

# ИНТЕРФЕЙСЫ ДЛЯ ОРГАНИЗАЦИИ МАЛОЙ СВЯЗНОСТИ ОБОБЩЁННОЕ ПРОГРАММИРОВАНИЕ (GENERICS)

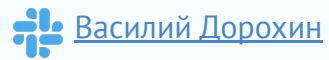


ВАСИЛИЙ ДОРОХИН



**ВАСИЛИЙ ДОРОХИН**

QuadCode, QA Engineer



---

# План занятия

1. [Задача](#)
2. [Утилитные классы](#)
3. [Сортировка](#)
4. [Интерфейсы](#)
5. [Comparable](#)
6. [Generics](#)
7. [Итоги](#)



# ЗАДАЧА



# ЗАДАЧА

Сортировка элементов по различным критериям — одна из ключевых функциональностей современных систем.

Например, товары мы можем сортировать по цене, пользовательскому рейтингу и т.д.

А авиабилеты — по цене, длительности перелёта, количеству пересадок и т.д.

---

# ЗАДАЧА

```
@NoArgsConstructor  
@AllArgsConstructor  
@Data  
public class Product {  
    private int id;  
    private String name;  
    private int price;  
    private double rating;  
}
```

Так как можно сортировать только набор объектов, нам понадобится массив, как минимум, с двумя элементами.

# ЗАДАЧА

Но теперь вопрос: как сортировать?

```
class ProductsTest {  
    private Product first = new Product( id: 1, name: "First", price: 1_000, rating: 3.5);  
    private Product second = new Product( id: 2, name: "Second", price: 2_000, rating: 4.5);  
    private Product third = new Product( id: 3, name: "Third", price: 3_000, rating: 5.0);  
  
    @Test  
    public void shouldSortById() {  
        Product[] expected = new Product[]{first, second, third};  
        Product[] actual = new Product[]{third, first, second};  
        // TODO: sort  
        assertEquals(expected, actual);  
    }  
}
```

Массивы — достаточно специфический тип данных (у него нет как такового класса, от него нельзя отнаследоваться), и в нём нет никаких методов сортировки.

# TODO

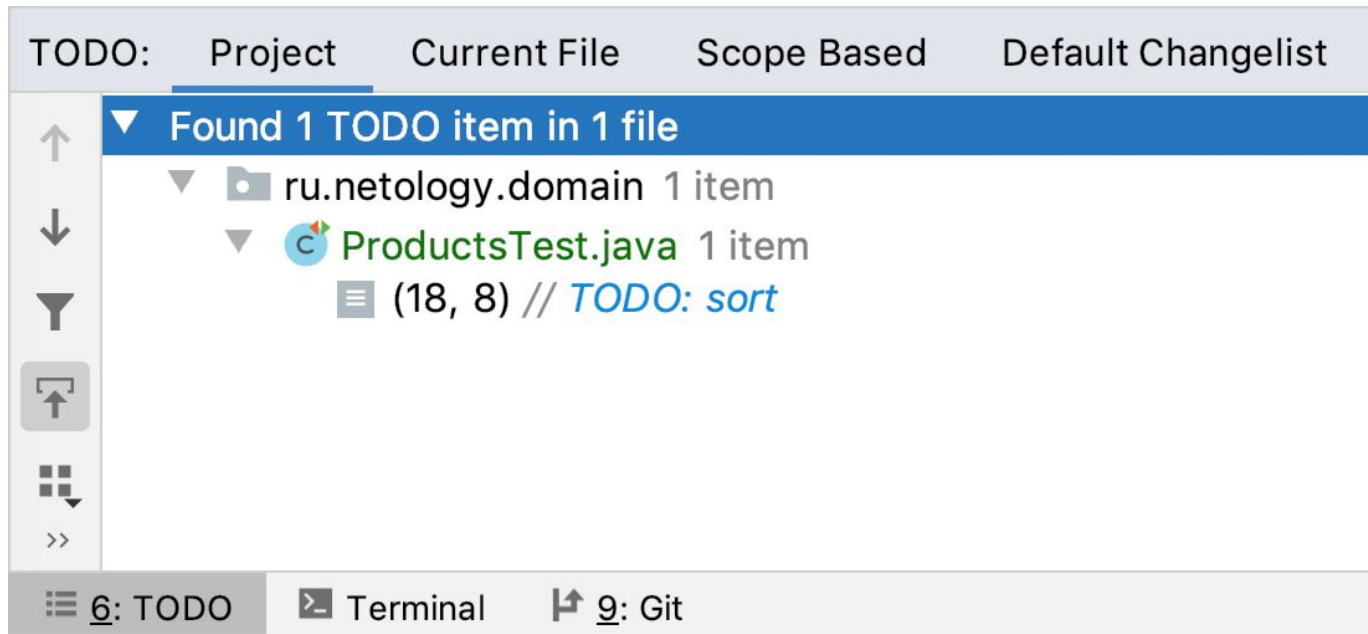
```
class ProductsTest {  
    private Product first = new Product( id: 1, name: "First", price: 1_000, rating: 3.5);  
    private Product second = new Product( id: 2, name: "Second", price: 2_000, rating: 4.5);  
    private Product third = new Product( id: 3, name: "Third", price: 3_000, rating: 5.0);  
  
    @Test  
    public void shouldSortById() {  
        Product[] expected = new Product[]{first, second, third};  
        Product[] actual = new Product[]{third, first, second};  
        // TODO: sort  
        assertEquals(expected, actual);  
    }  
}
```

В IDEA комментарии, начинающиеся с *TODO* или *FIXME*, интерпретируются особым образом: они аккуратно собираются в панельку TODO (Alt + 6) и анализируются перед коммитом.



# TODO

Обязательно пользуйтесь этой возможностью и помечайте такими комментариями места в коде, к которым нужно вернуться и что-то доделать:





# УТИЛИТНЫЕ КЛАССЫ



# УТИЛИТНЫЕ КЛАССЫ

В рамках ООП мы с вами говорили о том, что методы не могут существовать сами по себе. Нужен объект, на котором эти методы можно вызывать.

В итоге схема была такая:

1. Объявляем класс и методы
2. Создаём объект
3. Вызываем на объекте метод

# УТИЛИТНЫЕ КЛАССЫ

Это не всегда нужно и удобно: допустим, если метод не использует никаких полей объекта, то получается «он сам по себе» и объект для его работы вроде бы и не нужен.

Мы можем устранить шаг создания объекта, создав так называемые **статические методы** (помеченные ключевым словом `static`), — это такие методы, которым не нужен объект:

1. Объявляем класс со статическими методами
2. Вызываем методы (не создавая объект)

# УТИЛИТНЫЕ КЛАССЫ

Эталонный пример в этом плане — класс `Math`.

Он содержит только статические методы и статические поля.

Мы можем обращаться к полям и методам следующим образом:

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

```
public class MathTest {
```

```
    @Test
```

```
    public void shouldCalculateSin() {
```

```
        double result = Math.sin(Math.PI / 2);
```

```
        double expected = 1.0;
```

```
        double delta = 0.01;
```

```
        assertEquals(expected, result, delta);
```

```
    }
```

```
}
```

ИмяКласса.СтатическийМетод(...)  
или  
ИмяКласса.СтатическоеПоле(...)

# УТИЛИТНЫЕ КЛАССЫ

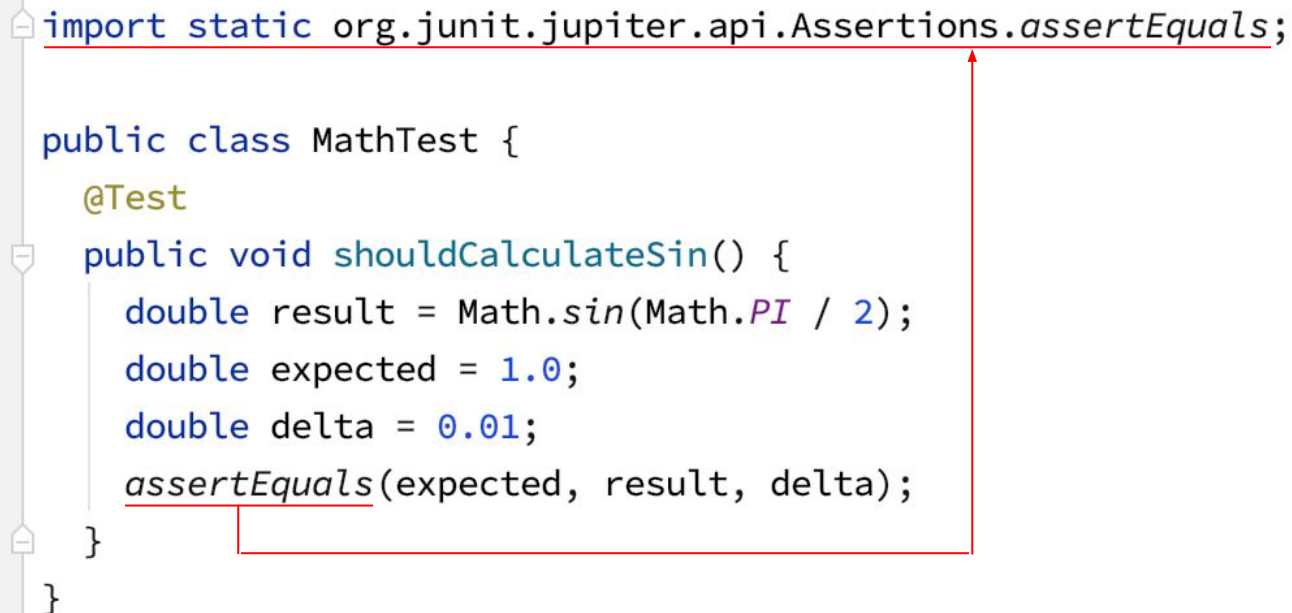
Классы, которые содержат только статические методы, принято называть утилитными (в их названии есть буква s на конце: Assertions, Arrays).

Например, все наши assert'ы — это на самом деле статические методы класса Assertions:

```
public class MathTest {  
    @Test  
    public void shouldCalculateSin() {  
        double result = Math.sin(Math.PI / 2);  
        double expected = 1.0;  
        double delta = 0.01;  
        Assertions.assertEquals(expected, result, delta);  
    }  
}
```

# STATIC IMPORT

Мы так никогда не писали, потому что в тестах принято использовать `static import` — специальный `import`, который позволяет использовать имена статических методов без имени класса:



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

public class MathTest {
    @Test
    public void shouldCalculateSin() {
        double result = Math.sin(Math.PI / 2);
        double expected = 1.0;
        double delta = 0.01;
        assertEquals(expected, result, delta);
    }
}
```

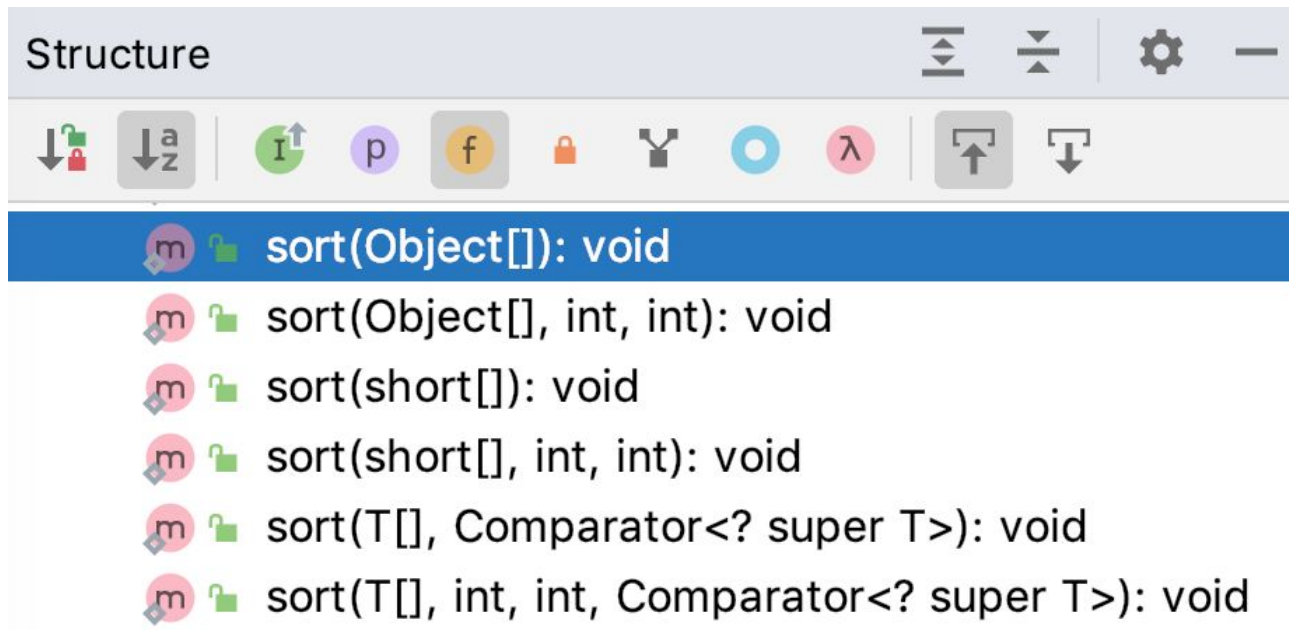
The screenshot shows a code editor with a vertical toolbar on the left containing icons for running, testing, and other IDE functions. The code defines a `MathTest` class with a `@Test` method `shouldCalculateSin`. A red line connects the `assertEquals` call in the test method to the `import static org.junit.jupiter.api.Assertions.assertEquals;` statement at the top, illustrating how the static import allows the method to be called without the `Assertions` prefix.

Важно: так обычно делают только в тестах!

## Q & A

Q: Как это нам поможет в задаче сортировки?

A: Для массивов есть свой утилитный класс `Arrays`. Именно в нём есть метод сортировки массива `Object[]`:



Комбинация клавиш `Alt + 7` в IDEA позволяет вам посмотреть на структуру класса.



# ARRAYS.SORT

```
@Test
public void shouldSortById() {
    Product[] expected = new Product[]{first, second, third};
    Product[] actual = new Product[]{third, first, second};

    Arrays.sort(actual);

    assertEquals(expected, actual);
}
```

Ключевой вопрос: откуда этот метод знает, как сортировать наши объекты, если мы это нигде не указывали?

# ARRAYS.SORT

После запуска тестов выяснится, что он «не знает»:

```
java.lang.ClassCastException: class ru.netology.domain.Product cannot be cast to class java.lang.Comparable...
```

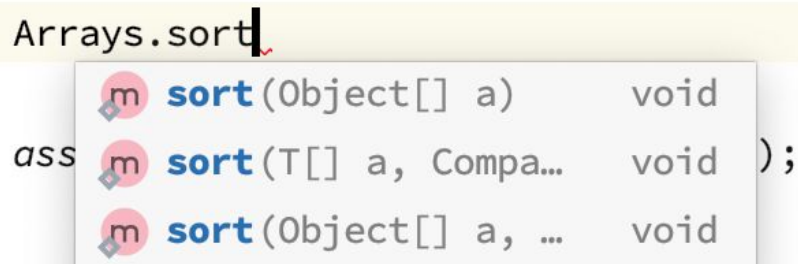
Вообще говоря, это было написано в JavaDoc'e на метод:

```
* @param a the array to be sorted  
* @throws ClassCastException if the array contains elements that are not  
* <i>mutually comparable</i> (for example, strings and integers)  
* @throws IllegalArgumentException (optional) if the natural  
* ordering of the array elements is found to violate the  
* {@link Comparable} contract  
*/
```

```
@Contract(mutates = "param1")  
public static void sort( @NotNull Object[] a) {
```

## РЕМАРКА: JAVADOC

Большинство «программистов» программируют следующим образом: находят подходящий класс (а то и целый кусок кода со StackOverflow), ставят точку и смотрят на подсказку IDEA, выбирая первый подходящий метод:



```
Arrays.sort  
m sort(Object[] a) void  
ass m sort(T[] a, Compa... void );  
m sort(Object[] a, ... void
```

При этом про JavaDoc'и они либо вообще не слышали, либо никогда не пользовались. Мы настоятельно вам рекомендуем **всегда читать JavaDoc'и** на те классы и методы, которые вы используете.



# СОРТИРОВКА



# СОРТИРОВКА

Задача сортировки набора объектов (массива, в частности) встречается настолько часто, что уже реализована в стандартной библиотеке Java.

Алгоритм сортировки основан на сравнении двух объектов: мы показываем алгоритму как сравнить два любых объекта в наборе, а алгоритм затем сравнивает все объекты и выполняет необходимые перестановки.



# СОРТИРОВКА

При сравнении двух объектов возникает два закономерных вопроса:

1. Как их сравнивать?
2. Какой результат должен возвращать метод сравнения?



## КАК СРАВНИВАТЬ

Например, можно оттолкнуться от идеи `equals` и реализовать нечто подобное.



# КАК СРАВНИВАТЬ

Аналогия из реальной жизни: представим, что у нас есть команда сотрудников. Им нужно выбрать лучшего.

*Вопрос к аудитории: как они могут это сделать?*





# КАК СРАВНИВАТЬ

Варианта два:

1. Они могут сами договориться и выбрать кого-то.
2. Мы можем к ним приставить менеджера, который и будет решать.



# КАК СРАВНИВАТЬ

То же самое в Java:

1. Мы можем сделать умные объекты, которые могут сравниваться друг с другом.
2. Мы можем сделать отдельного «менеджера», который будет сравнивать два объекта.

## ТИП РЕЗУЛЬТАТА

Понятно, что мы не можем сделать как в `equals`, возвращая `true` или `false`, потому что у нас будет целых три варианта:

1. Первый меньше второго
2. Первый равен второму
3. Первый больше второго

*Вопрос к аудитории: какой тип данных вы бы предложили использовать?*



## ТИП РЕЗУЛЬТАТА

В Java решили возвращать `int` по следующим правилам:

1. Отрицательное число (первый меньше второго)
2. Ноль (первый равен второму)
3. Положительное число (первый больше второго)

И по умолчанию сортировка выполняется по возрастанию (от меньшего к большему).



## ВОПРОСЫ

**Q:** Как Arrays узнает, какой метод из наших объектов использовать? И какова сигнатура этого метода?

**Q:** В Object никаких подобных методов не определено, а значит, мы должны наследоваться от другого класса (со всеми вытекающими последствиями)?

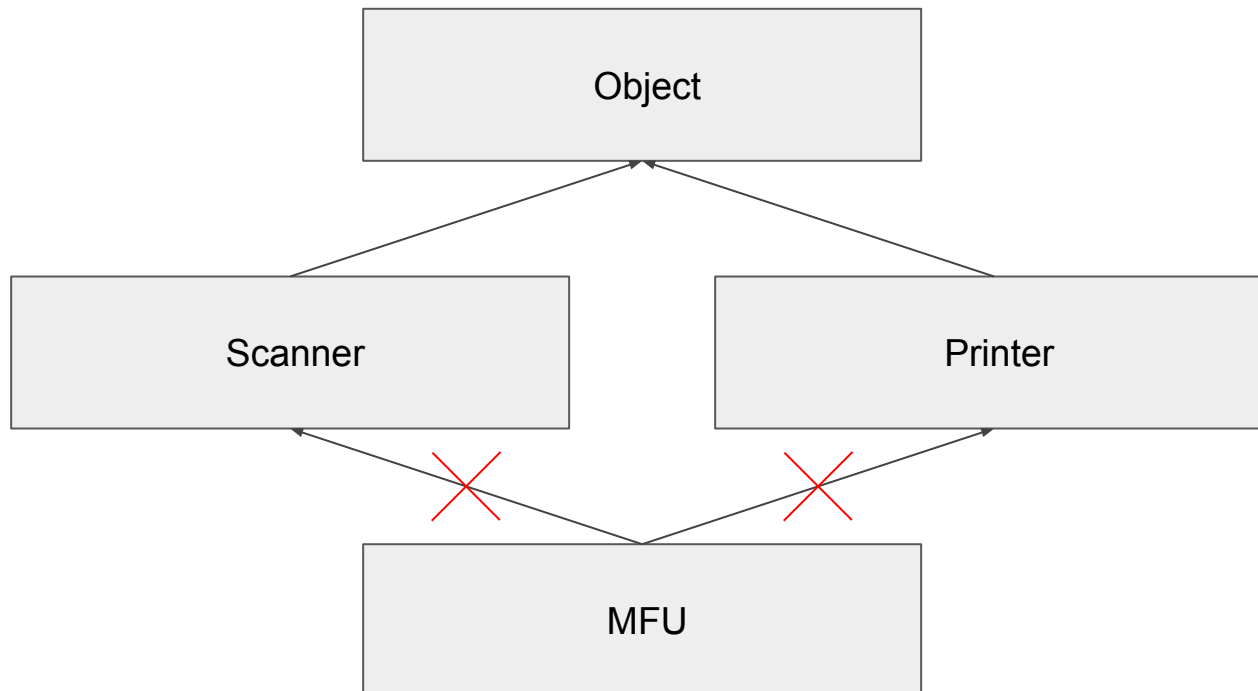
**A:** Для решения подобных задач в Java есть интерфейсы.



# ИНТЕРФЕЙСЫ

# ПРОБЛЕМЫ

На время вернёмся к одной из прошлых лекций, где мы говорили об ограничениях наследования в Java и разбирали задачу про МФУ, принтер и сканер, — **наследоваться можно только от одного класса**, поэтому вот эта схема не реализуема:



---

# АБСТРАКЦИЯ

Давайте рассмотрим задачу с другой стороны. По большому счёту нам часто не важно, печатаем мы на принтере или на МФУ, **важно, что он умеет печатать.**

То же самое относится к сканированию: например, нам нужно в формате PDF отправить скан-копию документа. Мы можем это сделать на реальном сканере, на МФУ или вообще с помощью смартфона сфотографировать и перевести в PDF.

Ключевое для нас то, что указанные **устройства умеют это делать.**





# ИНТЕРФЕЙСЫ

**Интерфейс** — определение набора методов, которые должен реализовывать объект.

Если объект содержит реализацию указанных методов, то говорят, что **объект реализует интерфейс**.



# ИНТЕРФЕЙСЫ

Изначально, интерфейс — **это чистая абстракция**: мы определяем набор методов, которые должны быть в объекте, а потом используем этот интерфейс в качестве типа данных (для переменных, параметров и полей).

# ИНТЕРФЕЙСЫ

```
public interface Printer {  
    void print(Document document);  
}
```

```
public interface Scanner {  
    Document scan();  
}
```

Как вы видите, интерфейс объявляется достаточно просто:

1. Имя интерфейса и модификатор доступа
2. Метод и возвращаемый тип

# ИНТЕРФЕЙСЫ

С одной стороны, всё просто, а с другой стороны, в определение интерфейса внесены идеи по умолчанию:

1. Интерфейс определяет публичное поведение объекта, поэтому все методы по умолчанию публичные:

```
public interface Printer {  
    public abstract void print(Document document);  
}
```

Modifier 'public' is redundant for interface methods

[Remove unnecessary 'public abstract'](#)



[More actions...](#)



# ИНТЕРФЕЙСЫ

2. Интерфейс требует наличия метода, но не накладывает ограничений на саму реализацию, поэтому все методы — абстрактные:

```
public interface Printer {  
    public abstract void print(Document document);  
}
```

Modifier 'abstract' is redundant for interface methods

[Remove unnecessary 'public abstract'](#)



[More actions...](#)





# ABSTRACT

**abstract** — ключевое слово, используемое в двух контекстах:

1. С методом класса — подразумевает, что этот метод должен быть реализован дочерними классами
2. С классом — подразумевает, что нельзя создавать объекты этого класса

Есть важное правило: если в классе есть хотя бы один абстрактный метод, то весь класс должен быть абстрактным, но не наоборот (в абстрактном классе может не быть ни одного абстрактного метода).

# ABSTRACT ДЛЯ НАС

Для нас **abstract** будет означать лишь одно: мы не можем создать неабстрактный класс, если наследуемся от абстрактного или реализуем интерфейс, нам обязательно нужно реализовать все абстрактные методы интерфейса или родительского класса:

реализуем интерфейс Printer

```
public class LJ1210 implements Printer {
```

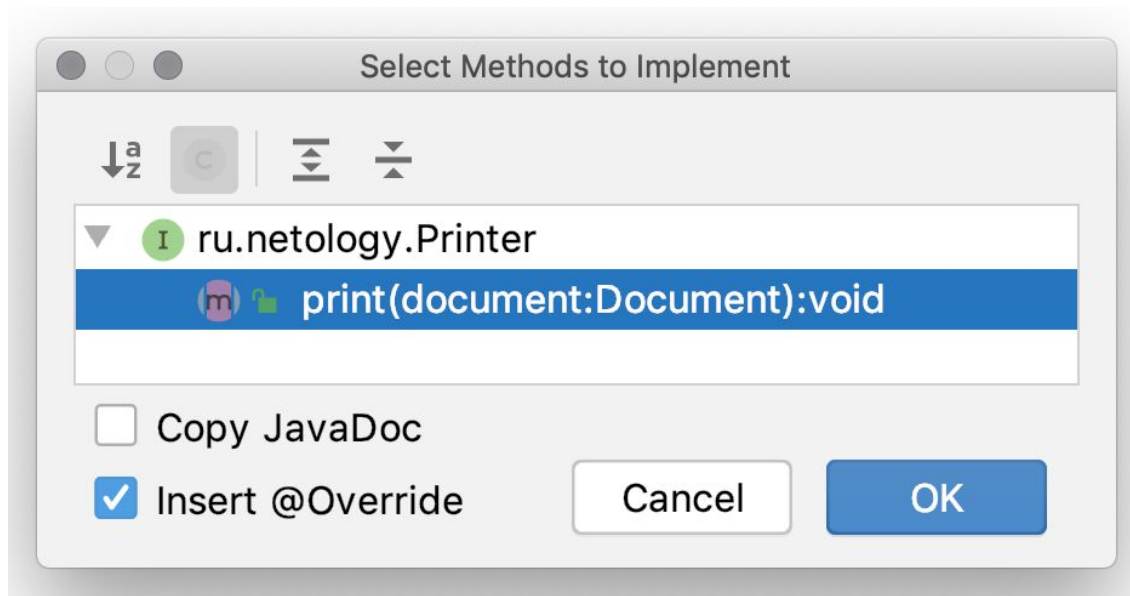
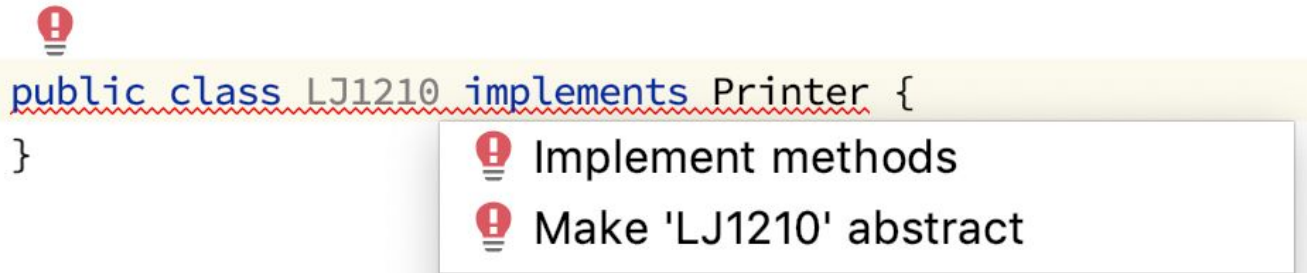
Class 'LJ1210' must either be declared abstract or implement abstract method 'print(Document)' in 'Printer'

Implement methods ↶⬆↷

More actions... ↶↷

# IDEA

IDEA нам всегда помогает, достаточно нажать Alt + Enter:





# ИНТЕРФЕЙСЫ

Обратите внимание: интерфейсы не накладывают ограничений на реализацию, они только требуют, чтобы она была (т.е. такая реализация вполне легитимна):

```
public class LJ1210 implements Printer {  
    @Override  
    public void print(Document document) {  
    }  
}
```

# ЧТО ЭТО НАМ ДАЁТ?

Мы можем использовать интерфейсы как типы для переменных, параметров и полей. Таким образом, если у нас есть сервис, отвечающий за работу с документами, он может требовать не конкретный класс, а класс, имплементирующий нужный интерфейс:

```
public class OfficeService {  
    public void print(Document document, Printer printer) {  
        printer.print(document);  
    }  
}
```

видим все методы,  
определённые в  
интерфейсе

указываем интерфейс

---

## Q & A

**Q:** Это похоже на то, как мы работали с наследованием.

**A:** Совершенно верно.

**Q:** Если наследование умеет делать так же, то зачем нам интерфейсы?

**A:** Интерфейсы — это слабая связность (в отличие от наследования).  
Один класс может реализовывать несколько интерфейсов.

# МФУ

```
public class LJProM283 implements Printer, Scanner {  
    @Override  
    public void print(Document document) {  
    }  
  
    @Override  
    public Document scan() {  
        return null;  
    }  
}
```

реализуемые интерфейсы

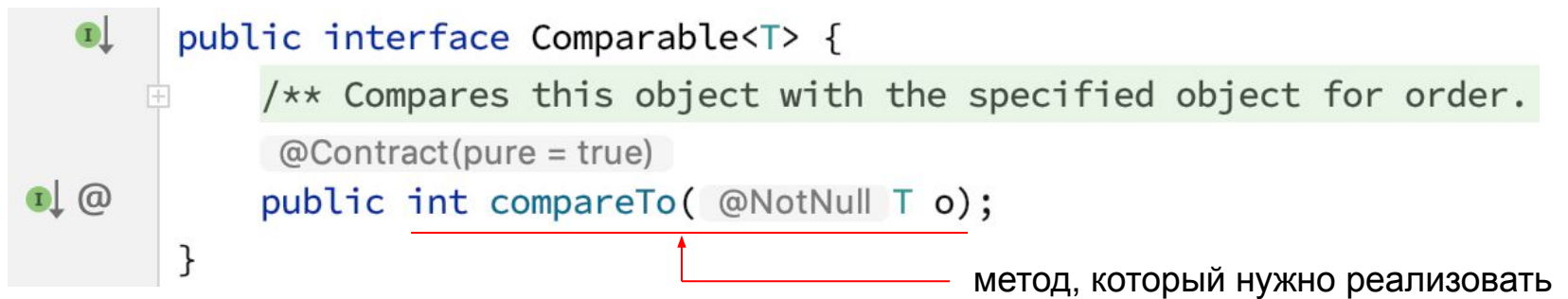


**COMPARABLE**

# СОРТИРОВКА

Вернемся к задаче.

В стандартной библиотеке Java существует специальный интерфейс:



```
public interface Comparable<T> {  
    /** Compares this object with the specified object for order.  
     * @Contract(pure = true)  
     */  
    public int compareTo( @NotNull T o);  
}
```

method, which needs to be implemented

Этот интерфейс определяет «натуральный» порядок сортировки объектов нашего класса.

«Натуральный» означает общепринятый (например, для строк — это алфавитный порядок).

# COMPARABLE

```
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Product implements Comparable {
    private int id;
    private String name;
    private int price;
    private double rating;

    @Override
    public int compareTo(Object o) {
        Product p = (Product) o;
        return id - p.id;
    }
}
```

имеем доступ к приватным полям другого объекта того же класса

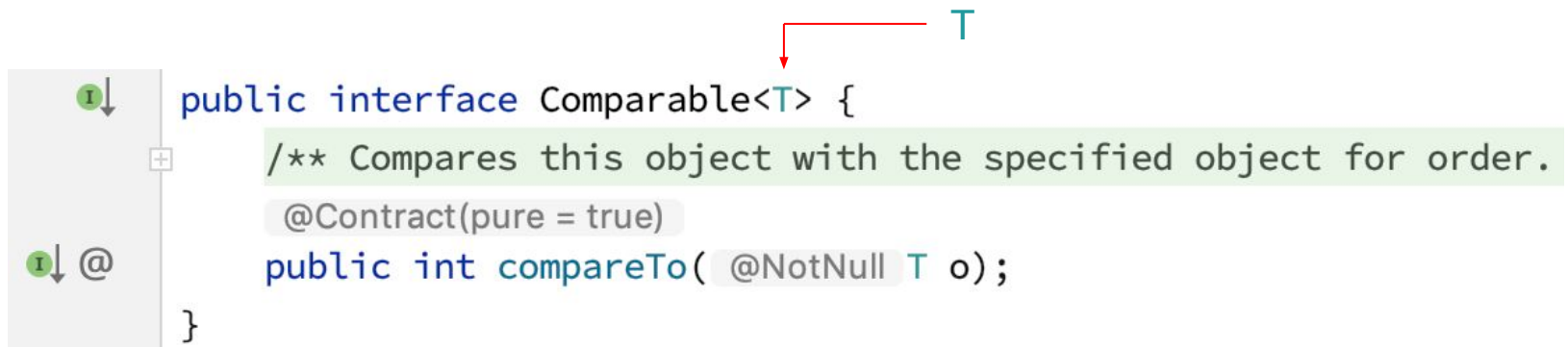
После этого наш тест на сортировку пройдёт ✓

# РЕЗУЛЬТАТЫ

Мы реализовали интерфейс, показав, как сравнивать наш объект с другим объектом.

Но есть ряд моментов:

1. Выбранный нами способ позволяет сортировать только в одном порядке
2. Непонятно, что это за буква **T** в определении интерфейса:



```
public interface Comparable<T> {  
    /** Compares this object with the specified object for order.  
     * @Contract(pure = true)  
     * @NotNull  
     */  
    public int compareTo( @NotNull T o);  
}
```

The diagram illustrates the definition of the `Comparable` interface. It includes annotations such as `@Contract(pure = true)` and `@NotNull`. A red arrow points from the letter `T` in the generic type parameter `<T>` to a separate `T` label, highlighting its role as a type parameter.





# GENERIC

# GENERIC

**Generics** — механизм, позволяющий нам писать обобщённый код, работающий со множеством типов. При этом проверка типов осуществляется на этапе компиляции самим компилятором.

Мы можем приводить каждый раз к нужному типу, используя `Object`, но это не безопасно. Если мы забудем поставить `instanceof`, то получим исключение в момент выполнения приложения.

А с `generic`'ами мы получим ошибку на этапе компиляции. Как тестировщики вы уже знаете, что чем раньше обнаружена ошибка, тем дешевле её исправить.

# BOX

Для того, чтобы понять предназначение generic'ов, мы решим небольшую задачу.

Давайте избавимся от NPE путём создания специального класса Box:

```
public class Box {  
    private Object value;  
  
    public Box(Object value) {  
        this.value = value;  
    }  
  
    public boolean isEmpty() {  
        return value == null;  
    }  
  
    public Object getValue() {  
        return value;  
    }  
}
```

# BOX

Наш класс Box обладает недостатком. Мы указали тип поля Object, поэтому, какой бы объект мы туда не клали, возвращать нам будут Object (хотя за ссылкой на Object будет именно наш объект):

```
class BoxTest {  
    @Test  
    public void shouldSaveAndReturnValue() {  
        Product product = new Product();  
        Box box = new Box(product);  
  
        box.g  
    }  
}
```

Autocomplete popup for `box.g` showing methods: `getValue()` (returns `Object`), `getClass...` (returns `Class<? extends Box>`), and `toString()` (returns `String`). A hint at the bottom says "Press ^. to choose the sele...Next Tip".

Diagram illustrating the transformation of the code. Two red circles with green arrows point from the original code to the transformed code. The transformed code is as follows:

```
class BoxTest {  
    @Test  
    public void shouldSaveAndReturnValue() {  
        Product product = new Product();  
        Box box = new Box(product);  
  
        Object value = box.getValue();  
        assertEquals(product, value);  
    }  
}
```

# BOX

Если мы захотим хранить только объекты `Product`, то можно поменять `Object` на `Product`, но тогда `Box` потеряет свою универсальность (придётся делать «точно такой же» для другого типа). Вот именно здесь нам и пригодятся `generic`'и:

```
public class GenericBox<T> {  
    private T value;  
  
    public GenericBox(T value) { this.value = value; }  
  
    public boolean isEmpty() { return value == null; }  
  
    public T getValue() { return value; }  
}
```

Тип-параметр: везде\* внутри `T` для конкретного объекта будет заменено на тот тип, что указан здесь

Примечание\*: как всегда есть исключения и нюансы, но для общих случаев это так.

В рамках этого объекта `T` везде  
«заменится» на `Product`

```
public class GenericBox<T> {  
    private T value;  
  
    public GenericBox(T value) { this.value = value; }  
  
    public boolean isEmpty() { return value == null; }  
  
    public T getValue() { return value; }  
}
```

```
class GenericBoxTest {  
    @Test  
    public void shouldParametrize() {  
        Product product = new Product();  
        GenericBox<Product> productBox = new GenericBox<>(product);  
    }  
}
```

`productBox.get`

m **getValue()** **Product**  
m getClass() Class<? extends GenericBox>  
^↓ and ^↑ will move caret down an... Next Tip

Уже не `Object` а `Product`

Ключевое: никаких приведений типов, `instanceof` и т.д. — мы «скидываем» эту работу на компилятор.

Для этого объекта `T = Product`

Для этого объекта `T = String`

```
public class GenericBox<T> {  
    private T value;  
    // constructor + isEmpty + getValue  
}
```

```
class GenericBoxTest {
```

```
    @Test
```

```
    public void shouldParametrizedWithProduct() {
```

```
        Product product = new Product();
```

```
        GenericBox<Product> productBox = new GenericBox<>(product);
```

```
        Product value = productBox.getValue();
```

```
        assertEquals(product, value);
```

```
    }
```

```
    @Test
```

```
    public void shouldParametrizedWithString() {
```

```
        String str = "Hello world";
```

```
        GenericBox<String> stringBox = new GenericBox<>(str);
```

```
        String value = stringBox.getValue();
```

```
        assertEquals(str, value);
```

```
    }
```

```
}
```



# ПАРАМЕТРИЗАЦИЯ

На самом деле для параметризации есть два ключевых варианта:

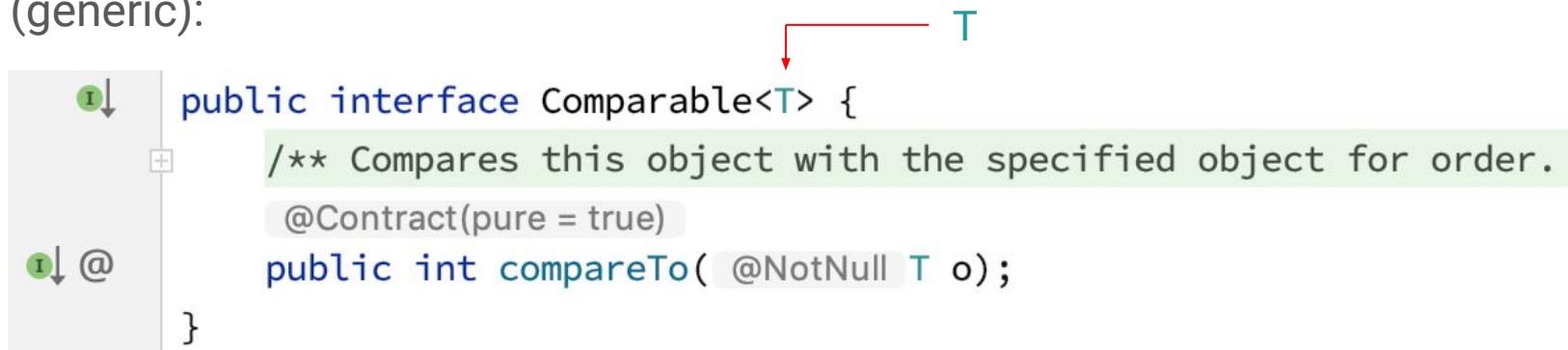
1. При создании объекта в угловых скобках переменной указать нужный тип (этот вариант мы только что рассмотрели)
2. При наследовании/реализации указать тип (рассмотрим на следующих слайдах). Мы как тестировщики чаще будем использовать уже готовые параметризованные типы (а не писать свои).

Давайте используем вариант 2 для нашей задачи с сортировкой.



# GENERIC

Каждый раз, когда вы видите определение класса, интерфейса или метода с угловыми скобками, это означает параметризацию (generic):



```
public interface Comparable<T> {  
    /** Compares this object with the specified object for order.  
     * @Contract(pure = true)  
     * @NotNull T o;  
    public int compareTo( @NotNull T o);  
}
```

The image shows a code editor snippet of the `Comparable` interface. On the left, there are two icons: a green circle with a white 'I' and a green circle with a white '@'. A red arrow points from the letter 'T' in the generic parameter `<T>` to the `@NotNull T o` parameter in the `compareTo` method signature.

Как это работает:

1. Если ничего не написать в угловых скобках, то везде буква `T` (кроме самих угловых скобок) заменяется на `Object`.
2. Если написать в угловых скобках тип, то везде буква `T` заменится на ЭТОТ тип.

# GENERIC

```
public class Product implements Comparable {  
    private int id;  
    private String name;  
    private int price;  
    private double rating;
```

угловых скобок нет

@Override

```
    public int compareTo(Object o) {  
        Product p = (Product) o;  
        return id - p.id;  
    }  
}
```

поэтому здесь *Object*

компилятор будет проверять  
на соответствие этому интерфейсу

```
public interface Comparable {  
    public int compareTo(Object o);  
}
```

# GENERICICS

```
public class Product implements Comparable<Product> {  
    private int id;  
    private String name;  
    private int price;  
    private double rating;
```

угловые скобки есть

```
    @Override  
    public int compareTo(Product o) {  
        return id - o.id;  
    }  
}
```

поэтому здесь Product

компилятор будет проверять  
на соответствие этому интерфейсу

```
public interface Comparable {  
    public int compareTo(Product o);  
}
```

# GENERIC

Если мы попробуем «обмануть» компилятор и написать что-то другое, то нам об этом сразу скажут:

```
public class Product implements Comparable<Product> {
```

Class 'Product' must either be declared abstract or implement abstract method 'compareTo(T)' in 'Comparable'

Implement methods



More actions...



```
    @Override  
    public int compareTo(Object o) {  
        return id - o.id;  
    }  
}
```



# GENERICs

На данном этапе для нас generic'и предоставляют удобный механизм контроля типов: нам не приходится работать с *Object* и «руками» приводить типы — компилятор сам за этим будет следить и выполнять необходимые манипуляции.



# ИТОГИ



## ИТОГИ

Сегодня мы рассмотрели важную тему интерфейсов и generic'ов.

В основном это была вводная информация, целиком всю мощь интерфейсов и generic'ов мы с вами увидим на следующей лекции, когда будем разбирать Collections API.



## ВАЖНО

Экосистема интерфейсов и generic'ов формировалась многие версии Java, поэтому «понимание» её может показаться сложным.

На самом деле, если вы будете использовать наши приёмы и аналогии, постепенно приобретая опыт работы с ними, не пытаясь изучить сразу все возможности и нюансы, то у вас всё получится.





# 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**.



## ДОМАШНЕЕ ЗАДАНИЕ

В рамках ДЗ вы научитесь использовать ещё один интерфейс — `Comparator`, предназначенный для определения произвольного порядка (не натурального как в `Comparable`) объектов.



Задавайте вопросы и напишите отзыв о лекции!

**ВАСИЛИЙ ДОРОХИН**

 Василий Дорохин