

# Object oriented programming

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

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

Именно абстракция позволяет нам строить управляемые системы.

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.

## 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.

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

**Q:** что такое фабрика?

**A:** в программировании очень любят "яркие" аналогии, чтобы

абстрактные вещи воспринимались проще

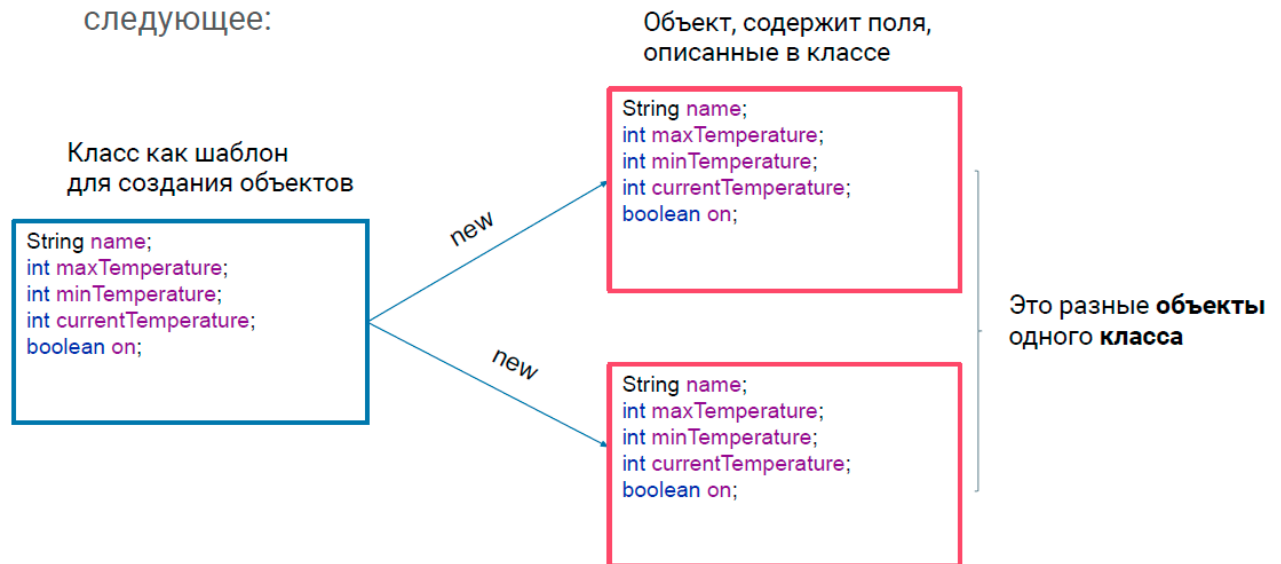
Фабрики в реальной жизни, например, кондитерская фабрика, занимаются "штампованием" **(массовым производством) однотипных объектов\***

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

Как вы уже знаете, объект создаётся из класса с помощью

ключевого слова `new`.

Важно понимать, что при создании класса происходит примерно следующее:



**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.

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

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.

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

**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\*.

Аннотация **@Disabled** позволяет отключить "падающий" тест

Есть две формы доступа к полям:

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

## 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\*.

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`)

### 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

## GETTERS & SETTERS

В Java используется концепция **getter'ов** и **setter'ов** - специальных методов, задача которых - получать текущее значение поля (`get`) и устанавливать (`set`). IDEA зашит shortcut для их генерации (`Alt + Insert`)

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

**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.


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!

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

## EARLY EXIT

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

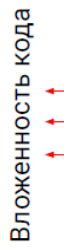


```
public void setCurrentTemperature(int currentTemperature) {  
    if (currentTemperature > maxTemperature) {  
        return;  
    }  
    if (currentTemperature < minTemperature) {  
        return;  
    }  
    // здесь уверены, что все проверки прошли  
    this.currentTemperature = currentTemperature;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

`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:



```
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.

# 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;
}
```



`this` - это просто ключевое слово, позволяющее не придумывать имя.

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

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

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

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.

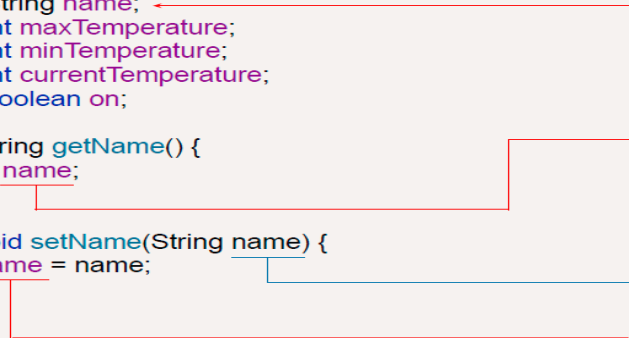
# THIS

```
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