Objects

Класс: `java.util.Objects`

Описание

Класс-утилита, предоставляющий чисто статические методы для работы с объектами и массивами объектов, бóльшая часть которых предназначена для обеспечения *нулебезопасности* (null safety).

Нулебезопасность

Исключение `NullPointerException` — достаточно распространённый бич программ, работающих с API, которые могут возвращать `null` как признак отсутствия значения, если клиентский код забывает проверить результат на `null`. Например, что, если мы пишем вики-движок, который должен открыть страницу редактирования, когда в URL страницы встречается параметр запроса (query parameter) `action=edit`?

`http://example.com/wiki/Article_name?action=edit`

Наивное сравнение строк на равенство таит в себе ошибку:

```
String action = request.getParameter("action");

// Не нулебезопасно - никогда так не делайте!
if (action.equals("edit")) {
    // вывести страницу редактирования
}
```

Если в URL страницы вообще не окажется параметра `action`, то метод `getParameter` вернёт `null`, и вместо красивого экрана редактирования пользователь увидит страницу трассировки стека для нашего любимого исключения... правильно, `NullPointerException`.

У опытных программистов уже выработался рефлекс в сравнении со строковой константой ставить её на первое место: `"edit".equals(action)`. Но что, если обе ссылки на сравниваемые объекты могут быть равны `null`? Что, если нужно вызвать другой метод у ссылки, которая может быть равна `null`?

Наконец, значения `null` создают неудобства при работе с функциональными интерфейсами:

```
// Ой, сломались
Stream<String> stringValues = Stream.of(42, null, "test").map(Object::toString);
```

Класс `Objects` предоставляет статические нулебезопасные аналоги для трёх методов, поддерживаемых всеми объектами: `equals`, `hashCode` и `toString`.

Обычный вызов	Нулебезопасный вызов
`a.equals(b)`	`Objects.equals(a, b)`
`a.hashCode()`	`Objects.hashCode(a)`
`a.toString()`	`Objects.toString(a)`

Все они делают что-то осмысленное, если `a == null`.

Нулебезопасные `equals` и `hashCode`

Вызов `Objects.equals(a, b)` возвращает `true` тогда и только тогда, когда:

- либо `a == null` и `b == null`;
- либо `a != null` и `a.equals(b)`.

Вызов `Objects.hashCode(a)` возвращает `a.hashCode()`, если `a != null`, и `0` иначе. То есть, по сути, `0` считается хэш-кодом значения `null`.

Нулебезопасный `toString`

Вызов `Objects.toString(a)` возвращает `a.toString()`, если `a != null`, и строку из четырёх символов `"null"` иначе.

Иначе говоря, этот метод полностью аналогичен `String.valueOf` (на самом деле его он и вызывает) и возвращает ту строку, которую печатают `print(a)` и `println(a)`. Следующие три вызова всегда выводят одну и ту же строку:

```
Object x = ...;
System.out.println(x);
System.out.println(String.valueOf(x));
System.out.println(Objects.toString(x));
```

Интересен второй вариант метода `Objects.toString` с двумя параметрами:

`static String toString(Object o, String nullDefault)`

Если первым параметром передать `null`, то он вернёт строку, переданную вторым параметром. Это может быть полезно, например, если требуется обрабатывать значения `null` как пустые строки (некоторые СУБД и библиотеки графического интерфейса не делают между ними различий).

```
// Преобразовать все null в пустые строки
public Stream<String> nullStringsToEmpty(Stream<String> input) {
    return input.map(str -> Objects.toString(str, ""));
}
```

Ещё один часто встречающийся случай — использование `null` как признака отсутствия значения (начиная с Java 8, для значений, которые могут отсутствовать, лучше использовать `Optional`, но уже существует большое количество старого кода, в котором используется такая идиома). Например, пусть мы храним список людей с датами рождения, но не все даты рождения известны:

```
public final class Person {
    private final String name;
    private final LocalDate dateOfBirth;

    public Person(final String name, final LocalDate dateOfBirth) {
        this.name = name;
        this.dateOfBirth = dateOfBirth;
    }

    public String getName() { return name; }
    public LocalDate getDateOfBirth { return dateOfBirth; }
}

List<Person> greatConquerors = new ArrayList<>();
IsoChronology chrono = IsoChronology.INSTANCE;

greatConquerors.add(new Person("Александр Македонский",
        chrono.date(IsoEra.BCE, 356, Month.JULY, 20)));
greatConquerors.add(new Person("Чингисхан", null));
greatConquerors.add(new Person("Наполеон Бонапарт",
        chrono.date(1769, Month.AUGUST, 15));
```

Тогда при выводе нашего списка мы можем подставить специальную строку для неизвестных значений:

```
for (Person p : greatConquerors) {
    System.out.printf("Завоеватель: %s, дата рождения: %s\n",
            p.getName(), Objects.toString(p.getDateOfBirth(), "неизвестна"));
}
```

Проверка предусловий

К сожалению, язык Java не позволяет на синтаксическом уровне запретить передавать `null` в качестве параметра методов. Остаётся только описать в документации, какие параметры метода могут и не могут принимать `null` в качестве допустимого значения.

Желательно, чтобы метод выбрасывал исключение сразу же, как только он увидит, что значения его параметров недопустимы. Это частный случай более общего принципа, предписывающего сигнализировать об ошибке как можно раньше *(fail-fast)* вместо того, чтобы продолжать выполнение и получить ошибку или, того хуже, некорректный результат работы метода где-нибудь дальше.

Семейство методов `requireNonNull` служит для подобной *проверки предусловий* и, как правило, вызывается в самом начале метода.

`static <T> T requireNonNull(T obj)`

`static <T> T requireNonNull(T obj, String message)`

`static <T> T requireNonNull(T obj, Supplier<String> messageSupplier)`

Эти методы принимают ссылку на объект, возвращают её же и выбрасывают `NullPointerException` в случае, если она равна `null`. Можно задать сообщение (при желании — составляемое только по необходимости через `Supplier`) для выбрасываемого исключения.

Например, можно вставить такую проверку в конструктор класса Person:

```
public Person(String name, LocalDate dateOfBirth) {
    this.name = Objects.requireNonNull(name, "Name cannot be null");
    this.dateOfBirth = dateOfBirth;
}
```

Важно! В библиотеке Guava есть класс `Preconditions`, методы которого позволяют подобным образом проводить не только проверку параметров на `null`, но и другие предусловия, выбрасывая для них нужный тип исключения, включая проверки на диапазон индексов (`IndexOutOfBoundsException`) и проверки произвольных ограничений на значения параметров (`IllegalArgumentException`).

Предикаты для проверки на `null` и не-`null`

В классе `Objects` есть ещё два статических метода, связанных с `null`:

`static boolean isNull(Object obj)`

`static boolean nonNull(Object obj)`

Они эквивалентны соответственно `obj == null` и `obj != null` и применяются в основном для удобства операций с функциональными интерфейсами типа `Predicate`:

```
List<Object> values = Arrays.asList(42, null, null, "test");
values.stream().filter(Objects::isNull)   // null, null
values.stream().filter(x -> x == null)    // null, null
values.stream().filter(Objects::nonNull)  // 42, "test"
values.stream().filter(x -> x != null)    // 42, "test"
```

Хэш-код последовательности значений

Последний заслуживающий интереса метод класса `Objects` никак не связан с проверками на `null`.

`static int hash(Object... values)`

Он предназначен для удобства вычисления одного `hashCode` сразу по нескольким значениям. Как правило, он применяется для реализации `hashCode` в классах, объединяющих несколько значений. Например, мы могли бы так реализовать методы `equals` и `hashCode` для класса Person:

```
public final class Person {
    private final String name;
    private final LocalDate dateOfBirth;
    ...

    @Override
    public boolean equals(Object other) {
        if (this == other) {
            return true;
        }

        if (!(other instanceof Person)) {
            return false;
        }

        Person p = (Person) other;
        return name.equals(p.name) && Objects.equals(dateOfBirth, p.dateOfBirth);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, dateOfBirth);
    }
}
```

При этом следующие три вызова всегда дают один и тот же результат:

```
Objects.hash(obj1, ..., objN)
Arrays.hashCode(new Object[] { obj1, ..., objN })
Arrays.asList(obj1, ..., objN).hashCode()
```

Важно! Не путайте методы `Objects.hash` и `Objects.hashCode`. Если `Objects.hash` вызвать с одним параметром (например, `x`), он вернёт значение, отличающееся от результата `Objects.hashCode(x)` (которое, как мы помним, возвращает `x.hashCode()` либо `0` для `null`). Иначе говоря, `Objects.hash(x) != Objects.hashCode(x)`.