1)SOLID

**S: Single Responsibility Principle (Принцип единственной ответственности).**

Класс должен быть ответственен лишь за что-то одно. Он может иметь несколько методов, но они должны использоваться лишь для решения общей задачи, те служить одной цели. Если класс имеет несколько значений, то его нужно заменить на несколько отдельных классов

**O: Open-Closed Principle (Принцип открытости-закрытости).**

Программные сущности (классы, модули, функции) должны быть открыты для расширения, но не для модификации.

**L: Liskov Substitution Principle (Принцип подстановки Барбары Лисков).**

Наследники должны соблюдать логику родителей. Например, если есть две уточки, механическая и живая, наследующиеся от абстрактной уточки, умеющей плавать, то обе эти уточки должны уметь плавать, причём не обязательно одним и тем же способом.

**I: Interface Segregation Principle (Принцип разделения интерфейса).**

много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения(цель-минимизация побочных эффектов и повторов за счет разделения ПО на независимые части)

**D: Dependency Inversion Principle (Принцип инверсии зависимостей).**

Объектом зависимости должна быть абстракция, а не что-то конкретное.

а ) Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.

б) Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Применяя этот принцип, одни модули можно легко заменять другими и тогда никакие перемены на низкоуровневом модуле не повлияют на высокоуровневый

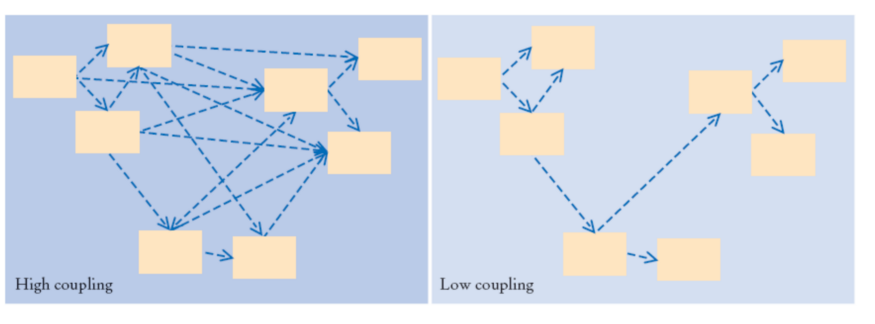
STUPID

**Синглтон**

Это объект с глобальной областью видимости создается, который ровно 1. создание такого объекта ограничено. Его дословный перевод – одиночка. Этот паттерн гарантирует, что у класса есть только один объект (один экземпляр класса) и к этому объекту предоставляется глобальная точка доступа. Проблемы: такие программы трудно тестировать, они скрывают свои зависимости

**Сильная Связанность/Tight Coupling**

Сильная зависимость классов между собой, большое количество связей. Если внесение изменения в одном модуле в вашем приложении требует, чтобы вы изменили другой модуль, тогда связанность существует. Сильно связанные модули трудно повторно использовать, и также сложно тестировать. 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

****

**Невозможность тестирования**

В большинстве случаев невозможность тестирования вызвана сильной связанностью.

**Преждевременная оптимизация**

Фактически, оптимизированные системы гораздо сложнее, чем просто написание цикла или использование преинкремента вместо постинкремента. В конечном итоге, вы останетесь с нечитабельным кодом. Именно поэтому *Преждевременную Оптимизацию* часто считают ошибочной.

**Не дескриптивное присвоение имени** (непонятные нечитаемые имена, сокращения)

**Дублирование кода** (например просто код без функций.) **Дублированный** **код** - это термин компьютерного программирования, обозначающий последовательность исходного **кода**, которая встречается более одного раза либо в программе, либо в разных программах, принадлежащих или поддерживаемых одним и тем же объектом.

Причины:

* Умышленное копирование фрагментов программы
* Многократное использование сложного API
* Повторная реализация существующей функциональности
* Слабая выразительность используемого языка
* Недостаток прав для модификации исходного кода

**2)Objec**t – класс, от которого наследуются все остальные классы, поэтому все типы и классы могут реализовывать те методы, которые есть в этом классе. Основные:

toString()

Метод нужен для получения представления данного объекта как строки. По дефолту для объекта будет выводиться полное название класса и его хэш-код (я так понимаю, это он) через @. Его часто переопределяют по этой причине. Возвращает очевидно стринг.

hashCode()

Хэшкод – это целое число, которое является уникальным идентификатором  содержимого объекта.Возвращает интовое значение, являющееся хэш-кодом объекта.

getClass()

Позволяет получить тип данного объекта. Выводит что-то типа class Human

equals()

Тупо сравнивает два объекта на равенство. По умолчанию сравнивает содержимое объектов, выводит булеан.

Clone()

Копирует объекты, именно создаёт копию, то есть при клонировании создаётся один новый объект и его полям присваиваются значения полей объекта-образца.

Есть ещё всякие finalize(), wait(), notify(), notifyAll()…..

**4)abstract class**

класс от которого нельзя создать объект.

может содержать abstract методы, типа ф-ции без реализации. каждый наследующийся класс должен их реальзовать

5)**interface**

Проблему наследования от одного класса в жабе частично решают интерфейсы. Интерфейс – по факту шаблон для создания класса, они определяют некоторый функционал, не имеющий конкретной реализации, а затем реализуются классы с применением этого функционала. Один класс может применить несколько интерфейсов.

Интерфейсы не имеют модификаторов доступа, по дефолту паблик, так как они по задумке должны определять функционал для реализации его классом. Если класс применяет интерфейс, то он должен использовать все его методы. Если класс не делает этого, то он должен быть абстрактным, а его неабстрактные потомки должны будут реализовать потом его методы.

Нельзя напрямую создать объект интерфейса.

можно наследоваться ТОЛЬКО ОТ 1 КЛАССА НО ОТ НЕСКОЛЬКИХ ИНТЕРФЕЙСОВ

**реализация по умолчанию** - Метод, реализованный в интерфейсе, называется методом по умолчанию и обозначается ключевым словом default. Если класс реализует интерфейс, он может, но не обязан, реализовать методы, реализованные в интерфейсе. Класс наследует реализацию по умолчанию

Функциональный Интерфейс - интерфейс с 1м абс. методом

К функциональному интерфейсу также можно добавить аннотацию @FunctionalInterface, чтобы компилятор дальше сам проверял, что у нас всего один абстрактный метод.

6)**ENUM**

**логически - набор констант. правильный singletone**

это класс, создание экземпляров которого ограничено. все экземпляры прописываются через запятую

7) Применяя final модификатор Вы говорите, что поля не могут быть изменены, методы переопределены

static для поля/метода означает что они принадлежат классу а не объекту. для доступа не надо создавать объект

static для класса только для вложенного класса.

означает что создать объект этого класса можно без создания объекта род. класса

public class A{

static public class B{ }

public class C{ }

}

…

A a = new A(); //обычный класс

A.B b = new A.B()// статический класс

A.C c = a.new C()// вложенный класс, связан с экземпляром А

A.C c = new A.C() //ошибка

8)

Сигнатура метода — это имя метода , и порядок параметров. ТИП ВОЗВРАЩАЕМОГО ЗНАЧЕНИЯ НЕ ВХОДИТ В СИГНАТУРУ

перегрузка это возможность определить метод с тем же именем но другим набором параметров

void min(Array a);

void min(List b);

void min(int a, in b);

Переопределение используется тогда, когда вы переписываете (переделываете, переопределяете) УЖЕ существующий метод.

@override говорит компилятору что вы хотите именно ПЕРЕОПРЕДЕЛИТЬ а не перегрузить метод. должна быть такая же сигнатура. чтобы вы не могли случайно написать тот же метод с другой сигнатурой

**ковариант возвращаемого типа**

Возвращение коварианта означает, что при переопределении метода возвращаемый тип переопределяющего метода наследуется от возвращаемого типа переопределенного метода.

кароче возвращаемый тип переопределяющего метода наследуется от возвращаемого типа переопределенного метода.

**class One{**

**One get(){return this;}**

**}**

**class Two extends One{**

**Two get(){return this;}**

**void message(){**

**System.out.println("After Java5”)**

**}**

**public static void main(String args[]){**

**new Two().get().message();**

**}**

**}**

9) **ПРОСТО ЖЕСТЬ (Элементы функционального программирования в синтаксисе Java. Функциональные интерфейсы, лямбда-выражения. Ссылки на методы.)**

**функциональный интерфейс**- содержит только 1 абстрактный метод. К функциональному интерфейсу также можно добавить аннотацию @FunctionalInterface, чтобы компилятор дальше сам проверял, что у нас всего один абстрактный метод.

анонимный класс. Они позволяют вам одновременно объявлять и создавать экземпляры класса. Они похожи на местные классы, за исключением того, что у них нет названия. Используйте их, если вам нужно использовать локальный класс только один раз.

Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в **функциональном интерфейсе**

**Ссылка на метод - это сокращенный синтаксис выражения лямбда, который выполняет только один метод. Это позволяет нам ссылаться на конструкторы или методы, не выполняя их. работают только при совпадении сигнатуры**

**ТРЕНДЕЦ**