3️⃣

Лаба 3

2. Методы класса object:

   Принципы solid:

   1. S(olid) Single responsibility principle. Принцип единственной ответственности.
      суть:
      1 сущность = 1 задача
   2. (s)O(lid)
      Open-closed principle
      принцип открытости и закрытости.
      суть:
      Сущности должны быть открыты для расширения и закрыты для модификации
   3. (so)L(id)
      Liskov substitution principle
      Принцип постановки Барбары Лисков
      Суть:
      сущности, которые наследуют родительский тип, должны точно также работать и с дочерними классами, при это программа не должна ломаться. (другими словами: наследуемый класс должен дополнять а не заменять базовый класс)
   4. (sol)I(d)
      Interface segregation principle
      Принцип разделения интерфейса
      суть:
      сущности не должны зависеть от методов, которые они не используют
   5. (soli)D
      Dependency inversion principle

Принцип инверсии зависимости

суть:

Модули высокого уровня не должны зависеть от модулей более низкого уровня, все они должны зависеть от абстракций. а абстракции не должны зависеть от деталей. в свою очередь детали должны зависеть от абстракций

Принципы STUPID

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

Класс object - базовый класс для всех классов. все классы наследуют его методы

методы:

- **hashCode()**
  - Возвращает целочисленное значение указывающее на место в памяти с помощью алгоритма хэша
  - если два объекта ссылаются на один класс произойдет **коллизия:**
    - hashCode() возвращает одинаковое значение на два “разных” объекта (в реале они ссылаются на один)
    - чтобы не возникала коллизия можно переопределить метод ниже
- **clone()**
  - по дефолту является protected, нужен для создания нового объекта для избегания **коллизии** и записи в него данных из оригинального чтобы создать копию, с другим хэшкодом
- **toString()**
  - дефолтное определение

```
public boolean toString(Object y) {
    return getClass() + "@" + hashCode();
}
```

  - Этот метод позволяет получить текстовое описание любого объекта

- **getClass()**
  - возвращает класс объекта
- **equals()**
  - == сравнивает значения хэша (два объекта указывают на одну память, одинаковый хэш)
  - дефолтное определение
  - нужно переопределять для сравнения объектов с разным хэшом

```
public boolean equals(Object y) {
    return this == y;
}
```

- методы используются для многопоточного программирования
  - `public final native void notify()`
  - `public final native void notifyAll()`
  - `public final native void wait( long timeout)`
  - `public final void wait( long timeout, intnanos)`
  - `public final void wait()`

> 💡 Ключевое слово `native` обозначает что этот метод иплементирован в C или C++ в Java Native Interface (JNI)

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

множественного наследования в джаве нет!!!! нельзя!! но вместо этого есть интерфейсы!
(проблема ромбовидного наследования)

3. Понятие абстрактного класса. Модификатор `abstract`.
   Используется только для классов и методов.
   Если в классе есть абстрактный метод, то класс должен быть абстрактным
   - Невозможно создать экземпляр абстрактного класса
   - Абстрактный метод должен быть определен классом наследником или снова помечен абстрактным

   -может содержать метод main, может содержать статические методы, может имплементироваться и быть наследованным от одного класса
   Конфликтующие ключевые слова:

4. *Final u Abstract* — классы и методы не могут быть одновременно абстрактными (что в большинстве случаев подразумевает необходимость их уточнения для реализации) и финальными, т.е. неизменяемыми. Получается, что в инструкции написано, как создать хороший прочный шлем из любого материала (абстрактная часть), но для этого в нем обязательно не должно быть отверстий (финальная обязательная часть, изменению не подлежит).

5. *Abstract u Static* — абстрактный метод не может одновременно быть статическим Статический абстрактный метод не имеет смысла, ведь он мало того, что ничего не делает, так еще и принадлежит целому классу — бесполезная штука получается.

6. Понятие интерфейса. Реализация интерфейсов в Java, методы по умолчанию. Отличия от абстрактных классов.
   - В интерфейсе только public методы
   - все поля могут быть только
   `public static final`
   - методы должны объявляться без тела
   -если не помечены
   `default`
   - можно наследовать интерфейс интерфейсом используя ключ. слово `extends` (работает как в классах), во всех остальных случаях `implements`
   - может быть множественное наследовательность используя интерфейсы
   - могут вложенные классы и интерфейсы (static по дефолту)

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

```
public enum Auth {
    ADMIN("Write"); //Константе ADMIN мы задаем значение "Wri
    public final String permission; //все поля являются final
    private Auth(String permission) {
        this.permission = permission;
    }
}
```

можно создать конструктор, который будет вызываться для каждой константы

```
Auth.ADMIN.permission == "Write" //true
```

с помощью метода ordinal() можно получить индекс как в массиве

Функциональное программирование - это программирование, в котором функции являются объектами, и их можно присваивать переменным, передавать в качестве аргументов другим функциям, возвращать в качестве результата от функций и т. п.

> Lambda-выражения - это метод без объявления, т.е. без модификаторов доступа, возвращающие значение и имя. нахуя нужны? они вносят в язык функциональность на равне с объектами.Короче говоря, они позволяют написать метод и сразу же использовать его. Особенно полезно в случае однократного вызова метода, т.к. сокращает время на объявление и написание метода без необходимости создавать класс.
>
> функциональные интерфейсы (Functional Interface) – это интерфейсы только с одним абстрактным методом, объявленным в нем.
>
> Ссылка на метод - это сокращенный синтаксис для лямбда-выражения, которое содержит только один вызов метода.

```
interface Callable<T> { //функциональный интерфейс
    void call();
```

```
    default void test() {

    }
}
```

Лямбда выражения:

```
void Test (Callable callable) {
    callable.call();
}

Test(() -> { //параметры лямбды
    //код лямбды

});

Test(new Callable() {
    @Override
    void c
});
Test(Class::nehod);
```

По факту, лямбда выражения является

Типы ссылок на методы

Существуют следующие ссылки на методы четырех типов:

1. Ссылка на статический метод.
2. Ссылка на метод экземпляра конкретного объекта.
3. Ссылка на метод экземпляра произвольного объекта определенного типа.
4. Ссылка на конструктор.

Дефолтные методы интерфейса