S.O.L.I.D принципы - способ регламентировать код таким образом, чтобы он был понятен, удобен для повторного использования, дополнения.

1) Принцип единственной обязанности

У каждого класса должна быть единственная по функциям обязанность. Содержать только те методы, которые реально связанны с объектом.

За классом должна быть очевидно закреплена его обязанность.

2) Принцип открытости/закрытости

Классы должны быть закрыты для модификации, но открыты для расширения. В хорошо спроектированных программах новая функциональность вводится путем добавления нового кода, а не изменения старого. private поля будут участвовать только в логике работы этого самого класса. Данные закрывать, а методы открывать. Хорошим тоном является для каждого поля создавать метод, читающий его и меняющий.

3) Принцип подстановки Барбары Лисков

Разделение регламентации поведения. В любой точки использования объекта в случае подстановки вместо него наследника должно работать корректно, согласно ожиданиям. Наследуйся правильно! Если метод вкладывает в себя "плыть"(уточки), то он должен работать корректно во всех случаях. (Прямоугольник - квадрат). Решение - добавлять доп методы, выносить общие методы в отдельные интерфейсы. Зависит от контекста требований программы.

4) Принцип разделения интерфейса

То же самое, что в первом принципе, только для интерфейсов. Разделять интерфейсы по логике, не создавать один интерфейс, делающий все, и не разделять один смысловой интерфейс на два. Иначе всем наследникам придется реализовывать все методы, указанные в интерфейсе.

5) Принцип инверсии зависимостей

Можно либо жестко связывать две сущности, либо создавать. Поля являются зависимостью для класса; Например, класс автомобиль зависит от объекта колесо. Зависеть — это не наследование. Можно добавить конструктор, в котором мы присваиваем автомобилю колесо. Не сам автомобиль должен создавать себе объекты зависимостей, а тот, кто снаружи создает объект автомобиля. Прямая зависимость - объект сам заботится о своих зависимостях, а инверсия зависимостей - тот, кто создает объект заботится о них. Если жестко привязываться к одному экземпляру, придется им ограничиваться; если использовать инвертированные зависимости, можно использовать много разных.

STUPID код, серьезно?

Это может задеть Ваше самолюбие, но вы уже, вероятно, написали много STUPID кода. Я тоже. Но, что это значит?

* Синглтон
* Сильная Связанность/Tight Coupling
* Невозможность тестирования
* Преждевременная оптимизация
* Не дескриптивное присвоение имени
* Дублирование кода

Ниже в тексте я остановлюсь на каждом пункте более подробно. Это, можно сказать, стенограммы моего рассказа в общих чертах.

Синглтон

Шаблон «одиночка» — вероятно, самый известный шаблон разработки, а также и самый недооцененный. Вы знаете о синдроме «одиночки»? Это — когда вы думаете, что шаблон «синглтон» — наиболее подходящий шаблон для текущего варианта использования, который у вас есть. Другими словами, вы используете его везде. Это определенно не круто.  
  
Одиночные элементы противоречивы, и их часто считают ошибочными шаблонами. Вы должны избегать их. Фактически, использование синглтона не проблема, а признак проблемы. Вот две причины почему:

* Программы, использующие глобальное состояние очень сложно протестировать;
* Программы, которые зависят от глобального состояния, скрывают свои зависимости.

Но должны ли вы действительно избегать их все время? Я сказал бы да, потому что часто можно заменить использование синглтона чем-то лучшим. Избежание всего статического очень важно, чтобы не допустить сильной связанности.

Сильная связанность

[Сильная связанность](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D1%86%D0%B5%D0%BF%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) — это ьшить связь между своими модулями. Связанность — это мера того, насколько связанобобщение проблемы синглтона. Одним словом, вы должны умены подпрограммы или модули.  
  
Если внесение изменения в одном модуле в вашем приложении требует, чтобы вы изменили другой модуль, тогда связанность существует. Например, вы инстанцируете объекты в классе своего конструктора вместо передачи экземпляров как параметров. Это плохо, потому что он не допускает дальнейших изменений, такие как замена экземпляра экземпляром подкласса, объектом-mock или что бы то ни было.  
  
Сильно связанные модули трудно повторно использовать, и также сложно тестировать.

Невозможность тестирования

По моему мнению, тестирование не должно быть трудным! Нет, правда. Каждый раз, когда вы не пишете unit тесты, потому что у вас нет времени, реальная проблема заключается в том, что ваш код не так уж эффективен, но это — другая история.  
  
В большинстве случаев невозможность тестирования вызвана сильной связанностью.

Преждевременная оптимизация

Дональд Эрвин Кнут сказал: «преждевременная оптимизация — корень всех зол. Только одни затраты, и никакой пользы». Фактически, оптимизированные системы гораздо сложнее, чем просто написание цикла или использование преинкремента вместо постинкремента. В итоге, вы останетесь с нечитабельным кодом. Именно поэтому *Преждевременную Оптимизацию* часто считают ошибочной.  
  
Мой друг часто говорит, что есть два правила для оптимизации приложения:

* Не делайте этого;
* (только для профессионалов!) пока не делайте этого.

Не дескриптивное присвоение имени

Это должно быть, очевидно, но все же нужно это сказать: назовите свои классы, методы, атрибуты и переменные должным образом. Ох, и не сокращайте их! И да, вы пишете код для людей, не для машин. Они не понимают то, что вы пишете, так или иначе. Компьютеры понимают только 0 и 1. Языки программирования предназначены для людей.

Дублирование кода

Дублированный код неэффективен, таким образом, не повторяйтесь, и также делайте его короче и проще. Пишите код только один раз!  
  
Теперь, когда я объяснил, что собой являет код STUPID, вы можете подумать, что ваш код является кодом STUPID. Это (еще) не имеет значения. Не расстраивайтесь, сохраняйте спокойствие и вместо этого напишите SOLID код!

2. Класс Object и его методы

Метод toString служит для получения представления данного объекта в виде строки. При попытке вывести строковое представления какого-нибудь объекта, как правило, будет выводиться полное имя класса. Например:

        Person tom = new Person("Tom");

        System.out.println(tom.toString()); // Будет выводить что-то наподобие Person@7960847b

метод **hashCode** позволяет задать некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты.

|  |  |  |  |
| --- | --- | --- | --- |
| 1  2 | public int hashCode() {  return J9VMInternals.fastIdentityHashCode(this);  }  Возможно, вас напугала строчка J9VMInternals.fastIdentityHashCode(this), но пока для  простоты считайте, что эта реализация зависит от JVM.  Метод getClass позволяет получить тип данного объекта:   |  |  | | --- | --- | | 1  2 | Person tom = new Person("Tom");  System.out.println(tom.getClass()); // class Person  Метод equals  Из примера выше видно, что == сравнивает не свойства объектов, не состояние, а ссылки на объекты. Такой способ сравнения подходит, если мы сравниваем уникальные объекты, существующие в одном экземпляре.  Но чаще всего это не то, что требуется, так как обычно при сравнении необходимо, чтобы сравнивались не ссылки, а состояния объектов. При этом, не всегда требуется, чтобы сравнивались все свойства объектов. Ведь логика сравнения может быть своя. Например, мы можем сказать, что две машины равны, если они одинаково стоят. Но при этом у машины есть еще другие свойства, такие как цвет, тип кузова и т.д.  public boolean equals(Object obj) {  return (this == obj);  }  Т.е по-умолчанию производится сравнение **по ссылке**. | |

3. Особенности реализации наследования в Java. Простое и множественное наследование.

Множественное наследование – способность создавать классы с множеством классов-родителей. В отличии от других популярных объектно-ориентированных языков, вроде С++, язык Java не поддерживает множественное наследование классов. Не поддерживает он его из-за вероятности столкнуться с «проблемой алмаза» и вместо этого предпочитает обеспечивать некий комплексный подход для его решения, используя лучшие варианты из тех, которыми мы можем достичь аналогичный результат наследования.

## «Проблема Алмаза»

Для более простого понимания проблемы алмаза допустим, что множественное наследование поддерживается в Java. В этом случае, мы можем получить классы с иерархией показанной на рисунке ниже. Предположим ,что SuperClass — это абстрактный класс, описывающий некоторый метод, а классы ClassA и ClassB — реальные классы. **SuperClass.java**

**package** com.journaldev.inheritance;

**public** **abstract** **class** SuperClass {

**public** **abstract** **void** doSomething();

}

**ClassA.java**

**package** com.journaldev.inheritance;

**public** **class** ClassA **extends** SuperClass{

@Override

**public** **void** doSomething(){

System.out.println("Какая-то реализация класса A");

}

//собственный метод класса ClassA

**public** **void** methodA(){

}

}

Теперь, давайте предположим, что класс ClassC наследуется от ClassA и ClassB одновременно, и при этом имеет следующую реализацию:

**package** com.journaldev.inheritance;

**public** **class** ClassC **extends** ClassA, ClassB{

**public** **void** test(){

//вызов метода родительского класса

doSomething();

}

}

Заметьте, что метод test() вызывает метод doSomething() родительского класса, что приведет к неопределенности, так как компилятор не знает о том, метод какого именно суперкласса должен быть вызван. Благодаря очертаниям диаграммы наследования классов в этой ситуации, напоминающим очертания граненого алмаза проблема получила название «проблема Алмаза». Это и есть основная причина, почему в Java нет поддержки множественного наследования классов.

Наследование - расширение функционала уже имеющихся классов за счет добавления нового функционала или изменения старого. При наследовании класс наследует все те же поля и методы, которые есть в родительском, кроме тех, которые имеют модификатор private.

4. Понятие абстрактного класса. Модификатор abstract.

# Абстракция

Выделение главных, наиболее значимых характеристик предмета, и наоборот — отбрасывание второстепенных, незначительных. Просто чертеж для будущих «нормальных» классов

Экземпляр (объект) абстрактного класса создать нельзя

Базовый класс, который не предполагает создания экземпляров

 Абстрактный класс похож на обычный класс. В абстрактном классе также можно определить поля и методы, но в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал.

При определении абстрактных классов используется ключевое слово **abstract**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public abstract class Human{        private String name;        public String getName() { return name; }  } |

Но главное отличие состоит в том, что мы не можем использовать конструктор абстрактного класса для создания его объекта. Например, следующим образом:

|  |  |
| --- | --- |
| 1 | Human h = new Human(); |

Кроме обычных методов абстрактный класс может содержать **абстрактные методы**. Такие методы определяются с помощью ключевого слова abstract и не имеют никакой реализации:

|  |  |
| --- | --- |
| 1 | public abstract void display(); |

Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе. Также следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как абстрактный.

1. Понятие интерфейса. Реализация интерфейсов в Java, методы по умолчанию. Отличия от абстрактных классов.

# Интерфейс

Абстрактный тип, который используется для указания поведения, которое должны реализовывать классы.

Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Если класс применяет интерфейс, то он должен реализовать все методы интерфейса. Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный, а его неабстрактные классы-наследники затем должны будут реализовать эти методы.

* Интерфейсы описывают только часть функциональности объекта — определённые признаки. Абстрактный класс же может описывать целую категорию разных объектов, а его характеристики имеют право наследовать только те объекты, которые являются частью этой категории. Например, собаки и волки — часть общей категории «Животные», а интерфейс, описывающий умение бегать, может реализовать и человек, и робот, и собака.
* Интерфейс описывает только поведение (методы), и у него нет полей. Точнее, есть возможность их объявить, но они будут public static final. В то же время абстрактный класс может содержать классические поля, которые будут принадлежать разным объектам.
* Наследник абстрактного класса обязан наследовать все его составляющие, а интерфейс создан только для реализации (имплементирования). Поэтому в Java мы можем наследовать класс только от одного класса, а на реализацию интерфейсов внутри одного класса ограничений нет.

Когда лучше использовать абстрактные классы:

* Вы хотите избежать дублирования кода, реализуя несколько тесно связанных классов из одной семантической категории.
* Классы, которые будут расширять ваш абстрактный класс, имеют много общих свойств и будут реализовывать много похожих методов.
* Наследуемый класс используется в отношении IS-A — то есть класс-наследник только расширяет функциональность абстрактного класса.
* Проект уже частично написан, а вы знаете, что выбранные классы будут часто меняться и дополняться новыми методами и полями. То есть поддерживать абстрактный класс и дополнять его намного проще, чем проектировать интерфейс и добавлять его новые методы во все места в коде, где они должны реализовываться.

Когда лучше использовать интерфейсы:

* Вам нужно описать определённую логику, которую должны поддерживать не связанные между собой объекты.
* Вам нужно привести к одному типу группу объектов и гарантировать схожую функциональность.
* Вам необходимо добавить какой-то маркер, который будет говорить о том, что выбранные классы поддерживают определённую логику.
* Вы хотите использовать множественное наследование.
* Методы по умолчанию похожи на статические методы тем, что для них вы также должны предоставлять тело. Чтобы объявить метод по умолчанию, просто используйте ключевое слово default.
* **public** **interface** **MyInterface** {
* **default** **int** **doSomething**() {
* **return** 0;
* }
* }
* В отличие от статических методов, методы по умолчанию наследуются классами, реализующими интерфейс. Что важно, такие классы могут при необходимости переопределять их поведение.
* Теперь один и тот же метод по умолчанию с одной и той же сигнатурой унаследован от двух разных интерфейсов. У каждого интерфейса своя реализация этого метода.  
    
  Итак, как наш класс узнает, какую из двух различных реализаций использовать?  
    
  Он и не узнает. Приведенный выше код приведет к ошибке компиляции

1. Перечисляемый тип данных (enum) в Java. Особенности реализации и использования.

Enum нужен для того, чтобы контролировать передаваемые аргументы.

Набор констант можно завершить “;” и дальше писать поля, конструкторы, методы.

Объект создается: SignalTraffic red = SignalTraffic.RED;

Можно после конструктора(Green “зеленый ”) открыть блок инициализации и определить метод. Он будет принадлежать только этому объекту. Тут же можно и переопределить метод.

Yellow(“желтый”){public void glow(){}},

Компилятор преобразовывает enum в следующий код: public class SignalTraffic extends Enum{

ST RED = new ST(“красный”);

1. Методы и поля с модификаторами static и final.

final — это **модификатор, позволяющий объявлять константные поля в классе**. Если у вас есть некоторое свойство проектируемого вами объекта, значение которого не будет меняться, то вы можете воспользоваться этим модификатором. Любая попытка переопределить значение поля с модификатором final приводит к выбросу исключения.

Думаю, слово final говорит само за себя. Применяя final модификатор Вы говорите, что поля не могут быть изменены, методы переопределены, а классы нельзя наследовать (о наследовании будет отдельная статья). Этот модификатор применяется только к классам, методам и переменным (также и к локальным переменным).

static модификатор перед методом или полем говорит о том, что они не принадлежат к экземпляру данного класса. Что это означает для нас? Когда мы описали поле класса или метод как static, его можно вызвать без использования экземпляра класса. То есть вместо такой конструкции: Cat cat = new Cat(); cat.method(), можно написать просто Cat.method(). При условии, что метод объявлен как static. Статические переменные едины для всех объектов класса. У них одна ссылка.

Еще одно важное замечание, которое нужно сказать по поводу static модификаторов: статические поля инициализируются во время загрузки класса.

1. Перегрузка и переопределение методов. Коварианты возвращаемых типов данных.

Перегрузка метода – название метода оставляем то же, а тип данных, с которым работает метод, меняет. У дочернего класса будут два метода move.

Коварианты – переопределяет в методе тип возвращаемого значения (нормализация вектора возвращает не двумерный, а 3-мерный вектор)

Методы вызываются одни и те же, а способы взаимодействия с ними разные.

# Переопределение методов

Производный класс может определять свои методы, а может переопределять методы, которые унаследованы от базового класса. Перед переопределяемым методом указывается аннотация @Override (проверяет переопределен ли метод, вызывает ошибку если метод не найден в родительском классе).

1. Элементы функционального программирования в синтаксисе Java. Функциональные интерфейсы, лямбда-выражения. Ссылки на методы.

[Функциональные интерфейсы и лямбда-выражения в Java / Skillbox Media](https://skillbox.ru/media/base/funktsionalnye_interfeysy_i_lyambda_vyrazheniya_v_java/)

Вложенные классы

Имеет смысл, если такой класс будет использоваться только внутри одного класса, в который он вкладывается. Такой класс может быть с любым модификатором доступа. Объект вложенного класса можно создать вне класса, в который он вложен. Для этого нужно создать экземпляр верхнего класса (Car car = new Car();) а потом экземпляр вложенного (Car.Wheel wheel = car.new Car.Wheel;). Как из вложенного класса вызвать приватный метод верхнеуровнего класса?

public void crash() {

Car.this.crash();

}

(Если просто писать this., то это будет ссылка на вложенный класс)

Внутренние классы

это такой же вложенный класс, отмеченный модификатором static. Этим классом мы начинаем пользоваться как любым другим статическим элементом данного класса. Для создания объекта этого класса не нужен объект внешнего класса. Из объекта вложенного класса нельзя обращаться к не статическим членам внешнего класса. Создание: Car.Wheel wheel = new Car.Wheel();

Локальные классы

Это классы, которые можно создать вообще везде. Похожи на локальные переменные. Правила:

Модификатор доступа не указывается

Невозможно объявление статических методов и иных статических членов, но

Возможно использование констант

Захват внешних локальных переменных возможен, если они определены, как effectively final (это переменные, которые никогда не меняются вне локального класса, константы по смыслу)

Не могут быть статичными

Анонимные классы

Создание объекта интерфейса на лету и передача его в качестве параметра методу. В нем мы переовпределяем run на лету.

public void test(){

start(new Runnable() {

public void run(){ }

});

}

Нет создания отдельного класса, неизвестен тип объекта Runnable(). Заставляет java сделать класс, реализующий объект. Тип неизвестен, поэтому класс анонимный.

Анонимный класс - синтаксический сахар. Это аналог локального класса. Разница только в наличии у класса имени для повторного использования.

public void test(){

class X implements Runnable{

public void run(){ }

};

start (new X() );

}

То же самое можем делать с любым типом данных, необязательно с интерфесом. Тогда будет ожидаться extends вместо implements. Внутри фигурных скобок можно делать что угодно.

Исключения

Это конструкция, позволяющая обрабатывать возникающие ошибки. Java предлагает порождать объекты определенных типов данных, которые будут содержать состояние и поведение возникшей ошибки.

Ключевое слово, предваряющее блок кода

try {

Блок кода, который необходимо выполнить

int x = ...;

int y = 5 / x;

Если в блоке возникает ошибка, то упарвление будет передано блоку catch, выполнение кода прервется. Таких блоков может быть несколько.

} catch (ArithmeticException e) {

// обработка

Блок finally выполняется в любом случае: была ошибка или нет. Он не является обязательным.

} finally {

//обработка

}

Object -> Throwable ->

-> Error (непроверяемое)

-> Excepton (проверяемое) ->

-> RunTimeException (непроверяемое)

-> Other Exception (проверяемое)

Можно создавать свои классы - исключения. Для этого нужно наследовать его от Exception. Чтобы выбрасывать такое исключение, используется ключевое слово throw. После такого как мы его выбрасываем, новое исключение можно обрабатывать в catch.