**Концепция чистого кода**

- предполагает написание кода, который легко читать, понимать, сопровождать и расширять.

**Форматирование:**

1. Использовать форматирование в соответствии с [конвенцией](https://www.oracle.com/technetwork/java/codeconventions-150003.pdf) или принятом на проекте стандартом.
2. Последовательность в порядке объявления членов класса: Поля, конструкторы, public методы, private методы.
3. Использовать логическое разделение класса на блоки с помощью дополнительных пустых строк.

**Именование**:

1. Должно раскрывать суть программного члена – должны быть достаточно информативными, чтобы читатель мог сразу понять его назначение.
2. Легко читаемо (не использовать нотации, сокращения, leet (буквы меняются на цифры) и другие вводящие в заблуждение конструкции).
3. Имя должно даваться в соответствии с областью видимости (переменные, используемые в небольших блоках кода, могут иметь короткие имена, тогда как глобальные переменные или переменные класса должны иметь более описательные имена).
4. Если переменная окружена контекстом, то его не нужно дублировать в имени (контекст, в котором используется переменная, часто уже даёт достаточно информации о её назначении).
5. Имена классов (подбираются исходя из их состояния, а не поведения) и переменных существительные, функций/методов – глаголы. Boolean переменные и методы, возвращающие Boolean называются предикативно (с использованием can/is). Утилитарные классы обычно заканчиваются на s (Arrays, Collections и др.)

**Методы**:

1. Длина метода не более 10 строк (в идеале) или чтобы метод помещался на экран. Для достижения использовать **extract till you drop** – методика рефакторинга, при котором функционал распухшего метода выделяется в более мелкие и простые, процесс повторяется до тех пор, пока извлекать станет нечего.
2. Соблюдение SRP – Single Responsibility Principle (также достигается за счет методики **ETYD**)
3. Избегать Магических чисел – не хардкодить в методах значения, а выносить их в именованные константы (создавать для них понятный контекст).
4. Объявлять переменные как можно ближе к их первому использованию (объявление всех переменных в начале вынуждает погружаться в лишний контекст)
5. Использовать подходящую конструкцию условного выражения для упрощения восприятия кода (вроде как switch предпочтительно избегать из-за громоздкости и нетипичного синтаксиса).
6. Использовать описательные переменные или методы для пересечения флагов в условных выражениях.

//без использования описательной переменной  
if (user.isActive() && !user.isSuspended() && user.hasPermission("edit")) {  
 editContent();  
}  
//с использованием описательной переменной  
boolean canEditContent = user.isActive() && !user.isSuspended() && user.hasPermission("edit");  
if (canEditContent) {  
 editContent();  
}

1. Предпочитать позитивные условные выражения (если использовать негативный, то при необходимости проверки положительного утверждения нужно использовать двойное отрицание)

//отрицательный флаг  
if (!(order.isNotProcessed())) {  
 processOrder();  
}  
//положительный флаг  
if (order.isProcessed()) {  
 processOrder();  
}

1. Использовать **guard clause** – техника, которая подразумевает проверку предусловий в начале метода и выход в случае невыполнения условий (улучшает читаемость, избавляет от вложенных if, предотвращает исполнение лишнего кода).

// Без использования guard clause  
if (b != 0) {  
 int result = a / b;  
 System.out.println("Результат деления: " + result);  
} else {  
 System.out.println("Деление на ноль");  
}  
//развернем условие,   
// тем самым вынесем условие для основного кода метода наверх  
if (b == 0) {  
 System.out.println("Деление на ноль");  
 return;  
}  
int result = a / b;  
System.out.println("Результат деления: " + result);

1. В теле функции все должно быть на одном уровне абстракции.
2. Минимизация количества аргументов (затрудняет понимание назначения метода). В идеале метод должен получать аргументы из окружения. Для достижения использовать объекты для группировки связанных аргументов, дефолтные значения и перегрузку методов.
3. Избегать использование Boolean аргументов для методов. Предпочтительным способом будет разделить метод на несколько, но без флага, а его назначение отразить в названии.

public void configure(String host, int port, boolean useSSL) {  
 if (useSSL) {  
 // Логика для использования SSL  
 } else {  
 // Логика для обычного подключения  
 }  
}

//разделенный метод  
public void configureWithSSL(String host, int port) {  
 // Логика для использования SSL  
}  
  
public void configureWithoutSSL(String host, int port) {  
 // Логика для обычного подключения  
}

1. Метод не должен иметь сайд эффектов – то есть метод должен выполнять операцию, для которой он предназначен, не изменяя при этом окружения.
2. Менять объект, переданный в качестве параметров может только void т.к. если возвращается что-то другое, подразумевается, что объект, на котором вызвался метод не изменялся.
3. Не возвращать null:
   1. вызывающий такой метод код обязан проверять на null
   2. в разработке предпочтительным является подход fail fast (раннее сигнализирование об ошибке) – если, например, вызывается метод, который положит возвращаемое значение в коллекцию, то этот null обнаружится спустя время и обнаружить следы его возникновения будет проблематично.

Альтернативой null является: если ситуация нештатная, то бросается исключение (например, objectNotFound). Если семантика метода подразумевает возвращение null как значение, то использовать **Null-object** или Optional. Если метод все-таки возвращает null, то клиент метода должен быть осведомлен об этом с помощью названия метода или аннотации @Nullable. [Материал.](https://javarush.com/groups/posts/1147-pochemu-null---ehto-plokho)

**Комментарии:**

В идеале код должен быть самодокументирован (должен быть понятен без комментария) за счет соответствующего именования его структур. Должны использоваться только в случае, если нельзя раскрыть детали реализации в виде кода (то есть отвечать на вопрос почему – например описание выбранного алгоритма, почему совершаются неочевидные действия в коде, предупреждение о последствиях рефакторинга). Если нет возможности отрефакторить здесь и сейчас, то возможно написание комментария в виде toDO.

**Прочие рекомендации:**

1. Следование закону **Деметры** (принцип наименьшего знания) – каждый программный модуль должен обладать ограниченным знанием о других модулях (знает только о модулях, которые имеют непосредственное к нему отношение). Или же – никакой объект не должен раскрывать свою структуру тем, кто к нему обращается. Направлен на достижение слабой связанности между модулями (law coupling по GRASP паттернам) и улучшение инкапсуляции (минимизации зависимостей между классами).

*Пример:*

public class PaperBoy {  
   
 public void DeliverMagazine(int cost, Customer customer) {  
  
 Wallet wallet = customer.getWallet();  
 //логика взятия оплаты  
 }  
}

Класс PaperBoy напрямую взаимодействует с полем класса Customer и изменяет его состояние. В данном случае класс PaperBoy взаимодействует сразу с двумя классами (Customer и Wallet), в таком случае изменения в классах повлекут за собой необходимость изменения логики класса PaperBoy.

public class PaperBoy {  
  
 public void DeliverMagazine(int cost, Customer customer) {  
 customer.getPayment(cost);  
 //логика взятия оплаты  
 }  
}

В исправленном варианте классу Customer был добавлен метод getPayment(), который инкапсулирует работу с Wallet, теперь изменения в этом классе никак не повлияют на класс PaperBoy.

\*\*Управлять не ногами собаки, если нужно чтобы она подошла, а самой собакой, она знает как ими управлять ☺ \*\*