**Руководство Google по форматированию кода на Java.**

**https://google.github.io/styleguide/javaguide.html#s5-naming**

**1 Введение**

В данном документе описаны стандарты написания кода на языке программирования Java в корпорации Google. Исходный код Java считается соответствующим этим стандартам тогда и только тогда, когда он удовлетворяет всем описанным в документе правилам.

Затронутые в руководстве темы касаются не только эстетической стороны форматирования кода, но и других типов соглашений или стандартов кодирования. Тем не менее, данный документ концентрируется в первую очередь на определении **строгих правил**, которым мы повсеместно следуем, и не дает рекомендаций, которые могут быть неверно реализованы (как человеком, так и машинными инструментами).

**1.1 Терминология**

В рамках этого руководства определены следующие термины:

1. Термин *класс* используется для обозначения «обычного» класса, перечислений, интерфейса или типа аннотации (@interface)

2. Термин *член класса* используется для обозначения вложенного класса, поля, метода или *конструктора*, то есть для всех элементов класса высокого уровня, кроме блоков инициализации и комментариев

3. Термин *комментарий* всегда относится к комментариям *реализации*. Мы не используем словосочетание «комментарии к документации», а вместо этого используем термин «Javadoc»

Остальная терминология будет даваться по мере необходимости в разных местах данного руководства.

**1.2 Примечание к руководству**

Примеры кода в данном документе не демонстрируют единственно верный стилистический подход к его форматированию, хотя и используются в нашем руководстве.

**2 Исходный файл. Основы**

**2.1 Имя файла**

Имя исходного файла состоит из имени одного конкретного класса высшего уровня, находящегося в нем. Имя является регистрозависимым и заканчивается расширением *.java*

**2.2 Кодировка файла: UTF-8**

Для файлов с кодом используется кодировка **UTF-8.**

**2.3 Специальные символы**

**2.3.1 Символы пробела**

Помимо последовательности символов конца строки, **горизонтальный символ пробела ASCII (0×20)** является единственным символом-разделителем, встречающимся в исходном файле. Это означает, что:

1. Все прочие пробельные символы в символьных и строковых литералах экранируются
2. Символы табуляции **не** используются для отступов

**2.3.2 Специальные экранирующие последовательности**

Для каждого символа, для которого существует специальная экранирующая последовательность (\b, \t, \n, \f, \r, \", \' и \\), предпочтительнее использовать именно ее вместо соответствующего ей восьмеричного значения (например, \012) или кода Unicode (например, \u000a).

**2.3.3 Символы не из таблицы ASCII**

Для символов не из таблицы ASCII используется символ Unicode (например, ∞) или эквивалентная экранирующая последовательность (например, \u221e). Выбор отдается в пользу тех символов, которые делают код более понятным и читаемым.

При использовании символов экранирования в кодировке Unicode и в тех случаях, когда применяются действующие символы Unicode, убедительно рекомендуем такой код сопровождать объясняющими комментариями

| **Пример** | **Примечание** |
| --- | --- |
| String unitAbbrev = "μs" | Отлично: прекрасно воспринимается даже без комментария |
| String unitAbbrev = "\u03bcs" // "μs" | Допускается, но нет причин так делать |
| String unitAbbrev = "\u03bcs" // Греческая буква mu, "s" | Допускается, но неуклюже и может привести к ошибкам |
| String unitAbbrev = "\u03bcs" | Плохо: читающий понятия не имеет, что это такое |
| return '\ufeff' + content // знак порядка байтов | Хорошо: используйте экранирование для непечатаемых символов и комментируйте при необходимости |

Никогда не делайте ваш код менее читаемым только из опасения, что некоторые программы не смогут правильно обработать символы не из таблицы ASCII. Если такое происходит, такие программы работают неверно и их нужно исправить

**3 Структура исходного файла**

Исходный файл состоит из следующих элементов (в указанном порядке**)**:

1. Информация о лицензии или авторских правах, если имеется
2. Объявление пакета
3. Объявление импортов
4. Объявление класса

**Ровно одна пустая строка** разделяет каждый присутствующий раздел.

**3.1 Информация о лицензии или авторских правах, если имеется**

Информация о лицензии или авторских правах должна быть размещена в том файле, к которому она относится.

**3.2 Объявление пакета**

Пакет объявляется без переноса строки. Ограничение на ширину строки (Раздел 4.4) на объявление пакета не распространяется.

**3.3 Объявления импортов**

**3.3.1 Символ подстановки при объявлении импортов**

**Символ \* (подстановки)** при объявлении импортов, статических или нет, **не используется**.

**3.3.2 Перенос строки**

Импорты объявляются **без переноса строки**. Ограничение на ширину строки на них не распространяется.

**3.3.3 Упорядочивание и интервал**

Импорты упорядочиваются следующим образом:

1. Все статические импорты размещаются и группируются в одном блоке
2. Все не статические импорты размещаются в другом блоке

Если объявляются как статические, так и не статические импорты, то эти блоки нужно разделить пустой строкой. Других пустых строк между ними быть не должно.

Внутри каждого блока импортируемые классы следуют в порядке сортировки ASCII-символов.

**3.3.4 Статический импорт вложенных классов**

Статический импорт не используется для статических вложенных классов. Такие классы импортируются с объявлением обычного импорта.

**3.4 Объявление класса**

**3.4.1 Объявляется ровно один класс верхнего уровня**

Каждый класс верхнего уровня располагается в своем исходном файле.

**3.4.2 Упорядочивание содержимого класса**

Порядок, выбираемый при расположении членов и блоков инициализации класса, может оказать большое влияние на легкость понимания кода. Так или иначе, для этого не существует простого и правильного рецепта; разные классы могут упорядочивать свое содержимое по-разному.

Важно то, чтобы у каждого класса был **такой логический порядок** расположения содержимого, который позволил бы программисту, читающему код, объяснить его.

**3.4.2.1 Перегруженный код не должен быть разделен**

Когда у класса есть несколько конструкторов или методов с одинаковым именем, они должны располагаться последовательно, без вставок другого кода между ними.

**4 Форматирование**

**Терминология**

Тело класса, метода или конструктора относится к *блочной конструкции.*  
  
Обратите внимание, что согласно Разделу 4.8.3.1, любой инициализатор массива также может рассматриваться как блочная конструкция.

**4.1 Фигурные скобки**

**4.1.1 Фигурные скобки используются везде, где они могут быть использованы**

Фигурные скобки используются в if, else, for, while и do-while, даже если тело выражения пустое или содержит лишь одну строку кода.

**4.1.2 Непустые блоки: стиль K & R**

Фигурные скобки ставятся согласно стилю Кернигана и Ритчи («Египетские скобки») для непустых блоков и блочных конструкций:

* Перед открывающейся скобкой переход на новую строку не делается:

*// правильно*

**if** (**true**) {

*// неправильно*

**if** (**true**)

{

* Переход на новую строку делается после открывающей скобки:

*// правильно*

**while** (**true**) {

*// some code*

*// неправильно*

**while** (**true**) { *// some code*

* Переход на новую строку делается перед закрывающей скобкой:

*// правильно*

**for** (**true**) {

*// some code*

}

*// неправильно*

**while** (**true**) { */\* some code \*/* }

*// неправильно*

**if** (**true**) {

*/\* some code \*/* }

* Переход на новую строку делается после закрывающей скобки, только если эта скобка завершает выражение или тело метода, конструктора или не анонимный класс. Переход на новую строку не делается после скобки, если за ней следует else, catch или точка с запятой

Примеры правильно выполненных правил:

**return** () -> {

**while** (condition()) {

method();

}

};

**return** **new** MyClass() {

@Override **public** **void** **method**() {

**if** (condition()) {

**try** {

something();

} **catch** (ProblemException e) {

recover();

}

} **else** **if** (otherCondition()) {

somethingElse();

} **else** {

lastThing();

}

}

};

Некоторые исключения для перечислений приведены в Разделе 4.8.1

**4.1.3 Пустые блоки могут быть сжатыми**

Пустой блок или пустая блочная конструкция может следовать стилю K & R (как описано в Разделе 4.1.2). Также возможно, чтобы такой блок был закрыт сразу же после открытия, без символов или разрыва строки внутри {}. Это правило не относится к случаю, когда блок является частью многоблочного выражения, которое содержит if-else или try-catch-finally.  
  
Примеры:

*// Приемлемо*

**void** **doNothing**() {}

*// Также приемлемо*

**void** **doNothingElse**() {

}

*// Неприемлемо: нельзя использовать пустые блоки в многоблочном выражении*

**try** {

doSomething();

} **catch** (Exception e) {}

**4.2 Два пробела для отступа**

Каждый раз, когда открывается новый блок или блочная конструкция, смещение вправо увеличивается на два пробела. Когда блок заканчивается, начало следующей строки кода смещается на предыдущий уровень смещения. Уровень смещения применяется как к блоку, так и к комментариям в этом блоке (см. пример в Разделе 4.1.2).

**4.3 Одно выражение на каждую строку**

Каждое выражение завершается переходом на новую строку.

**4.4 Ограничение ширины строки в 100 символов**

Java-код имеет ограничение в 100 символов по ширине строки. Под «символом» понимается любой из элементов Unicode. За исключением случаев, описанных ниже, каждая строка с превышением ограничения по ширине, должна быть перенесена так, как это объяснено в Разделе 4.5.

**Исключения:**

1. Строки, в которых соблюдение ограничения по ширине невозможно (например, длинная ссылка URL в Javadoc или длинная JSNI-ссылка на метод)
2. Объявления package и import (см. Разделы 3.2 и 3.3)
3. Строки с командами в комментариях, которые могут быть скопированы и вставлены для выполнения в терминале

**4.5 Перенос строки**

Терминология

Когда код, который в другом случае мог бы быть расположен на одной строке, разделяется на несколько строк, это явление называется переносом строки.

Не существует общепринятой однозначной формулы, определяющей, как именно надо делать перенос строки в каждой ситуации. Очень часто существует несколько способов переноса строки с одним и тем же фрагментом кода.

Обычно перенос делается во избежание переполнения строки по ширине. Но даже если код остался бы в пределах разрешенной ширины, то и он, по решению автора, может быть перенесен на новую строку.

Выделение вспомогательного метода или локальной переменной может решить проблему переполнения строки по ширине без переноса кода

**4.5.1 Где делать перенос**

Первое указание по переносу строки гласит: предпочтительнее делать перенос на более высоком синтаксическом уровне. Также:

1. Когда строка разрывается не на операторе присваивания, разрыв делается перед символом.

Это правило также применимо к следующим «оператороподобным» символам:

* разделяющая точка «.»
* двойное двоеточие ссылочного метода «::»
* амперсанд в скобках дженерика <T extends Foo & Bar>
* разделитель в catch-блоке catch (FooException | BarException e)

2. Когда строка переносится на операторе присваивания, перенос обычно делается после символа, но приемлемо и другое решение  
  
Это также применимо к двоеточию для цикла for-each.

3. Имя метода или конструктора при переносе строки остается присоединенным к открывающей скобке «(»

4. Запятая «,» при переносе строки остается с элементом, который ей предшествует

5. Строка никогда не переносится непосредственно у стрелки лямбда-выражения, кроме случаев, когда его тело состоит из одного выражения без фигурных скобок:

MyLambda<String, Long, Object> lambda =

(String label, Long value, Object obj) -> {

...

};

Predicate<String> predicate = str ->

longExpressionInvolving(str);

Главная цель переноса строки — добиться ясности кода, но не обязательно наименьшего количества строк

**4.5.2 Смещение продолжения строки на 4 и более пробелов**

При переносе строки каждая следующая ее подстрока (каждое продолжение строки) смещается как минимум на 4 пробела относительно предыдущей.  
  
Когда продолжений строки несколько, смещение может варьироваться в пределах 4 пробелов по желанию автора. По общему правилу, два продолжения строки могут иметь одинаковое смещение тогда и только тогда, когда они начинаются с синтаксически параллельных элементов.  
  
В Разделе 4.6.3 даются указания по использованию различного количества пробелов для выравнивания элементов кода относительно предыдущих строк.

**4.6 Пробелы и отступы**

**4.6.1 Отступы**

Одна пустая строка всегда ставится:  
  
1. Между следующими друг за другом членами или инициализаторами класса: полями, конструкторами, методами, вложенными классами, статическими и динамическими блоками инициализации

* исключение: пустая строка между двумя последовательными полями (без кода между ними) используется опционально. При необходимости пустые строки используются для логического группирования полей
* исключение: пустые строки между константами класса enum (см. Раздел 4.8.1)

2. В соответствии с другими разделами данного документа (например, с Разделом 3 и Разделом 3.3)  
  
Пустая строка также может быть использована повсеместно для повышения читаемости кода, например, между выражениями для организации кода в логические подразделы. Пустая строка перед первым членом класса, или блоком инициализации, или после последнего члена, или блока инициализации класса не приветствуется, но и не возбраняется.  
  
Несколько последовательных пустых строк делать разрешается, но это не необходимо и не приветствуется.

**4.6.2 Пробелы**

Помимо требований самого языка или прочих правил данного документа, а также не считая литералов и комментариев (в т.ч. Javadoc), одиночные пробелы из таблицы ASCII могут присутствовать только в следующих местах:  
  
1. При разделении любого зарезервированного слова, такого как if, for или catch, и открывающей круглой скобки «(», которая следует за ним  
  
2. При разделении любого зарезервированного слова, такого как else или catch, и закрывающей фигурной скобки «}», которая следует за ним  
  
3. Перед любой открывающей фигурной скобкой «{», за исключением двух ситуаций:

* @SomeAnnotation({a, b})
* String[][] x = {{"foo"}}; - пробел между {{ не нужен согласно п. 8 ниже

4. По обе стороны от любого бинарного или тернарного оператора  
  
Это правило также применимо к следующим операторам:

* амперсанд внутри угловых скобок: <T extends Foo & Bar>
* разделитель в блоке catch, содержащий несколько исключений: catch (FooException | BarException e)
* двоеточие «:» в  for-each
* стрелка в лямбда-выражении: (String str) -> str.length()

Но это правило не применимо к операторам:

* двойное двоеточие «::» ссылочного метода, которое пишется как Object::toString
* разделяющая точка «.», которая пишется как object.toString()

5. После «,:;» или закрывающей круглой скобки «)» при приведении типа  
  
6. По обе стороны от двойной косой черты «//» при создании комментария в той же строке кода. Здесь разрешены, но не необходимы несколько пробелов  
  
7. Между объявлением типа и именем переменной: List<String> list  
  
8. Опционально: внутри скобок инициализатора массива

* new int[] {5, 6} и new int[] { 5, 6 } — оба варианта верные

9. Между аннотацией типа и [] или ...  
  
Это правило не требует наличия или отсутствия пробелов в начале или конце строки; оно относится лишь к внутренним пробелам.

**4.6.3 Горизонтальное выравнивание не требуется никогда**

Терминология

Горизонтальное выравнивание — это практика добавления различного числа дополнительных пробелов в вашем коде с целью сделать так, чтобы определенные элементы были расположены под другими элементами с предыдущей строки.

Эта практика разрешена, но ее использование не требуется данным руководством. Также нет необходимости поддерживать соблюдение выравнивания в тех участках кода, где оно уже было применено.  
  
Пример с выравниванием и без него:

**private** **int** x; *// хорошо*

**private** Color color; *// и это тоже*

**private** **int** x; *// разрешено, но при редактировании в будущем*

**private** Color color; *// можно оставить без выравнивания*

Выравнивание способствует читаемости кода, но создает проблемы с поддержкой такого кода в будущем. Допустим, требуется изменить только одну строку. Это изменение может исказить форматирование ранее принятого кода, что допускается. Но скорее всего, программист (возможно, вы) начнет корректировку количества пробелов на рядом стоящих строках, что, возможно, запустит целую череду исправлений. Изменение одной строки может активировать «взрывную волну» из бессмысленного труда (в худшем случае). Или, в лучшем случае, приведет к искажению информации в истории версий, ухудшит чтение кода и обострит конфликты слияния.

**4.7 Группирующие скобки рекомендованы**

Допускается не ставить группирующие скобки только тогда, когда автор кода и рецензент согласны в том, что нет разумной вероятности, что код будет неверно истолкован без скобок, и при этом скобки не облегчили бы его чтение. Нет причины полагать, что каждый, кто читает код, помнит наизусть всю таблицу приоритетов использования операторов Java.

**4.8 Особые конструкции**

**4.8.1 Классы-перечисления**

После каждой запятой, которая следует за константой перечисления, разрыв строки необязателен. Дополнительные пустые строки (обычно только одна) также допускаются. Вот пример такого кода:

**private** **enum** Answer {

YES {

@Override **public** String **toString**() {

**return** "yes";

}

},

NO,

MAYBE

}

Код класса-перечисления без методов и комментариев, описывающих его константы, может быть представлен, как инициализатор массива (см. Раздел 4.8.3.1):

**private** **enum** Suit { CLUBS, HEARTS, SPADES, DIAMONDS }

Так как перечисления — это классы, для них следует применять все прочие правила, применимые к классам.

**4.8.2 Объявления переменных**

**4.8.2.1 Одна переменная на одно объявление**

Каждая переменная (поле или локальная) объявляется только по одной за раз: такие объявления, как int a, b; не используются.  
  
**Исключение:**Множественные объявления переменных допускаются в заголовке цикла for.

**4.8.2.2 Объявляйте переменные тогда, когда они нужны**

Локальные переменные необязательно должны объявляться в начале блока или блочной конструкции. Наоборот, локальные переменные необходимо объявлять непосредственно перед местом их первого использования, чтобы минимизировать область их действия. Обычно локальные переменные инициализируются в момент объявления, или же сразу после него.

**4.8.3 Массивы**

**4.8.3.1 Инициализаторы массивов могут быть «блочными»**

Любой массив может быть инициализирован так, как если бы он был блочной конструкцией. К примеру, весь следующий код допустим (список примеров не полный):

**new** **int**[] {

0, 1, 2, 3

}

**new** **int**[] {

0, 1,

2, 3

}

**new** **int**[]

{0, 1, 2, 3}

**new** **int**[] {

0,

1,

2,

3,

}

**4.8.3.2 Никаких объявлений массивов в стиле языка С**

Квадратные скобки ставятся после типа, а не после переменной: String[] args, а не String args[].

**4.8.4 Оператор switch**

Терминология

Внутри блока switch располагается одна или более групп операторов. Каждая группа состоит из одной или более меток (как case FOO:, так и default:), за которыми следует один или более операторов (или, в случае последней группы, ни одного или более).

**4.8.4.1 Смещение**

Как и в случае с любым другим блоком, содержимое блока switch смещается на 2 пробела.  
  
После метки блока switch делается перенос строки, а смещение увеличивается на 2 пробела, так же как при открытии блока. Каждая следующая метка возвращается на предыдущий уровень смещения, как при закрытии блока.

**4.8.4.2 Сквозной проход комментируется**

Внутри блока каждая группа операторов либо завершает досрочно switch (с помощью break, continue, return или выбросом исключения), либо помечается комментарием, чтобы обозначить, что выполнение кода будет или может быть продолжено в следующей группе. Достаточно любого комментария, передающего идею сквозного прохода (обычно // fall through). Такой особый комментарий не требуется в последней группе блока switch. Пример:

**switch** (input) {

**case** 1:

**case** 2:

prepareOneOrTwo();

*// fall through*

**case** 3:

handleOneTwoOrThree();

**break**;

**default**:

handleLargeNumber(input);

}

Обратите внимание, что комментарий ставится не после case 1, а лишь в конце группы операторов.

**4.8.4.3 Всегда используйте default**

Оператор switch должен содержать метку default, даже если в ней не присутствует код.  
  
**Исключение:**блок switch для типа enum может не использовать default, если он содержит явные случаи, охватывающие все возможные значения этого типа. Это позволяет среде разработки или другим инструментам статического анализа выдавать предупреждение о том, что некоторые случаи не охвачены.

**4.8.5 Аннотации**

Аннотации, применяемые к классу, методу или конструктору, следуют непосредственно после блока документации. Каждая аннотация указывается на собственной строке (то есть по одной аннотации на строку). Эти разрывы строки не являются переносами строки (см. Раздел 4.5), поэтому уровень отступа не увеличивается. Пример:

@Override

@Nullable

**public** String **getNameIfPresent**() { ... }

**Исключение:** одиночная аннотация без параметров может отображаться вместе с первой строкой сигнатуры, например:

@Override **public** **int** **hashCode**() { ... }

Аннотации, применяемые к полю, также появляются сразу после блока документации, но в этом случае несколько аннотаций (возможно, параметризованных) могут быть перечислены в одной строке, например:

@Partial @Mock DataLoader loader;

Не существует особых правил для форматирования аннотаций к параметрам, локальным переменным или типам.

**4.8.6 Комментарии**

Этот раздел посвящен комментариям реализации. Javadoc рассматривается отдельно в Разделе 7.  
  
Любому разрыву строки может предшествовать произвольное количество пробелов, за которым следует комментарий реализации. Такой комментарий делает строку непустой.

**4.8.6.1 Стиль блочного комментария**

Уровень отступа блочного комментария совпадает с окружающим кодом. Блочные комментарии могут быть как /\* … \*/, так и // … Для многострочных комментариев вида /\* … \*/ последующие строки должны начинаться с символа \*, выравненного с символом \* с предыдущей строки.

*/\**

*\* This is // And so /\* Or you can*

*\* okay. // is this. \* even do this. \*/*

\*/

Комментарии не заключаются в прямоугольники, изображенные звездочками или другими символами.

При написании многострочных комментариев используйте стиль /\* … \*/, если хотите, чтобы средства автоматического форматирования кода делали перенос строки при необходимости (в стиле параграфов). Большинство средств форматирования не могут делать этого с блоками однострочных комментариев // …

**4.8.7 Модификаторы**

Модификаторы класса и полей, если они присутствуют, отображаются в порядке, рекомендованном спецификацией языка Java:

**public** **protected** **private** **abstract** **default** **static** **final** **transient** **volatile** **synchronized** **native** **strictfp**

**4.8.8 Числовые литералы**

Тип long использует прописную букву L, а не строчную (чтобы не перепутать с цифрой 1). Например, 300\_000\_000L вместо 300\_000\_000l.

**5 Именование**

**5.1 Общие правила для всех идентификаторов**

Идентификаторы используют только буквы и цифры ASCII и, в некоторых отмеченных ниже случаях, подчеркивания.

Таким образом, каждому действительному имени идентификатора соответствует регулярное выражение \w+ (буквенно-цифровой символ, встречающийся один или более раз).

Стилю данного руководства не соответствуют имена, использующие специальные суффиксы или префиксы, например: name\_, mName, s\_name или kName.

**5.2 Правила для разных типов идентификаторов**

**5.2.1 Имена пакетов**

Имена пакетов должны быть записаны в нижнем регистре, без использования camelCase или подчеркиваний.

Правильно: com.example.deepspace  
Неправильно: com.example.deepSpace или com.example.deep\_space

**5.2.2 Имена классов**

Имена классов пишутся в стиле UpperCamelCase (с заглавной первой буквой).

Имена классов обычно являются существительными или словосочетаниями с существительными. Например, Character или ImmutableList.

Имена интерфейсов также могут быть существительными или словосочетаниями с существительными (например, List), но иногда могут быть и прилагательными или сочетаниями прилагательных (например, Readable).

Не существует конкретных правил или даже устоявшихся соглашений для именования типов аннотаций.

Тестовые классы носят имя, которое начинается с имени класса, который они тестируют, и заканчивается словом Test. Например, HashTest или HashIntegrationTest.

**5.2.3 Имена методов**

Имена методов пишутся в стиле lowerCamelCase.

Имена методов обычно являются глаголами или словосочетаниями с глаголами. Например, sendMessage или stop.

Подчеркивания могут присутствовать в именах тестовых методов JUnit для разделения логических компонентов в имени. При этом каждый компонент пишется в стиле lowerCamelCase. Типичным шаблоном является <methodUnderTest>\_<state>, например, pop\_emptyStack. Не существует единственно верного пути в именовании тестовых методов.

**5.2.4 Имена констант**

Константы именуются в стиле CONSTANT\_CASE: все буквы в верхнем регистре, каждое слово отделено от следующего подчеркиванием. Но что именно является константой?

Константы — это статические финальные поля, содержимое которых является неизменным, и методы, которые не имеют видимых побочных эффектов. Это относится к примитивам, String, неизменяемым типам и неизменяемым коллекциям неизменяемых типов. Если какое-либо наблюдаемое состояние объекта может измениться, он не является константой. Простого намерения никогда не изменять объект недостаточно.

Примеры:

*// Константы*

**static** **final** **int** NUMBER = 5;

**static** **final** ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");

**static** **final** ImmutableMap<String, Integer> AGES = ImmutableMap.of("Ed", 35, "Ann", 32);

**static** **final** Joiner COMMA\_JOINER = Joiner.on(','); *// because Joiner is immutable*

**static** **final** SomeMutableType[] EMPTY\_ARRAY = {};

**enum** SomeEnum { ENUM\_CONSTANT }

*// Не константы*

**static** String nonFinal = "non-final";

**final** String nonStatic = "non-static";

**static** **final** Set<String> mutableCollection = **new** HashSet<String>();

**static** **final** ImmutableSet<SomeMutableType> mutableElements = ImmutableSet.of(mutable);

**static** **final** ImmutableMap<String, SomeMutableType> mutableValues =

ImmutableMap.of("Ed", mutableInstance, "Ann", mutableInstance2);

**static** **final** Logger logger = Logger.getLogger(MyClass.getName());

**static** **final** String[] nonEmptyArray = {"these", "can", "change"};

Имена констант обычно являются существительными или словосочетаниями с существительными.

**5.2.5 Имена не константных полей**

Имена полей, не являющихся константами (статических или нет), пишутся в стиле lowerCamelCase.

Имена таких полей обычно являются существительными или словосочетаниями с существительными. Например, computedValues или index.

**5.2.6 Имена параметров**

Имена параметров пишутся в стиле lowerCamelCase.

В публичных методах следует избегать имен параметров, состоящих из одного символа.

**5.2.7 Имена локальных переменных**

Имена локальных переменных пишутся в стиле lowerCamelCase.

Даже будучи финальными и неизменяемыми, локальные переменные не считаются константами и не должны писаться в том же стиле, что и константы.

**5.2.8 Имена переменных типа**

Каждая переменная типа именуется согласно одному из двух стилей:

* Одиночная заглавная буква, за которой может следовать обычное число (например, E, T, X, T2)
* Имя в виде имени класса (см. Раздел 5.2.2), за которым следует заглавная буква T (примеры: RequestT, FooBarT).

**5.3 «Верблюжий» стиль (camelCase)**

Иногда существует более одного способа преобразовать фразу на английском языке в «верблюжий» стиль, как например в случае с аббревиатурами или нетипичными выражениями вроде «IPv6» или «iOS».  
Для повышения предсказуемости, данное руководство задает следующую (примерную) схему.

Начинаем с исходной формы имени:

1. Преобразуйте фразу в обычный ASCII и удалите все апострофы. Например, «Müller's algorithm» можно преобразовать в «Muellers algorithm»  
2. Разделите полученный результат на слова, отбрасывая пробелы и оставшиеся знаки пунктуации (обычно дефисы):

* рекомендация: если какое-либо слово уже имеет общепринятую форму в обычном «верблюжьем» стиле, разделите его на составные части (например, «AdWords» преобразуется в «ad words»). Обратите внимание, что такое слово, как «iOS», на самом деле не совсем в «верблюжьем» стиле; оно не соответствует каким-либо соглашениям, поэтому данная рекомендация не применяется.

3. Теперь переведите все в нижний регистр (включая аббревиатуры), затем переведите в верхний регистр первый символ:

* … в каждом слове, чтобы достичь стиля UpperCamelCase, или
* … в каждом слове, кроме первого, чтобы достичь стиля lowerCamelCase

4. Наконец, соедините все слова в одиночный идентификатор

Обратите внимание, что регистр исходных слов почти полностью игнорируется.  
  
Примеры:

| **Исходная форма** | **Правильно** | **Неправильно** |
| --- | --- | --- |
| "XML HTTP request" | XmlHttpRequest | XMLHTTPRequest |
| "new customer ID" | newCustomerId | newCustomerID |
| "inner stopwatch" | innerStopwatch | innerStopWatch |
| "supports IPv6 on iOS?" | supportsIpv6OnIos | supportsIPv6OnIOS |
| "YouTube importer" | YouTubeImporter YoutubeImporter\* |  |

\*Допускается, но не рекомендуется.

**Примечание:**

Некоторые слова в английском языке используют дефис неоднозначно: например, оба слова «nonempty» и «non-empty» являются правильными, поэтому имена методов checkNonempty и checkNonEmpty также являются правильными.

**6 Практика программирования**

**6.1 Используйте всегда аннотацию @Override**

Метод помечается аннотацией @Override всякий раз, когда он действительно переопределяется. Это относится как к методу класса-потомка, переопределяющему метод класса-родителя, так и к методу интерфейса, переопределяющему метод супер-интерфейса.

**Исключение:**

Аннотация может быть опущена, если родительский метод помечен аннотацией @Deprecated.

**6.2 Не игнорируйте пойманные исключения**

Очень редко возникают ситуации, когда не нужно предпринимать ни каких действий в ответ на пойманное исключение (типичным решением является занесение в лог или, если это считается «невозможным», перебросить исключение как AssertionError).

Ниже приведен пример с поясняющим комментарием, когда действительно уместно не предпринимать никаких действий в блоке catch:

**try** {

**int** i = Integer.parseInt(response);

**return** handleNumericResponse(i);

} **catch** (NumberFormatException ok) {

*// it's not numeric; that's fine, just continue*

}

**return** handleTextResponse(response);

**Исключение:**

В тестах пойманное исключение может быть проигнорировано и не сопровождено комментарием, если именем теста является expected или же, если имя начинается с expected. Ниже приводится очень распространенная идиома, показывающая, что тестируемый код генерирует исключение ожидаемого типа, поэтому здесь комментарии не нужны:

**try** {

emptyStack.pop();

fail();

} **catch** (NoSuchElementException expected) {

}

**6.3 Для статических членов используйте имя класса**

Обращаться к члену статического класса необходимо через имя класса, а не по ссылке на объект класса или по выражению, возвращающему этот объект:

Foo aFoo = ...;

Foo.aStaticMethod(); *// good*

aFoo.aStaticMethod(); *// bad*

somethingThatYieldsAFoo().aStaticMethod(); *// very bad*

**6.4 Не используйте финализаторы**

Крайней редко необходимо переопределить метод Object.finalize.

**Подсказка:**

Не делайте этого. Если вам действительно это нужно, сначала прочитайте и очень тщательно осмыслите Effective Java Item 7, «Avoid Finalizers», а затем — не делайте этого.

**7. Javadoc**

**7.1 Форматирование**

**7.1.1 Основная форма**

Простое форматирование блоков Javadoc соответствует данному примеру:

*/\*\**

*\* Multiple lines of Javadoc text are written here,*

*\* wrapped normally...*

*\*/*

**public** **int** **method**(String p1) { ... }

… или в одну строку:

*/\*\* An especially short bit of Javadoc. \*/*

Простая форма применима всегда. Однострочная форма может быть применена, когда весь блок Javadoc (включая маркеры комментариев) может поместиться на одной строке. Обратите внимание, что это применимо лишь тогда, когда в блоке нет таких тэгов, как @return.

**7.1.2 Параграфы**

Одна пустая строка, то есть строка, содержащая только выровненную ведущую звездочку (\*), присутствует между абзацами и перед группой блочных тэгов, если таковые имеются. Каждый абзац, кроме первого, содержит <p> непосредственно перед первым словом, без пробела после.

**7.1.3 Блочные тэги**

Все блочные тэги следуют в таком порядке: @param, @return, @throws, @deprecated, и эти четыре типа никогда не присутствуют с пустым описанием. Если тег блока не помещается на одной строке, строки продолжения имеют отступ в четыре (или более) пробела от @.

**7.2 Итоговый фрагмент**

Каждый блок Javadoc начинается с краткого итогового фрагмента. Этот фрагмент очень важен: это единственный текст, который присутствует в определенном контексте, таком как индексы классов и методов.

Этот фрагмент — словосочетание с существительным или глаголом, а не с полным предложением. Он не начинается с A {@code Foo} is a… или This method returns…, и не образует законченного утвердительного предложения, как например Save the record. Однако этот фрагмент пишется с заглавной буквы, и в нем проставляются знаки пунктуации, как если бы это было законченное предложение.

**Подсказка:**

Распространенной ошибкой является написание простого Javadoc в виде /\*\* @return the customer ID \*/.  
Это неверно и должно быть исправлено на /\*\* Returns the customer ID. \*/.

**7.3 Когда применяется Javadoc**

Javadoc присутствует по меньшей мере в каждом public классе и в каждом public и protected члене такого класса, за исключением некоторых случаев, описанных ниже.

Дополнительный Javadoc может присутствовать, как это разъяснено в Разделе 7.3.4, Необязательный Javadoc.

**7.3.1 Исключение: методы, описывающие себя**

Javadoc опционален для простых и очевидных методов, таких как getFoo, в тех случаях, когда действительно нельзя сказать ничего более, чем «Возвращает foo».

**Важно**: неуместно ссылаться на это исключение, чтобы оправдать пропуск соответствующей информации, которую может потребоваться обычному читателю.

Например, для метода с названием getCanonicalName не опускайте документацию (с тем основанием, что имя метода говорит лишь /\*\* Возвращает каноническое имя. \*/), если обычный человек, читающий код, может и не подозревать, что означает термин «каноническое имя»!

**7.3.2 Исключение: переопределение**

Javadoc не всегда сопровождает метод, который переопределяет метод из супер-класса (или супер-интерфейса).

**7.3.4 Необязательный Javadoc**

Прочие классы и члены сопровождаются Javadoc по необходимости или по желанию.

Всякий раз, когда комментарий реализации будет использоваться для определения общей цели или поведения класса или члена, этот комментарий записывается как Javadoc (с использованием /\*\*).

Необязательный Javadoc не обязательно должен следовать правилам форматирования Разделов 7.1.2, 7.1.3 и 7.2, хотя это, конечно, рекомендуется.