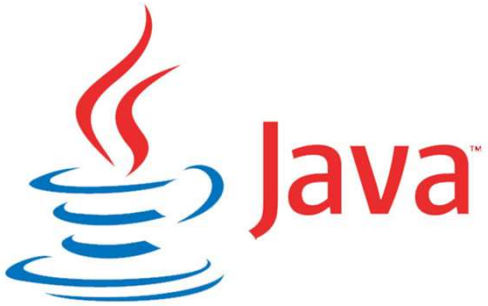
Тема 7.
Интерфейсы и абстрактные классы



Парадигмы ООП

Парадигмы ООП

Наследование

Наследование – механизм, позволяющий описать класс на основе уже существующего.

Родитель – класс, от которого происходит наследование.

Потомок/наследник – класс, который произошел (был унаследован) от какого-то базового класса (класса-родителя).

Между родителем и потомком устанавливается отношение ***«является»***.

Парадигмы ООП

Наследование

Рассмотрим автомобиль и нашего жука***.***

Все мы знаем, что, вообще говоря, ***наш жук – это автомобиль***.
Как и VW Passat, как и BMW X6 и все остальные.

Жук ЯВЛЯЕТСЯ автомобилем.

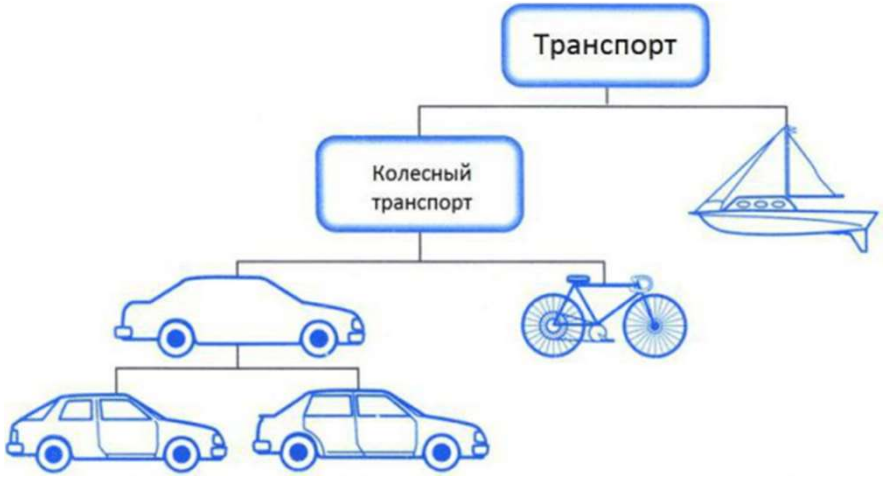
BMW X5 ЯВЛЯЕТСЯ автомобилем.
Ferrari F40 ЯВЛЯЕТСЯ автомобилем.

«Класс-потомок» ЯВЛЯЕТСЯ «Класс-родитель»

Парадигмы ООП

А в чем смысл?

- ***Класс-потомок может пользоваться данными и методами класса-родителя!***
- ***Класс-потомок может добавлять новые данные и методы!***
- ***Класс потомок может переопределять методы класса-родителя!***
- **А ЗНАЧИТ мы можем строить иерархии классов!**
- **Следовательно, нам нужно писать меньше кода!**



Парадигмы ООП

Наглядный пример

Давайте опишем класс Person, представляющий отдельного человека:

```java
public class Person {

    private String name;
    private String surname;

    public Person(String name, String surname){

        this.name=name;
        this.surname=surname;
    }

    public String getName() {
        return name;
    }

    public String getSurname() {
        return surname;
    }

    public void displayInfo(){

        System.out.println("Имя: " + name + " Фамилия: " + surname);
    }
}
```

Парадигмы ООП

Наглядный пример

- В последствии, мы хотели бы расширить имеющуюся систему классов, добавив в нее класс, описывающий сотрудника предприятия – класс Employee;
- Сотрудник предприятия является человеком и, следовательно, имеет тот же функционал, что и класс Person;
- Поэтому, ничто не мешает нам сделать класс Employee производным от класса Person:

```
1 class Employee extends Person{
2
3 }
```

- Чтобы объявить один класс наследником другого, нужно использовать после имени класса-наследника ключевое слово extends, после которого указывается имя базового класса.

Парадигмы ООП

Наглядный пример

- В классе Employee могут быть определены свои поля, методы и конструктор:

```
1  class Employee extends Person{
2
3      private String company;
4
5      public Employee(String name, String surname, String company) {
6
7          super(name, surname);
8          this.company=company;
9      }
10 }
```

Парадигмы ООП

Запрет наследования

- Хотя наследование очень интересный и эффективный механизм, но в некоторых ситуациях его применение может быть нежелательным.
- В этом случае можно запретить наследование с помощью ключевого слова final:

```
1 public final class Person {
2
3 }
```

- Если бы класс Person был бы определен таким образом, то следующий код был бы ошибочным и не сработал, так как наследование было запрещено:

```
1 class Employee extends Person{ {
2
3 }
```

Парадигмы ООП

Запрет наследования

- Кроме запрета наследования можно также запретить переопределение отдельных методов.
- Запретим переопределение метода displayInfo():

```
1  public class Person {
2
3      //.....................
4
5      public final void displayInfo(){
6
7          System.out.println("Имя: " + name + " Фамилия: " + surname);
8      }
9  }
```

Модификаторы доступа

Все члены класса в языке Java - поля и методы, свойства - имеют модификаторы доступа. В прошлых темах мы уже сталкивались с модификатором **public**.

Модификаторы доступа позволяют задать допустимую область видимости для членов класса, то есть контекст, в котором можно употреблять данную переменную или метод.

Модификаторы доступа

В Java используются следующие модификаторы доступа:

- **public**: публичный, общедоступный класс или член класса. Поля и методы, объявленные с модификатором public, видны другим классам из текущего пакета и из внешних пакетов.

- **private**: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе.

- **protected**: такой класс или член класса доступен из любого места в текущем классе или пакете или в производных классах (классах наследниках), даже если они находятся в других пакетах

- **Модификатор по умолчанию**. Отсутствие модификатора у поля или метода класса предполагает применение к нему модификатора по умолчанию. Такие поля или методы видны всем классам в текущем пакете.

```java
public class Program{

    public static void main(String[] args) {

        Person kate = new Person( name: "Kate",  age: 32,  address: "Baker Street",  phone: "+12334567");
        kate.displayName();     // норм, метод public
        kate.displayAge();      // норм, метод имеет модификатор по умолчанию
        kate.displayPhone();    // норм, метод protected
        //kate.displayAddress();  // ! Ошибка, метод private

        System.out.println(kate.name);      // норм, модификатор по умолчанию
        System.out.println(kate.address);   // норм, модификатор public
        System.out.println(kate.age);       // норм, модификатор protected
        //System.out.println(kate.phone);   // ! Ошибка, модификатор private
    }
}
```

```java
class Person{

    String name;
    protected int age;
    public String address;
    private String phone;

    public Person(String name, int age, String address, String phone){
        this.name = name;
        this.age = age;
        this.address = address;
        this.phone = phone;
    }
    public void displayName(){
        System.out.printf("Name: %s \n", name);
    }
    void displayAge(){
        System.out.printf("Age: %d \n", age);
    }
    private void displayAddress(){
        System.out.printf("Address: %s \n", address);
    }
    protected void displayPhone(){
        System.out.printf("Phone: %s \n", phone);
    }
}
```

В данном случае оба класса расположены в одном пакете, поэтому в классе Program мы можем использовать все методы и переменные класса Person, которые имеют модификатор по умолчанию, **public** и **protected**.

А поля и методы с модификатором **private** в классе Program не будут доступны.

Если бы класс Program располагался бы в другом пакете, то ему были бы доступны только поля и методы с модификатором **public**.

Абстрактные классы

- Кроме обычных классов в Java есть абстрактные классы.
- Абстрактный класс похож на обычный класс, в нем 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, в свою очередь, реализуют этот функционал.
- При определении абстрактных классов используется ключевое слово abstract:

```
public abstract class Human{

    private String name;

    public String getName() { return name; }
}
```

- Но главное отличие абстрактного класса состоит в невозможности использовать конструктор абстрактного класса для создания его объекта. Например, следующим образом:

  Human h = new Human();

Абстрактные классы

- Кроме обычных методов абстрактный класс может содержать **абстрактные методы**.
- Такие методы определяются с помощью ключевого слова **abstract** и не имеют никакого функционала: **public abstract void display()**;
- Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе.
- Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе.
- Следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как **абстрактный**.

Абстрактные классы

Зачем нужны абстрактные классы

Предположим, что перед разработчиком стоит задача написать программу для обслуживания банковских операций.

В программе будут определены 3 класса:

- Person, который описывает человека;
- Employee, который описывает банковского служащего;
- Client, который представляет клиента банка.

Очевидно, что классы Employee и Client будут производными от класса Person, так как оба класса имеют некоторые общие поля и методы.

И так как все объекты будут представлять либо сотрудника, либо клиента банка, то напрямую от класса Person создаваться объекты не будут, поэтому имеет смысл сделать его абстрактным.

Абстрактные классы

Зачем нужны абстрактные классы

```java
public abstract class Person {

    private String name;
    private String surname;

    public String getName() { return name; }
    public String getSurname() { return surname; }

    public Person(String name, String surname){

        this.name=name;
        this.surname=surname;
    }

    public abstract void displayInfo();
}
```

```java
class Employee extends Person{

    private String bank;

    public Employee(String name, String surname, String company) {

        super(name, surname);
        this.bank=company;
    }

    public void displayInfo(){

        System.out.println("Имя: " + super.getName() + " Фамилия: "
                + super.getSurname() + " Работает в банке: " + bank);
    }
}
```

```java
class Client extends Person
{
    private String bank;

    public Client(String name, String surname, String company) {

        super(name, surname);
        this.bank=company;
    }

    public void displayInfo(){

        System.out.println("Имя: " + super.getName() + " Фамилия: "
                + super.getSurname() + " Клиент банка: " + bank);
    }
}
```

Интерфейсы

- Механизм наследования очень удобен, но он имеет свои ограничения – наследование возможно только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование.
- В языке Java подобную проблему частично позволяют решить интерфейсы.
- Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы.
- Один класс может применить ***множество*** интерфейсов.
- Чтобы определить интерфейс, используется ключевое слово interface:

```
interface Printable{

    void print();
}
```

Интерфейсы

- В IDE интерфейс создается так же, как и обычный класс.
- В Intelij IDEA это можно сделать следующим образом: **File -> New -> Java Class** и в открывшемся окне выбирать тип **Interface;**
- Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации.
- Методы без реализации похожи на абстрактные методы абстрактных классов.
- В примере выше, в интерфейсе Printable объявлен один метод, который не имеет реализации.
- Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ у них public, так как цель интерфейса – определение функционала для реализации его классом, и весь функционал должен быть открыт для реализации.
- При объявлении интерфейса надо учитывать, что только один интерфейс в файле может иметь тип доступа public, а его название должно совпадать с именем файла.
- Остальные интерфейсы (если такие имеются в файле java) НЕ должны иметь модификаторов доступа.

Интерфейсы

- Чтобы класс применил интерфейс, надо использовать ключевое слово **implements**.

```
class Book implements Printable{
    String name;
    String author;
    int year;

    Book(String name, String author, int year){
        this.name = name;
        this.author = author;
        this.year = year;
    }

    @Override
    public void print() {
        System.out.printf("Книга '%s' (автор %s) была издана в %d году \n", name, author, year);
    }
}
```

- При этом надо учитывать - если класс применяет интерфейс, то он должен реализовать все методы интерфейса, как в случае выше реализован метод print().

Интерфейсы

- Потом, в главном классе мы можем использовать данный класс и его метод print():

```
1  Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863);
2  b1.print();
```

- В то же время нельзя напрямую создавать объекты интерфейсов, поэтому следующий код не будет работать:

```
1  Printable pr = new Printable();
2  pr.print();
```

Интерфейсы

Для чего нужны интерфейсы

- Простой пример интерфейса из повседневной жизни — пульт от телевизора.
- Он связывает два объекта, *человека* и *телевизор*, и выполняет разные задачи: *прибавить или убавить звук*, *переключить каналы*, *включить или выключить телевизор*.
- Одной стороне (*человеку*) нужно обратиться к интерфейсу (*нажать на кнопку пульта*), чтобы вторая сторона выполнила действие, например, чтобы телевизор переключил канал на следующий.
- При этом пользователю не обязательно знать устройство телевизора и то, **как** внутри него реализован процесс смены канала.
- Все, к чему пользователь имеет доступ — это ***интерфейс***.
- Главная задача — получить нужный результат.

Интерфейсы

Для чего нужны интерфейсы

- Интерфейс описывает поведение, которым должны обладать классы, реализующие этот интерфейс.
- «Поведение» — это совокупность методов.

Предположим, что мы хотим создать несколько мессенджеров.

Что должен уметь любой мессенджер?

В упрощенном виде, принимать и отправлять сообщения.

Давайте создадим интерфейс Messenger:

```
public interface Messenger{

    public void sendMessage();

    public void getMessage();
}
```

Интерфейсы

Для чего нужны интерфейсы

- И теперь мы можем просто создавать наши классы-мессенджеры, имплементируя этот интерфейс.
- Компилятор сам «заставит» нас реализовать их внутри классов.

```
public class Telegram implements Messenger {

    public void sendMessage() {

        System.out.println("Отправляем сообщение в Telegram!");
    }

     public void getMessage() {
         System.out.println("Читаем сообщение в Telegram!");
     }
}
```

Интерфейсы

Для чего нужны интерфейсы

WhatsApp

```
public class WhatsApp implements Messenger {

    public void sendMessage() {

        System.out.println("Отправляем сообщение в WhatsApp!");
    }

     public void getMessage() {
         System.out.println("Читаем сообщение в WhatsApp!");
     }
}
```

Viber

```
public class Viber implements Messenger {

    public void sendMessage() {

        System.out.println("Отправляем сообщение в Viber!");
    }

     public void getMessage() {
         System.out.println("Читаем сообщение в Viber!");
     }
}
```

Какие преимущества это дает? Самое главное из них — **слабая связанность**.

Интерфейсы

Для чего нужны интерфейсы

- Давайте представим, что мы проектируем программу, в которой у нас будут собраны данные клиентов.
- В классе Client обязательно нужно поле, указывающее, каким именно мессенджером клиент пользуется.

Без интерфейсов это выглядело бы следующим образом:

```
public class Client {

    private WhatsApp whatsApp;
    private Telegram telegram;
    private Viber viber;
}
```

Интерфейсы

Для чего нужны интерфейсы

- В данном случае, у клиента мы создали три поля для трех разных мессенджеров.
- Но у клиента может быть только один мессенджер, просто мы не знаем какой именно.
- Получается, один или два из этих полей будут всегда равны null, да и вообще не нужны для работы программы.
- Так не лучше ли использовать наш интерфейс?

```
public interface Messenger{

    public void sendMessage();

    public void getMessage();
}
```

Это и есть пример «слабой связанности»! Вместо того, чтобы указывать конкретный класс мессенджера в классе Client, мы просто упоминаем, что у клиента есть мессенджер, а какой именно — определится в ходе работы программы.

Интерфейсы

Для чего нужны интерфейсы

Но зачем нам для этого именно интерфейсы? Зачем их вообще добавили в язык?

Давайте разберемся.

Предположим, класс Messenger — родительский, а Viber, Telegram и WhatsApp — наследники.

Но есть одна загвоздка - множественного наследования в Java нет, а вот множественная реализация интерфейсов — есть.

Т.е. класс может реализовывать сколько угодно интерфейсов.

Представьте, что у нас есть класс ***Smartphone***, у которого есть поле ***Application*** — установленное на смартфоне приложение.

```
public class Smartphone {

    private Application application;
}
```

Интерфейсы

Для чего нужны интерфейсы

Приложение и мессенджер, конечно, похожи, но все-таки это разные вещи.

Мессенджер может быть и мобильным, и десктопным, в то время как Application — это именно *мобильное* приложение.

Если бы мы использовали наследование, то не смогли бы добавить объект Telegram в класс Smartphone.

Ведь класс Telegram не может наследоваться одновременно от Application и от Messenger!

А мы уже успели унаследовать его от Messenger, и в таком виде добавить в класс Client.

Но вот реализовать оба интерфейса класс Telegram запросто может!

Поэтому в классе Client мы сможем внедрить объект Telegram как Messenger, а в класс Smartphone — как Application.

Вот как это делается:

Интерфейсы

Для чего нужны интерфейсы

```
public class Telegram implements Application, Messenger {

    //...методы
}

public class Client {

    private Messenger messenger;

    public Client() {
        this.messenger = new Telegram();
    }
}


public class Smartphone {

    private Application application;

    public Smartphone() {
        this.application = new Telegram();
    }
}
```

Интерфейсы

Для чего нужны интерфейсы

Теперь можно использовать класс Telegram и в роли Application, и в роли Messenger.

Стоит еще раз обратить внимание, что методы в интерфейсах всегда пустые, т.е., не имеют реализации.

Причина этого проста - интерфейс ***описывает*** поведение, а не реализует его.

Интерфейсы – методы по умолчанию

- Ранее до JDK 8 при реализации интерфейса необходимо было обязательно реализовать все его методы в классе, а сам интерфейс мог содержать только определения методов без конкретной реализации.
- В JDK 8 была добавлена такая функциональность как методы по умолчанию.
- Начиная с этой версии интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод.

Пример:

```
interface Printable {

    default void print(){

        System.out.println("Undefined printable");
    }
}
```

```
class Journal implements Printable {

    private String name;

    String getName(){
        return name;
    }
    Journal(String name){

        this.name = name;
    }
}
```

Интерфейсы – статические методы

- Начиная с JDK 8 в интерфейсах доступны также статические методы - они аналогичны методам класса:

```
interface Printable {

    void print();

    static void read(){

        System.out.println("Read printable");
    }
}
```

- Чтобы обратиться к статическому методу интерфейса также, как и в случае с классами, пишут название интерфейса и метод:

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

    Printable.read();
}
```

Интерфейсы – private методы

- По умолчанию все методы в интерфейсе фактически имеют модификатор public.
- Однако начиная с Java 9 стало возможным определять в интерфейсе методы с модификатором private.
- Они могут быть статическими и нестатическими, но они не могут иметь реализации по умолчанию и должны использоваться только внутри самого интерфейса:

```
interface Calculatable{

    default int sum(int a, int b){
        return sumAll(a, b);
    }
    default int sum(int a, int b, int c){
        return sumAll(a, b, c);
    }

    private int sumAll(int... values){
         int result = 0;
         for(int n : values){
             result += n;
         }
         return result;
    }
}
```

```
class Calculation implements Calculatable{

}
```

```
public class Program{

    public static void main(String[] args) {

        Calculatable c = new Calculation();
        System.out.println(c.sum(1, 2));
        System.out.println(c.sum(1, 2, 4));
    }
}
```

Константы в интерфейсах

- Кроме методов в интерфейсах могут быть определены статические константы:

```
class WaterPipe implements Stateable{

    public void printState(int n){
        if(n==OPEN)
            System.out.println("Water is opened");
        else if(n==CLOSED)
            System.out.println("Water is closed");
        else
            System.out.println("State is invalid");
    }
}
```

```
interface Stateable{

    int OPEN = 1;
    int CLOSED = 0;

    void printState(int n);
}
```

```
public class Program{

    public static void main(String[] args) {

        WaterPipe pipe = new WaterPipe();
        pipe.printState(1);
    }
}
```

- Хотя такие константы также не имеют модификаторов, но по умолчанию они имеют модификатор доступа ***public static final***, и поэтому их значение доступно из любого места программы.

Наследование интерфейсов

Интерфейсы, как и классы, могут наследоваться:

```
interface BookPrintable extends Printable{

    void paint();
}
```

При применении этого интерфейса класс Book должен будет реализовать как методы интерфейса BookPrintable, так и методы базового интерфейса Printable.

Вложенные интерфейсы


```
class Printer{
    interface Printable {

        void print();
    }
}
```

Как и классы, интерфейсы могут быть вложенными, то есть могут быть определены в классах или других интерфейсах.

```
public class Journal implements Printer.Printable {

    String name;

    Journal(String name){

        this.name = name;
    }
    public void print() {
        System.out.println(name);
    }
}
```

При применении такого интерфейса нам надо указывать его полное имя вместе с именем класса.

```
Printer.Printable p =new Journal("Foreign Affairs");
p.print();
```

Использование интерфейса будет аналогично предыдущим случаям.

Интерфейсы как параметры и результаты методов

Интерфейсы могут использоваться в качестве типа параметров метода или в качестве возвращаемого типа:

```
public class Program {
    public static void main(String[] args) {

        Printable printable = createPrintable( name: "Foreign Affairs", option: false);
        printable.print();

        read(new Book( name: "Java for impatients", author: "Cay Horstmann"));
        read(new Journal( name: "Java Dayly News"));
    }

    static void read(Printable p){

        p.print();
    }

    static Printable createPrintable(String name, boolean option){

        if(option)
            return new Book(name, author: "Undefined");
        else
            return new Journal(name);
    }
}
```

```
public class Journal implements Printable {
    private String name;

    String getName(){
        return name;
    }

    Journal(String name){

        this.name = name;
    }
    public void print() {
        System.out.println(name);
    }
}
```

```
public interface Printable {
    void print();
}
```

```
public class Book implements Printable {
    String name;
    String author;

    Book(String name, String author){

        this.name = name;
        this.author = author;
    }

    public void print() {

        System.out.printf("%s (%s) \n", name, author);
    }
}
```

Интерфейсы как параметры и результаты методов

- Метод read() в качестве параметра принимает объект интерфейса Printable, поэтому в этот метод мы можем передать как объект Book, так и объект Journal.

- Метод createPrintable() возвращает объект Printable, поэтому мы можем возвратить как объект Book, так и Journal.

Консольный вывод:

```
Foreign Affairs
Java for impatients (Cay Horstmann)
Java Dayly News
```

Интерфейсы в механизме обратного вызова

Одним из распространенных способов использования интерфейсов в Java является создание обратного вызова.

Суть обратного вызова состоит в том, что программист создает действия, которые вызываются при других действиях.

Стандартный пример - нажатие на кнопку:

- когда пользователь нажимает на кнопку - он производит действие, но, в ответ на это нажатие, запускаются другие действия.
- Например, нажатие на значок принтера запускает печать документа на принтере и т.д.

Интерфейсы в механизме обратного вызова

Давайте рассмотрим пример.

```
1  public class EventsApp {
2      public static void main(String[] args) {
3          Button button = new Button(new ButtonClickHandler());
4          button.click();
5          button.click();
6          button.click();
7      }
8  }
9
10 class ButtonClickHandler implements EventHandler{
11     public void execute(){
12         System.out.println("Кнопка нажата!");
13     }
14 }
15
16 interface EventHandler{
17     void execute();
18 }
19
20 class Button{
21     EventHandler handler;
22
23     Button(EventHandler action){
24         this.handler = action;
25     }
26
27     public void click(){
28         handler.execute();
29     }
30 }
```

Интерфейсы в механизме обратного вызова

- Итак, определим класс Button, который в конструкторе принимает объект интерфейса EventHandler и в методе click() (имитация нажатия) вызывает метод execute() этого объекта.
- Далее определяется реализация EventHandler в виде класса ButtonClickHandler.
- В основной программе объект этого класса передается в конструктор класса Button.
- Таким образом, через конструктор устанавливается обработчик нажатия кнопки, который будет вызываться при каждом вызове метода button.click().
- В итоге программа выведет на консоль следующий результат:

```
Кнопка нажата!
Кнопка нажата!
Кнопка нажата!
```

Интерфейсы в механизме обратного вызова

Но, казалось бы, зачем выносить все действия в интерфейс, а потом его реализовывать?
Почему бы не написать класс Button с методом execute() и вызвать этот метод у объекта класса Button?

```
1 ▼ class Button{
2 ▼     public void click(){
3           System.out.println("Кнопка нажата!");
4 ▲     }
5 ▲ }
```

Дело в том, что на момент определения класса не всегда бывают точно известны те действия, которые должны производиться.
Особенно если класс Button и класс основной программы находятся в разных пакетах, библиотеках и могут проектироваться разными разработчиками.
К тому же может быть несколько кнопок - объектов Button - для каждого из которых надо определить свое действие.

Интерфейсы в механизме обратного вызова

Давайте изменим главный класс программы.

```
public class EventsApp {
    public static void main(String[] args) {
        Button tvButton = new Button(new EventHandler(){
            private boolean on = false;
            public void execute(){
                if(on) {
                    System.out.println("Телевизор выключен..");
                    on=false;
                } else {
                    System.out.println("Телевизор включен!");
                    on=true;
                }
            }
        });

        Button printButton = new Button(new EventHandler(){
            public void execute(){
                System.out.println("Запущена печать на принтере...");
            }
        });
        tvButton.click();
        printButton.click();
        tvButton.click();
    }
}
```

Интерфейсы в механизме обратного вызова

- Здесь имеется две кнопки - одна для включения-выключения телевизора, а другая для печати на принтере.
- Вместо того, чтобы создавать отдельные классы, реализующие интерфейс EventHandler, обработчики задаются в виде анонимных объектов, которые реализуют интерфейс EventHandler.
- Причем обработчик кнопки телевизора хранит дополнительное состояние в виде логической переменной on.
- В итоге консоль выведет нам следующий результат:

```
Телевизор включен!
Запущена печать на принтере...
Телевизор выключен..
```

- Интерфейсы в данном качестве особенно широко используются в различных графических API - AWT, Swing, JavaFX, где обработка событий объектов - элементов графического интерфейса – особенно актуальна.

Практика

Реализовать следующую структуру используя абстрактные классы и наследование.