразу.

Запустите тест, чтобы еще раз убедиться, что только первое утверждение не работает, мы понятия не имеем, что другие также не работают.



Это могло быть проблемой - мы бы вернулись и исправили первое утверждение, повторно запустили тест, нам нужно было исправить следующее, повторно запустить тест и так далее. Это не та быстрая обратная связь, которую мы ищем.

JUnit 5 поддерживает [утверждение assertAll](https://junit.org/junit5/docs/current/user-guide/#writing-tests-assertions) . Это проверит каждое утверждение, даже если одно из них не работает. Мы делаем это, помещая все утверждения, которые хотим сгруппировать, в assertAllвызов в виде серии лямбда-выражений.

@Тест

@ DisplayName ( «Следует проверить все элементы в списке» )

**void** shouldCheckAllItemsInTheList () {

Список < Целое число > чисел = Список. из ( 2 , 3 , 5 , 7 ) ;

Утверждения. assertAll (() - > assertEquals ( 1 , числа. получить ( 0 )) ,

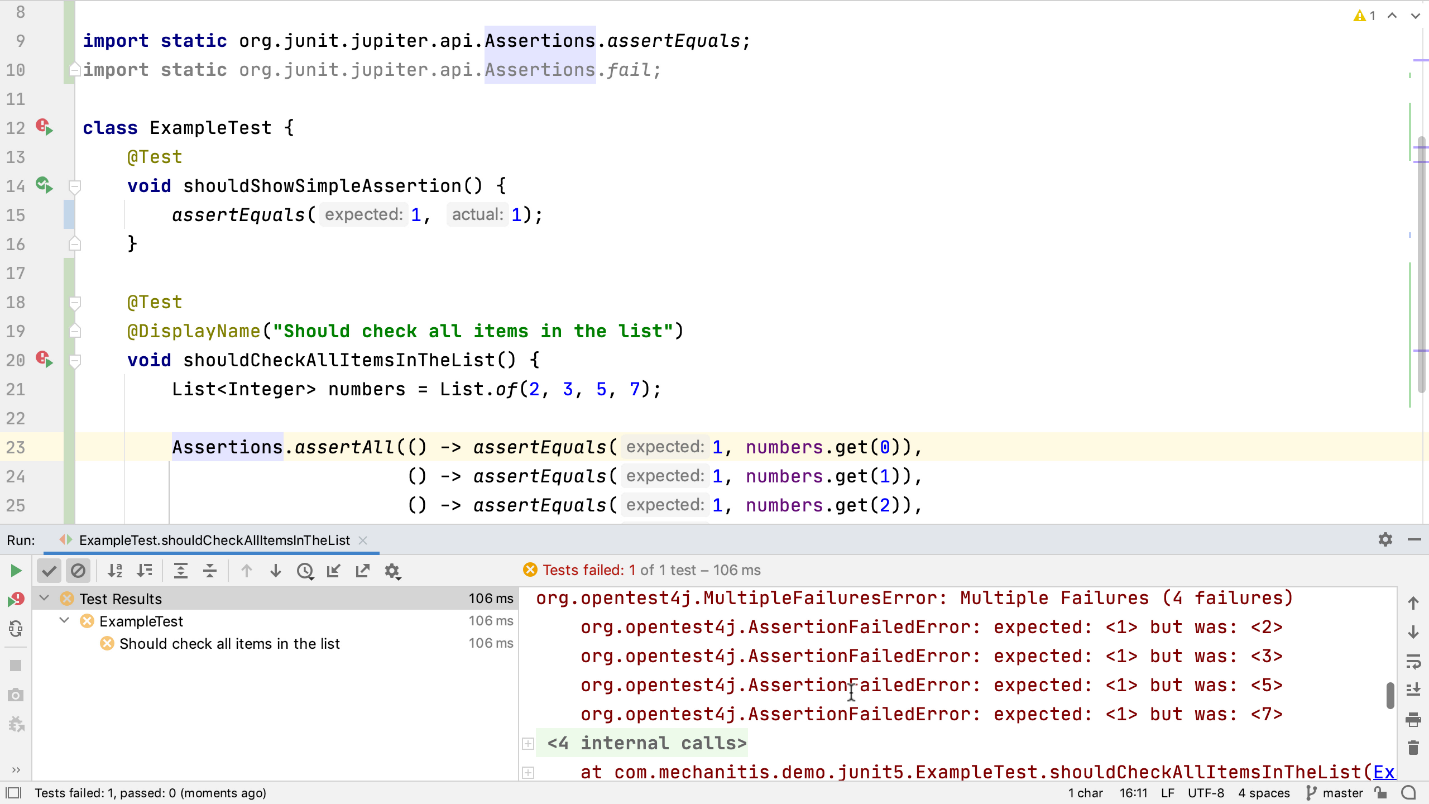
() - > assertEquals ( 1 , числа. получить ( 1 )) ,

() - > assertEquals ( 1 , числа. получить ( 2 )) ,

() - > assertEquals ( 1 , числа. получить ( 3 ))) ;

}

Давайте сохраним тест со значениями, которые должны завершиться неудачей, чтобы мы могли увидеть, что происходит, когда мы запускаем ошибку assertAll:



Мы видим, что *все* утверждения потерпели неудачу - все они были выполнены, несмотря на то, что первое из них не удалось. Это значительно упрощает нам выявление проблем и их устранение за один проход, вместо того, чтобы повторно запускать тест повторно.

Внесите изменения, чтобы исправить тест:

@Тест

@ DisplayName ( «Следует проверить все элементы в списке» )

**void** shouldCheckAllItemsInTheList () {

Список < Целое число > чисел = Список. из ( 2 , 3 , 5 , 7 ) ;

Утверждения. assertAll (() - > assertEquals ( 2 , числа. получить ( 0 )) ,

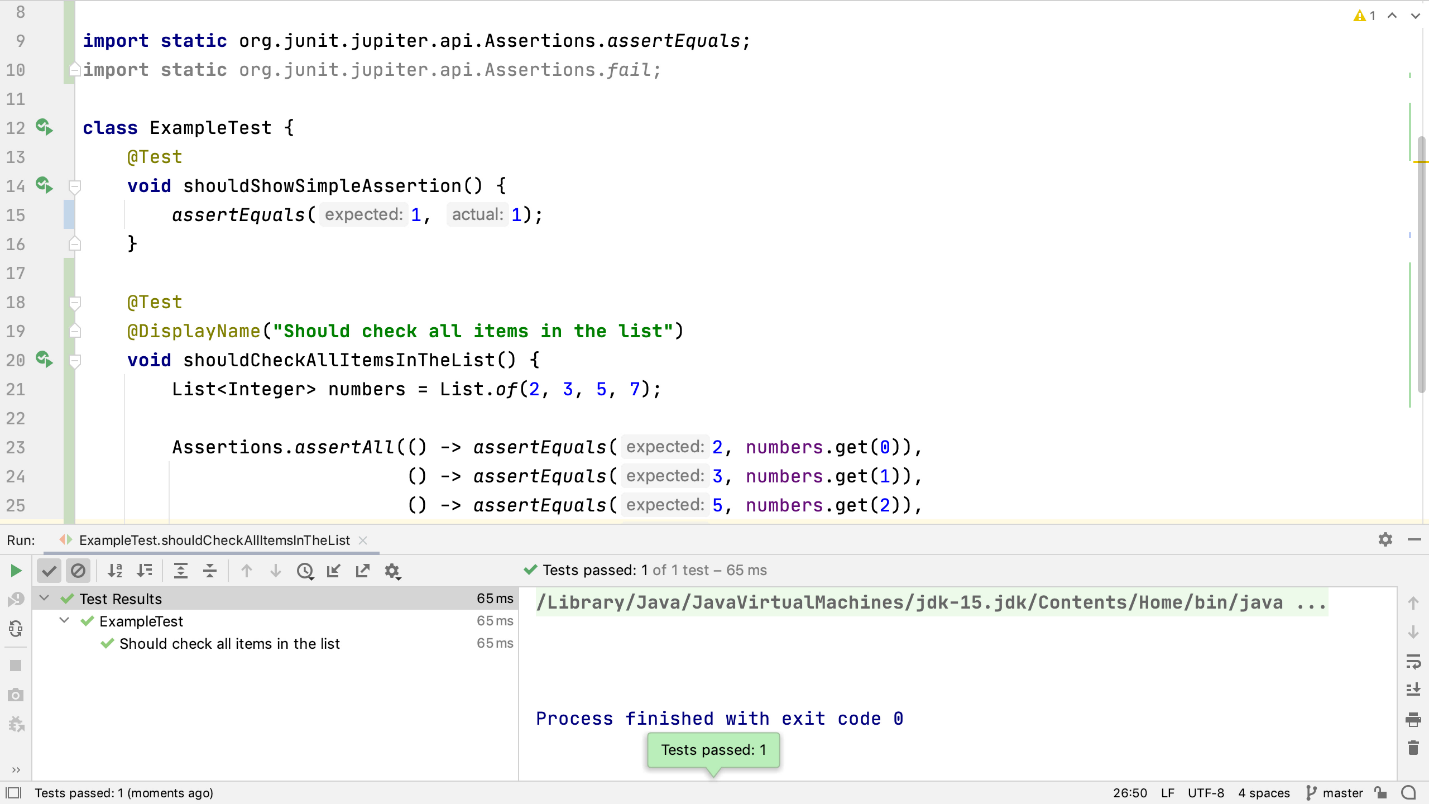
() - > assertEquals ( 3 , числа. получить ( 1 )) ,

() - > assertEquals ( 5 , числа. получить ( 2 )) ,

() - > assertEquals ( 7 , числа. получить ( 3 ))) ;

}

Повторный запуск теста должен показать, что все работает:



Предположения

Теперь давайте посмотрим на [предположения](https://junit.org/junit5/docs/current/user-guide/#writing-tests-assumptions) в JUnit 5. [Более поздние версии JUnit 4 поддерживали предположения](https://junit.org/junit4/javadoc/4.12/org/junit/Assume.html) , но те из нас, кто привык работать со старыми тестами, возможно, не сталкивались с этой концепцией раньше. Мы можем захотеть написать тесты, которые выполняются только при определенных обстоятельствах, которые верны - например, если мы используем определенный тип хранилища или используем определенную версию библиотеки. Это может быть более применимо к системным или интеграционным тестам, чем к модульным тестам. В этих случаях мы можем установить предположение в начале теста, и тест будет запущен только в том случае, если критерии для этого предположения соблюдены. Давайте напишем тест, который следует запускать только в том случае, если мы используем версию API выше десяти.

@Тест

@ DisplayName ( «Следует запускать тест только при соблюдении некоторых критериев» )

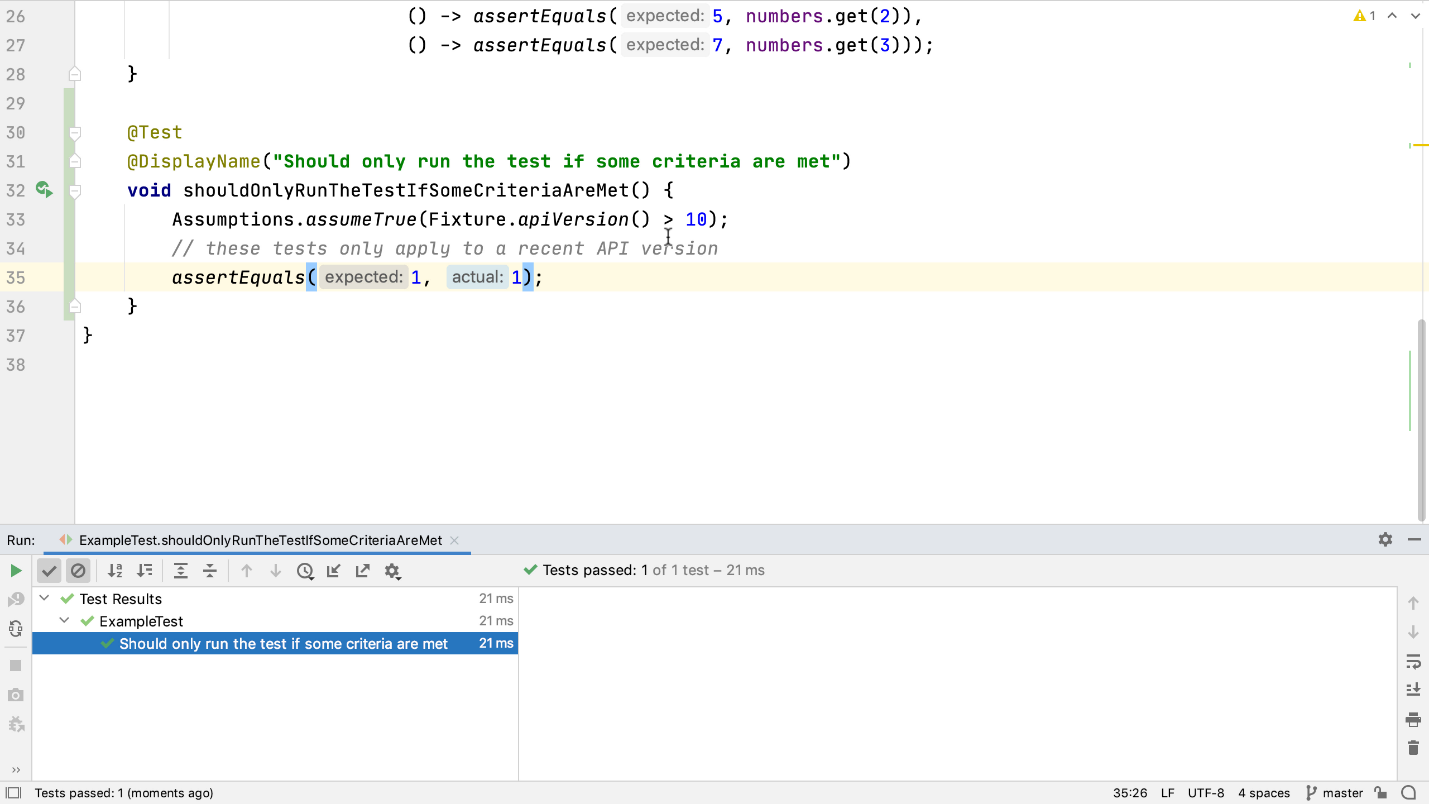
**void** shouldOnlyRunTheTestIfSomeCriteriaAreMet () {

Предположения. acceptTrue ( Fixture. apiVersion () > 10 ) ;

assertEquals ( 1 , 1 ) ;

}

Когда мы запускаем тест, мы видим, что этот тест выполняется и проходит, как ожидалось, потому что [Fixture](https://github.com/trishagee/junit5-showcase/blob/b70dacfa5cd0d24388656e6504be4a98991994e5/src/test/java/com/mechanitis/demo/junit5/Fixture.java) возвращает версию API выше 10 (для этого руководства Fixture.apiVersion()возвращает 13).

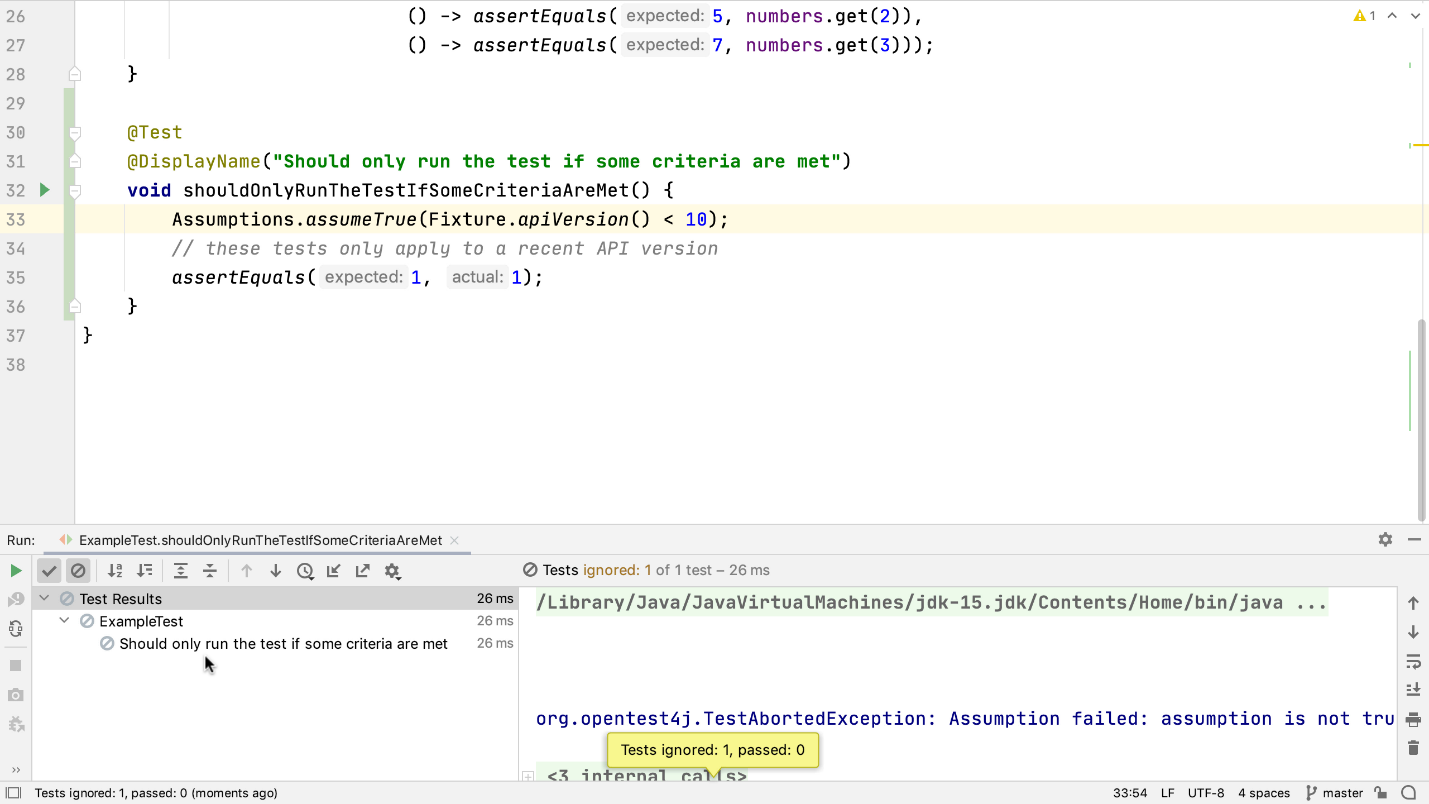


Давайте перевернем проверку в предположении, чтобы тест запускался только в том случае, если версия API меньше 10:

Предположения. acceptTrue ( Fixture. apiVersion () < 10 ) ;

assertEquals ( 1 , 1 ) ;

Повторите тест - он не должен стать зеленым. Поскольку наша версия API выше десяти, эта проверка возвращает false, предположение не выполняется и тест не запускается. Он отображается как отключенный или проигнорированный тест:



Тесты на основе данных

Ранее мы видели, что можем использовать assertAllдля группировки ряда утверждений и обеспечения их выполнения. Это один из способов выполнения нескольких проверок. Есть и другие случаи, когда мы можем захотеть выполнить один и тот же набор проверок для разных наборов данных. Для этого мы можем использовать [параметризованные тесты](https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests) . Параметризованные тесты - это то, где мы можем передавать данные в тест в качестве параметров, и с JUnit 5 есть несколько разных способов сделать это ( [см. Документацию](https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests) , это очень хорошо). Мы рассмотрим простейший подход, чтобы показать, как он работает.

Давайте воспользуемся @ValueSourceаннотацией, чтобы дать методу тестирования серию отдельных значений для тестирования.

@ParameterizedTest

@ DisplayName ( «Следует создавать фигуры с разным количеством сторон» )

@ ValueSource ( Интс = { 3 , 4 , 5 , 8 , 14 })

**void** shouldCreateShapesWithDifferentNumbersOfSides ( int expectedNumberOfSides ) {

}

JUnit 5 поддерживает множество различных типов ввода массива для этой аннотации, давайте воспользуемся массивом жестко запрограммированных целых чисел для этого теста. Каждое из этих значений будет передано в метод индивидуально, поэтому для метода тестирования требуется один intпараметр expectedNumberOfSides, чтобы передать значение.

ПРИМЕЧАНИЕ. IntelliJ IDEA может помочь нам с параметризованными тестами в JUnit 5. Он позволяет нам знать, что если мы используем ValueSourceаннотацию, мы не должны использовать @Testаннотацию, а ParameterizedTestвместо этого. Мы можем использовать **Alt + Enter,** чтобы IntelliJ IDEA изменила любые @Testаннотации @ParameterizedTest.

Внутри тестового метода вызовите конструктор [Shape](https://github.com/trishagee/junit5-showcase/blob/f562c1f14ecbdf784dabd45651905e439ad6e42c/src/main/java/com/mechanitis/demo/junit5/Shape.java) , передав количество сторон, предоставленных нам, и убедитесь, что Shape может дать нам правильное количество сторон.

@ParameterizedTest

@ DisplayName ( «Следует создавать фигуры с разным количеством сторон» )

@ ValueSource ( Интс = { 3 , 4 , 5 , 8 , 14 })

**void** shouldCreateShapesWithDifferentNumbersOfSides ( int expectedNumberOfSides ) {

Форма формы = **новая** форма ( ожидаемыйНомерОфСайд ) ;

assertEquals ( ожидаемое числоОфСайд , форма. числоОфСидес ()) ;

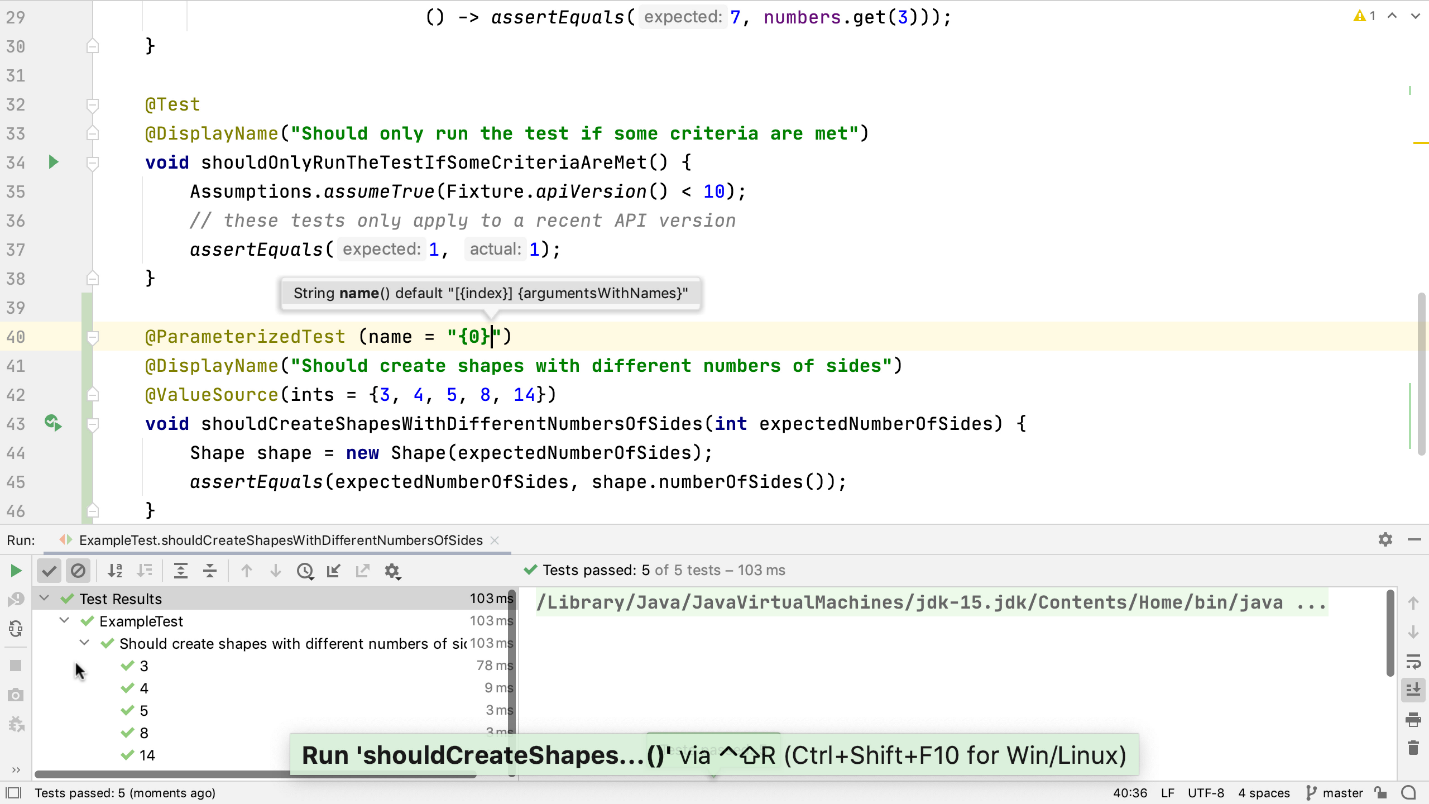
}

Запустите тест. На самом деле тест запускается не один раз. Тест запускается для каждого из intзначений, которые мы помещаем в ValueSourceаннотацию.

Мы можем изменить способ отображения этих отдельных тестов в результатах, создав собственное имя в ParameterizedTestаннотации. Для этого теста покажите значение количества сторон, с которыми создается форма, используя первый параметр ( expectedNumberOfSides) в качестве имени тестового экземпляра:

@ ParameterizedTest ( имя = "{0}" )

Когда тест запущен, мы видим, что окно запуска показывает количество сторон, используемых в качестве имени для каждого экземпляра теста:



Проверка исключений

Параметризованные тесты очень полезны для тестирования больших наборов достоверных данных, но они также действительно полезны для проверки большого количества недопустимых входных данных с теми же утверждениями.

Создайте новый тест, чтобы проверить недопустимый ввод. Установите новый ValueSource для ints, но на этот раз все intзначения будут недопустимыми числами сторон многоугольника. Предположим, что вам нужно проверить слишком мало сторон, и предположим, что код не поддерживает создание фигур с очень большим количеством сторон:

@ParameterizedTest

@ DisplayName ( «Не следует создавать фигуры с недопустимым количеством сторон» )

@ ValueSource ( Интс = { 0 , 1 , 2 , целое число. MAX\_VALUE })

**void** shouldNotCreateShapesWithInvalidNumbersOfSides ( int expectedNumberOfSides ) {

}

На этом этапе мы должны спросить себя: «Каково ожидаемое поведение, когда ввод неверен?». Если мы решим, что конструктор должен генерировать исключение, когда ему передаются недопустимые значения, мы можем проверить это с помощью [assertThrows](https://junit.org/junit5/docs/current/user-guide/" \l "writing-tests-assertions) . Мы Exceptionсообщаем ему, что мы ожидаем сгенерировать, и используем лямбда-выражение для передачи метода, который, как мы ожидаем, вызовет исключение.

@ ParameterizedTest ( имя = "{0}" )

@ DisplayName ( «Не следует создавать фигуры с недопустимым количеством сторон» )

@ ValueSource ( Интс = { 0 , 1 , 2 , целое число. MAX\_VALUE })

**void** shouldNotCreateShapesWithInvalidNumbersOfSides ( int expectedNumberOfSides ) {

assertThrows ( IllegalArgumentException. класс ,

() - > **новая** форма ( expectedNumberOfSides )) ;

}

Группировка тестов с @Nested

В этом последнем разделе мы рассмотрим одну из моих любимых функций JUnit 5 - [вложенные тесты](https://junit.org/junit5/docs/current/user-guide/#writing-tests-nested) . Вложенные тесты позволяют нам группировать определенные типы тестов вместе внутри более крупного класса. Есть много причин, по которым мы могли бы захотеть это сделать. Например, чтобы сгруппировать тесты с аналогичной настройкой или разобрать, но которые не настолько отличаются от других тестов в классе, что они должны быть в отдельном тестовом файле.

Мы собираемся использовать эту функцию, чтобы сгруппировать вместе все тесты, требующие уже настроенной формы.

Создайте внутренний класс и добавьте аннотацию Nested. Мы также можем добавить DisplayName к этому так же, как и к методу тестирования.

class ExampleTest {

@Nested

@ DisplayName ( «Когда форма была создана» )

**class** WhenShapeExists {

}

}

Вложенный класс, конечно, может содержать поля, и мы можем использовать их для хранения значений, которые потребуются всем тестам внутри этого внутреннего класса. Давайте создадим простую фигуру для использования в этих тестах.

class ExampleTest {

@Nested

@ DisplayName ( «Когда форма была создана» )

**class** WhenShapeExists {

частная окончательная форма формы = **новая** форма ( 4 ) ;

}

}

Мы даже можем создавать вложенные классы внутри нашего вложенного класса. Это может быть полезно для дальнейшей группировки. Мы собираемся использовать его в этом примере, чтобы сгруппировать вместе тесты [Happy Path,](https://en.wikipedia.org/wiki/Happy_path) тесты, которые проверяют, что все работает должным образом при нормальных обстоятельствах.

class ExampleTest {

@Nested

@ DisplayName ( «Когда форма была создана» )

**class** WhenShapeExists {

частная окончательная форма формы = **новая** форма ( 4 ) ;

@Nested

@ DisplayName ( "Если разрешить" )

**class** ShouldAllow {

}

}

}

Теперь мы можем создавать наши специальные тесты внутри наших вложенных классов. С вложенными классами мы, вероятно, захотим определить соглашение об именах, которое имеет смысл при печати результатов теста, что мы увидим через минуту. Давайте сделаем этот первый тест счастливого пути простой проверкой, которая показывает, что Shape возвращает правильное количество сторон. Затем мы можем создать еще один тест, который проверяет, что для нашей формы возвращается правильное описание.

@Nested

@ DisplayName ( "Если разрешить" )

**class** ShouldAllow {

@Тест

@ DisplayName ( "просмотр количества сторон" )

**void** visibleTheNumberOfSides () {

assertEquals ( 4 , shape. numberOfSides ()) ;

}

@Тест

@ DisplayName ( "просмотр описания" )

**void** visibleTheDescription () {

assertEquals ( "Квадрат" , shape. description ()) ;

}

}

(Обратите внимание, что в этом фрагменте я просто показываю самый внутренний класс, но он по-прежнему является частью более крупного класса)

Теперь давайте создадим группу для тестов, которые покажут, какое поведение *не* поддерживается или не ожидается. Предположим, что в нашем примере две фигуры с одинаковым количеством сторон не должны быть одной и той же формы. Это список для [всего класса](https://github.com/trishagee/junit5-showcase/blob/c598110f44e441165e41f563bb335e038975d690/src/test/java/com/mechanitis/demo/junit5/ExampleTest.java) :

class ExampleTest {

@Nested

@ DisplayName ( «Когда форма была создана» )

**class** WhenShapeExists {

частная окончательная форма формы = **новая** форма ( 4 ) ;

@Nested

@ DisplayName ( "Если разрешить" )

**class** ShouldAllow {

@Тест

@ DisplayName ( "просмотр количества сторон" )

**void** visibleTheNumberOfSides () {

assertEquals ( 4 , shape. numberOfSides ()) ;

}

@Тест

@ DisplayName ( "просмотр описания" )

**void** visibleTheDescription () {

assertEquals ( "Квадрат" , shape. description ()) ;

}

}

@Nested

@ DisplayName ( «Не следует» )

**class** ShouldNot {

@Тест

@ DisplayName ( «быть равным другой фигуре с таким же количеством сторон» )

**void** beEqualToAnotherShapeWithTheSameNumberOfSides () {

assertNotEquals ( **новая** форма ( 4 ) , форма ) ;

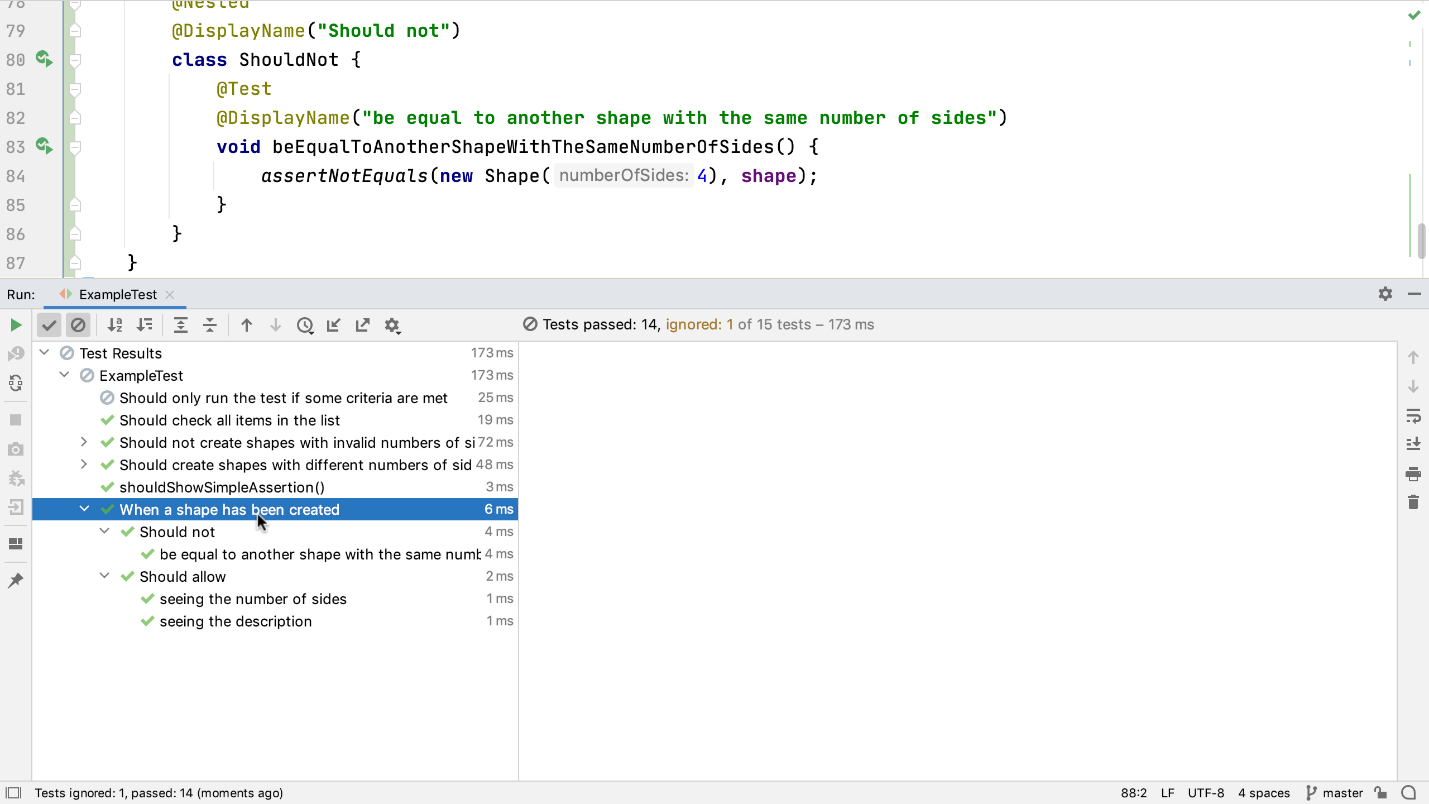
}

}

}

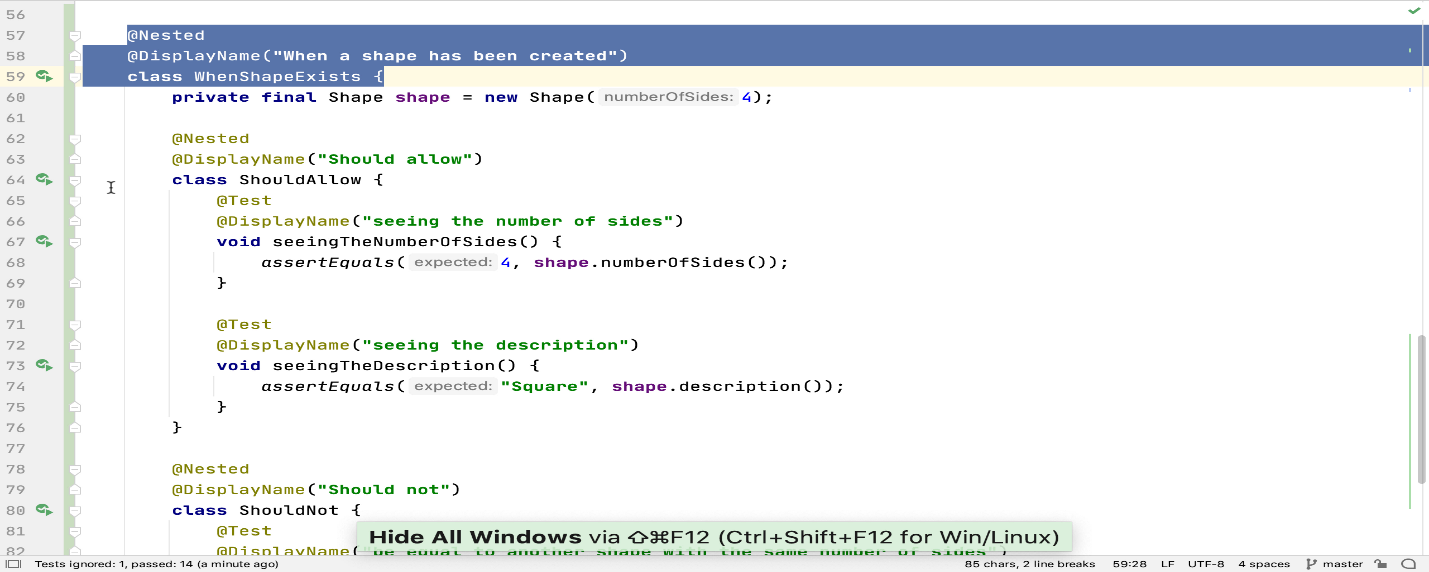
}

Если мы запустим все тесты в классе ( **⌃R** или **Shift + F10** ), мы сможем увидеть наши вложенные тесты в результатах тестов. Мы видим, что группировка означает, что результаты аналогичных тестов сгруппированы вместе. Мы также можем увидеть, как отображаемое имя может помочь нам понять группировку тестов.



Совет IDE: сворачивание кода

Если все эти аннотации добавляют слишком много шума к редактору, мы всегда можем [свернуть их](https://www.jetbrains.com/help/idea/working-with-source-code.html#code_folding) , нажав на минус в желобе или используя сочетание клавиш для свертывания кода, **⌘.**или **Ctrl +.**- куда "." - это точка или точка на клавиатуре. Мы можем навести указатель мыши на свернутые аннотации, чтобы увидеть их.



Вывод

В этом учебном пособии только поверхностно рассмотрены функции, предлагаемые JUnit 5. Чтобы узнать больше, перейдите к [документации JUnit 5](https://junit.org/junit5/docs/current/user-guide/) , она охватывает огромное количество тем, в том числе более подробно демонстрирует функции, которые мы видели в этом видео. Он также описывает шаги, которые необходимо предпринять для перехода на JUnit 5 с JUnit 4, что также было описано в [блоге и видео](https://blog.jetbrains.com/idea/2020/08/migrating-from-junit-4-to-junit-5/) .

Лучшие ярлыки

В этом сообщении блога есть некоторые ярлыки, но в видео было продемонстрировано гораздо больше, и не все из них были упомянуты здесь:

* **⌘ [** или **Ctrl + Alt + стрелка влево** Вернуться назад - позволяет легко перемещаться между всеми местами, где вы были при написании кода и выполнении тестов.
* **⌘⇧T** или **Ctrl** + **Shift** + **T** Перемещение между тестом и испытуемым. Это рассматривается в [5 лучших советах по навигации](https://blog.jetbrains.com/idea/2020/09/top-5-navigation-keyboard-shortcuts-in-intellij-idea-shortcuts/) (блог и видео).
* **⌃⇧J** или **Ctrl + Shift + J Соединить** линии для создания кода компиляции.
* **⇧⌘⏎** или **Ctrl + Shift + Enter**[Завершить](https://www.jetbrains.com/help/idea/working-with-source-code.html#56aee) оператор закрывают скобки и оператор, а также форматирует код.
* **⌃⇧Пробел** или **Ctrl + Shift + Пробел**[Умное завершение](https://www.jetbrains.com/help/idea/auto-completing-code.html?keymap=secondary_macos#smart_completion)

Смотрите также:

* [Переход с JUnit 4 на JUnit 5](https://blog.jetbrains.com/idea/2020/08/migrating-from-junit-4-to-junit-5/) - блог со встроенным видео
* [Модульное тестирование и покрытие в IntelliJ IDEA](https://youtu.be/QDFI19lj4OM) (видео)
* [Советы и приемы редактора](https://blog.jetbrains.com/idea/2020/08/editor-tips-and-tricks-in-intellij-idea/) (блог и видео)
* [Блог лучших 15 ярлыков IntelliJ IDEA](https://blog.jetbrains.com/idea/2020/03/top-15-intellij-idea-shortcuts/) (блог и видео