

# ОБЪЕКТЫ С ВНУТРЕННИМ СОСТОЯНИЕМ УПРАВЛЕНИЕ СОСТОЯНИЕМ ПРИ ТЕСТИРОВАНИИ



ОКСАНА МЕЛЬНИКОВА



**ОКСАНА МЕЛЬНИКОВА**

Software testing engineer





# План занятия

1. [Инициализация](#)
2. [Инициализаторы полей](#)
3. [Testability](#)
4. [Инициализаторы полей в тестах](#)
5. [Конструкторы](#)
6. [Перегрузка](#)
7. [Code Generation](#)
8. [Итоги](#)



# ИНИЦИАЛИЗАЦИЯ



# ИНИЦИАЛИЗАЦИЯ

*Вопрос к аудитории: давайте вспомним, как происходит процедура инициализации полей объектов. В какие значения инициализируются поля?*

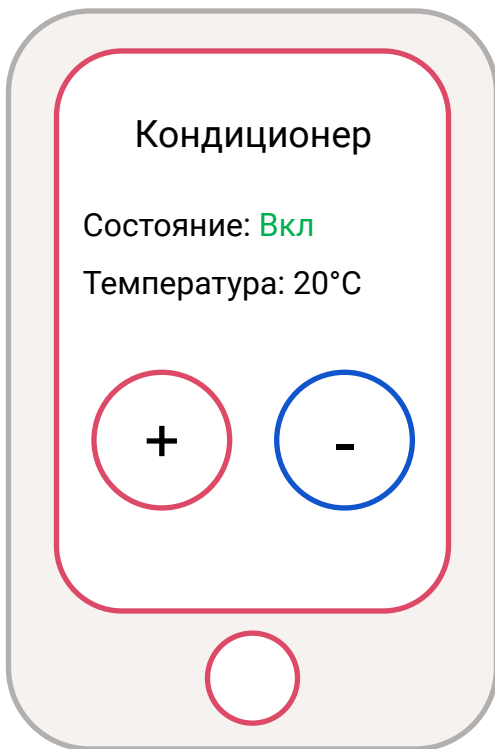


# ИНИЦИАЛИЗАЦИЯ

При создании объекта его поля инициализируются (т.е. им присваиваются) нулевые значения:

- для целых чисел — `0`
- для вещественных чисел — `0.0`
- для `boolean` — `false`
- для объектов — `null`

*Вопрос к аудитории: чем это может быть неудобно?*



## Афиша



Вперёд  
мультфильм



Отель «Белград»  
комедия



Остров фантазий  
ужасы



Джентльмены  
боевик

# НАЧАЛЬНОЕ СОСТОЯНИЕ

Не всегда нулевые значения являются валидными значениями для объекта, например:

- для кондиционера совсем неправильно, что минимальная, максимальная и текущая температуры устанавливаются в 0 (мы привыкли, что в реальной жизни там совсем другие значения)
- для менеджера фильмов (вспоминаем афишу): поле с фильмами устанавливается в `null` — это тоже плохо, было бы логичнее сделать пустой массив (`null` нам всегда грозит NPE)



# TESTABILITY

А теперь вспомним ДЗ «Радиоман»:

1. Клиент должен иметь возможность увеличивать и уменьшать уровень громкости звука (**в пределах от 0 до 10**)\*
2. Если уровень громкости звука достиг максимального значения, то дальнейшее нажатие на + не должно ни к чему приводить
3. Если уровень громкости звука достиг минимального значения, то дальнейшее нажатие на — не должно ни к чему приводить

Примечание\*: на следующей лекции мы поговорим, почему здесь 10

Представьте, что пределы от **0 до 100**, сколько бы раз в тесте вам пришлось вызывать метод увеличения громкости? Большое количество вызовов будет неудобно для тестирования.



# ОБЪЕКТЫ С ВНУТРЕННИМ СОСТОЯНИЕМ

Внутреннее состояние объектов является проблемой, потому что от него зависит поведение объекта, и иногда его очень сложно «подготавливать».

Например, мы хотим протестировать, что обычная шариковая ручка после первого использования (если не закрывать колпачок) «не засыхает» и продолжает писать даже через год.

Чтобы провести тест, мы можем купить новую ручку и подождать целый год, чтобы проверить это требование...

*Вопрос к аудитории: как нам это сделать быстрее?*



## ОБЪЕКТЫ С ВНУТРЕННИМ СОСТОЯНИЕМ

Было бы здорово найти ручку, которая уже была открыта и хранилась без колпачка 11-12 месяцев. Тогда бы мы смогли использовать её для тестов и не ждать целый год.

В обычной жизни это проблематично, но в мире ПО у нас есть для этого все возможности.



# УСТАНОВКА ЗНАЧЕНИЯ ПОЛЕЙ

Для установки значения полей в Java (если поля приватные) у нас есть следующие возможности:

1. инициализаторы полей
2. блоки инициализации
3. конструкторы
4. методы (включая setter'ы)
5. reflection API\*

Примечание\*: не рассматриваем в рамках этой лекции.



# ИНИЦИАЛИЗАТОРЫ ПОЛЕЙ

# ИНИЦИАЛИЗАТОРЫ ПОЛЕЙ

Инициализаторы полей — это выражения, позволяющие задать начальные значения полям создаваемых объектов:

```
package ru.netology.domain.field;  
  
public class Conditioner {  
    private int id;  
    private String name = "noname";  
    private int maxTemperature = 30;  
    private int minTemperature = 15;  
    private int currentTemperature = 22;  
    private boolean on;  
}
```

← инициализаторы полей

Как вы видите, не обязательно всем полям давать начальные значения. Если нас устраивают нулевые значения, то можно оставить их.

# ПРОВЕРЯЕМ

Удостоверимся, что теперь при создании объекта поля, для которого указаны инициализаторы, действительно проинициализированы в ненулевые значения:

```
package ru.netology.domain.field;

class ConditionerTest {
    @Test
    public void shouldInitFields() {
        Conditioner conditioner = new Conditioner();

        assertEquals(«noname», conditioner.getName());
        assertEquals(30, conditioner.getMaxTemperature());
        assertEquals(15, conditioner.getMinTemperature());
        assertEquals(22, conditioner.getCurrentTemperature());
    }
}
```

# ЧТО МОЖНО ИСПОЛЬЗОВАТЬ

В инициализаторах полей можно использовать не только литералы (вы ведь помните, что такое литералы?), но и:

1. Оператор `new` (для создания новых объектов)
2. Арифметические, логические и иные выражения
3. Вызовы методов
4. Обращение к собственным полям




# ЧТО МОЖНО ИСПОЛЬЗОВАТЬ

Несмотря на то, что в инициализаторах можно многое, мы не рекомендуем усложнять: инициализаторы должны быть короткими и ПОНЯТНЫМИ:

```
package ru.netology.domain.field;

public class Conditioner {
    private int id;
    private String name = «noname»;
    private int maxTemperature = 30;
    private int minTemperature = 15;
    private int currentTemperature = (maxTemperature + minTemperature) / 2;
    private boolean on;
}
```





## ЧТО НЕЛЬЗЯ ИСПОЛЬЗОВАТЬ

В инициализаторах полей можно использовать только выражения (синтаксические конструкции, возвращающие ровно одно значение).

К выражениям не относятся:

- условия
- циклы
- блоки кода



## ИНИЦИАЛИЗАТОРЫ ПОЛЕЙ

Что мы получили? Теперь наши объекты уже сразу после создания выглядят более-менее реалистично.

Но проблема по-прежнему осталась: если окажется, что setter'ов на поле температуры нет (как вы вспомните, на пульте управления кондиционером нет цифровой клавиатуры), то нам придётся **6 раз** вызывать метод увеличения температуры на единицу.

Неудобно.



# TESTABILITY

# TESTABILITY

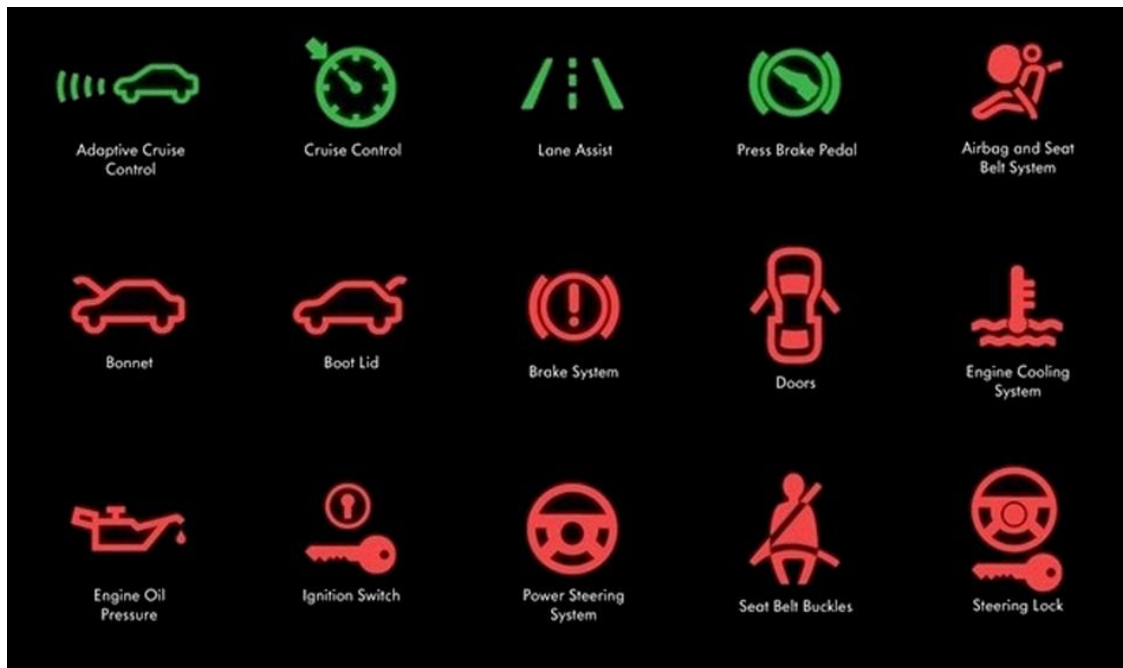
Важный момент: достаточно часто в классы добавляют методы для удобства тестирования, в том числе getter'ы и setter'ы.

Например, несмотря на то, что на пульте кондиционера физически нет кнопок установки температуры (кроме как +/-), мы можем добавить\* getter'ы и setter'ы для этого поля **только для удобства тестирования**.

Примечание\*: ни в коем случае **не добавляйте самовольно** код в чужие классы (написанные другими программистами) для упрощения тестирования! Обязательно согласовывайте это с ними, а ещё лучше — просите их самих добавить такой код.

# TESTABILITY

В реальной жизни мы видим то же самое: в автомобиле достаточно много датчиков (getter'ы), которые позволяют получать различную информацию о состоянии автомобиля:



# TESTABILITY

В то же время есть возможности, позволяющие настраивать некоторое состояние: например, можно сбросить «дневной пробег» (это не setter в чистом виде, но некий аналог установки состояния):





# **ИНИЦИАЛИЗАТОРЫ ПОЛЕЙ В ТЕСТАХ**



# AUTOMATION ENGINEER VIEW

С точки зрения автотестеров инициализаторы полей можно  
ИСПОЛЬЗОВАТЬ В ТЕСТАХ:

```
class StatisticsServiceTest {  
  
    @Test  
    void calculateSum() {  
        StatisticsService service = new StatisticsService();  
        ...  
        assertEquals(expected, actual);  
    }  
  
    @Test  
    void findMax() {  
        StatisticsService service = new StatisticsService();  
        ...  
        assertEquals(expected, actual);  
    }  
}
```

```
class StatisticsServiceTest {  
    StatisticsService service = new StatisticsService();  
    @Test  
    void calculateSum() {  
        ...  
        assertEquals(expected, actual);  
    }  
  
    @Test  
    void findMax() {  
        ...  
        assertEquals(expected, actual);  
    }  
}
```

Примечание\*: это тесты для одного из сервисов с предыдущих лекций

## КАК ЭТО РАБОТАЕТ

Напоминаем, что для запуска каждого метода с аннотацией `@Test` по умолчанию JUnit создаёт новый объект тестового класса и на нём уже вызывает этот метод.

Так как каждый раз создаётся новый объект класса, значит, инициализируется поле (`StatisticsService service = new StatisticsService();`). Поэтому, мы можем не писать эту строку каждый раз внутри тестовых методов.

# ИНКАПСУЛЯЦИЯ

Наверное вы обратили внимание, что внутри тестов мы не всегда соблюдаем свои же правила: `public class`'ы и методы, а все поля `private`. В коде, сгенерированном IDEA (Ctrl + Shift + T) всё `package-private`.

К тестам не всегда предъявляются те же требования, что и к обычному коду, но мы рекомендуем вам следовать правилам и вырабатывать единый стиль.

```
class StatisticsServiceTest {  
    StatisticsService service = new StatisticsService();  
    @Test  
    void calculateSum() {  
        ...  
        assertEquals(expected, actual);  
    }  
  
    @Test  
    void findMax() {  
        ...  
        assertEquals(expected, actual);  
    }  
}
```



# КОНСТРУКТОРЫ



# КОНСТРУКТОРЫ

**Конструкторы (constructors)** — именованные блоки кода, очень похожие на методы. Их ключевая задача — инициализация объекта.

Поясним на примере: завод выпускает кондиционеры.

Кондиционеры с завода приходят с «заводскими настройками», то есть там уже выставлены некоторые параметры — температура, режим и т.д.

У разных моделей кондиционеров разные температурные диапазоны и разные заводские настройки.



# КОНСТРУКТОРЫ

Мы, конечно, можем создать по отдельному классу на каждую модель кондиционера и выставить там необходимые значения по умолчанию через инициализаторы полей.

Но если значение полей — единственное отличие, то, возможно, существует способ не плодить кучу классов?



# КОНСТРУКТОРЫ

Давайте создадим новый package constructor, в который скопируем наш класс и скомпилируем приложение (mvn compile).

# DEFAULT CONSTRUCTOR

Прежде чем мы начнём создавать собственный конструктор, давайте посмотрим на сгенерированный class-файл\* (через декомпилятор IDEA):

```
package ru.netology.domain.constructor;

public class Conditioner {
    private int id;
    private String name = «noname»;
    private int maxTemperature = 30;
    private int minTemperature = 15;
    private int currentTemperature = 22;
    private boolean on;

    public Conditioner() { ← default constructor
    }
}
```

Примечание\*: для Maven-проекта находится в каталоге target/classes.



# DEFAULT CONSTRUCTOR

Итак, **default constructor** — это блок кода, генерируемый компилятором Java, который:

1. имеет такой же access modifier, как класс
2. не имеет возвращаемого значения
3. имеет название, соответствующее названию класса
4. не имеет параметров
5. имеет пустое тело\*

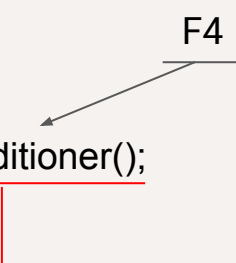
```
public class Conditioner {  
    ...  
    public Conditioner() { ← default constructor  
    }  
}
```

Примечание\*: не всегда, но в большинстве случаев.

# DEFAULT CONSTRUCTOR

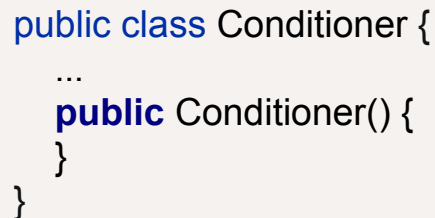
Раз эта конструкция существует, то она для чего-то нужна:

```
class ConditionerTest {  
    @Test  
    public void shouldUseConstructor() {  
        Conditioner conditioner = new Conditioner();  
    }  
}
```



A grey rectangular box containing the first code block. A line labeled 'F4' with an arrow points from the underlined `new Conditioner();` to the start of the second code block.

```
public class Conditioner {  
    ...  
    public Conditioner() {  
    }  
}
```



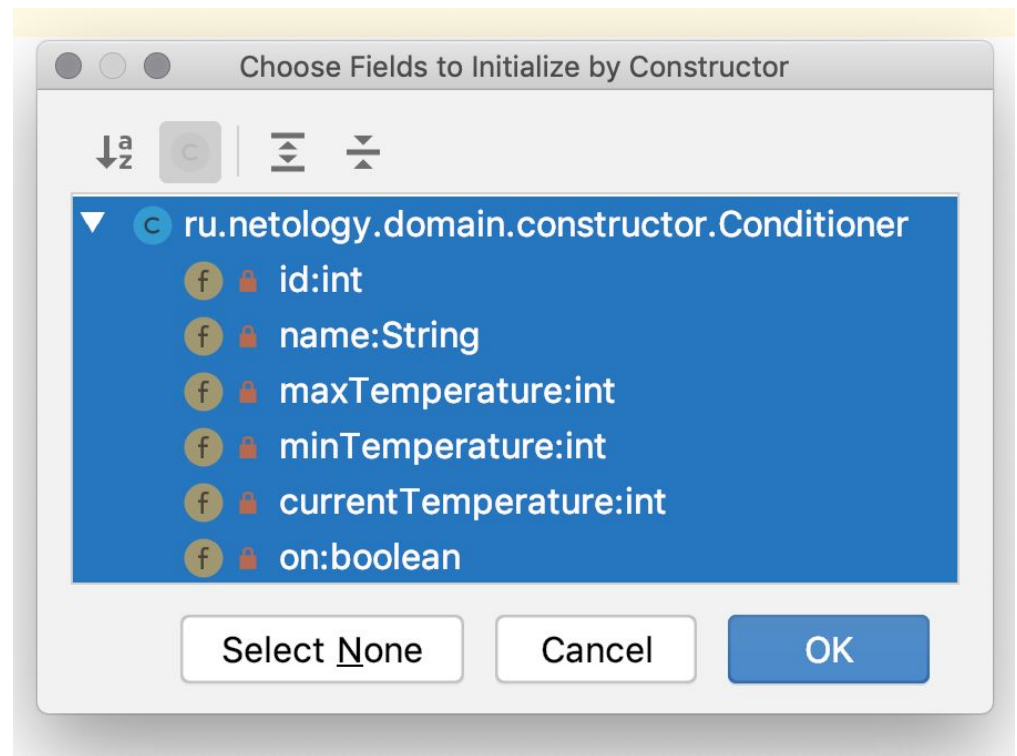
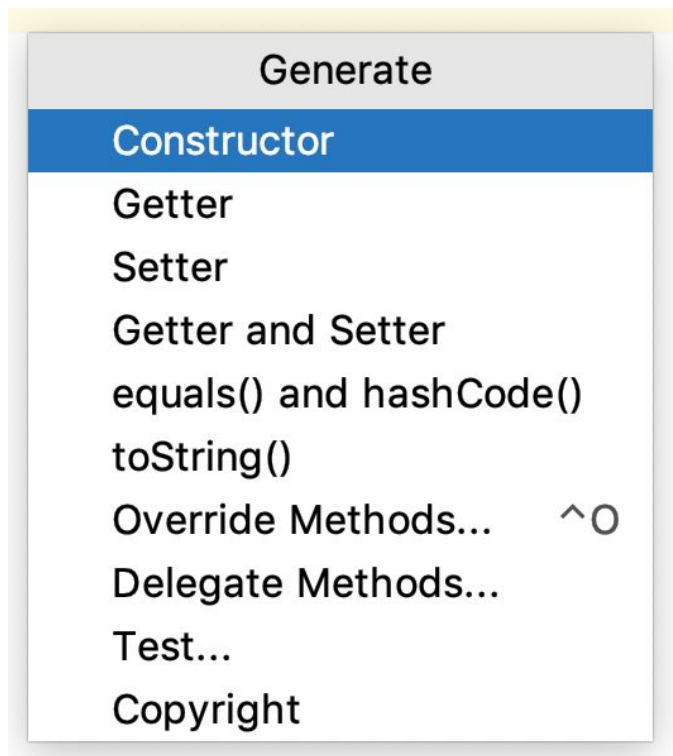
A grey rectangular box containing the second code block. A red line connects the underlined `new Conditioner();` from the first code block to the `public Conditioner() {` line in this block.

Когда мы пишем `new Conditioner()`, происходит вызов default constructor'a, который генерируется компилятором.

Именно поэтому мы не писали ничего в скобках (у дефолтного конструктора нет параметров).

# КОНСТРУКТОРЫ

Создадим свой конструктор: Alt + Insert (у IDEA есть конструктор), Ctrl + A, OK:



# CONSTRUCTOR

```
public class Conditioner {  
    ...
```

```
    public Conditioner(  
        int id,  
        String name,  
        int maxTemperature,  
        int minTemperature,  
        int currentTemperature,  
        boolean on  
    ) {  
        this.id = id;  
        this.name = name;  
        this.maxTemperature = maxTemperature;  
        this.minTemperature = minTemperature;  
        this.currentTemperature = currentTemperature;  
        this.on = on;  
    }  
}
```

мы немного изменили форматирование,  
чтобы умещалось на экран

Получился «один setter на всё».

Примечание\*: не обязательно делать конструктор на все поля, можно выбрать только нужные или не выбирать ни одного вообще.

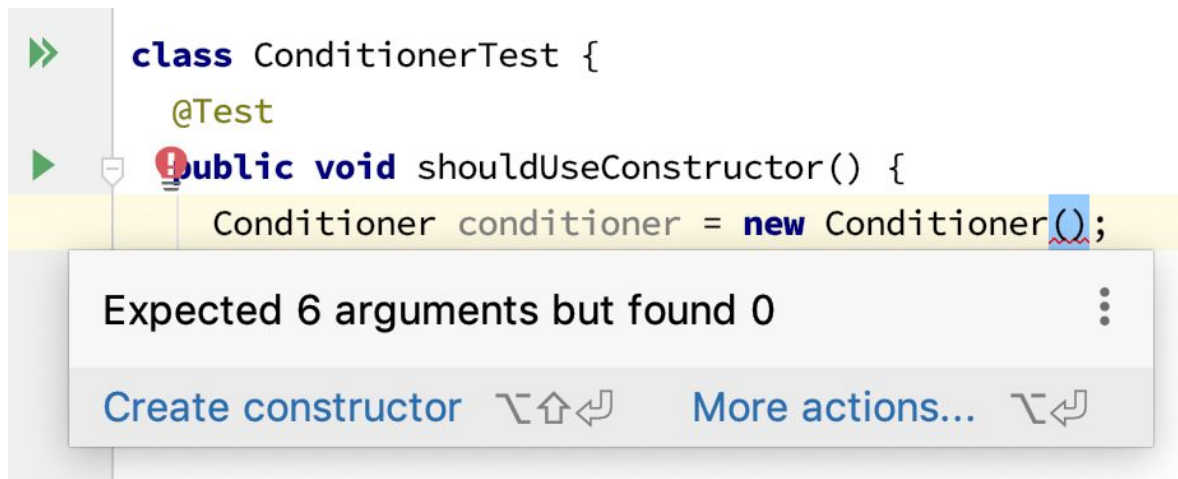


## NO ARGS & ALL ARGS

Такие конструкторы без параметров и со всеми параметрами, соответствующими полям, часто называют No Args Constructor и All Args Constructor соответственно.

# CONSTRUCTOR

После создания конструктора мы получим ошибку (класс теста перестанет компилироваться):



Компилятор Java говорит, что ожидается 6 параметров (а в нашем конструкторе их ровно 6), а передаём мы 0.



## DEFAULT CONSTRUCTOR

**Q:** Но до этого же работало? Куда девался default constructor?

**A:** Компилятор Java перестал его генерировать сразу после того, как мы объявили свой конструктор.

**Q:** Теперь мы не можем создать объект из класса, не указав все 6 параметров?

**A:** Совершенно верно, в этом плане конструктор выступает в качестве «правил», которые внешний мир (все остальные классы) должен выполнять для того, чтобы иметь возможность создавать объект этого класса.

# CONSTRUCTOR

Теперь мы можем сразу при создании указывать нужное состояние (и можно не вызывать по 10 раз метод увеличения температуры):

```
class ConditionerTest {  
    @Test  
    public void shouldUseConstructor() {  
        Conditioner conditioner = new Conditioner(  
            1,  
            "Winter Cold",  
            30,  
            10,  
            30,  
            true  
        );  
        assertEquals(30, conditioner.getCurrentTemperature());  
    }  
}
```





# DEFAULT CONSTRUCTOR

**Q:** Мы можем создать только один конструктор?

**A:** Хороший вопрос. На самом деле нет, мы можем создать несколько.



# **ПЕРЕГРУЗКА**



# OVERLOADING

**Перегрузка** (overloading) в Java — механизм, при котором два метода (включая конструкторы) в классе\* могут иметь **одинаковые имена, но разные сигнатуры**.

Примечание\*: при рассмотрении наследования и static мы уточним это определение.



# OVERLOADING

Сигнатура — это набор из декларации методов, состоящий из:

1. имени метода
2. списка параметров (количество, порядок и типы)

Обратите внимание:

1. в сигнатуру не входит тип возвращаемого значения
2. важно только количество, порядок и типы аргументов (а не их названия)

# CONSTRUCTOR

```
public class Conditioner {  
    ...  
  
    public Conditioner() {  
    }  
  
    public Conditioner(  
        int id,  
        String name,  
        int maxTemperature,  
        int minTemperature,  
        int currentTemperature,  
        boolean on  
    ) {  
        ...  
    }  
}
```

Разные сигнатуры  
(список параметров различается:  
у первого нет параметров, у второго — 6)

# OVERLOADING

Таким образом, мы предоставили две возможности для создания объектов нашего класса (у нас два конструктора):

```
class ConditionerTest {  
    @Test  
    public void shouldUseConstructor() {  
        Conditioner conditioner = new Conditioner(  
            1,  
            "Winter Cold",  
            30,  
            10,  
            30,  
            true  
        );  
        assertEquals(30, conditioner.getCurrentTemperature());  
    }  
  
    @Test  
    public void shouldUseNoArgsConstructor() {  
        Conditioner conditioner = new Conditioner();  
        assertEquals(22, conditioner.getCurrentTemperature());  
    }  
}
```

# OVERLOADING

**Q:** Зачем это может быть нужно?

**A:** По нескольким причинам:

1. чтобы каждый раз не придумывать новые имена методам
2. чтобы реализовать разную логику для разных параметров
3. чтобы предоставить значение параметров по умолчанию

**Q:** Как Java определяет, какой из двух методов использовать?

**A:** Ответ на этот вопрос не так прост (целиком он описан в спецификации). В упрощённой форме — по количеству, порядку и типам аргументов при вызове ищется наиболее подходящий.

---

# OVERLOADING & OVERRIDING

Не путайте термины Overloading и Overriding (в них даже количество букв разное 😊).

Перегрузка — это именно Overloading. Мы разберем Overriding на следующих лекциях.



# OVERLOADING

Пример использования перегрузки в библиотеке JUnit5:


```
/** <em>Assert</em> that {@code expected} and {@code actual} are equal. */  
public static void assertEquals(int expected, int actual) { AssertEquals.assertEquals(expected, actual); }  
  
/** <em>Assert</em> that {@code expected} and {@code actual} are equal. ...*/  
public static void assertEquals(Object expected, Object actual) { AssertEquals.assertEquals(expected, actual); }  
  
/** <em>Assert</em> that {@code expected} and {@code actual} are equal. ...*/  
public static void assertEquals(Object expected, Object actual, String message) {...}
```

Метод *assertEquals* перегруженный (меняется как количество, так и типы параметров), поэтому нам достаточно запомнить только его имя, а Java сама вызовет нужную версию.

# ВАЖНО

```
public Conditioner(  
    int id,  
    String name,  
    int maxTemperature,  
    int minTemperature,  
    int currentTemperature,  
    boolean on  
) {  
    ...  
}
```

```
public Conditioner(  
    int id,  
    String name,  
    int currentTemperature,  
    int maxTemperature,  
    int minTemperature,  
    boolean on  
) {  
    ...  
}
```



Пример выше не является перегрузкой, так как имена параметров не учитываются, только количество, типы и порядок.

Поэтому, данный код не скомпилируется.



# CODE GENERATION



## BOILERPLATE

**Q:** если конструкторы не содержат никакой логики (как getter'ы и setter'ы могут её не содержать), не слишком ли утомительно каждый раз писать практически одинаковый код?

**A:** верное замечание, такой код называют **boilerplate code** (кусочек кода, который встречается во множестве мест без изменения или с небольшими изменениями) и даже с генератором IDEA. Писать такой код неприятно.



# CODE GENERATION

Code Generation — термин, описывающий автоматическую генерацию кода.

Было бы здорово генерировать boilerplate код конструкторов и getter'ов/setter'ов.

И действительно, существует специальный инструмент (и плагин к Maven), который умеет генерировать такой код.

# LOMBOK

Lombok — библиотека, которая занимается генерацией кода.

С её использованием наш код будет выглядеть вот так:

<b>@NoArgsConstructor</b>	←	Конструктор без параметров
<b>@AllArgsConstructor</b>	←	Конструктор с параметрами на все поля*
<b>@Data</b>	←	getter'ы/setter'ы на все поля + ряд методов

```
public class Conditioner {  
    private int id;  
    private String name = «noname»;  
    private int maxTemperature = 30;  
    private int minTemperature = 15;  
    private int currentTemperature = 22;  
    private boolean on;  
}
```

Причём аннотации подозрительно похожи на те термины, которые мы используем (совпадение 😈?).



# LOMBOK

В разных командах существует разное отношение к Lombok, вплоть до полного неприятия.

Мы крайне настоятельно предостерегаем вас от подобного отношения к любым инструментам: если инструмент популярен, то вы должны уметь им пользоваться без эмоциональной окраски.

С Lombok вы познакомитесь в рамках выполнения ДЗ.



# ИТОГИ





## ИТОГИ

Сегодня мы поговорили о том, насколько важно иметь возможность устанавливать нужное нам состояние объектов. Это ключевой аспект тестопригодности (testability) всей системы.

Мы узнали, что иногда следует добавлять методы, которые позволяют легко устанавливать/проверять состояние объекта только для того, чтобы этот объект было легко тестировать.



# ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

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



**Задавайте вопросы и напишите отзыв о лекции!**

**ОКСАНА МЕЛЬНИКОВА**

 Оксана Мельникова