

Interfaces & Generics

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

Обязательно пользуйтесь этой возможностью и помечайте такими комментариями места в

коде, к которым нужно вернуться и что-то доделать `// TODO: sort`

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

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)

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);
    }
}
```

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(...)

STATIC IMPORT

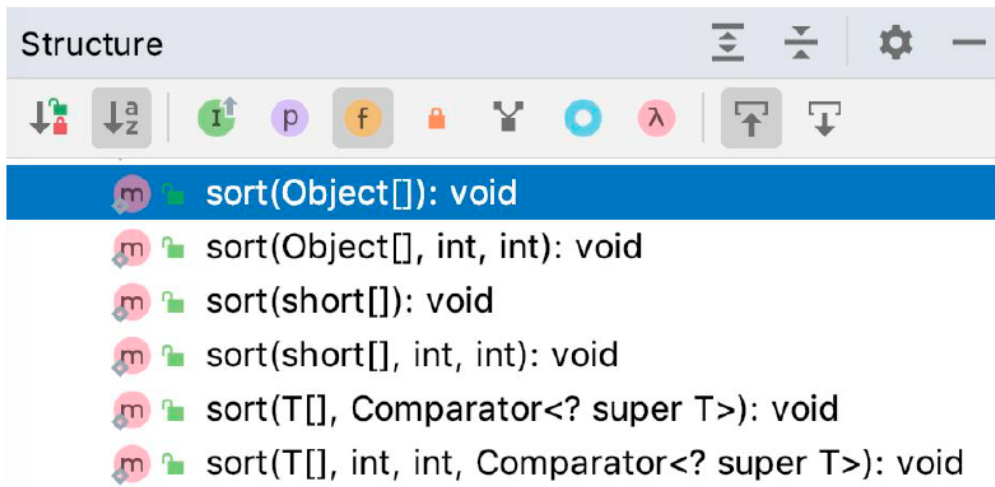
Мы так никогда не писали, потому что в тестах принято использовать `static import` — специальный `import`, который позволяет использовать имена статических методов без имени класса:

```
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);
    }
}
```

Важно: так обычно делают только в тестах!

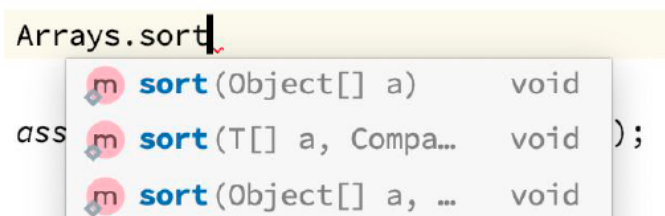
A: Для массивов есть свой утилитный класс `Arrays`. Именно в нём есть метод сортировки массива `Object[]`:



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

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:



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

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.

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:

Первый меньше второго

Первый равен второму

Первый больше второго

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.

Интерфейс — определение набора методов, которые должен реализовывать объект.

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

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).

```
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

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

1. Интерфейс определяет публичное поведение объекта, поэтому все методы по умолчанию публичные
2. 2. Интерфейс требует наличия метода, но не накладывает ограничений на саму реализацию, поэтому все методы — абстрактные

ABSTRACT

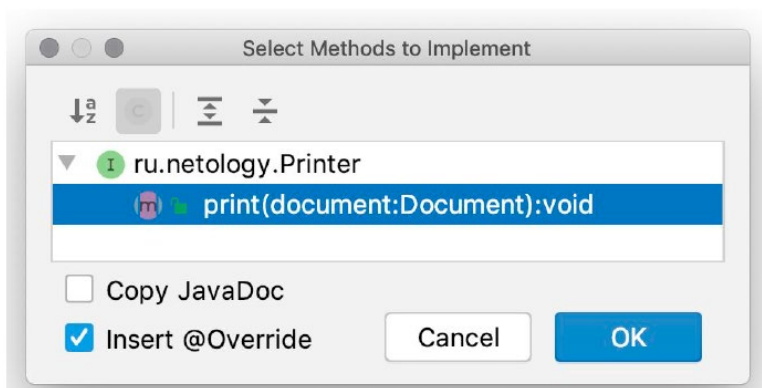
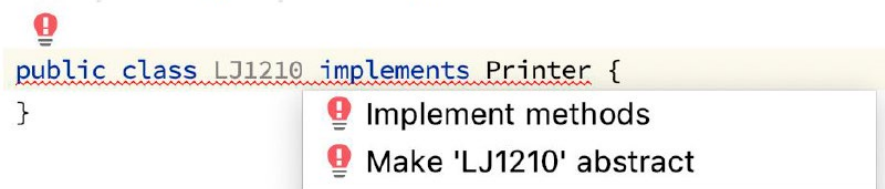
abstract — ключевое слово, используемое в двух контекстах:

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).

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

IDEA нам всегда помогает, достаточно нажать Alt + Enter:



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: 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

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)  
     * @param o the object to be compared.  
     * @return the result of the comparison.  
     */  
    public int compareTo(@NotNull T o);  
}
```

метод, который нужно реализовать

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 пройдет ✓

```
@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;
    }
}
```

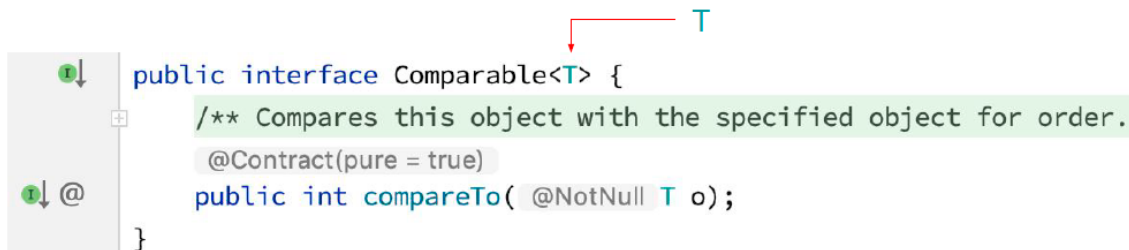
имеем доступ к приватным полям другого объекта того же класса

После этого наш тест на сортировку пройдет ✓

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)  
     */  
    public int compareTo( @NotNull T o);  
}
```

A red arrow points from the letter 'T' in the generic type parameter <T> to the parameter 'T' in the method signature compareTo(@NotNull T o).

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 задачку.

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)

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

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  
    }  
}
```

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`

```

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);
    }
}

```

```

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

```

55

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).

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...

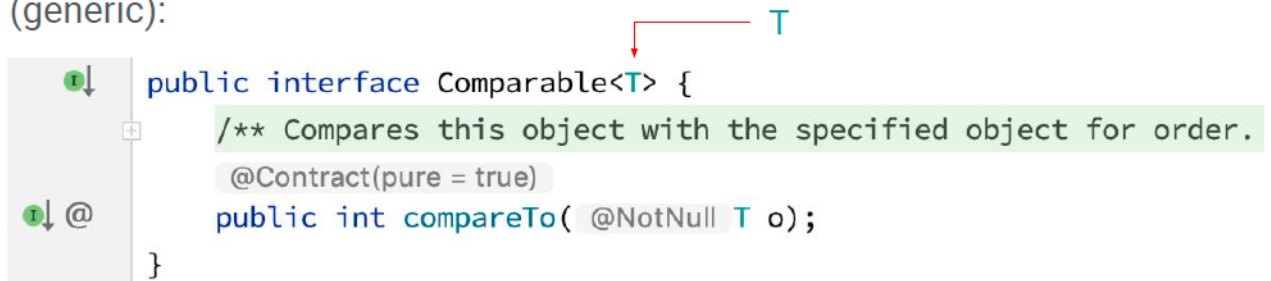
```

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

```

GENERICCS

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)  
     */  
    public int compareTo( @NotNull T o);  
}
```

The screenshot shows the Java code for the `Comparable` interface. A red arrow points from the `T` in the angle brackets to the `T` in the `compareTo` method signature. There are also green arrows pointing to the `@` symbol in the `@NotNull` annotation and the `+` symbol in the code editor's gutter.

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.

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.