



# ЦИКЛЫ, ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ И АННОТАЦИИ



Оксана Мельникова



**ОКСАНА МЕЛЬНИКОВА**

Software testing engineer





# ПЛАН ЗАНЯТИЯ

1. [Параметризованные тесты](#)
2. [Пакеты](#)
3. [Аннотации](#)
4. [Массивы и строки](#)
5. [Циклы](#)
6. [Итоги](#)



# **ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ**

# ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ

На прошлой лекции мы с вами написали автотесты для нашей утилиты.

Давайте на них взглянем:

```
@org.junit.jupiter.api.Test
void shouldCalculateForRegisteredAndUnderLimit() {
    BonusService service = new BonusService();

    long amount = 1000_60;
    boolean registered = true;
    long expected = 30;

    long actual = service.calculate(
        amount, registered
    );

    assertEquals(expected, actual);
}
```

```
@org.junit.jupiter.api.Test
void shouldCalculateForRegisteredAndOverLimit() {
    BonusService service = new BonusService();

    long amount = 1_000_000_60;
    boolean registered = true;
    long expected = 500;

    long actual = service.calculate(
        amount, registered
    );

    assertEquals(expected, actual);
}
```

Видно, что меняться будут только подсвеченные данные.



# ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ

В JUnit существуют Parameterized Tests. Они дают возможность прогонять один и тот же тест с разными данными (в т.ч. ожидаемый результат).

По аналогии с ручным тестированием, мы можем прикрепить к тест-кейсу табличку с тестовыми данными, в которой перечислить, для каких входных данных нужно прогнать этот тест-кейс (см. следующий слайд).

# ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ

	A	B	C	D
1	test	amount	registered	expected
2	registered user, bonus under limit	100060	true	30
3	registered user, bonus over limit	100000060	true	500

```
void shouldCalculate(  
    String test,   
    long amount,   
    boolean registered,   
    long expected   
) {  
    BonusService service = new BonusService();  
  
    // вызываем целевой метод:  
    long actual = service.calculate(amount, registered);  
  
    // производим проверку (сравниваем ожидаемый и фактический):  
    assertEquals(expected, actual);  
}
```



# ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ

JUnit активно использует аннотации — мы уже видели, что достаточно «повесить» на метод аннотацию `@Test`, чтобы метод стал тестом.

То же самое с параметризованными тестами — нужно написать аннотацию `@ParameterizedTest` и сообщить JUnit, откуда брать входные данные.





# АННОТАЦИИ

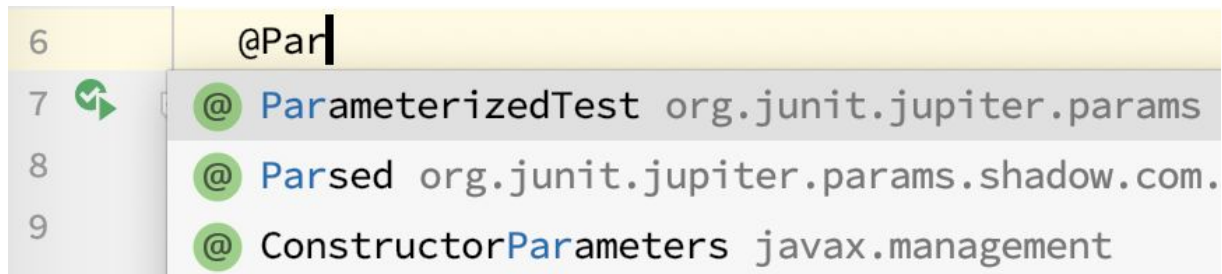
**Аннотации** — это мета-данные, прикрепляемые к коду.

Фактически, это специальные «пометки», которые программист может оставлять для того, чтобы на основе этих пометок JUnit и другие инструменты могли делать свою работу.

В большинстве случаев перед аннотацией будет символ @ (но не всегда).  
Аннотации можно писать над классами, методами, параметрами и т.д.

# ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ

Удалим аннотацию `@Test` и добавим аннотацию `@ParameterizedTest`:



**Важно:** IDEA значительно помогает благодаря автодополнению — активно используйте его, это позволит вам писать меньше и меньше ошибаться.

Продолжайте набирать, пока не будет подсвечен нужный вам вариант, или выберите его с помощью стрелок.

# import

IDEA сделала две вещи:

- дописала за нас `@ParameterizedTest`;
- добавила строку `import org.junit.jupiter.params.ParameterizedTest;`

```
import org.junit.jupiter.params.ParameterizedTest;
```

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

```
class BonusServiceTest {  
    @ParameterizedTest  
    void shouldCalculateForRegisteredAndUnderLimit() {  
        BonusService service = new BonusService();  
    }  
}
```



# ПАКЕТЫ

# ПАКЕТЫ

В Java все классы (и другие сущности) разделены по пакетам.

**Пакет** — это просто каталог (или их набор) на жёстком диске. Например, в стандартной библиотеке Java:



## ЗАЧЕМ ЭТО НУЖНО?

У нас есть онлайн-магазин, в котором продаются ноутбуки (и зарядники к ним), а также смартфоны (и зарядники к ним) и другие гаджеты.

Если товаров у нас больше хотя бы 20, то просто «вывалить» их на одну страницу — значит запутать пользователя.

И даже на сайтах фаст-фудов появляется категоризация:



**БУРГЕРЫ ИЗ  
ГОВЯДИНЫ**



**БУРГЕРЫ ИЗ КУРИЦЫ  
И РЫБЫ**



**ЗАКУСКИ**

Изображения с сайта Burger King

# ЗАЧЕМ ЭТО НУЖНО?

В разделе Ноутбуки, мы можем показывать ноутбуки, а зарядники (аксессуары) к ноутбукам можем поместить в подраздел.

- Ноутбуки
  - Acer
  - Samsung
  - Зарядники
  - ...
- Смартфоны
  - iPhone
  - Samsung
  - Зарядники
  - ...



## ЗАЧЕМ ЭТО НУЖНО?

Благодаря такой структуре:

- очень легко найти нужный товар;
- можно использовать короткие названия — т.е. если пользователь зашёл в раздел Ноутбуки и видит Зарядники, то это Зарядники для ноутбуков.

В реальной жизни, конечно же, не всегда так, но мы описали так, чтобы вы поняли, как это работает в Java.





# ПАКЕТЫ В JAVA

Когда классов становится очень много, возникает две проблемы:

1. Очень трудно найти нужный.
2. Очень трудно придумывать новые имена классам (чтобы они не совпадали с уже существующими).

В Java нельзя создать два одинаковых класса в одном пакете (попробуйте в своём проекте создать два класса `BonusService` — не получится).

Поэтому классы так же аккуратно раскладывают по «папочкам», организуя некоторую иерархию.



# ПАКЕТЫ В JAVA

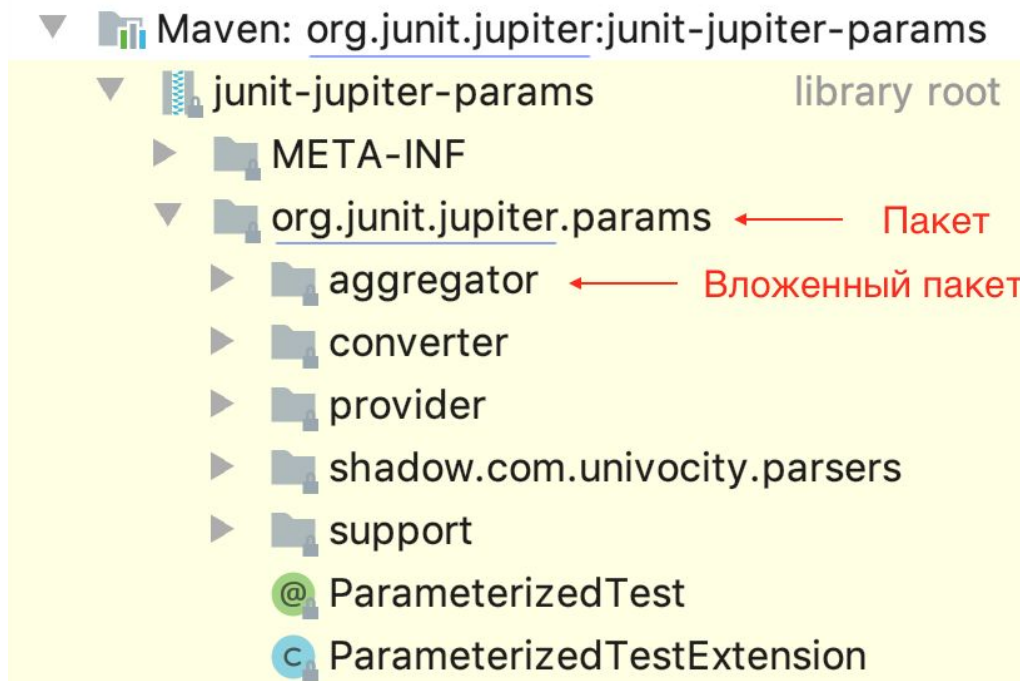
Таким образом, пакет — это каталог на жёстком диске, служащий для решения проблемы имён классов.

Т.е. если у нас два разных пакета: `notebook.charger` и `smartphone.charger`, то мы можем в них создавать классы с одинаковыми именами, но это будут «разные» классы.

# ПАКЕТЫ В JAVA

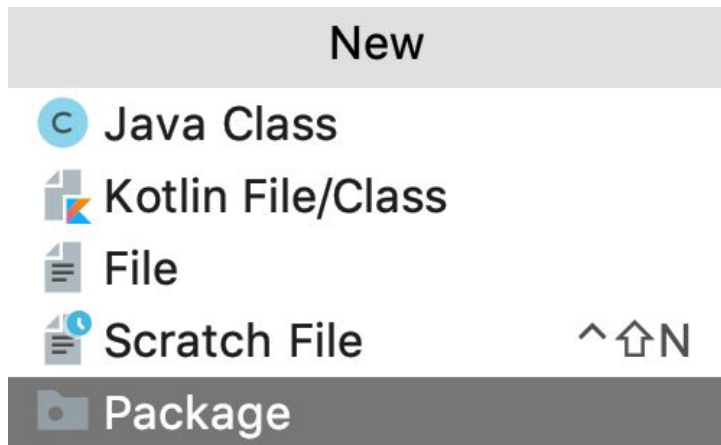
Все классы находятся в пакетах. До этого наши классы находились в безымянном пакете — это очень плохая практика.

Часто принято в название пакета включать `GroupId` артефакта:



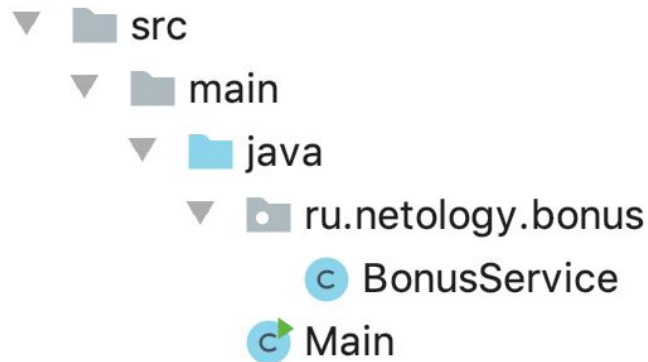
# ПАКЕТЫ В JAVA

Создадим пакет в IDEA (Alt + Insert на каталоге src):

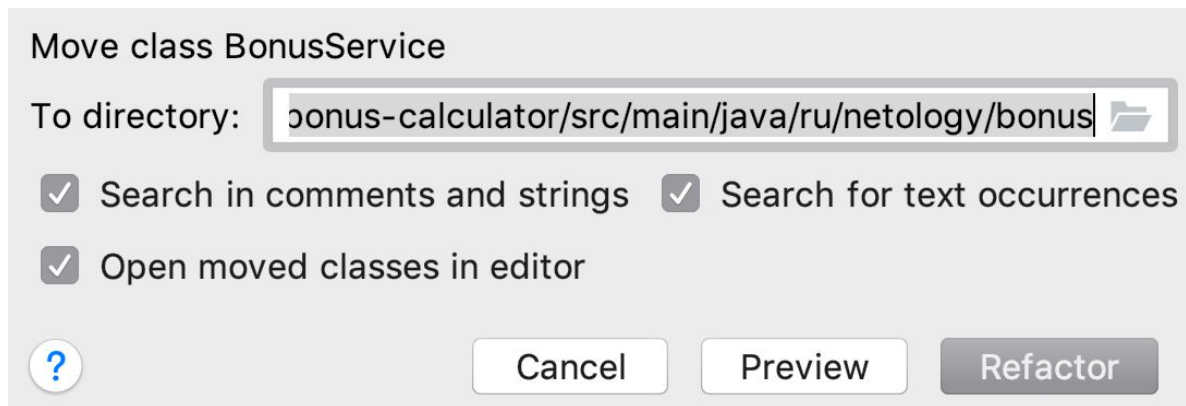


# ПАКЕТЫ В JAVA

Мышкой (или через Ctrl + X Ctrl + V) перетащим туда BonusService:



IDEA выдаст диалог подтверждения:



# ПАКЕТЫ В JAVA

Наше перемещение класса в пакет `ru.netology.bonus` привело к двум вещам:

1. В класс `BonusService` добавилась (автоматически) строка `package ru.netology.bonus;`
2. В классы `Main` и `BonusServiceTest` добавилась (автоматически) строка `import ru.netology.bonus.BonusService;`

## package И import

Когда классы находятся в разных пакетах, они должны использовать fully-qualified (полное) имя класса для обращения.

Полное имя класса состоит из имени пакета + имя класса:

`ru.netology.bonus.BonusService`.

Но это «очень длинно», поэтому есть конструкция `import`, которая буквально говорит: «берём вот этот класс по полному имени и дальше в этом файле будем использовать только короткое имя».

Т.е. если в `Main.java` мы написали `import ru.netology.bonus.BonusService;`, то дальше в файле `Main.java` можем просто писать `BonusService`.

# ПАКЕТЫ

Q: а если классы находятся в одном пакете?

A: если в одном — то не нужно. Перенесём **Main** в тот же пакет:

```
import ru.netology.bonus.BonusService;
```

Unused import statement



Optimize imports

More actions...





# ПАКЕТЫ И ТЕСТЫ

То же самое нужно сделать с тестом — перенесём его в тот же пакет (`ru.netology.bonus`).

**Q:** т.е. заберём из каталога тестов и перенесём к программе?

**A:** нет. Тесты должны лежать в каталоге для тестов. При работе с пакетами важно только то, в каком пакете лежит класс. Если имя пакетов совпадает — то классы лежат в одном пакете.



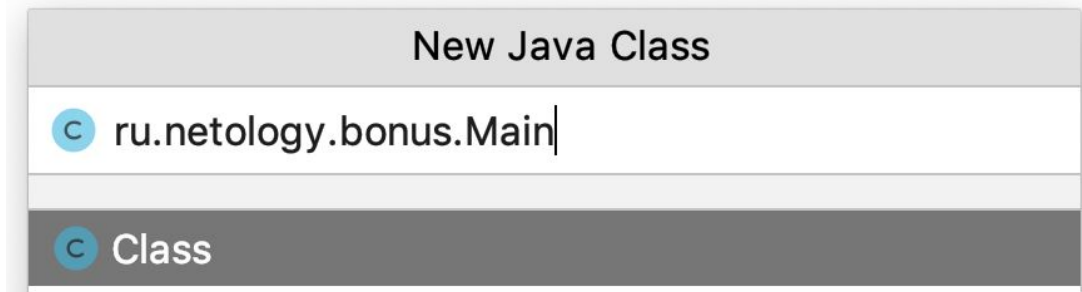
# ВАЖНО

Ключевые две вещи:

1. Вы не должны руками писать `package ...` и `import ...` — за вас это сделает IDEA.
2. Начиная с сегодняшнего дня все ваши классы должны быть в пакетах (без пакетов ДЗ не принимается).

# IDEA

Когда вы создаёте проект, вы можете сразу создавать класс в пакете:

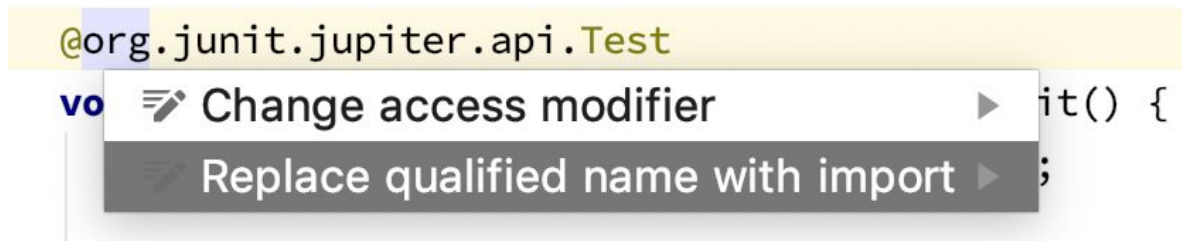


За вас создастся сразу и класс, и пакет.

Кроме того, автотесты сразу будут создаваться в нужном пакете (в том же, что и тестируемый класс).

# IDEA

Заменить полные (fully-qualified) имена можно с помощью установки курсора и нажатия **Alt + Enter**:



IDEA сама добавит нужный **import**.



# **ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ**

# ВОЗВРАЩАЕМСЯ К ТЕСТАМ

Итак, теперь понятно, откуда взялся `import` для `@ParameterizedTest` и что он значит.

```
import org.junit.jupiter.params.ParameterizedTest;
```

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

```
class BonusServiceTest {  
    @ParameterizedTest  
    void shouldCalculateForRegisteredAndUnderLimit() {  
        BonusService service = new BonusService();  
    }  
}
```

Мы пока не разбирались со `static` и `import *`: пока нужно запомнить, что конструкция `import static org.junit.jupiter.api.Assertions.*;` позволяет использовать методы `assertEquals` без указания класса.

Мы будем её использовать только для `assertEquals` и подобных.

# CSV

Аннотация `@ParameterizedTest` не может использоваться сама по себе — нужно указать, откуда брать данные.

Мы берём данные из общепринятого «табличного» формата данных [CSV](#), который позволяет указывать набор значений, разделённых определённым символом — разделителем.

Например, наши тест-кейсы в этом формате можно было записать вот так:

```
'registered user, bonus under limit',100060,true,30  
'registered user, bonus over limit',100000060,true,500
```

Обратите внимание:

1. CSV — это не Java, здесь мы не используем `_` в числах.
2. Если встречается значение, содержащее запятую (название теста), то оно заключается в кавычки.
3. Одна строка — один набор данных.

# @CsvSource

Для того, чтобы использовать этот формат в наших тестах, мы можем использовать аннотацию @CsvSource:

```
@ParameterizedTest
@CsvSource()
void shouldCalculateForRegisteredAndUnderLimit() {
    String[] value() new BonusService();
    char delimiter() default ',';
    // подготавливаем данные:
    long amount = 1000_60;
    boolean registered = true;
    long expected = 30;
```

Но использование этой аннотации больше похоже на вызов функции, чем на те аннотации, что мы видели.

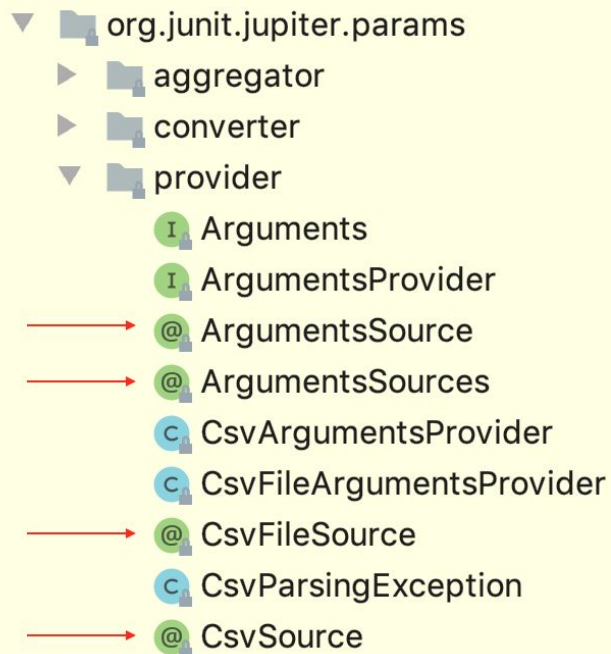


# АННОТАЦИИ

Q: где мне узнать про все аннотации?

A: варианта два:

- В руководстве [на официальном сайте](#);
- Изучая пакеты библиотек:



```
▼ org.junit.jupiter.params
  ► aggregator
  ► converter
  ▼ provider
    I Arguments
    I ArgumentsProvider
    → @ ArgumentsSource
    → @ ArgumentsSources
    C CsvArgumentsProvider
    C CsvFileArgumentsProvider
    → @ CsvFileSource
    C CsvParsingException
    → @ CsvSource
```

всё, что с @ - аннотации



# АННОТАЦИИ

## @CsvSource

**Научимся читать аннотации:** для этого надо на названии аннотации нажать **F4** и мы попадём в исходный файл, в котором описана аннотация.

Здесь особо ничего не понятно, но IDEA предлагает скачать «исходные коды» этого файла:

```
Decompiled .class file, bytecode ... Download Sources Choose Sources...
1  /.../
5
6  package org.junit.jupiter.params;
7
```

Естественно, для этого нужно соединение с Интернет.

# @CsvSource

```
public @interface CsvSource {  
    /**  
     * The CSV lines to use as source of arguments; must not be empty.  
     *  
     * Each value corresponds to a line in a CSV file and will be split using  
     * the specified {@link #delimiter delimiter}.  
     */  
    String[] value();  
    /**  
     * The column delimiter to use when reading the {@linkplain #value lines}.  
     *  
     * Defaults to {@code ','}.  
     */  
    char delimiter() default ',';  
}
```

`/** ... */` — это JavaDoc-комментарии, в которых описывается как работает эта аннотация.

Вам нужно привыкать их читать, потому что комментарии и сами исходные файлы — это первоисточник информации (не статьи в интернете, а именно исходники).

## @CsvSource

```
public @interface CsvSource {  
    String[] value();  
    char delimiter() default ',';  
}
```

Если убрать комментарии, то получается, что аннотация по структуре напоминает класс с набором методов.

Разберёмся теперь вот с этим: `String[]`.



# **МАССИВЫ И СТРОКИ**



# МАССИВЫ И СТРОКИ

Запись вида `String[]` — это массив (новый тип данных, который мы сегодня изучим).

`String` — это строковый тип данных в Java.

Оба этих типа не являются примитивными и выделяются среди других типов данных.

# СТРОКИ

Строки в Java представлены классом `String` (поэтому и написано с большой буквы).

Строки — особый класс в Java, один из немногих, которые позволяют инициализировать себя без\* вызова `new`:

```
String testName = "registered user, bonus under limit";  
// Кавычки обязательно должны быть двойными  
// Переноса строк внутри кавычек не допускается
```

Примечание\*: хотя с `new` тоже можно, но мы эти детали обсуждать не будем.



# ОПЕРАЦИИ СО СТРОКАМИ

У строк есть свой оператор `+`, который осуществляет конкатенацию (склеивание) двух строк в одну:

```
String testName = "registered user, " + «bonus under limit»;
// Это позволяет длинные строки разбивать на несколько:
String message = "Hello, dear User!" + "You registered on our service ...";
```

Все остальные операции представлены методами, которые мы пройдем позже.

# МАССИВЫ

**Массив** — это упорядоченный набор данных фиксированной длины.

Когда нам нужно хранить набор данных (объектов или примитивов), мы можем использовать массивы:

```
// Объявление массива:  
int[] numbers; // массив из целых чисел  
String[] names; // массив из строк  
  
// Но неинициализированные переменные использовать нельзя!  
  
// Инициализация массива (вариант 1):  
int[] numbers = new int[3]; // массив из 3 чисел  
String[] names = new String[3]; // массив из 3 строк  
  
// Инициализация массива (вариант 2):  
int[] numbers = {1, 2, 3}; // массив из 3 чисел  
String[] names = {"Vasya", "Petya", "Masha"}; // массив из 3 строк
```

# ОПЕРАЦИИ С МАССИВАМИ

С массивами можно выполнять две ключевые операции:

- чтение данных по индексу;
- запись данных по индексу.

```
String[] names = {"Vasya", "Petya", "Masha"}; // массив из 3 строк
```

```
// нумерация элементов идёт с нуля:
```

```
// — у «Vasya» индекс = 0
```

```
// — у «Petya» индекс = 1
```

```
// — у «Masha» индекс = 2
```

```
// Чтение по индексу:
```

```
System.out.println(names[0]);
```

```
// Запись по индексу:
```

```
names[0] = "Vasiliy Ivanovich";
```

```
// .length — свойство, хранящее длину массива
```

```
System.out.println(names.length);
```

# СВОЙСТВА (ПОЛЯ ОБЪЕКТОВ)

До этого мы с вами работали только с методами, сейчас же познакомимся со свойствами (или полями объектов)\*:

```
// .length — свойство, хранящее длину массива  
System.out.println(names.length);
```

Т.е. обращение к свойству идёт с помощью оператора `.` после которого следует имя свойства.

В отличие от методов, мы не ставим круглые скобки — поскольку `()` после имени означают «вызов метода».

Свойство `length` предназначено только для чтения — в него ничего нельзя записать.

Примечание\*: позже мы будем разграничивать эти термины.



# МАССИВЫ И СТРОКИ

**Итого:** мы разобрались, что `String[]` — это массив строк и посмотрели на варианты его инициализации.

Вернёмся же к аннотациями.



# АННОТАЦИИ

---

## @CsvSource

```
public @interface CsvSource {  
    String[] value(); // элемент аннотации  
    char delimiter() default ','; // элемент аннотации  
}
```

Аннотация — по структуре напоминает класс с набором методов.

# КАК С ЭТИМ РАБОТАТЬ?

Поскольку аннотации — это мета-данные (в данном случае для JUnit), мы можем в них прописывать доп.информацию, которую потом сам JUnit и

```
// Записываем данные в аннотацию:
@CsvSource(
    value={
        "registered user, bonus under limit',100060,true,30",
        "registered user, bonus over limit',100000060,true,500"
    },
    delimiter=',',
)
```

Где-то потом JUnit сам сделает:


```
// Упрощённый псевдокод (только для понимания)
String[] values = annotation.value(); // чтобы получить то, что мы записали
char delimiter = annotation.delimiter(); // чтобы получить то, что мы записали
// Затем для каждого значения из values делаем:
// 1. Разбиваем строку по delimiter
// 2. Вызываем тестовый метод, передавая в него все параметры
```



# ПАРАМЕТРЫ

Но параметров у нас нет, поэтому добавим сами:

```
@ParameterizedTest
@CsvSource(
    value={
        "registered user, bonus under limit',100060,true,30",
        "registered user, bonus over limit',100000060,true,500"
    },
    delimiter=',',
)
void shouldCalculate(String test, long amount, boolean registered, long expected) {
    BonusService service = new BonusService();
    // вызываем целевой метод:
    long actual = service.calculate(amount, registered);
    // производим проверку (сравниваем ожидаемый и фактический):
    assertEquals(expected, actual);
}
```

A diagram with red arrows illustrating the mapping of parameters from the CSV source to the test method arguments. The first CSV line "registered user, bonus under limit',100060,true,30" has arrows pointing to the arguments: "test" (String), "amount" (long), "registered" (boolean), and "expected" (long). The second CSV line "registered user, bonus over limit',100000060,true,500" has arrows pointing to the same four arguments in the same order.

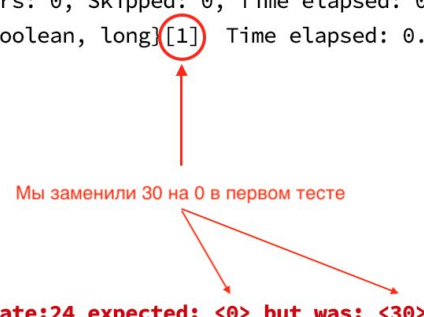
**Важно:** показать работу в дебаггере.

JUnit сам подставит в тестовый метод значения именно в том порядке, в котором мы указали.

# ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ

Попробуем «уронить» первый тест, чтобы посмотреть, как это выглядит:

```
[INFO] -----  
[INFO] T E S T S  
[INFO] -----  
[INFO] Running ru.netology.bonus.BonusServiceTest  
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.148 s <<< FAILURE!  
[ERROR] shouldCalculate{String, long, boolean, long}[1] Time elapsed: 0.044 s <<< FAILURE!  
  
[INFO]  
[INFO] Results:  
[INFO]  
[ERROR] Failures:  
[ERROR] BonusServiceTest.shouldCalculate:24 expected: <0> but was: <30>  
[INFO]  
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0  
[INFO]  
[INFO] -----  
[INFO] BUILD FAILURE  
[INFO] -----
```



Мы заменили 30 на 0 в первом тесте

Интересно, но JUnit нумерует тесты с 1 а не с 0. Т.е. по этим логам вы должны понимать, что упал именно первый тест.



# АННОТАЦИИ

Вы должны понимать: аннотации — это не вызов метода (хотя бывает похоже).

Аннотации — это установка мета-данных. Иногда в эти мета-данные можно передавать параметры (как в аннотации `@CsvSource`).

Но в `@Test` или `@ParameterizedTest` же мы ничего не передавали? Почему так?

# АННОТАЦИИ

// Внутри нет никаких элементов — поэтому передавать ничего нельзя

```
public @interface Test {  
}
```

Но в `@ParameterizedTest` есть элемент `name()` а мы туда ничего не передавали.

```
public @interface ParameterizedTest {  
    // опустили не интересующие нас конструкции  
  
    String name() default DEFAULT_DISPLAY_NAME;  
}
```

У элементов аннотаций может быть значение по умолчанию. Это значит, что указывать его необязательно.

# DEFAULT

```
public @interface CsvSource {  
    String[] value(); // элемент аннотации  
    char delimiter() default ','; // элемент аннотации  
}
```

Значит, `delimiter` можно не указывать (если нас устраивает значение по умолчанию):

```
@ParameterizedTest  
@CsvSource(  
    value={  
        "registered user, bonus under limit',100060,true,0",  
        "registered user, bonus over limit',100000060,true,500"  
    }  
  
    void shouldCalculate(String test, long amount, boolean registered, long expected) {  
        ...  
    }
```

# value

Кроме того, имя `value` имеет специальное предназначение — если указывается только оно `value`, то его можно не писать:

```
@ParameterizedTest
@CsvSource({
    "registered user, bonus under limit',100060,true,0",
    "registered user, bonus over limit',100000060,true,500"
})
void shouldCalculate(String test, long amount, boolean registered, long expected) {
    ...
}
```

# АННОТАЦИИ

Итого:

- мы научились использовать параметризованные тесты и `@CsvSource`;
- мы начали учиться читать аннотации;
- немного поговорили о массивах и строках.

Вы должны понимать, что аннотации — это не классы и не вызов метода. Это мета-данные, в которые бывает можно передавать некоторые параметры.

Нужно обязательно научиться читать и использовать аннотации, поскольку на них будет строиться большая часть автоматизации.



# ЦИКЛЫ





# ЦИКЛЫ

Мы с вами посмотрели, как JUnit умеет много раз запускать одни и те же действия с разными параметрами.

Посмотрим, какие конструкции у нас есть в Java, для того, чтобы сделать нечто подобное.

На самом деле, вам как тестировщикам, придётся достаточно редко работать с циклами (есть более высокоуровневые инструменты), но основные принципы знать надо.



# ЦИКЛЫ

Итак, **цикл** — это повторение одного и того же блока кода, выполняемое по условию.

На примере JUnit: перебирались все наши тест-кейсы и для них запускался тестовый метод. Буквально: для каждого теста делаем одно и то же.

Какие задачи можно решать с помощью циклов: у нас есть данные о покупках конкретного пользователя за месяц и мы хотим посчитать статистику:

- общие расходы за месяц;
- сумму максимальной покупки;
- количество покупок, стоимостью выше определённого лимита;
- и любые производные (средняя стоимость покупки и т.д.).



# ЦИКЛЫ

В Java есть 4 вида циклов:

1. `foreach`;
2. `for`;
3. `while`;
4. `do-while`.

Сегодня мы рассмотрим только первый, как наиболее часто используемый при работе с набором элементов.

# foreach

Здесь буквально «перебираются» все элементы из массива и на каждой итерации кладётся следующий элемент в переменную `purchase`.

```
// Продемонстрировать работу в дебаггере
int[] purchases = {1_000, 2_000, 500, 5_000, 2_000};
// IDEA: iter + tab
for (int purchase : purchases) {
    System.out.println(purchase);
}
```

Цикл останавливается после перебора всех элементов.

**Важно:** несмотря на то, что `purchase` здесь объявлена «вне блока», по факту — она видна только внутри `for`.

# СУММА

Поиск суммы аналогичен подсчёту суммы «карандашиком» — мы начинаем с нуля и к нему «прибавляем» следующие элементы (числа).

Если проделать это со всеми элементами — то, в итоге получим сумму:

```
public long calculateSum(long[] purchases) {  
    long sum = 0; // начинаем с нуля  
    for (long purchase : purchases) {  
        // аналог sum = sum + purchase;  
        // каждый раз прибавляем к текущей сумме новый элемент  
        sum += purchase;  
    }  
    return sum;  
}
```

**Важно:** показать работу в дебаггере.

---

# TECT

@Test

```
void calculateSum() {  
    StatisticsService service = new StatisticsService();  
  
    long[] purchases = {1_000, 2_000, 500, 5_000, 2_000};  
    long expected = 10_500;  
  
    long actual = service.calculateSum(purchases);  
  
    assertEquals(expected, actual);  
}
```

# МАКСИМУМ

Поиск максимума можно рассмотреть по аналогии с тем, как маленькие дети иногда выбирают что-то из набора предметов: берут первое попавшееся в руку, затем берут второй, сравнивают и оставляют в руке лучший.

Если проделать это сравнение со всеми предметами, то в итоге у нас в руке останется лучший:

```
// показать работу в дебаггере
public long findMax(long[] purchases) {
    long currentMax = purchases[0]; // берём за точку отсчёта первый
    // перебираем по одному
    for (long purchase : purchases) {
        // каждый раз сравниваем:
        // если тот, что у нас — меньше, чем текущий
        if (currentMax < purchase) {
            // «выкидываем» наш, а вместо него «кладём» в переменную текущий
            currentMax = purchase;
        }
    }
    // возвращаем итоговый
    return currentMax;
}
```

# TECT

```
@Test
void findMax() {
    StatisticsService service = new StatisticsService();

    long[] purchases = {1_000, 2_000, 500, 5_000, 2_000};
    long expected = 5_000;

    long actual = service.findMax(purchases);

    assertEquals(expected, actual);
}
```



# Параллельные потоки

**Q:** зачем нам `@Parameterized`, мы же могли просто в тестах написать циклы?

**A:** обычно не принято в тестах размещать логику (тогда придётся тестировать ещё и эту логику), поэтому старайтесь не использовать циклы и условия в тестах.

**Q:** а как через `@Parameterized` и `@CsvSource` передавать массивы?

**A:** встроенных средств для этого нет (т.к. формат CSV не подразумевает, что вы можете в столбце записать несколько значений), но есть возможность создать "источник" для значений (мы этот метод рассматривать не будем, но если вам интересно, читайте документацию на `@ArgumentsSource` ).



**ИТОГИ**



## ИТОГИ

Сегодня была достаточно насыщенная лекция — мы обговорили важность параметризации тестов, то, как использовать и читать аннотации.

Поговорили о пакетах: для чего нужны, как их создавать и как использовать (не делайте этого руками, за вас сделает IDEA).

Кроме того, посмотрели на циклы (повторяющиеся наборы действий) и то, как их использовать с массивами.

Самое важное: привыкайте работать с исходниками и документацией, поскольку именно в них будут (чаще всего) ответы на все вопросы.



# ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в чате Slack!
- Задачи можно сдавать по частям.
- Зачет по домашней работе проставляется после того, как приняты **все задачи**.



Задавайте вопросы и напишите отзыв о лекции!

**ОКСАНА МЕЛЬНИКОВА**

 Оксана Мельникова