- **Что такое SOLID принципы**
  - *S – Single Responsibility principle*
    - Принцип единой ответственности
    - Через каждый класс проходит только одна ось изменений => он отвечает только за одну задачу
    - Если нам нужен класс для нескольких задач, то правильнее создать отдельные классы для более мелких задач, а после применяя наследование создать комбинированный класс
    - **Пример**
      - Есть датчик, измеряющий температуру и влажность => Если создадим единый класс для датчика (TempHygro) то если кому-то понадобится что-то отдельное, то ему придётся либо подключать то что ему не надо либо копипастить в свой личные класс что ещё хуже
      - Правильно – создать два класса Hydro и Temp, и если необходимо, то унаследовать от них класс HydroTemp
  - *O – Open-Closed principle*
    - Принцип открытости и закрытости
    - Любой блок кода открыт к добавлению дополнительного функционала НО закрыт для изменения существующей
    - **Польза**
      - Если какие-то блоки уже протестированы => добавляя новые и не меняя старые мы можем не искать баги в старом коде
      - Меньше денег на тестирование!!!!!!
    - Способы реализации принципа
      - Способ Бертрана Мейера
        - Пишем блок кода
        - Блокируем его для изменения(кроме багофикса)
        - Если хотим что-то добавить => наследование + новый код может иметь другой интерфейс

- Полиморфные принцип – Роберта Мартина
    - Интерфейс должен быть неизменным => новые реализации работают с одним и тем же интерфейсом, возможно частично делегируя задачи старой реализации
    - Благодаря статичному интерфейсу можно не менять пользовательский код

- *L - LCP*
    - Принципы Барбары Лисков
    - Если есть кусок кода, в который приходит базовые класс => туда должны спокойно приходить, и все наследники и ничего не должно падать, ломаться, взрываться и т. д.
    - Поведение наследников не должно противоречить базовым классам
    - Если мы наследуемся от класса и в классе наследнике блокаем некоторые методы родителя (которые нам типо ненужны…) – грубейшее нарушение LCP => трындец полный
    - Child не должен требовать от клиентского кода больше чем Parent и не должен давать ему меньший функционал чем Parent => клиентский код не должен парится о том с кем он сейчас работает – с Parent или с Child
- *I - ICP*
    - Принцип разделения интерфейса
    - Много интерфейсов для клиентов лучше, чем один общий интерфейс
    - Клиенты не должны завесить от методов, которые они не используют
    - **Пример**
        - Взять и вынести все методы из класса в один большое интерфейс – ГОВНО

                - Любое изменение методов в классе, даже тех, которые не используют некоторые клиенты, вызовет полное перекомпилирование всего
                - НАДО каждому клиенты свой интерфейс – тогда всё будет норм
            - Избыточные интерфейсы требуют от пользовательского кода определения кучи методов, которые ему нафиг не нужны
    - *D – DIP*
        - Принцип инверсии зависимостей
        - Нужно использовать все классы через интерфейсы
        - Абстракции не должны зависеть от деталей
        - Детали должны зависеть от абстракций
        - Добавление чего-то нужно делать через интерфейсы => интерфейсы от интерфейсов и т.д.
        - Очень упрощает расширение
- **Что такое STUPID принципы**
    - *Синглтон* – использование чего-то статического и глобального
    - *Сильная связность* – любое изменение в коде порождает каскад изменений в других участках кода
    - *Невозможность тестирования* – ну просто сложно тестировать и всё
    - *Преждевременная оптимизация* – плохо
    - *Не дескриптивное присвоений имени* – переменные a,b,c…
    - *Дублирование кода* – ПЛОХО
- **Класс Object. Реализация методов по умолчанию**
    - Любой класс в Java наследуется от класса Object
    - Стандартный методы
        - toString – строковая интерпретация объекта класса
        - hashCode – уникальный номер соответствующий объекту класса
        - equals – позволяет сравнивать два объекта класса на идентичность

- **Особенности наследования в Java. Простое и множественное наследование**
    - В Java нет множественного наследования классов, но можно наследоваться от множества интерфейсов
    - Позволяют расширять функционал существующих классов или изменять его в новых классах
    - Модификатор final запрещает наследование класса
- **Абстрактный модификаторы**
    - Абстрактный классы позволяют определять поля и методы(возможно с реализацией) => это типо чертежа, по которому можно создать конечный класс
    - Нельзя создавать объекты абстрактного класса
    - Методы могут быть абстрактными => тогда их реализацию надо будет прописывать уже в классе наследнике (как в интерфейсах)
    - *Правила и ограничения*
        - Абстрактный метод не может быть private или final
        - Наследник обязан реализовать все абстрактные методы
- **Понятие интерфейсов**
    - Позволяют прописать обязательные методы для объектов с похожим взаимодействием => описывают то, как пользовательская прога может взаимодействовать с классами, не описывая реализацию
    - Позволяет нормально сделать полиморфизм
- **Перечисляемый тип Enum**
    - Позволяет заранее определить набор объектов, которые могут быть созданы на основе данного класса
    - Вообще Enum это класс => в enum можно только implement
    - Мы можем как определить общие для всех объектов методы, так и перегрузить отдельный методы для конкретных объектов, или переопределить их, или вообще добавить новые…
    - Полезные методы
        - toString()

        - name()
        - valueOf(name) – получение объекта по строковой интерпретации имени
        - values() – получение всех объектов перечисления
- **Модификаторы final и static**
    - Final – задает константу => нельзя изменять поле/метод/класс
    - Static – одинаково для всех объектов класса => относится не к объекту а к классу => к нему неправильно обращаться от имени объекта
    - Меняя static переменную от имени объекта, мы меняем её для всех объектов этого класса
    - В static методах можно использовать только static поля и методы класса => другие нельзя т. к. непонятно какой объект использовать
    - В Нестатических методах можно использовать всё статическое НО не наоборот
    - В static логическом блоке те же ограничения
    - Статический блок нужен для инициализации static полей
    - Он срабатывает только один раз при создании объекта или обращении к статическим полям/методам класса
- **Перегрузка и переопределение методов. Ковариванты возвращаемых типов данных.**
    - *Переопределение (override)* – изменение реализации, НО не структуры(возвращаемые и принимаемые значения)
    - *Перегрузка (overload)* – изменение всего кроме имени функции
    - *Ковариванты возвращаемых типов данных* => при переопределении метода мы можем возвращать объекты которые являются наследниками к объекту который возвращается методом родителем
    - *Пример*
        - Есть метод product => возвращает тип данных Product (класс тут типо фабрика)

        - Мы можем для молочной фабрики возвращать Milk а для конфетной Sweet при условии что классы Milk и Sweet наследуются от Product
    - для лучшего понимания
- **Функциональные интерфейсы, лямбда-выражения. Ссылки на методы.**
    - Тут про ФИ и лямбды
    - Тут про ссылки на методы