

Вложенные классы. Документирование



ПРЕПОДАВАТЕЛЬ



Юрий Костяной

Java/Kotlin backend-разработчик

- 3+ года опыта в коммерческой разработке
- 2+ года опыта в преподавании
- Проекты по интеграции сторонних платформ, CRM
- Проблемно-ориентированный подход в преподавании



ВАЖНО:

- Камера должна быть включена на протяжении всего занятия.
- Если у Вас возник вопрос в процессе занятия, пожалуйста, поднимите руку и дождитесь, пока преподаватель закончит мысль и спросит Вас, также можно задать вопрос в чате или когда преподаватель скажет, что начался блок вопросов.
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях.
- Вести себя уважительно и этично по отношению к остальным участникам занятия.
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя.

ПЛАН ЗАНЯТИЯ

1. Основной блок
2. Вопросы по основному блоку
3. Домашняя работа

1

ОСНОВНОЙ БЛОК

Введение

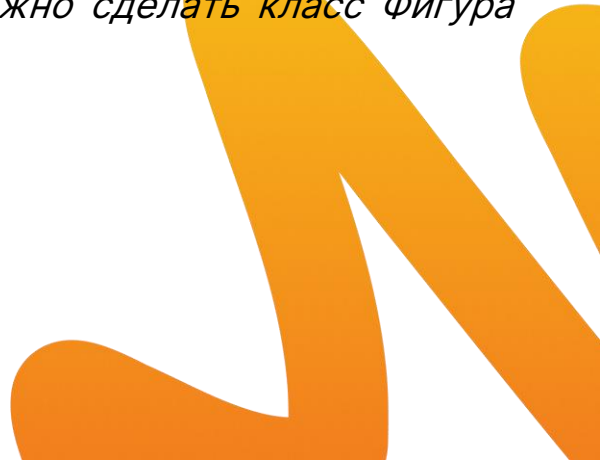
- Важно, какой ты внутри
- Документалочка



Проблема

Допустим, нам необходимо написать игру Шахматы. Мы можем создать класс Игровое поле, хранящий положение фигур. Сами фигуры тоже являются сущностями – хранят цвет, тип фигуры, как фигура ходит, особые свойства. Класс Фигура требуется классу Игровое поле, но другие классы им не пользуются.

В таком случае для более тесной работы Игрового поля и Фигуры, а также дополнительных возможностей сокрытия состояния фигур можно сделать класс Фигура внутренним классом в Игровом поле.



Важно, какой ты внутри

Вложенные классы

Java позволяет создавать одни классы внутри других. Такие классы называют вложенными (nested). Вложенными могут быть также перечисления (enum).

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
  
    class InnerClass {  
        ...  
    }  
}
```



Важно, какой ты внутри

Пример вложенного класса

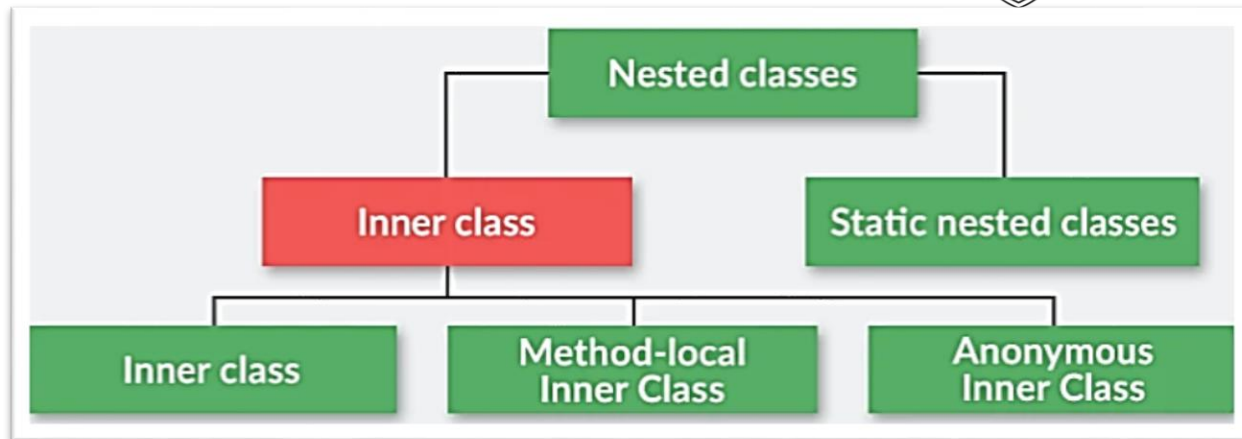
```
public class Question {  
    private Type type;  
  
    public Type getType() {  
        return type;  
    }  
  
    public void setType(Question.Type type) {  
        this.type = type;  
    }  
  
    public static class Type {  
        public static final Type SINGLE_CHOICE = new Type();  
        public static final Type MULTI_CHOICE = new Type();  
        public static final Type TEXT = new Type();  
  
        private Type() {  
        }  
    }  
}
```

Обращение к внутреннему static-классу

Внутренний static-класс

Важно, какой ты внутри Inner классы

Частным случаем
вложенных классов
являются **внутренние
классы** (*inner class*). У них
нет слова *static* в
определении класса



Внутренние классы бывают:

- обычными – полноценные вложенные классы, являются членами обрамляющего класса;
- локальными – определены внутри метода или другого блока кода, не являются членами обрамляющего класса;
- анонимными – наследуемые от какого-либо класса или интерфейса, в которых при объявлении **не задано имя класса**

Важно, какой ты внутри

Назначение inner-классов

1 Если объект реального мира, который мы описываем в классе, состоит из сложных составных частей, то для этих частей создаются отдельные классы. Например, для описания ноутбука логично создать классы материнской платы, монитора, оперативной памяти, клавиатуры пр. Каждая комплектующая часть будет иметь свой набор характеристик.



2 Если вспомогательный класс полезен только для одного другого класса, то вполне логично встроить его в этот класс и хранить их вместе. Использование вложенных классов увеличивает *инкапсуляцию*.

3 Когда классы должны взаимодействовать на уровне приватных полей. Скрывая класс «В» в пределах класса «А», члены класса «А» могут быть объявлены закрытыми, и «В» может получить доступ к ним. Кроме того, сам «В» может быть скрыт от внешнего мира модификатором `private`.

Важно, какой ты внутри

Пример внутреннего класса

```
class List {  
    private Object[] array;  
  
    public void print() {  
        Iterator iter = this.new Iterator();  
        Object obj;  
        do {  
            obj = iter.getNext();  
            System.out.println(obj);  
        } while (obj != null);  
    }  
    private class Iterator {  
        private int index = 0;  
  
        public Object getNext()  
            return (index < array.length) ? array[index++] : null;  
    }  
}
```

Создание экземпляра

Внешний объект

Внутренний класс

Обращение к полю
внешнего объекта

Важно, какой ты внутри

Пример локального класса

```
class Handler {  
    public void handle(String requestPath) {  
        class Path {  
            private List<String> parts;  
            private String path = "/";  
            Path(String path) {  
                if (path == null) return;  
                this.path = path;  
                this.parts = Arrays.asList(path.split("/"));  
            }  
            int size() { return parts.size(); }  
            String get(int i) { return this.parts.get(i); }  
            boolean startsWith(String s) { return path.startsWith(s); }  
        }  
        Path path = new Path(requestPath);  
        if (path.startsWith("/page")) {  
            String pageId = path.get(1);  
            // ...  
        }  
        if (path.startsWith("/post")) {  
            String categoryId = path.get(1);  
            String postId = path.get(2);  
            // ...  
        }  
        // ...  
    }  
}
```

Локальный класс

Создание экземпляра

Вызов методов
локального класса

Важно, какой ты внутри

Анонимные классы

Анонимный класс похож на локальный класс. Он также объявляется внутри метода, но имеет более простой синтаксис и, ко всему прочему, не имеет имени. Синтаксис также накладывает ограничения: анонимный класс всегда является наследником какого-либо класса или интерфейса, причем только одного. Объявление анонимного класса невозможно без создания экземпляра этого класса, причем ровно одного.

Используется, когда нужно создать объект здесь и сейчас, который более нигде не будет использоваться

```
button.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        Cell cell = field.getCell(buttonX, buttonY);  
        cell.onClick();  
    }  
});
```

Важно, какой ты внутри

Пример анонимных классов

```
public class MyIteratorExample {
    public static void main(String[] args) {
        List<String> names = List.of("Jim", "John", "Freddy", "Ringo", "Paul");
        // Итератор берёт строки в порядке увеличения длины
        Iterator<String> iterator = new Iterator<>() { // анонимный класс - создаём итератор на месте.
            private final List<String> innerList = new ArrayList<>(names); // копируем исходную коллекцию

            {
                innerList.sort(Comparator.comparing(String::length)); // сортируем коллекцию по длине строк
            }

            @Override public boolean hasNext() { return !innerList.isEmpty(); }

            @Override public String next() {
                if (!hasNext()) throw new NoSuchElementException();
                return innerList.remove(0);
            }
        };

        while (iterator.hasNext()) { System.out.println(iterator.next()); }
    }
}
```

Важно, какой ты внутри

Задание



1 Создайте класс библиотечной карточки. В карточке хранятся данные читателя (фамилия, имя, отчество, дата рождения, номер читательского билета, телефон), данные взятых книг (дата, когда книга была взята, название книги, автор, номер экземпляра, ФИО выдавшего книгу сотрудника, отметка о возврате и дата возврата). Используйте вложенные классы Полное имя, Телефон (код страны, номер, добавочный, тип номера).

2 В отдельном классе создайте метод, который получает на вход набор библиотечных карточек и выбирает книги, которые не вернули в течение 14 дней после взятия. Используйте локальный класс для просроченных книг. Сформируйте строку по списку просроченных книг с указанием названия, автора, ФИО и телефона читателя. По данному списку библиотекарь обзвонит читателей и напомним им, что пора вернуть книгу. При желании можно в локальном классе сгенерировать toString() метод.

Проблема

От Junior-разработчика в первую очередь ожидают аккуратности, чистого кода и хорошего документирования. Умение писать чистый код придёт после понимания основных принципов проектирования и практик написания кода, которые мы изучим позже. Документирование кода – это навык, который можно освоить уже сейчас. В проекте документирование помогает на всех этапах разработки.



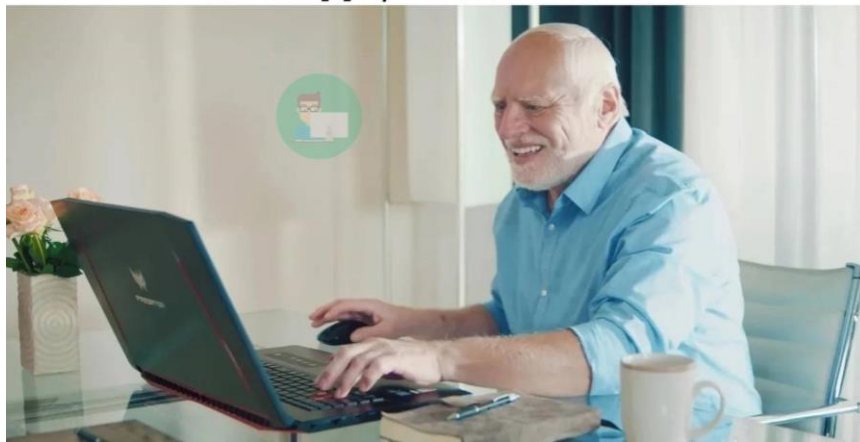
Документалочка

Вложенные классы

Документирование кода – это написание «комментариев» в коде в одном из общепринятых видов.

Основная цель документирования – повысить лёгкость обслуживания кода.

**Я УЖЕ В ТОМ ВОЗРАСТЕ, КОГДА САМ ПИШУ
КОММЕНТАРИИ К КОДУ ИНАЧЕ ЗАВТРА
ЗАБУДУ, ЧТО Я ПИСАЛ**



Самодокументируемый код

Правильные названия переменных, классов, полей и методов – первый шаг к документированию кода.

Но этого часто недостаточно, чтобы объяснить логику работы класса и его методов. Поэтому почти все классы стандартной библиотеки Java, начиная с Object, имеют поясняющую документацию.

Самодокументируемый код

Программист1: Почему нет комментариев?

Программист2: Мой код самодокументируемый

Программист1: Что это значит?

Программист2: Вместо "method1" или "val" я именую методы и переменные в соответствии с их содержанием или действием

Программист1:



Документалочка

Комментарии

Мы знаем об однострочных `//` и многострочных `/**` комментариях в Java. Такие комментарии будут полезны только тому, кто читает код Вашего класса. Этот подход не является универсальным, т.к. приходится залазить в каждый класс.

К тому же не все комментарии предназначены для программистов, которые используют Ваш код – некоторые мы пишем для себя или коллег, которые этот класс будут дорабатывать.

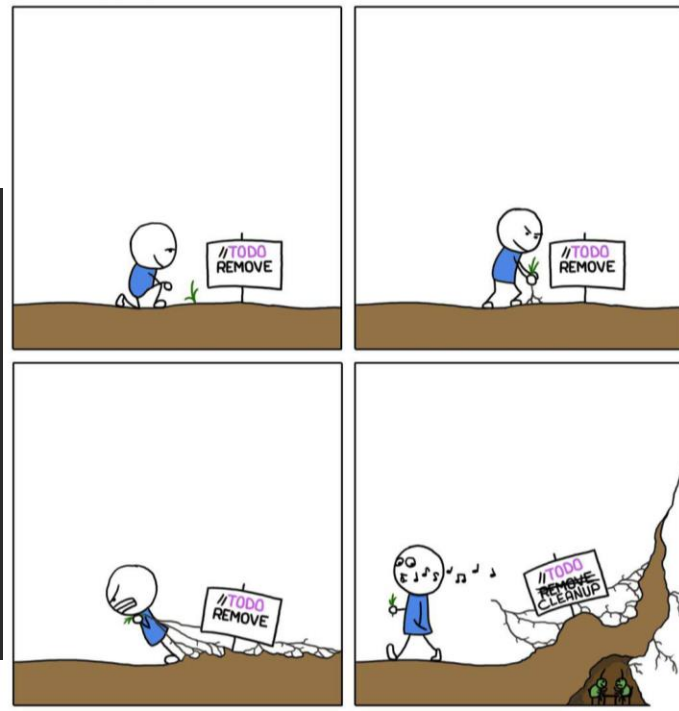
```
public class Documentation {  
    public static void main(String[] args) {  
        // Однострочный комментарий  
        PersonWithDocs person = new PersonWithDocs();  
        /* Многострочный  
        комментарий */  
        person.display();  
    }  
}
```

Документалочка

Комментарии

Для того, чтобы пометить доработки и исправления, которые нужно выполнить в коде в будущем, в комментарии указывают *TODO* или *FIX*.

```
public class Documentation {  
    public static void main(String[] args) {  
        // Однострочный комментарий  
        PersonWithDocs person = new PersonWithDocs();  
        /* Многострочный  
        комментарий */  
        person.display();  
        // TODO убрать комментарии после доработки класса  
    }  
}
```



Документалочка

Комментарии



1 Используйте комментарии только для неочевидных вещей в коде и пояснения основных блоков кода внутри методов, т.к. перегруженный комментариями код тяжело читается.

2 Комментарии не должны повторять очевидные вещи.

КОММЕНТАРИИ В КОДЕ ВСЕГДА ТАКИЕ:



Документалочка

Комментарии



- 1 Когда код доделан и выдан (в Git-репозиторий или на code-review), в нём не должно остаться закомментированного кода (особенно недоделанного или неправильного!), кроме случаев, когда ваш код написан ранее, чем его необходимо внедрить в проект.
- 2 Перед выдачей кода проверьте *TODO* и *FIX-комментарии*: если какие-то из них неактуальны, то удалите их.



Документалочка

Документирование

Для документирования используется третий тип комментариев, начинающийся с `/**` (`/**` затем Enter – шорткат в IDE).

```
public class Person {  
    private final String name; // Полное имя человека  
    /* Метод отображения данных о человеке */  
    public void display() {  
        System.out.println("Name: " + name);  
    }  
}
```

Вариант справа выглядит более объёмным, но он позволяет автоматизировать сбор документации с помощью технологии **JavaDoc**.

```
/**  
 * Класс для хранения данных о человеке  
 */  
public class PersonWithDocs {  
    /**  
     * Полное имя человека  
     */  
    private final String name;  
  
    /**  
     * Метод вывода информации о человеке  
     */  
    public void display() {  
        System.out.println("Name: " + name);  
    }  
}
```

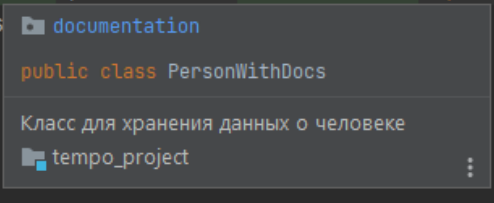

Документалочка

Документирование

JavaDoc – инструмент JDK, который позволяет описывать классы, поля и методы с помощью специальных тегов и затем формировать документацию в удобном виде.

Примеры использования JavaDoc:

```
public class Documentation {
    public static void main(String[] args) {
        PersonWithDocs person = new PersonWithDocs();
        person.display();
    }
}
```



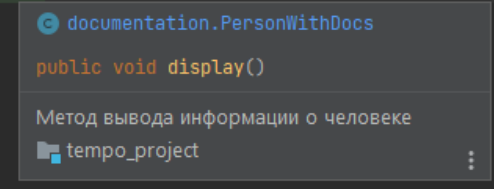
documentation

```
public class PersonWithDocs
```

Класс для хранения данных о человеке

tempo_project

```
public class Documentation {
    public static void main(String[] args) {
        PersonWithDocs person = new PersonWithDocs();
        person.display();
    }
}
```

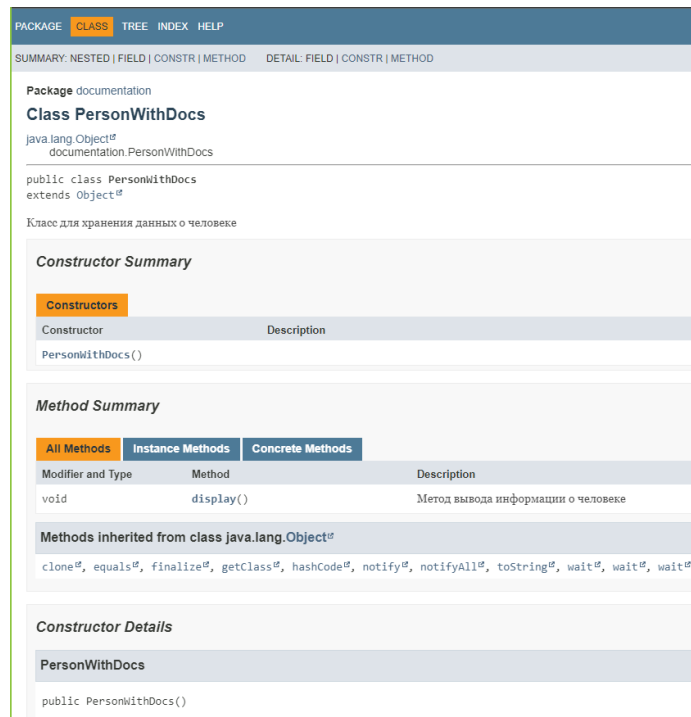


documentation.PersonWithDocs

```
public void display()
```

Метод вывода информации о человеке

tempo_project



PACKAGE CLASS TREE INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Package documentation

Class PersonWithDocs

java.lang.Object[®]
documentation.PersonWithDocs

public class PersonWithDocs
extends Object[®]

Класс для хранения данных о человеке

Constructor Summary

| Constructors | |
|------------------|-------------|
| Constructor | Description |
| PersonWithDocs() | |

Method Summary

| All Methods | Instance Methods | Concrete Methods |
|-------------------|------------------|------------------------------------|
| Modifier and Type | Method | Description |
| void | display() | Метод вывода информации о человеке |

Methods inherited from class java.lang.Object[®]

clone[®], equals[®], finalize[®], getClass[®], hashCode[®], notify[®], notifyAll[®], toString[®], wait[®], wait[®], wait[®]

Constructor Details

PersonWithDocs

public PersonWithDocs()



Example.zip

Документалочка

Документирование



JavaDoc может быть указан на уровне

- класса или интерфейса – включает описание класса (см. пример выше);
- поля – включает описание поля (см. пример выше);
- метода – включает описание метода, описание входных параметров, описание возвращаемого результата и описание бросаемых исключений

```
/**
 * Метод разделения полного имени на отдельные части
 * @param delimiter разделитель, по которому выделяются части имени
 * @return список частей имени человека
 * @throws Exception если имя человека не содержит указанного разделителя
 */
public List<String> getSeparatedName(String delimiter) throws Exception {
    if (!name.contains(delimiter)) throw new Exception("Имя не содержит разделитель " +
delimiter);
    return List.of(name.split(delimiter));
}
```

```
documentation.PersonWithDocs

public List<String> getSeparatedName(
    String delimiter
)
throws Exception

Метод разделения полного имени на отдельные
части

Params: delimiter – разделитель, по которому
выделяются части имени

Returns: список частей имени человека

Throws: Exception – если имя человека не
содержит указанного разделителя

tempo_project
```

Документалочка

Документирование

[Документация](#) Javadoc представлена на сайте Oracle. Основные теги:

- *@param*, *@return*, *@throws* – для описания методов;
- *{@link}* и *@see* – для создания встроенной ссылки на указанную часть исходного кода или внешнего ресурса;
- *@author* – имя автора, который добавил класс, метод или поле;
- *@since* указывает, какая версия класса, поля или метода была добавлена в проект;
- *@version* – указывает версию программного обеспечения;
- *@deprecated* – дает объяснение того, почему код является устаревшим, когда он стал устаревшим, и каковы альтернативы;
- *@code* – отображает текст шрифтом code без интерпретации текста как HTML-разметки или вложенных тегов javadoc;
- *{@inheritDoc}* – наследует Javadoc родительского класса или метода.

Документалочка

Документирование

Кроме перечисленных тегов JavaDoc поддерживает html-теги. Например,

1 `doc` = **doc**

2 `@see label` = see [label](#)

3 Список

`<dl>`

`<dt>Coffee</dt>`

`<dd>Black hot drink</dd>`

`<dt>Milk</dt>`

`<dd>White cold drink</dd>`

`</dl>`

Coffee

Black hot drink

Milk

White cold drink

Документалочка

Документирование

Запуск генерации документации:

- запуск из командной строки производится из директории проекта

*projectFolder: javadoc -d doc src|**

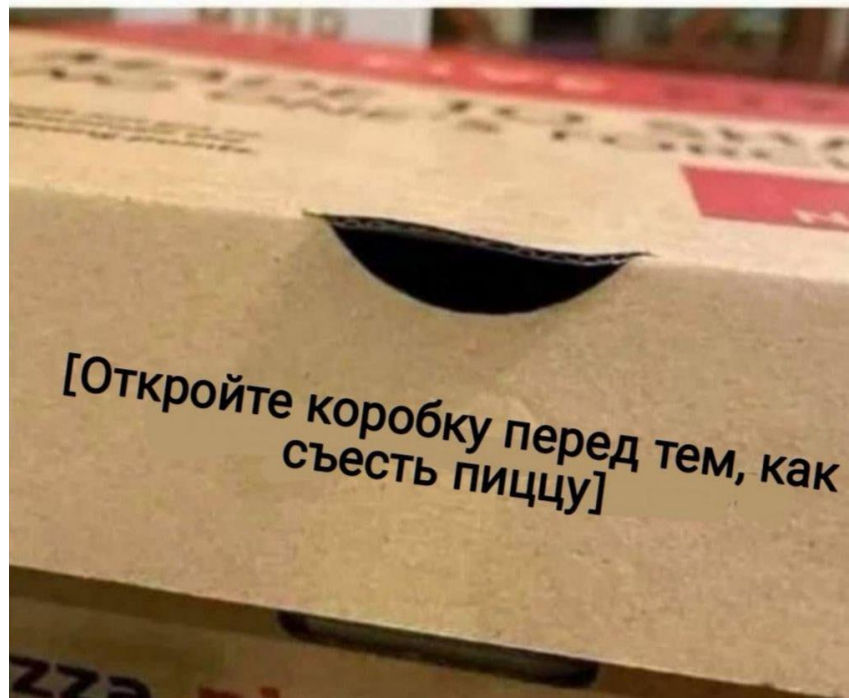
- для создания JavaDoc из IntelliJ IDEA используйте в главном меню

Tools -> Generate JavaDoc

Если Вы делали документацию на русском, то в Command Line Arguments нужно указать -
encoding utf8 -docencoding utf8 -charset utf8

- [пример](#) запуска с помощью плагина Maven (Maven изучим позже).

Каждый раз, когда я пишу документацию:



Задание

Для класса библиотечной карточки и его методов напишите Javadoc.

Сгенерируйте документацию.



2

Домашнее задание

Домашнее задание

- 1 Отгадайте загадку: с крыльями, но не летает, с цепью, но не лаает, со спицами, но не вяжет, с седлом, но не скачет, с рамой, но не дует.
- 2 Создайте класс Велосипед. Типы полей этого класса должны быть объявлены как внутренние классы (руль, седло, колесо, передачи, рама и т.д.). Каждая часть велосипеда помимо описания характеристик хранит запас прочности. Когда запас прочности равен 0, часть велосипеда ломается.
- 3 В основной программе создайте велосипед. Велосипед должен ехать и встречать на пути препятствия (яма, бордюр, битое стекло, лужа и т.д.). Каждое препятствие уменьшает показатель прочности на случайную величину у случайной части велосипеда. Программа выполняется, пока у велосипеда нет сломанных частей.
- 4 Напишите JavaDoc для написанных классов и методов. Одним из способов сгенерируйте документацию.

ЗАКЛЮЧЕНИЕ

