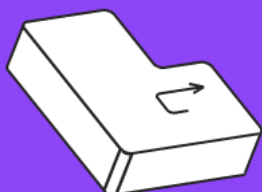




Введение в юнит-тестирование

Знакомство с тестовыми
фреймворками



Оглавление

Введение	3
Термины, используемые в лекции	3
Что такое юнит-тесты	3
Качество кода тестов и кто за этим следит	5
Интеграционное тестирование	7
Тестовые Фреймворки	8
JUnit	9
Семейство xUnit	10
Фреймворки для тестирования Java	11
JUnit 5	15
JUnit 5 Работа с утверждениями	16
Установка JUnit 5	17
Тестовый и рабочий код	20
@Test	21
Допишем класс Calculator	22
Тестируем новые функции	24
Аннотации	27
Подведем итоги	
Используемая литература	29

Введение

На прошлой лекции мы рассмотрели тестирование, про его цель, разные виды ошибок, 7 принципов тестирования и цикл разработки. На практике рассмотрели пограничные случаи. И начали погружаться в практическую часть тестирования: попробовали написать примитивные тесты, узнали об утверждениях Assert и о библиотеке AssertJ.

В сегодняшней лекции:

- Юнит-тесты
- Интеграционное тестирование
- Тестовые Фреймворки
- JUnit
- Подключение JUnit 5 в IDEA
- Создание тестов
- Аннотации JUnit

Термины, используемые в лекции

Моки — объекты-имитаторы, которые реализуют поведение реальной подсистемы. Моки используются как замена зависимостей.

API (аббр. от англ. Application Programming Interface) — описание способов взаимодействия одной компьютерной программы с другими.

Обратная совместимость — наличие в новой версии компьютерной программы или компьютерного оборудования интерфейса, присутствующего в старой версии, в результате чего другие программы (или человек) могут продолжать работать с новой версией без значительной переделки.

Что такое юнит-тесты

Существует много определений юнит-теста. Все эти определения сводятся к трём важным атрибутам, перечисленным ниже:

1. Проверяет правильность работы небольшого фрагмента кода, также называемого юнитом (модулем). Нет единого мнения по поводу того, что можно считать неделимым элементом, то есть единицей кода для

проверки. Это не всегда будет один метод `calculation()`, как у нас в примере с калькулятором.

```
public class CalculationTest {
    public static void main(String[] args) {
        assertEquals(9, Calculator.calculation(2, 6, '+'));
        assertEquals(0, Calculator.calculation(2, 2, '-'));
        assertEquals(14, Calculator.calculation(2, 7, '*'));
        assertEquals(2, Calculator.calculation(100, 90, '/'));
        assertEquals(
            Calculator.calculation(8, 4, '_'),
            new IllegalStateException());
    }
}
```

Это может быть целый класс или несколько классов, всё зависит от сложности и специфики кода проекта.



При выборе объекта для тестирования старайтесь думать не о коде, а о предметной области, то есть выделить не единицу кода, а единицу поведения. Количество классов, необходимых для реализации такой единицы поведения, не имеет значения.



В идеале тест должен рассказывать о проблеме, решаемой кодом проекта, этот рассказ должен быть связным и понятным даже для непрограммиста.

Пример связного рассказа:

Когда я зову свою собаку, она идёт ко мне.

Теперь сравните со следующим рассказом:

Когда я зову свою собаку, она сначала выставляет вперёд левую переднюю лапу, потом правую переднюю лапу, поворачивает голову, начинает вилять хвостом...

Второй рассказ не кажется особо вразумительным. Так начинают выглядеть ваши тесты, когда вы ориентируетесь на отдельные классы (лапы, голова, хвост) вместо фактического поведения (собака идёт к хозяину).

2. Юнит-тест должен быстро тестировать. Тестирование в идеальном мире проводится каждый раз при любом изменении кода проекта, это не должно занимать много времени.

Существуют разногласия относительно того, что именно можно считать быстрым юнит-тестом, так как это довольно субъективная метрика. Но в целом это не так важно — если вас устраивает скорость работы ваших тестов, это означает, что они достаточно быстры.

3. Изоляция от другого кода. Зависимости в тестовом коде заменяются на тестовые заглушки — моки. Это позволит вам сосредоточиться исключительно на тестируемом классе, изолировав его поведение от внешнего влияния.

Владимир Хориков в «Принципы юнит-тестирования» говорит о двух разных подходах к пониманию и реализации изоляции кода. Лондонская школа, более строгая, и Классическая школа. Их различия указаны в таблице:

	Изоляция	Что считается юнитом (единицей кода)	Что именно изолируется
Лондонская школа	Юнитов	Класс	Любые другие классы
Классическая школа	Юнит-тестов	Класс или набор классов	Уже связанные с другими классами классы (совместные зависимости)

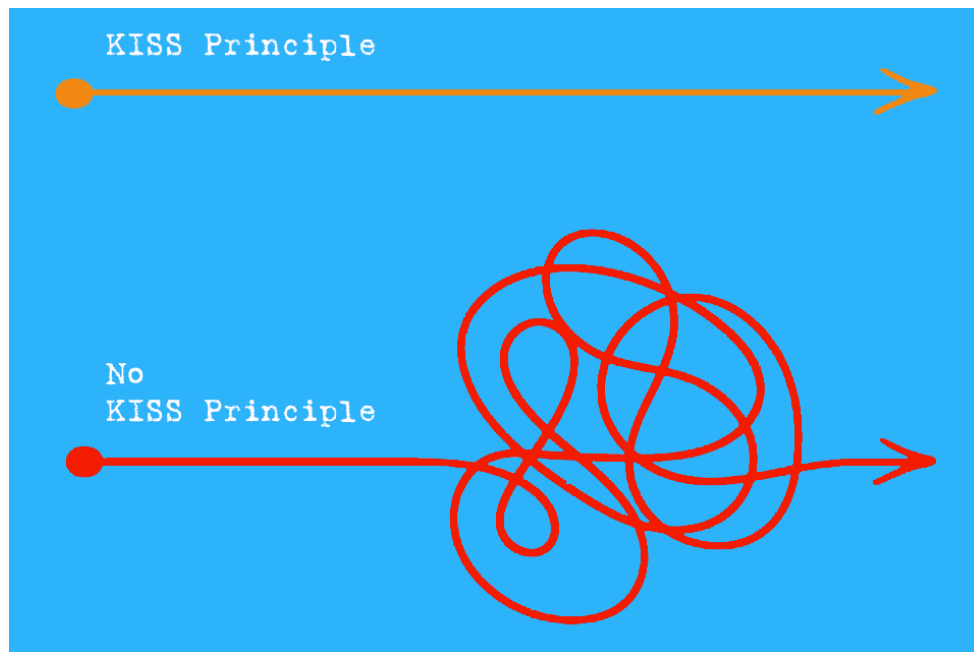
Качество кода тестов и кто за этим следит

Качество кода тестов должно не уступать основному коду, чтобы тратить меньше времени на поддержку и рефакторинг, как минимум. Причина этому одна — это его размер. На 1 строку рабочего кода в среднем приходится 2-3 строки тестового кода, то есть его в 2-3 раза больше, чем основного кода, но это не значит, что если в вашем случае тестового кода меньше, чем тестируемого, то вы что-то делаете неправильно.

В этих условиях код тестов должен хорошо читаться, быть структурированным, иметь хорошую типизацию и быть очень дружелюбным к инструментам автоматического рефакторинга для интеграции в процесс разработки.

Для поддержания качества тестового кода, так же как и основного, используются принципы KISS, DRY:

- **KISS (Keep It Simple Stupid)** — не усложняй! Смысл этого принципа программирования заключается в том, что стоит делать максимально простую и понятную архитектуру, применять шаблоны проектирования и не изобретать велосипед.



Как это относится к тестированию?

Чем меньше кода в тесте, тем проще он читается. Также небольшие тесты проще изменить при необходимости. За качеством кода тестов, как и за качеством рабочего кода, следит разработчик, но они должны быть написаны максимально читаемо любому непрограммисту в команде, ведь не всегда за тестами следит тот же человек, что и написал их. Вспомните fluent API, его цель сделать код читаемым. С первого взгляда можно сказать, что проверяет тест, нужно только перевести предложение.

```
assertThat(fellowshipOfTheRing).hasSize(9)
                                .contains(frodo, sam)
                                .doesNotContain(sauron);
```

- **DRY (Don't Repeat Yourself)** — не повторяйся, также известен как DIE — Duplication Is Evil. Этот принцип заключается в том, что нужно избегать повторений одного и того же кода. Лучше использовать универсальные свойства и функции.

I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code

Интеграционное тестирование

Юниты-модули могут быть сколь угодно идеальными, но в «вакууме». Важнее, чтобы юниты взаимодействовали между собой, как задумывалось, во внешнем окружении, реальной среде. Другими словами, интеграционное тестирование проверяет насколько хорошо тот или иной юнит способен взаимодействовать с внешним миром не ломая его, позволяет понять действительно ли API юнитов подходят друг другу (ближайшему окружению). Интеграционные тесты выполняются QA-отделом (или QA-командой), который выполняет тест-кейсы, проверяя производительность и функциональность приложения.

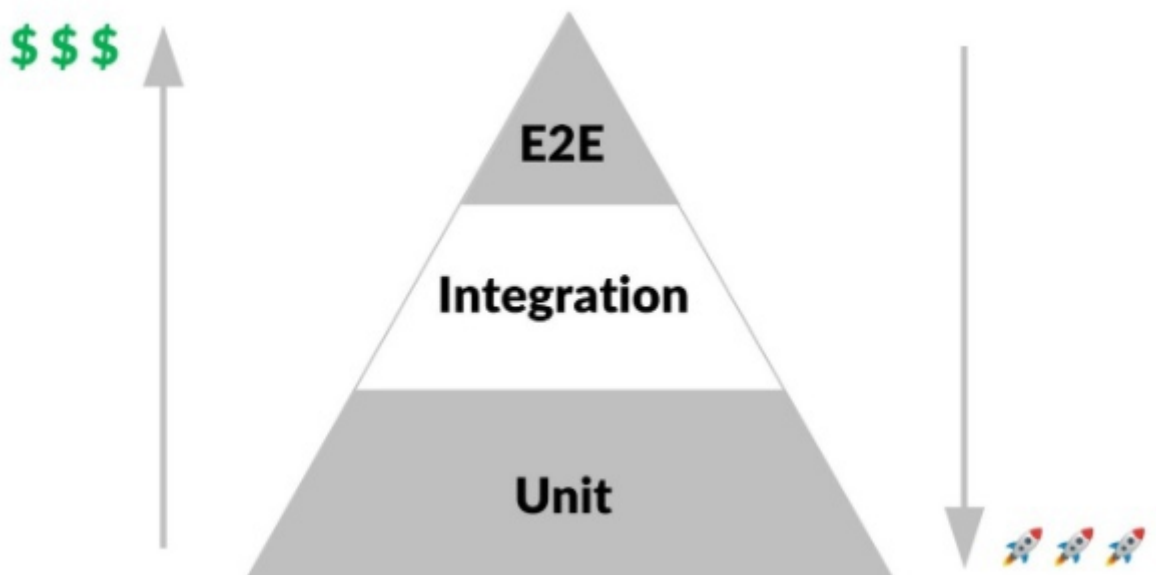


Границы между unit и интеграционным тестированием всегда размыты, но основное практическое правило таково:

Если тест:

- использует базу данных;
- использует сеть для вызова другого приложения;
- использует внешнюю систему (например, почтовый сервер);
- выполняет операции ввода-вывода;
- полагается не на исходный код, а на бинарник приложения.

То это интеграционный, а не модульный тест.



Пирамида автоматизации Майка Кона отлично иллюстрирует эффективный подход. Ширина каждого уровня пирамиды показывает, сколько тестов должно быть на каждом уровне по сравнению с другими. Чем выше уровень, тем меньше вам нужно тестов. Уровни в пирамиде автоматизации должны быть построены так, чтобы каждый уровень содержал тесты, которые могут быть выполнены без запуска другого уровня. Также видно, что юнит-тесты самые быстрые по скорости исполнения и самые дешёвые для бизнеса.

Но это не аксиома, приложениям требуется разное сочетание тестов, в зависимости от сложности и специфики.

Тестовые Фреймворки

Фреймворки для модульного тестирования предоставляют предварительно написанный код (шаблон), в который нам нужно только подставить свои значения. Фреймворки часто включают библиотеки и их цель максимально упростить жизнь программиста.

Основные преимущества фреймворков:

1. **Простой процесс диагностики и отладки.** Помимо сокращения времени работы, фреймворки также упрощают отладку и обслуживание программного обеспечения. Процесс отладки подразумевает пошаговое выполнение кода в поисках точки, в которой была допущена ошибка при написании программы. Некоторые имеют собственную внутреннюю систему тестирования кода, что позволяет программистам выполнять

модульные тесты одновременно. Такой процесс дает возможность больше времени посвятить тестированию, а не исправлению ошибок.

2. **Повышенная эффективность кода.** Фреймворки также способствуют повторному использованию кода, что, в свою очередь, повышает его эффективность. Чтобы не писать сложные структуры, содержащие сотни строк кода с нуля, можно обратиться к базе платформы. Надежный код в данном случае - это автогенерируемый код, мы стараемся избавиться от человеческого фактора. Используя такой метод разработчик получает код, в котором легко внести изменения и применить дополнительные функции. Также можно создать собственный код, чтобы использовать его в последующих проектах, помните про принцип DRY
3. **Ускоренная разработка.** Фреймворки содержат базовые программные модули, библиотеки, удобный интерфейс, гибкую среду кодирования и другие функции, которые упрощают работу. Разработчикам не нужно заботиться об обезличивании данных, управлении сессиями, обработке ошибок, аутентификации и т.д. Платформа отлично справляется с большинством из этих функций. Это позволяет программисту сразу начать писать код, не отвлекаясь на другие задачи

Вот некоторые широко используемые фреймворки, с которыми мы так или иначе столкнёмся на курсе:

- Mockito
- JBehave
- Spock
- TestNG
- JUnit

JUnit

JUnit является одним из наиболее широко распространённых фреймворков модульного тестирования с открытым исходным кодом. Он используется довольно давно и до сих пор популярен. На его основе создан TestNG, Spock.



JUnit, Созданный Кентом Бекон и Эриком Гаммой, JUnit принадлежит семье фреймворков xUnit, это собирательное название фреймворков с общей архитектурой, названия фреймворков этого семейства образованы аналогично «SUnit», обычно заменяется буква «S» на первую букву (или несколько первых) в названии предполагаемого языка («JUnit» для Java, «NUnit» для программной платформы .NET и т. д.).

JUnit породил экосистему расширений — jMock, EasyMock, DbUnit, HttpUnit и т. д.

JUnit переведён на другие языки, включая PHP (PHPUnit), C# (NUnit), Python (PyUnit), Fortran (fUnit), Delphi (DUnit), Free Pascal (FPCUnit), Perl (Test::Unit), C++ (CppUnit), Flex (FlexUnit), JavaScript (JSUnit), COS (COSUnit).



По словам Мартина Фаулера, одного из первых пользователей JUnit:

JUnit родился во время перелёта из Цюриха в 1997 году на ежегодную научно-исследовательскую конференцию OOPSLA в Атланте. Кент летел с Эриком Гаммой, а что ещё оставалось делать двум гикам в долгом полёте, кроме как программировать? Первая версия JUnit была собрана там, прямо во время полёта.

Семейство xUnit

Все фреймворки из семейства xUnit имеют следующие базовые компоненты архитектуры, которые в различных реализациях могут слегка варьироваться.

Общие архитектурные принципы семейства xUnit:

1. **Test runner** (Средство выполнения тестов) — это исполняемая программа, которая запускает тесты, реализованные с использованием фреймворка xUnit, и сообщает о результатах тестирования.
2. **Test case** (Тестовый пример) — это класс, от которого мы хотим наследоваться, если пишем свой тест.
3. **Test fixtures** (Тестовые инструменты) — это набор предварительных условий или состояний, необходимых для запуска теста.
4. **Test suites** (Наборы тестов) — позволяет упаковать много тестов, вроде контейнера, и можно запускать сразу несколько (набор) тестов.
5. **Test execution** (Выполнение теста) — это выполнение отдельного теста, когда сначала подготавливается контекст теста, потом тело теста и

завершающая часть.

6. **Test result formatter** — модуль форматирования результатов теста и их анализ.
7. **Assertions** (Ассерты) — уже знакомые нам по принципу проверки.

Фреймворки для тестирования Java

Ниже представлен примерный список библиотек и фреймворков для тестирования, с указанием принадлежности к семейству xUnit

Название	Семейство xUnit?	Описание
Agitar	Yes	Автоматизирует создание детальных тестов JUnit в коде.
Artos	Yes	Фреймворк с открытым исходным кодом для написания модульных, интеграционных и функциональных тестов. Он включает в себя предварительно настроенную структуру ведения журнала и отчёты по экстендам, утилиты для написания потока для ручного / полуавтоматического тестирования. Он поддерживает тестирование с использованием сценариев <i>Cucumber</i> .
Arquillian	Yes	Фреймворк с открытым исходным кодом для написания интеграционных и функциональных тестов.
AssertJ		Fluent-утверждения для java.
beanSpec		Behavior-driven development.
BeanTest	No	Небольшая среда веб-тестирования Java, созданная для использования WebDriver/HTMLUnit в сценариях BeanShell.
Cactus		Расширение JUnit для тестирования Java EE и веб-приложений. Тесты Cactus выполняются внутри контейнера Java EE/web.
Concordion		Разработка на основе приёмочных испытаний, Разработка на основе поведения, Спецификация на

		примере.
Concutest		Фреймворк для тестирования параллельных программ.
Cucumber-JVM		Разработка, основанная на поведении, заменяет устаревший Cuke4Duke на основе JRuby.
Cuppa		Behavior-driven development-фреймворк для Java 8.
DbUnit		Расширение JUnit для выполнения модульного тестирования программ, управляемых базой данных.
EasyMock		Фреймворк для работы с зависимостями.
EtlUnit	Yes	Платформа модульного тестирования для процессов извлечения – преобразования – загрузки, написанная на Java. Возможность тестирования Oracle, Informatica, SqlServer, PostGreSQL, MySQL и т. д.
EvoSuite		Инструмент генерации тестовых случаев, который может автоматически генерировать тесты JUnit.
GrandTestAuto		GrandTestAuto (GTA) — это платформа для полного автоматизированного тестирования программного обеспечения Java. Тесты могут быть распределены по нескольким машинам в сети.
GroboUtils		Расширение JUnit, обеспечивающее автоматическую документацию, модульное тестирование иерархии классов, покрытие кода и многопоточные тесты
Hamcrest		Создание настраиваемых средств сопоставления утверждений, которые можно использовать вместе со средами модульного тестирования
HavaRunner	Yes	Средство запуска JUnit со встроенной поддержкой параллелизма, комплектами и сценариями.
Instinct		Behavior-driven development
Java Server-Side Testing framework (JSST)		Фреймворк Java Server-Side Testing, основанный на той же идее, что и Apache CACTUS, но, в отличие от CACTUS, он не связан с JUnit 3.x и может использоваться в сочетании с любым фреймворком тестирования.
JBehave		Behavior-driven development

JDave		Behavior-driven development
JExample	Yes	Расширение JUnit, использующее зависимости между тестовыми примерами для уменьшения дублирования кода и улучшения локализации дефектов.
JGiven		Behavior-driven development
JMock		Макет фреймворка
JMockit		Фреймворк с открытым исходным кодом. Можно легко написать тесты, которые будут имитировать окончательные классы, статические методы, конструкторы и т. д. Ограничений нет.
Jnario	Yes	Разработка, основанная на поведении, как Cucumber
jqwik		Тестовый движок JUnit 5 для тестирования на основе свойств
Jtest	Yes	Коммерческий. Автоматическая генерация и выполнение модульных/компонентных тестов с охватом кода и обнаружением ошибок во время выполнения. Также обеспечивает статический анализ и экспертную оценку кода.
Jukito		Объединяет Mockito и Guice, чтобы обеспечить автоматическую фикцию и создание экземпляров зависимостей.
JUnit	Yes	
JUnitEE		Расширение JUnit для тестирования приложений Java EE.
JWalk		Быстрое полуавтоматическое создание исчерпывающих наборов модульных тестов.
Mockito		Мок-фреймворк с использованием библиотеки.
Mockrunner		Расширение JUnit для тестирования сервлетов, фильтров, классов тегов, действий и форм Struts.
Needle		Платформа с открытым исходным кодом для изолированного тестирования компонентов Java EE вне контейнера.
NUTester		Платформа тестирования, разработанная в Северо-восточном университете для помощи в преподавании вводных курсов по информатике на

		языке Java.
OpenPojo		Платформа с открытым исходным кодом, используемая для проверки и обеспечения поведения POJO, а также для управления идентификацией — equals, hashCode и toString.
PowerMock		Расширение как для Mockito, так и для EasyMock, позволяющее имитировать статические методы, конструкторы, окончательные классы и методы, частные методы, удалять статические инициализаторы и многое другое.
Randoop	Yes	Автоматически находит ошибки и генерирует модульные тесты для Java посредством случайного тестирования с обратной связью (вариант фаззинга).
Spock		Spock — это среда тестирования и спецификации для приложений Java и Groovy. Spock поддерживает спецификацию на примерах и тестирование в стиле BDD.
SpryTest	Yes	Коммерческий. Платформа автоматизированного модульного тестирования для Java.
SureAssert		Интегрированное решение для модульного тестирования Java для Eclipse. Contract-First Design и разработка через тестирование.
Tacinga		Использует чистый подход объектно-ориентированного программирования и предлагает коммерческую лицензию и бесплатную поддержку.
TestNG	Yes	Тесты могут включать модульные тесты, функциональные тесты и интеграционные тесты. Имеет средства для создания даже нефункциональных тестов (таких как нагрузочные тесты, временные тесты).
Unitils		Предлагает общие утилиты и функции для помощи в тестировании уровня сохраняемости и тестировании с фиктивными объектами. Предлагает специальную поддержку для тестирования кода приложения, использующего JPA, hibernate и spring. Unitils интегрируется со средами тестирования JUnit и

		TestNG.
XMLUnit		JUnit и NUnit тестирование для XML.

Источник: [List of unit testing frameworks — Wikipedia](#)

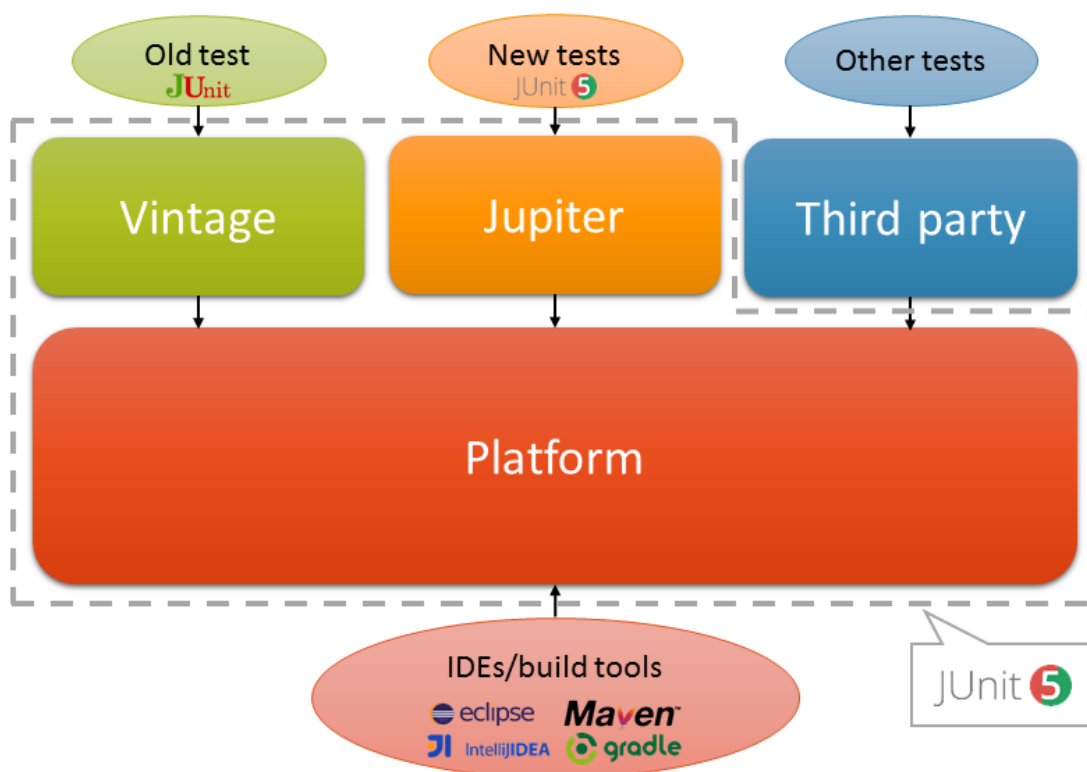
JUnit 5

JUnit 5 вышла в далёком 2016 году. Но из-за недостатка документации и настроенной интеграции с основными решениями, используемыми в разработке на Java, довольно популярной остаётся версия JUnit 4. На курсе мы рассмотрим последнюю (5) версию, так как новые проекты уже пишутся в основном на ней.

JUnit 5 можно разделить на 3 различных проекта (в отличие от 4 версии, которая была монолитной):

1. **JUnit Platform**
2. **Jupiter**
3. **Vintage**

Рассмотрим подробнее архитектуру фреймворка:



JUnit 5 начинается с платформы (**Platform**), разработчик не взаимодействует напрямую с ядром фреймворка, для этого предоставлено API (**Jupiter**).

Vintage — ещё один API, но для старых версий JUnit, он обеспечивает обратную совместимость со старыми версиями, т. е. можно безболезненно перейти со старой версии и не обязательно будет переписывать старые тесты, они могут работать все одновременно в одном проекте.

Разработчики предоставили возможность написать своё API (**Third party**), которое будет обращаться к тестовому движку.

Все популярные IDE поддерживают JUnit (**IDEs**), могут напрямую подключаются к движку JUnit, для запуска тестов. Мы позже увидим, что можно запускать тесты и без точки входа в Main.

Build tools (Maven, gradle) — это системы сборки проектов, которые берут на себя настройку зависимостей, скачивание библиотек, автоматизацию процессов тестирования. Они подключаются JUnit по такому же принципу как и IDE.

JUnit 5 Работа с утверждениями

Фреймворк имеет встроенный инструмент работы с утверждениями.

Механизм, основанный на Assert из JUnit, на самом деле, несколько устарел и не поддерживает некоторые новые функции языка, а также менее объектно-ориентирован, но всё ещё широко используется, так как идёт вместе с фреймворком JUnit по умолчанию.

- junit.framework.Assert
 - assertEquals
 - assertFalse
 - assertNotNull
 - assertNull
 - assertNotSame
 - assertEquals
 - assertTrue

Синтаксис тестов на JUnit Assert похож на AssertJ:

```
public class MathTest {
    @Test
    public void testEquals() {
        Assert.assertEquals(4, 2 + 2);
        Assert.assertTrue(4 == 2 + 2);
    }
}
```



```
}

@Test
public void testNotEquals() {
    Assert.assertFalse(5 == 2 + 2);
}
}
```

В выборе инструментов написания кода JUnit нас не ограничивает, многие используют **AssertJ**. мы уже с ним знакомы. Преимущество библиотеки в «естественном» синтаксисе — Fluent API и поддержке современных функций языка, к тому же его просто освоить.

Утверждение, написанное с помощью AssertJ, выглядит следующим образом:

```
assertThat(notificationText).contains("testuser@google.com");
```

Как альтернатива JUnit Assert достаточно популярен фреймворк **Hamcrest** — это известный фреймворк, используемый для модульного тестирования в экосистеме Java. Он может быть встроен в JUnit и, использует существующие предикаты, называемые классами сопоставления, для создания утверждений.

Утверждение, написанное с помощью Hamcrest, выглядит так:

```
assertThat(notificationText,
containsString("testuser@google.com"));
```

Установка JUnit 5

Перейдём к практике. Чтобы подключить зависимости JUnit, скачаем jar архивы библиотеки:

1. [junit-jupiter-api » 5.9.1](#)
2. [junit-platform-commons » 1.9.1](#)



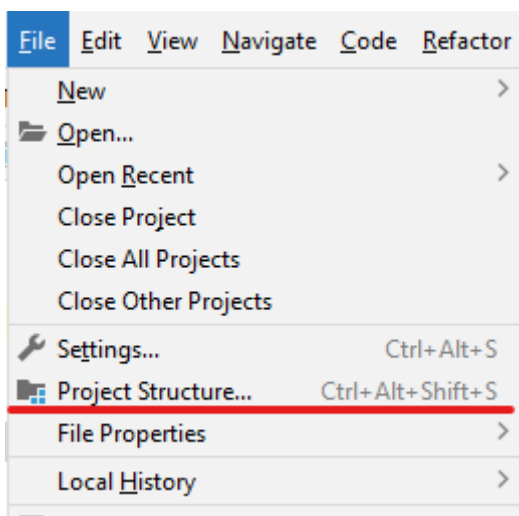
JUnit Jupiter Params » 5.9.1

JUnit Jupiter extension for running parameterized tests.

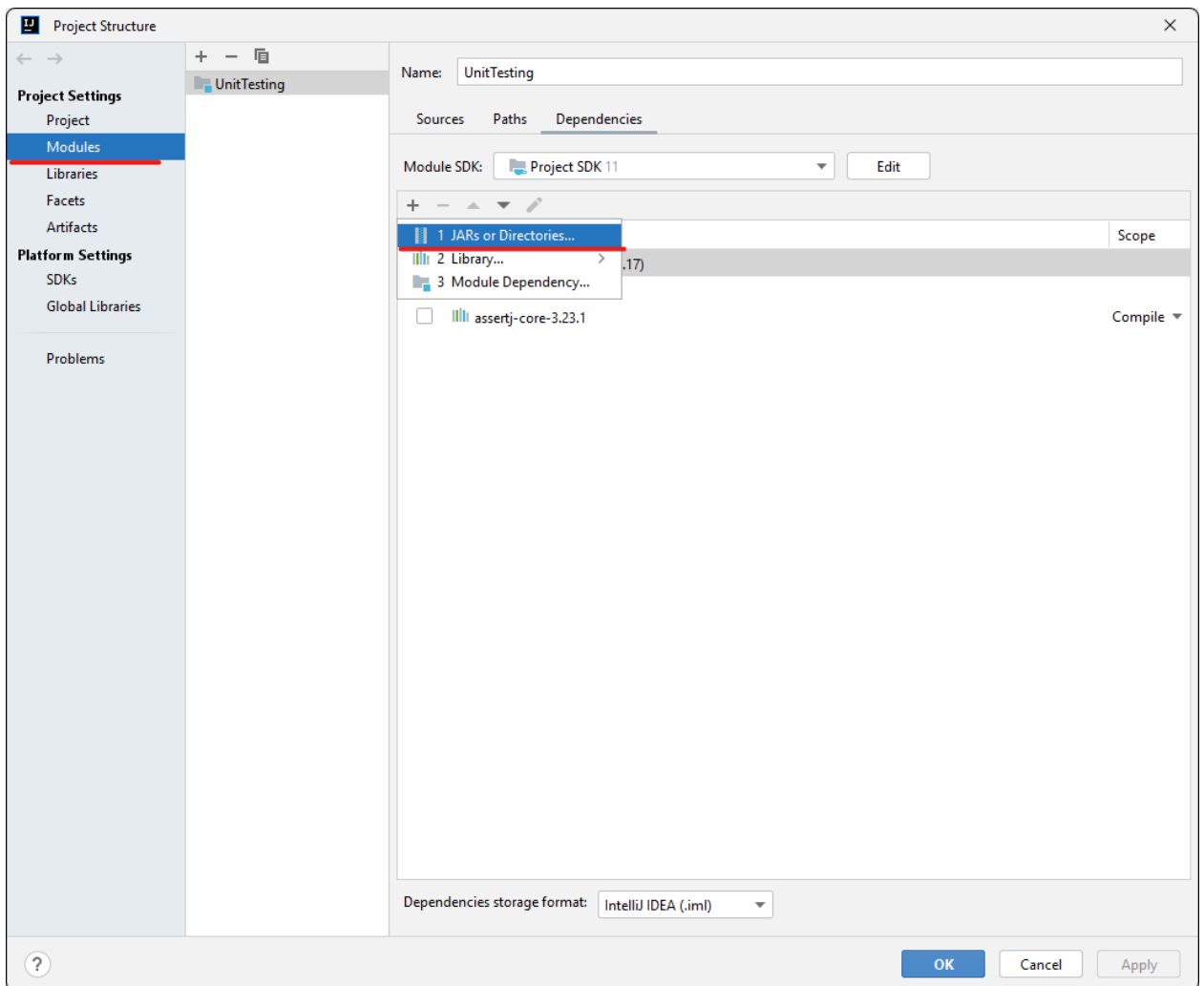
License	EPL 2.0
Categories	Testing Frameworks
Tags	junit testing
HomePage	https://junit.org/junit5/
Date	Sep 20, 2022
Files	pom (2 KB) jar (565 KB) View All
Repositories	Central
Ranking	#101 in MvnRepository (See Top Artifacts) #13 in Testing Frameworks
Used By	4,354 artifacts

И подключим так же, как на прошлом уроке:

1. Переходим в меню File → Project Structure

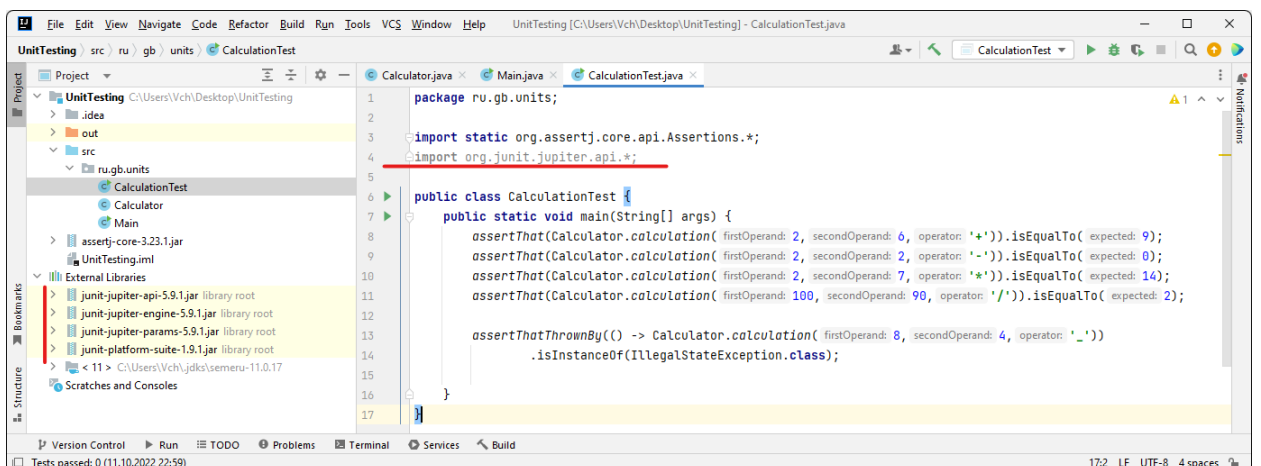


2. Добавляем в меню Modules скачанные JAR-архивы





3. Проверяем в CalculationTest — импортируем пакет.

```
import org.junit.jupiter.api.*;
```



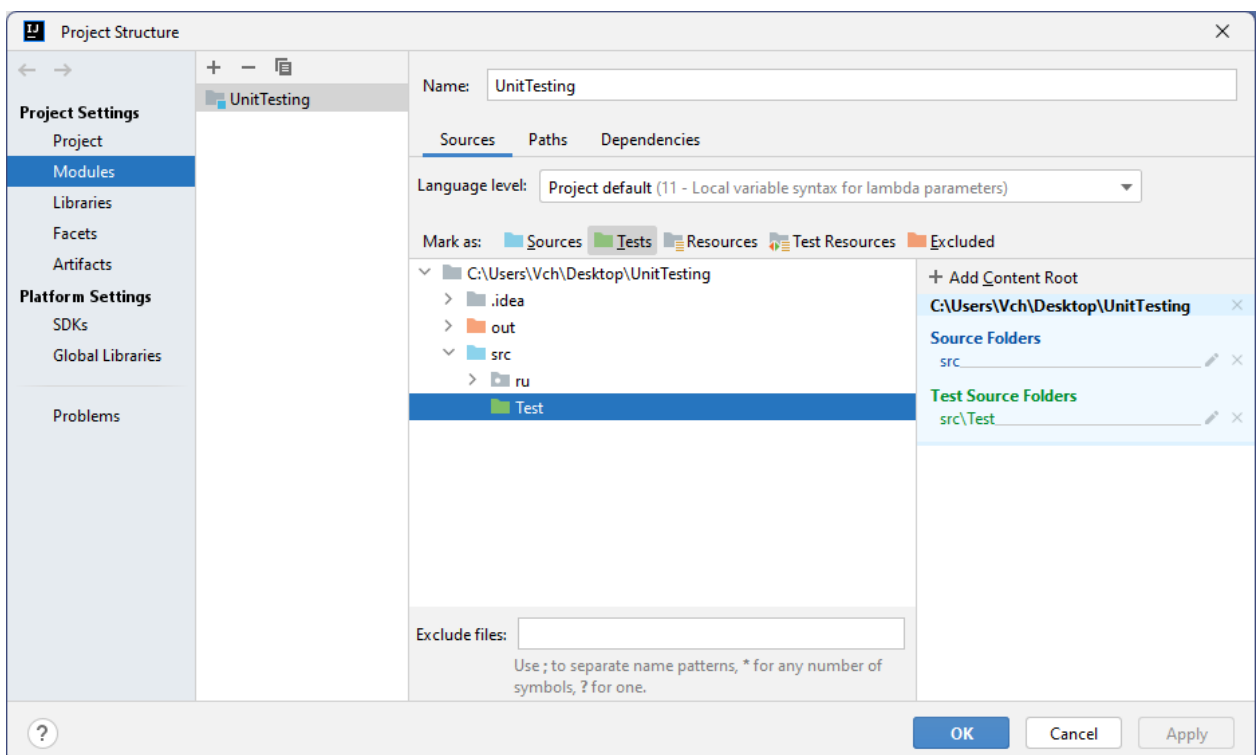
Тестовый и рабочий код

Прежде чем приступить к написанию тестов, нужно настроить корневой каталог для тестов. Корневой каталог Test Sources — это папка, в которой хранится ваш тестовый код. В окне Project tool эта папка отмечена значком . В этих папках код, относящийся к тестированию, хранится отдельно от рабочего кода.

Среда IDE обрабатывает код из разных источников по-разному. Например, результаты компиляции для исходных текстов и тестовых источников обычно помещаются в разные папки. Вот почему, если корень тестовых источников отсутствует, нам необходимо его создать. В противном случае наш код может быть обработан неправильно. В папке с типом  Test Resources можно хранить файлы ресурсов, связанных с вашими тестами.

Настроить тестовую папку можно следующим образом:

1. Создадим директорию Test.
2. Разметим её, как каталог для тестов, для этого перейдём в настройки проекта (IDEA: File → Project Structure — вкладка Sources) и отметим папку Test как Tests, папка должна стать зелёной.



3. Перенесём старый тестовый класс CalculationTest в новую Test Sources.

Теперь перепишем тестовый класс, с использованием JUnit 5:

```

import org.junit.jupiter.api.Test;
import ru.gb.units.Calculator;
import static org.assertj.core.api.Assertions.assertThat;
import static
org.assertj.core.api.Assertions.assertThatThrownBy;
public class CalculationTest {
    @Test
    void additionExpressionEvaluation() {
        Calculator calculator = new Calculator();
        assertThat(calculator.calculation(2, 6, '+')).isEqualTo(8);
    }
    @Test
    void subtractionExpressionEvaluation () {
        Calculator calculator = new Calculator();

        assertThat(calculator.calculation(2, 2, '-')).isEqualTo(0);
    }
    @Test
    void multiplicationExpressionEvaluation () {
        Calculator calculator = new Calculator();

        assertThat(calculator.calculation(2, 7, '*')).isEqualTo(14);
    }
    @Test
    void divisionExpressionEvaluation () {
        Calculator calculator = new Calculator();

        assertThat(calculator.calculation(100, 90, '/')).isEqualTo(2);
    }
    @Test
    void expectedIllegalStateExceptionOnInvalidOperatorSymbol ()
    {
        Calculator calculator = new Calculator();
        assertThatThrownBy(() ->
calculator.calculation(8, 4, '_'))
                .assertInstanceOf(IllegalStateException.class);
    }
}

```

@Test

Аннотация @Test в JUnit 5 используется для указания того, что аннотированный метод является методом тестирования. Находится аннотация в пакете junit.jupiter.api. Вот ее исходный код:

```

package org.junit.jupiter.api;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.apiguardian.api.API;
import org.apiguardian.api.API.Status;
import org.junit.platform.commons.annotation.Testable;
@Target({ElementType.ANNOTATION_TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@API(
    status = Status.STABLE,
    since = "5.0"
)
@Testable
public @interface Test {
}

```

Допишем класс Calculator

Добавим в класс функционал работы в консоли.

В main вызов методов getOperator() и getOperand(), с их помощью мы будем запрашивать и получать от пользователя значения для вычисления.

```

package ru.geekbrains;
import java.util.Scanner;
public class Calculator {
    private static final Scanner scanner = new
Scanner(System.in);
    public static void main(String[] args) {
        int firstOperand = getOperand();
        int secondOperand = getOperand();
        char operator = getOperator();
        int result = calculation(firstOperand, secondOperand,
operator);
        System.out.println("Результат операции: " + result);
    }
    public static int calculation(int firstOperand, int
secondOperand, char operator) {
        int result;

```

```

        switch (operator) {
            case '+':
                result = firstOperand + secondOperand;
                break;
            case '-':
                result = firstOperand - secondOperand;
                break;
            case '*':
                result = firstOperand * secondOperand;
                break;
            case '/':
                if (secondOperand != 0) {
                    result = firstOperand / secondOperand;
                    break;
                } else {
                    throw new ArithmeticException("Division by
zero");
                }
            default:
                throw new IllegalStateException("Unexpected
Operator");
        }
        return result;
    }
    public static char getOperator() {
        System.out.println("Введите операцию:");
        char operation = scanner.next().charAt(0);
        return operation;
    }
    public static int getOperand() {
        System.out.println("Введите число:");
        int operand = scanner.nextInt();
        return operand;
    }
}

```

Пример работы программы:

```

Введите число:
3
Введите число:
9
Введите операцию:
+
Результат операции: 12

```

Тестируем новые функции

В результате ручного тестирования была выявлена ошибка:

IllegalStateException: Unexpected Operator

В случае, когда в метод **getOperand** передаётся не цифра. Исправим метод в классе **Calculator**:

```
package ru.gb.units;

import java.util.Scanner;

public class Calculator {
    private static final Scanner scanner = new
Scanner(System.in);

    public static void main(String[] args) {
        int firstOperand = getOperand();
        int secondOperand = getOperand();
        char operator = getOperator();
        int result = calculation(firstOperand, secondOperand,
operator);
        System.out.println("Результат операции: " + result);
    }

    public static int calculation(int firstOperand, int
secondOperand, char operator) {
        int result;

        switch (operator) {
            case '+':
                result = firstOperand + secondOperand;
                break;
            case '-':
                result = firstOperand - secondOperand;
                break;
            case '*':
                result = firstOperand * secondOperand;
                break;
            case '/':
                if (secondOperand != 0) {
                    result = firstOperand / secondOperand;
                    break;
                } else {
                    throw new ArithmeticException("Division by
zero");
                }
            }
        }
    }
}
```



```

        }
        default:
            throw new IllegalStateException("Unexpected
Operator");
    }
    return result;
}

public static char getOperator() {
    System.out.println("Введите операцию:");
    char operation = scanner.next().charAt(0);
    return operation;
}

public static int getOperand() {
    System.out.println("Введите число:");
    int operand;
    if (scanner.hasNextInt()) {
        operand = scanner.nextInt();
    } else {
        System.out.println("Вы допустили ошибку при вводе числа.
Попробуйте еще раз");
        if (scanner.hasNext()) {
            scanner.next();
            operand = getOperand();
        } else {
            throw new IllegalStateException("Ошибка в вводимых
данных");
        }
    }
    return operand;
}
}

```

И напишем соответствующие тесты:

- Для случая, когда в метод передаётся корректное значение:
getOperandCompletesCorrectlyWithNumbers
В этом случае мы ожидаем завершение без ошибок.
- Для случая, когда в метод передается некорректное значение:
getOperandCompletesCorrectlyWithNotNumbers
В этом случае мы хотим увидеть ожидаемое исключение.

Метод **getOperand** работает со сканером, и должен корректно принимать цифры. Для теста мы должны имитировать ввод пользователем значений. Это делается с помощью `System.setIn(...)`:

```
import org.assertj.core.api.Assert;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import ru.gb.units.Calculator;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.io.PrintStream;
import java.util.Scanner;

import static org.assertj.core.api.Assertions.*;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculationTest {
    ...
    @Test
    void getOperandCompletesCorrectlyWithNumbers() {
        String testedValue = "9"; // Значение для тестов
        ByteArrayInputStream in = new
        ByteArrayInputStream(testedValue.getBytes());
        InputStream inputStream = System.in; // Сохраняем ссылку
на ввод с клавиатуры
        System.setIn(in); // Подменяем ввод

        Calculator.getOperand(); // Вызываем метод

        System.out.println(testedValue); // Для наглядности
вывода
        System.setIn(inputStream); // Подменяем обратно
    }
}

@Test
void getOperandCompletesCorrectlyWithNotNumbers() {
    String testedValue = "k";
    ByteArrayInputStream in = new
    ByteArrayInputStream(testedValue.getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
```

```

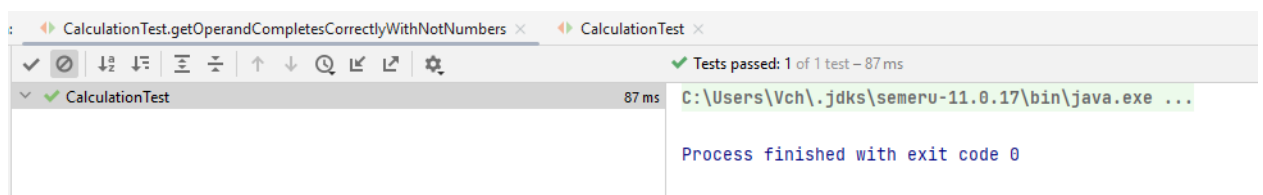
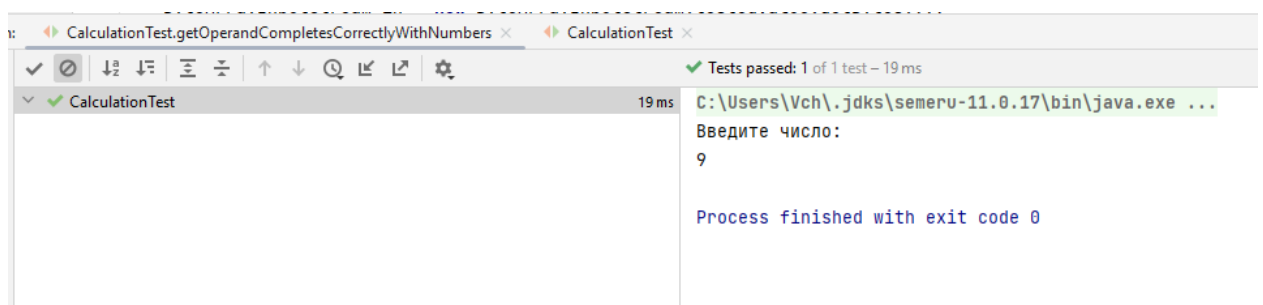
    InputStream inputStream = System.in;
    System.setIn(in);
    System.setOut(new PrintStream(out));

    assertThatThrownBy(() -> Calculator.getOperand())
        .assertInstanceOf(IllegalStateException.class).describedAs("Ошибка
        в вводимых данных");

    System.setIn(inputStream);
    System.setOut(null);
}

```

Результаты работы тестов:



Аннотации

Рассмотрим некоторые важные аннотации в JUnit 5 Jupiter API. Большинство из них находятся в пакете **org.junit.jupiter.api** в модуле **junit-jupiter-api**.

@BeforeEach

Означает, что аннотированный метод должен выполняться перед каждым методом @Test, @RepeatedTest, @ParameterizedTest или @TestFactory в текущем классе.

```

@BeforeEach
public void initEach() {
    //test setup code
}

```

@AfterEach

Означает, что аннотированный метод должен выполняться после каждого метода @Test, @RepeatedTest, @ParameterizedTest или @TestFactory в текущем классе.

```
@AfterEach
public void cleanUpEach() {
    //Test cleanup code
}
```

@BeforeAll

Означает, что аннотированный метод должен выполняться перед всеми методами @Test, @RepeatedTest, @ParameterizedTest и @TestFactory в текущем классе; аналогично @BeforeClass в JUnit 4. Такие методы должны быть статическими, если не используется жизненный цикл тестового экземпляра «для каждого класса».

```
@BeforeAll
public static void init() {
    System.out.println("BeforeAll init() method called");
}
```

@AfterAll

Означает, что аннотированный метод должен выполняться после всех методов @Test, @RepeatedTest, @ParameterizedTest и @TestFactory в текущем классе. Такие методы должны быть статическими, если не используется жизненный цикл экземпляра теста «для каждого класса».

```
@AfterAll
public static void cleanUp() {
    System.out.println("After All cleanUp() method called");
}
```

@DisplayName

Объявляет пользовательское отображаемое имя для тестового класса или тестового метода.

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

@Disable

Используется для отключения тестового класса или тестового метода;

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

@RepeatedTest

Означает, что метод является шаблоном теста для повторного тестирования.

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

@ParameterizedTest

Означает, что метод является параметризованным тестом.

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

Используемая литература

1. [Xunit](#)
2. [List of unit testing frameworks — Wikipedia](#)
3. [JUnit — Wikipedia](#)
4. [Prepare for testing | IntelliJ IDEA Documentation](#)
5. [Официальная документация junit5](#)
6. Хориков Владимир «Принципы юнит-тестирования». — СПб.: Питер, 2021. — 320 с.
7. [AssertJ — fluent assertions java library](#)
8. [Overview \(Hamcrest 2.2 API\)](#)