Table of Contents

Чем абстрактный класс отличается от интерфейса? В каких случаях следует использовать абстрактный класс, а в каких интерфейс?

+ В Java класс может одновременно реализовать несколько интерфейсов, но наследоваться только от одного класса. + Абстрактные классы используются только тогда, когда присутствует тип отношений «is a» (является).Интерфейсы могут реализоваться классами, которые не связаны друг с другом. + Абстрактный класс - средство, позволяющее избежать написания повторяющегося кода, инструмент для частичной реализации поведения. Интерфейс - это средство выражения семантики класса, контракт, описывающий возможности.Все методы интерфейса неявно объявляются как `public abstract` или (начиная с Java 8) `default` - методами с реализацией по-умолчанию, а поля - `public static final`. + Интерфейсы позволяют создавать структуры типов без иерархии. + Наследуясь от абстрактного, класс «растворяет» собственную индивидуальность. Реализуя интерфейс, он расширяет собственную функциональность.

Абстрактные классы содержат частичную реализацию, которая дополняется или расширяется в подклассах. При этом все подклассы схожи между собой в части реализации, унаследованной от абстрактного класса и отличаются лишь в части собственной реализации абстрактных методов родителя. Поэтому абстрактные классы применяются в случае построения иерархии однотипных, очень похожих друг на друга классов. В этом случае наследование от абстрактного класса, реализующего поведение объекта по умолчанию может быть полезно, так как позволяет избежать написания повторяющегося кода.Во всех остальных случаях лучше использовать интерфейсы.

Что имеет более высокий уровень абстракции: класс, абстрактный класс или интерфейс?

Интерфейс.

Immutable class

Неизменяемый класс - это класс, состояние которого нельзя изменить после создания. Пример: `String` - лучший пример неизменяемого класса. Создав строку, вы не сможете ее изменить.

Неизменяемый класс очень прост для понимания, он имеет только одно состояние. Неизменяемые классы являются потокобезопасными. Это самое большое преимущество неизменяемого класса, потому что, - вам не нужно применять синхронизацию для неизменяемых объектов. Также, неизменяемый класс может быть полезен при помещении объекта неизменяемого класса в HashMap или может использоваться для целей кэширования, поскольку его значение не изменится.Неизменяемые объекты по умолчанию являются потокобезопасными.

Создания неизменяемого класса: - *Финализируйте свой класс:* Если вы финализируете свой класс - ни один класс не сможет его расширить, следовательно, не сможет переопределить методы этого класса. - *Пометьте все переменные класса модификаторами доступа private и final:* Если вы сделаете переменную экземпляра `private` - ни один внешний класс не сможет получить доступ к переменным экземпляра, и, если вы сделаете их `final` - вы не сможете их изменить. - *Скажите «нет» методам-мутаторам:* Не создавайте метод set для некоторых переменных класса, тогда не будет возможности явно изменить состояние переменных экземпляра. - *Выполните клонирование изменяемых объектов при возврате из метода получения:* Если вы вернете клон объекта из метода get, то вернется объект. При этом, ваш оригинальный объект останется без изменений.

```
class ImmutableClass {
   private String name;
   private String value;

   public ImmutableClass(String name, String value) {
      this.name = name;
      this.value = value;
      System.out.println("New ImmutableClass created! Name " + this.name + " value " +
this.value);
   }

   public ImmutableClass setName(String name) {
      return new ImmutableClass(name, this.value);
   }

   public ImmutableClass setValue(String value) {
      return new ImmutableClass(this.name, value);
   }

   public String getName() {
      return name;
   }

   public String getValue() {
```

```
        return value;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        ImmutableClass that = (ImmutableClass) o;
        return Objects.equals(name, that.name) &&
            Objects.equals(value, that.value);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, value);
    }

    @Override
    public String toString() {
        return "ImmutableCat{" +
            "breed='" + name + '\'' +
            ", eyeColor='" + value + '\'' +
            '}';
    }
}
```