



# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ И ПРОЕКТИРОВАНИЕ



ОКСАНА МЕЛЬНИКОВА



**ОКСАНА МЕЛЬНИКОВА**

Software testing engineer





# План занятия

1. [Задача](#)
2. [ООП](#)
3. [Инициализация](#)
4. [Модификаторы доступа](#)
5. [Getters & Setters](#)
6. [this](#)
7. [Итоги](#)



# ЗАДАЧА



# ЗАДАЧА

Перед нами поставили следующую задачу: **спроектировать систему умного дома, которая будет управлять домашними устройствами.**

Среди устройств:

1. Кондиционер
2. Холодильник
3. Телевизор



# ЗАДАЧА

Важно: мы смотрим на систему с точки зрения возможности управления ею именно конечным пользователем.

А именно: **как он может управлять** этими устройствами.

Например, кондиционер можно вкл/выкл, изменять температуру.



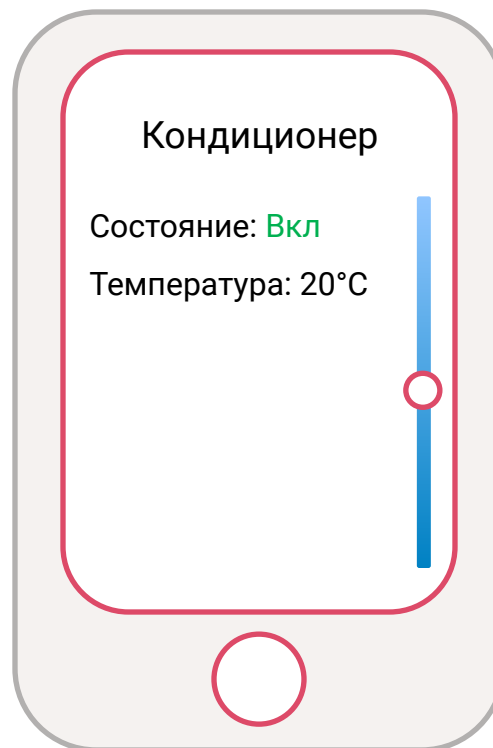
# АБСТРАКТНОЕ МЫШЛЕНИЕ

Очень сложно "проектировать" что-то, не имея картинки, как это должно работать, и не имея чёткого описания.

К этому нужно привыкать: строить в голове "абстрактную" картинку и по ней решать задачу.

# АБСТРАКТНОЕ МЫШЛЕНИЕ

Например, мы можем представить, что управлять нашими объектами можно будет через мобильное приложение, выставляя для кондиционера температуру:








# ЗАДАЧА

Важно: нам не нужно создавать самим кондиционер, холодильник и телевизор.

Нам нужно их описать таким образом, чтобы можно было с ними программно (через программу) взаимодействовать.



# **ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ**



# ООП

ООП (Объектно-ориентированное программирование) - подход к моделированию реального мира в программировании, когда мы всё **описываем в виде объектов**, обладающих **определёнными свойствами и поведением**.

В нашем примере и кондиционер, и холодильник, и телевизор - это объекты.



# СВОЙСТВА И МЕТОДЫ

*Вопрос к аудитории: давайте определим свойства для*

- кондиционера
- холодильника
- телевизора

*Вопрос к аудитории: давайте определим методы для*

- кондиционера
- холодильника
- телевизора



# ООП

Если подумать, то можно выделить тысячи свойств для наших объектов: от размера до материалов, из которых они сделаны.

То же самое с методами.

Но давайте подумаем, все ли они имеют значение для нашей конкретной задачи?



# АБСТРАКЦИЯ

**Абстракция** - выделение ключевых (имеющих значение для решаемой задачи) свойств и связей.

Именно абстракция позволяет нам строить управляемые системы.



# АБСТРАКЦИЯ

Например, для кондиционера ключевыми свойствами являются:

- max и min температуры
- включен он или нет
- текущая температура

При этом, ряд свойств можно менять (текущая температура и вкл/выкл), а ряд других - нельзя (max и min температура задаются на заводе).



# СВОЙСТВА

Остановимся пока на свойствах и попробуем описать наш объект на Java, чтобы с ним можно программно взаимодействовать.

Создадим проект по нашему стандартному шаблону (Maven + Surefire + JUnit5).



# КЛАССЫ

Java предлагает нам концепцию классов, на базе которых можно создавать объекты:

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

```
public class Conditioner {  
    String name;  
    int maxTemperature;  
    int minTemperature;  
    int currentTemperature;  
    boolean on;  
}
```

} Поля (**fields**) объектов этого класса

# PROPERTY VS FIELD

В Java для обозначения `name`, `min/maxTemperature` и других используют термин **поле (field)**, а не **свойство (property)**.

Поле (field) отвечает за хранение данных в объекте. Поле описывается в классе (см. предыдущий слайд) и хранит своё собственное значение для каждого объекта.

Начиная с этого слайда (и сегодняшней лекции), мы будем говорить **поле (field)**, а значение термина **свойство (property)** мы поясним позднее.



# СОСТОЯНИЕ

**Состоянием объекта** мы называем текущее (установленное в данный момент) значение всех полей.

Мы уже работали с объектами без состояния, теперь пришла пора научиться работать и с теми, которые имеют состояние.

Состояние вносит **дополнительную сложность** как в систему, так и в автотесты. В зависимости от состояния, поведение одного и того же объекта может меняться кардинальным образом.

Всё как в реальной жизни: например, если у человека хорошее настроение - у него одно поведение, если плохое - другое.



# DOMAIN

**Q:** почему класс `Conditioner` находится в `package domain`?

**A:** под *domain* чаще всего понимают предметную область, для которой выполняется моделирование. Поскольку наш класс описывает сущность из предметной области, то мы положили этот класс в `package domain`.

# КЛАСС

**Класс - это "фабрика" по созданию объектов.**

**Q:** что такое фабрика?

**A:** в программировании очень любят "яркие" аналогии, чтобы абстрактные вещи воспринимались проще

Фабрики в реальной жизни, например, кондитерская фабрика, занимаются "штампованием" (**массовым производством**) **однотипных объектов\*** (например, шоколад "Алёнка").

Примечание\*: используется упрощённая аналогия (фабрика может производить только один тип объектов).



# КЛАСС

Все шоколадки однотипные, но при этом каждая из них:

1. Является обособленным объектом (если у одной шоколадки откусить "кусочек", это никак не повлияет на другие шоколадки)
2. Имеет "собственную судьбу" (жизненный цикл)

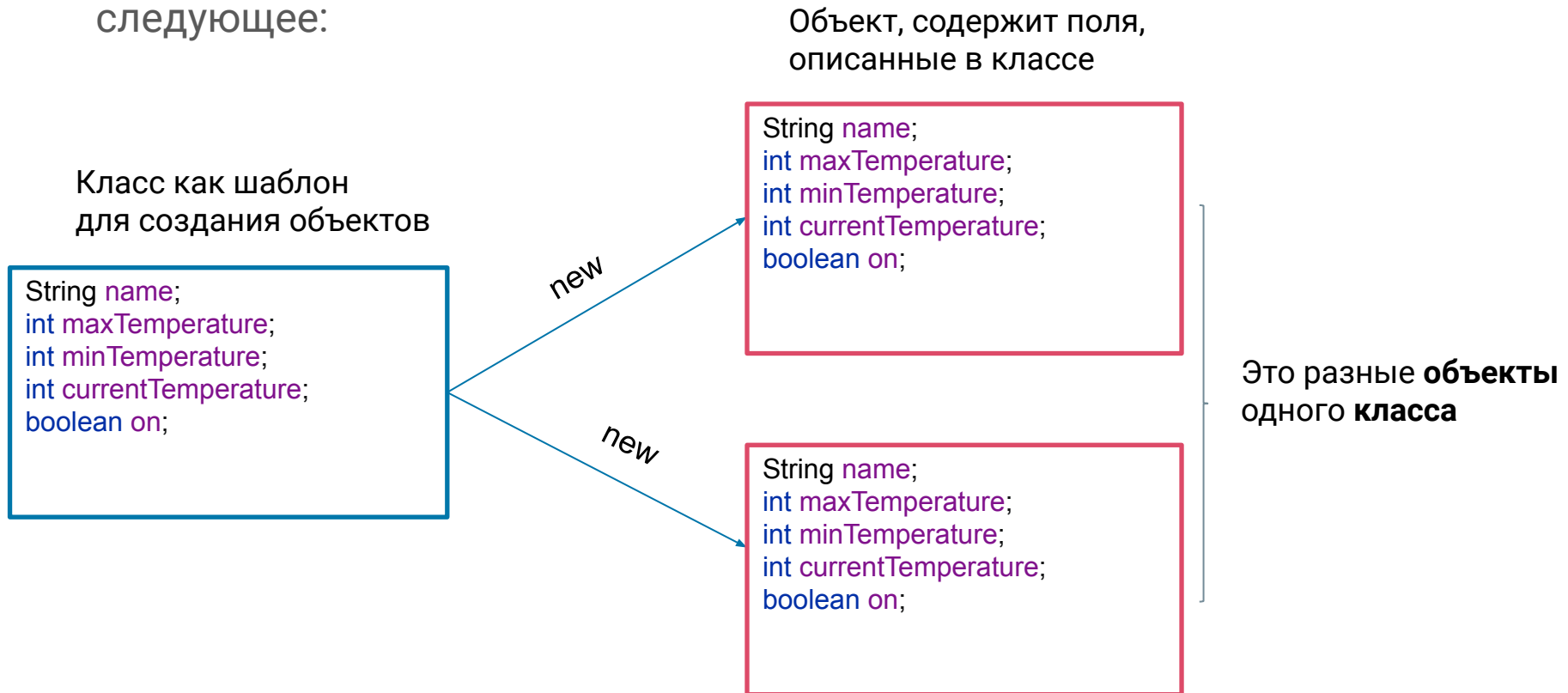
# СОЗДАНИЕ ОБЪЕКТА

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

Как вы уже знаете, объект создаётся из класса с помощью ключевого слова **new**.

# СОЗДАНИЕ ОБЪЕКТА

Важно понимать, что при создании класса происходит примерно следующее:





---

# ОБЪЕКТЫ

**Q:** зачем нам нужны объекты? Мы могли ведь просто хранить всё в переменных.

**A:** объекты нужны (в том числе), чтобы управлять набором данных разного типа, как единым целым.

Именно так устроено наше мышление, мы воспринимаем окружающие нас вещи **целостно, а не по отдельности**.

Например, мы чаще говорим "автомобиль движется по трассе ", но не говорим "колёса, кузов и двигатель движутся по трассе".

# АБСТРАКЦИЯ

Выбирая разный уровень детализации мы можем работать на разных уровнях абстракции, например:

- **Дом** - это один уровень абстракции
- **Улица** - более высокий
- **Город, Страна** и т.д.

В зависимости от того, какую конкретно задачу мы решаем, мы сами выбираем необходимый уровень абстракции.

Например, если наша задача построить маршрут автомобилисту для проезда, то дома и улицы имеют значение. А если мы строим человеку авиа-маршрут перелёта (см. сайты покупки авиа-билетов), то важны только города вылета и прилёта.



# ИНИЦИАЛИЗАЦИЯ

# ИНИЦИАЛИЗАЦИЯ

Когда мы работали с переменными, то брали за правило инициализировать переменные перед их использованием.


С полями объектов всё немного по-другому: 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

# ИНИЦИАЛИЗАЦИЯ

```
@Test
public void shouldInitFieldToZeroValues() {
    Conditioner conditioner = new Conditioner();
    assertNull(conditioner.name);
    assertEquals(0, conditioner.maxTemperature);
    assertEquals(0, conditioner.minTemperature);
    assertEquals(0, conditioner.currentTemperature);
    assertFalse(conditioner.on);
}
```

Обращение к свойству

Как видно из примера, обращение к свойству объекта осуществляется с помощью оператора : `conditioner.on`

Тест зелёный, а значит всё действительно так.



# NULL

`null` - специальный литерал, указывающий на то, что имя не ссылается ни на какой объект.

Т.е. у полей примитивного типа всегда есть значения, а поля "объектного" (их называют ссылочного) типа содержат значение `null`.

Вы достаточно часто будете сталкиваться с `null`, поэтому во избежание проблем с ним нужно хорошо познакомиться.

# NPE

**Важно:** `null` - это "маркер" отсутствия объекта, а значит на нём нельзя вызывать методы или пытаться получить доступ к полям.

Представьте, что вы вызываете лифт, двери открываются, а кабины лифта нет, только шахта. Попытка зайти в такой "лифт" не увенчается ничем хорошим:

```
@Test
public void shouldThrowNPE() {
    Conditioner conditioner = new Conditioner();
    assertEquals(0, conditioner.name.length());
}
```

! Tests failed: 1 of 1 test – 10 ms

`java.lang.NullPointerException` ← Сокращённо NPE

# NPE

! Tests failed: 1 of 1 test – 10 ms

`java.lang.NullPointerException` ← Сокращённо NPE

**NullPointerException (NPE)** - исключительная ситуация, возникающая при попытке обращения к null, как к объекту.

При возникновении такой ситуации JVM аварийно будет завершать работу вашего приложения, а автотесты будут падать\*.

Примечание\*: на самом деле можно обрабатывать подобные ситуации (что и делает JUnit, но об этом мы будем говорить позже).



# @Disabled

```
@Test
@Disabled
public void shouldThrowNPE() {
    Conditioner conditioner = new Conditioner();
    assertEquals(0, conditioner.name.length());
}
```

Аннотация **@Disabled** позволяет отключить "падающий" тест. Мы пометим этой аннотацией метод, в котором возникла исключительная ситуация (только для учебных целей).

На практике использовать эту аннотацию стоит крайне редко, т.к. зачем вам хранить неиспользуемые тесты? Удаляйте.



# ДОСТУП К ПОЛЯМ

# ДОСТУП К ПОЛЯМ

```
@Test
public void shouldChangeFields() {
    Conditioner conditioner = new Conditioner();
    assertEquals(0, conditioner.currentTemperature);
    conditioner.currentTemperature = -100;
    assertEquals(-100, conditioner.currentTemperature);
}
```

Есть две формы доступа к полям:

1. На чтение: `conditioner.currentTemperature`
2. На запись: `conditioner.currentTemperature = -100`

Причём тест зелёный, а значит мы смогли записать значение в поле.



## -100

**Q:** стоп-стоп, но мы же записали совсем "невалидное" значение в поле! Оно меньше минимального значения!

**A:** совершенно верно. Поля - это просто поля, они не содержат логики и в данный момент в них можно записывать всё, что допустимо правилами Java.



## РАЗГРАНИЧЕНИЕ ДОСТУПА

В реальной жизни, конечно, всё не так: **кондиционер закрыт "коробкой"**, а у вас **есть пульт, с которого вы можете выставлять температуру** (причём **вы не можете** выставить температуру больше максимальной или меньше минимальной).

Причём он закрыт коробкой специально, чтобы вы **ничего не испортили внутри**.



# РАЗГРАНИЧЕНИЕ ДОСТУПА

Java нам предоставляет возможность ограничить прямой доступ к полям вне методов класса, в котором эти поля объявлены (например, запретить из теста или Main).

Обратите внимание: Java разрешает ограничить доступ только одновременно на чтение и запись\*.

Примечание\*: чуть позже мы поговорим о том, как сделать поля доступными только для чтения.

# МОДИФИКАТОРЫ ДОСТУПА

Модификаторы доступа - ключевые слова языка Java, определяющие доступ к некоторым элементам (в том числе полям).

В Java 8\* определены следующие модификаторы доступа:

- public (ключевое слово **public**)
- protected (ключевое слово **protected**)
- package-private (по умолчанию, ключевое слово отсутствует)
- private (ключевое слово **private**)

Примечание\*: в Java 9 внедрена система модулей и модификаторы доступа стали иметь расширенное значение, но система эта до сих пор не очень популярна, поэтому мы её рассматривать не будем.

# МОДИФИКАТОРЫ ДОСТУПА

Модификаторы доступа в применении к полям:

- **public** - доступно на чтение и запись отовсюду
- **protected** - доступно на чтение и запись в том же пакете и наследниках (в том числе из других пакетов)\*
- **package-private** - доступно на чтение и запись в том же пакете
- **private** - доступно на чтение и запись только внутри класса

Примечание\*: про наследование и `protected` мы будем говорить на следующих лекциях.





# МОДИФИКАТОРЫ ДОСТУПА

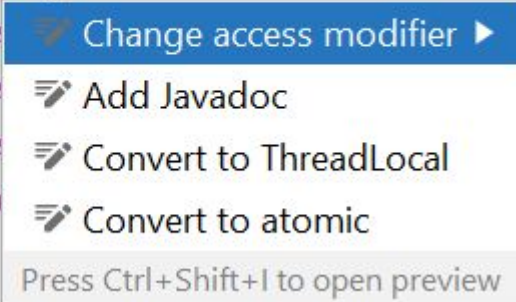
*Вопрос к аудитории: почему мы сейчас можем писать в поля объектов conditioner?*

# МОДИФИКАТОРЫ ДОСТУПА

Мы могли писать в поля объекта, потому что в классе поля имеют модификатор доступа package-private.

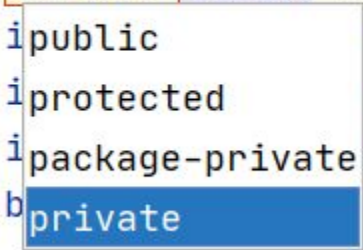
Используя наше правило (все поля - **private**) поменяем модификаторы доступа для полей (Alt + Enter):

```
public class Conditioner {  
    String name;  
    int maxTemperature;  
    int minTemperature;  
    int currentTemperature;  
    boolean on;  
}
```



The image shows an IDE context menu for the `String name;` field. The menu is open, and the first option, "Change access modifier", is highlighted with a blue background and a right-pointing arrow. Other options include "Add Javadoc", "Convert to ThreadLocal", and "Convert to atomic". At the bottom of the menu, it says "Press Ctrl+Shift+I to open preview".

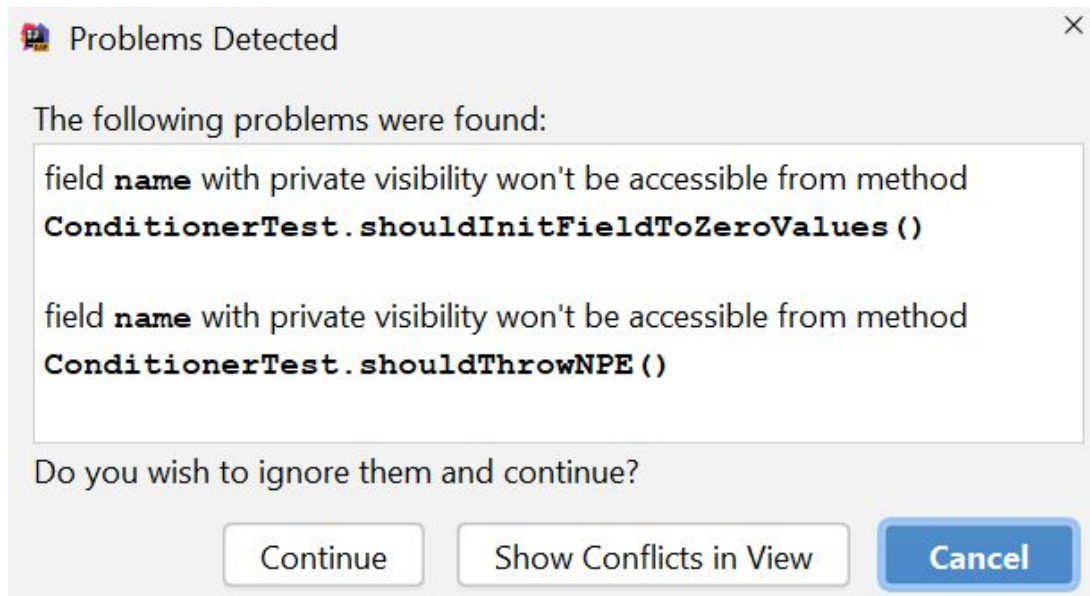
```
public class Conditioner {  
    private String name;  
    public int maxTemperature;  
    protected int minTemperature;  
    package-private int currentTemperature;  
    boolean on;  
}
```



The image shows the same IDE context menu for the `String name;` field, but now the `private` access modifier is selected from the list. The list includes `public`, `protected`, `package-private`, and `private`, with `private` highlighted in blue.

# МОДИФИКАТОРЫ ДОСТУПА

Но при попытке модификации поля получим предупреждение IDEA:



Т.к. `private` запретит прямой доступ из теста.

Но что тогда делать? Как получать доступ к полям?



# GETTERS & SETTERS

# GETTERS & SETTERS

Чтобы сохранить в исходниках проектов и первый вариант (без модификаторов доступа), и второй, мы создадим новый класс:

```
package ru.netology.domain;

public class ConditionerAdvanced {
    private String name;
    private int maxTemperature;
    private int minTemperature;
    private int currentTemperature;
    private boolean on;
}
```



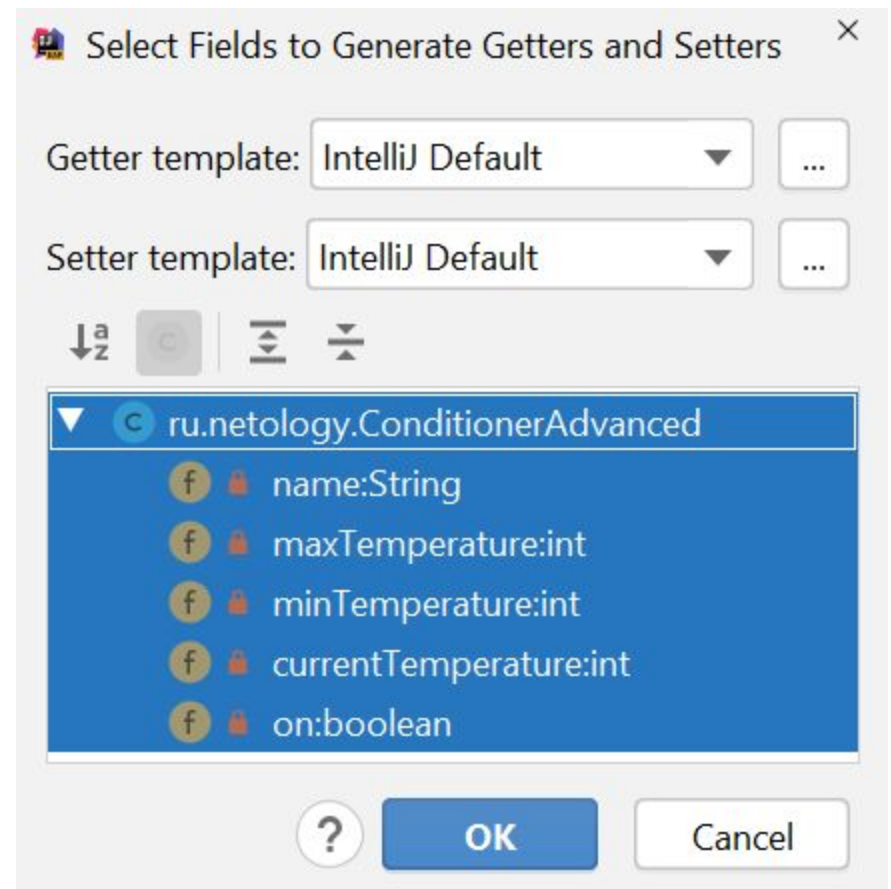
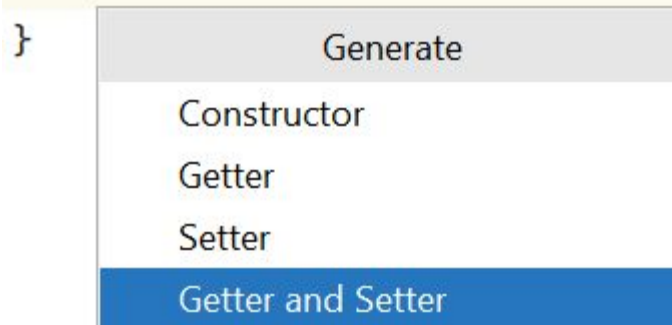
# GETTERS & SETTERS

В Java используется концепция **getter'ов** и **setter'ов** - специальных методов, задача которых - получать текущее значение поля (get) и устанавливать (set).

# GETTERS & SETTERS

Поскольку использование getter'ов и setter'ов - стандартная практика, в IDEA зашит shortcut для их генерации (Alt + Insert):

```
public class ConditionerAdvanced {  
    private String name;  
    private int maxTemperature;  
    private int minTemperature;  
    private int currentTemperature;  
    private boolean on;  
}
```



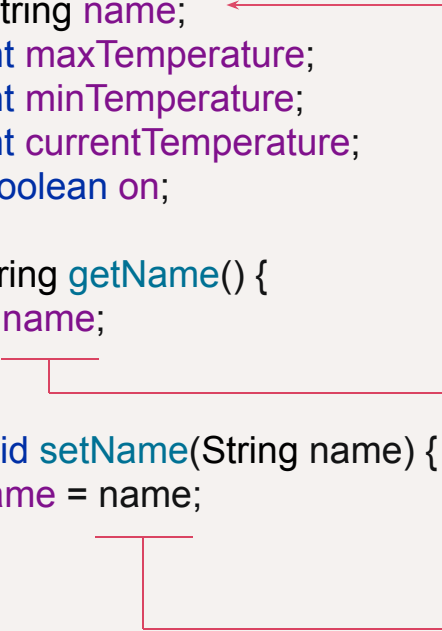
# GETTERS & SETTERS

```
package ru.netology.domain;

public class ConditionerAdvanced {
    private String name;
    private int maxTemperature;
    private int minTemperature;
    private int currentTemperature;
    private boolean on;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    ...
}
```



getName - getter

setName - setter



# GETTERS

Таким образом, getter - это всего лишь метод, который возвращает значение приватного поля:

```
public String getName() {  
    return name;  
}
```

**Важно:** стремитесь называть getter **именно в такой конвенции\*** (соглашении) **get + FieldName**, т.к. на это будут рассчитывать **большинство инструментов!**

Примечание\*: вы часто будете встречать этот термин (convention), он означает, что в определённом сообществе приняты определённые соглашения, например, по именованию getter'ов и setter'ов.

# SETTERS

С setter'ами всё немного сложнее:

```
public void setName(String name) {  
    this.name = name;  
}
```

Перепишем в эквивалентном варианте, чтобы стало понятнее (а с `this` разберёмся чуть позже):


```
public void setName(String newName) {  
    name = newName;  
}
```

Таким образом, setter - это всего лишь метод, который устанавливает новое значение приватного поля

**Важно:** стремитесь называть setter **именно в такой конвенции** `set + FieldName`, т.к. на это будут рассчитывать большинство инструментов!

# VOID

В setter'e тип возвращаемого значения указан как **void**:



```
public void setName(String name) {  
    this.name = name;  
}
```

**void** указывает на то, что метод ничего не возвращает (т.е. делает какую-то работу, но обратно никакого результата не возвращает).

Поэтому нельзя использовать оператор присваивания при вызове:

```
ConditionerAdvanced conditioner = new ConditionerAdvanced();  
void value = conditioner.setName("Кондей");
```

Так нельзя!

# GETTERS & SETTERS

Удостоверяемся, что setter'ы и getter'ы действительно работают:

```
@Test
public void shouldGetAndSet() {
    ConditionerAdvanced conditioner = new ConditionerAdvanced();
    String expected = "Кондишн";

    assertNull(conditioner.getName());
    conditioner.setName(expected);
    assertEquals(expected, conditioner.getName());
}
```

---

## Q & A

**Q:** но зачем это всё? Ведь результат такой же (как и при прямой записи в поле), а кода стало гораздо больше!

**A:** это нужно для того, чтобы не ломать "совместимость" систем, когда вы захотите видоизменить эти методы (особенно setter), добавив в них какую-то логику (см. пример на следующем слайде).

Кроме того, это общее соглашение: использовать getter'ы + setter'ы для доступа к полям извне класса.


## ЛОГИКА В SETTER'E

Например, нельзя выставлять температуру больше максимальной и ниже минимальной:

```
public void setCurrentTemperature(int currentTemperature) {  
    if (currentTemperature > maxTemperature) {  
        return;  
    }  
    if (currentTemperature < minTemperature) {  
        return;  
    }  
    this.currentTemperature = currentTemperature;  
}
```

# EARLY EXIT

Early Exit - подход, при котором мы проверяем условие и, если результат проверки нас не устраивает, выходим из функции:



```
public void setCurrentTemperature(int currentTemperature) {  
    if (currentTemperature > maxTemperature) {  
        return;  
    }  
    if (currentTemperature < minTemperature) {  
        return;  
    }  
    // здесь уверены, что все проверки прошли  
    this.currentTemperature = currentTemperature;  
}
```

**return** завершает выполнение функции, возвращая управления вызывающей функции (показать в дебаггере).

## БЕЗ EARLY EXIT

Без early exit мы получим меньше строчек кода, но "визуально" его станет читать тяжелее:

Вложенность кода

```
public void setCurrentTemperature(int currentTemperature) {  
    if (currentTemperature > maxTemperature) {  
        if (currentTemperature < minTemperature) {  
            this.currentTemperature = currentTemperature;  
        }  
    }  
}
```

Общая статистика говорит о том, что чем больше вложенность кода, тем выше частота ошибок в этом коде.

Поэтому мы будем требовать от вас использования early exit.



## Q & A

Q: нужно ли писать тесты на getter'ы и setter'ы?

A: только если в них есть логика. Т.е. в наших примерах на `get/setName` не нужно, а вот на `setCurrentTemperature` нужно, при этом `getCurrentTemperature` будет использоваться для проверки правильности установки значения.

Этот тест **не нужен** (если в `get/setName` нет логики):

```
@Test
public void shouldGetAndSet() {
    ConditionerAdvanced conditioner = new ConditionerAdvanced();
    String expected = "Кондишн";

    assertNull(conditioner.getName());
    conditioner.setName(expected);
    assertEquals(expected, conditioner.getName());
}
```

не делайте так

---

## Q & A

Q: а как мы можем проверить, что температура действительно установилась? Ведь getter может "врать".

A: проверить можно в дебаггере. Но тесты писать не нужно (в большинстве случаев)\*, т.к. **тесты проверяют поведение, а не внутреннюю реализацию.**

Примечание\*: как мы и говорили, нет единого мнения. Существует возможность и инструменты, которые позволяют "залезть внутрь" и посмотреть (либо установить) значение приватного поля напрямую.

Но на текущий момент более распространено мнение "что нам не важно внутреннее устройство, нам важно поведение".



# PROPERTY

Приватное поле + getter/setter к нему принято называть **property**.

При этом не обязательно, что должны быть и getter, и setter.



**THIS**

# THIS

**this** - это ключевое слово, которое используется для указания на объект, метод которого сейчас вызывается\*:

```
@Test
public void shouldGetAndSet() {
    ConditionerAdvanced conditioner = new ConditionerAdvanced();
    String expected = "Кондишн";

    assertNull(conditioner.getName());
    conditioner.setName(expected);

    assertEquals(expected, conditioner.getName());
}
```

```
public void setName(String name) {
    this.name = name;
}
```

Примечание\*: будут и другие значения этого ключевого слова.

# THIS

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

```
public class ConditionerAdvanced {
```

```
    private String name;
```

```
    private int maxTemperature;
```

```
    private int minTemperature;
```

```
    private int currentTemperature;
```

```
    private boolean on;
```

```
    public String getName() {
```

```
        return name;
```

```
    }
```

```
    public void setName(String name) {
```

```
        this.name = name;
```

```
    }
```

```
    ...
```

```
}
```

ничего не скрывается  
**this не нужен**

это имя скрывает  
поле (shadows)

**this** позволяет "добраться"  
до поля



# THIS

В отличие от локальных переменных, параметры методов могут иметь те же имена, что и поля (как в нашем примере).

При этом имя параметра "скрывает" (shadows) поле, т.е. везде внутри метода **name** - это именно параметр, а не поле.

Чтобы добраться до поля и нужен **this**.

**this** - это просто ключевое слово, позволяющее не придумывать имя (единообразно для всех классов).

# THIS

Пример из жизни: представим, что вас зовут Вася и вы участвуете в проекте в роли тестировщика.

На совещании руководитель проекта говорит: "Задача Василия - протестировать функцию оплаты по номеру телефона":

```
vasya.setTask("Протестировать функцию оплаты");
```

При этом Вася внутри себя думает\*: "Моя задача - протестировать...":

```
public void setTask(String task) {  
    this.task = task;  
}  
    моя задача
```



---

## Q & A

Q: вот такая форма записи была понятнее и проще:

```
public void setName(String newName) {  
    name = newName;  
}
```

A: это верно, но:

1. IDEA сама за вас генерирует getter/setter
2. Почти все используют вариант с `this`, поэтому мы тоже его будем использовать.

---

## Q & A

**Q:** почему тогда везде не писать [this](#)?

**A:** во-первых, это избыточно, во-вторых, так почти никто не делает.



# ИТОГИ



# ИТОГИ

Сегодня мы поговорили об ООП и рассмотрели базовые конструкции ООП в Java:

- классы
- объекты
- поля
- модификаторы доступа
- getters & setters



## ИТОГИ

Вам может показаться, что это всё сложно. На самом деле, это лишь вопрос практики и восприятия.

Обязательно практикуйтесь и выполняйте ДЗ.



# 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