

## ЗАНЯТИЕ 1.6

### ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

**Тема:** Создание объектно-ориентированного приложения для обработки строк. Изучение полиморфизма и абстракции на примерах.

#### 1 упражнение

Создайте новое java-приложение. Добавьте в него класс `Animal`, определите для него переменные класса (например, название животного). Реализуйте геттеры и сеттеры для этих переменных.

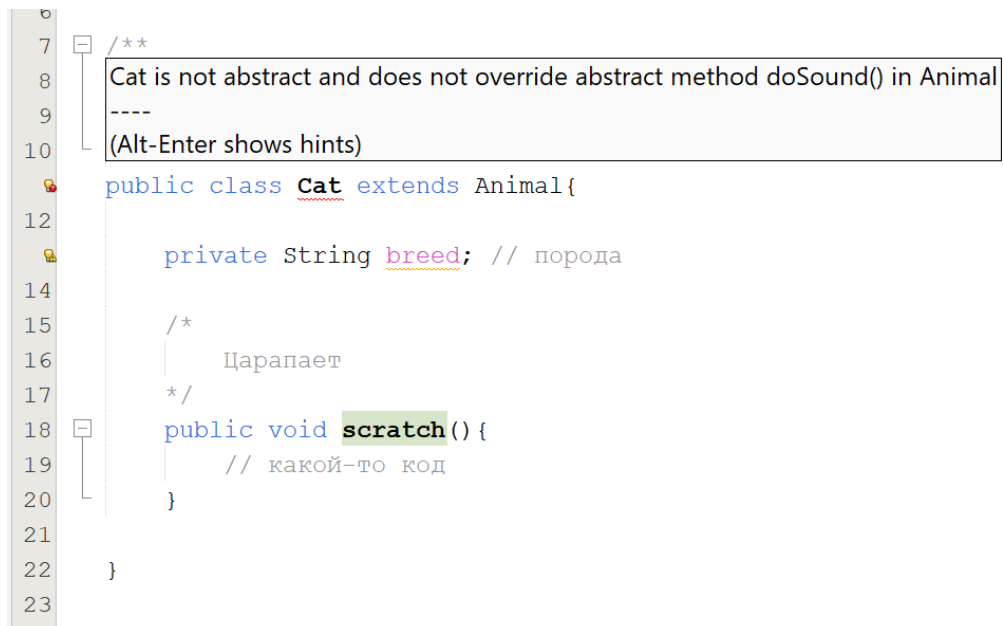
Так как данный класс описывает «общие» характеристики и «обобщенное» поведение (без конкретной реализации), то его необходимо сделать абстрактным. Определим один метод – `doSound()`. Метод будет отображать в консоль звук животного. Но так как для класса `Animal` этот метод реализовать логически трудно (какой звук издает животное?) – пометим его ключевым словом **abstract**. Это абстрактный метод. **У него нет тела.** В абстрактном классе могут быть и обычные методы.

```
12 public abstract class Animal {
13     private String title;
14
15     public Animal(String title) {
16         this.title = title;
17     }
18
19     public String getTitle() {
20         return title;
21     }
22
23     public void setTitle(String title) {
24         this.title = title;
25     }
26
27     public abstract void doSound();
28
29 }
```

Рисунок 1

Далее необходимо создать класс Cat (кошка). В нем, для примера, создадим переменную **breed** (порода), а также метод **scratch** (царапать).

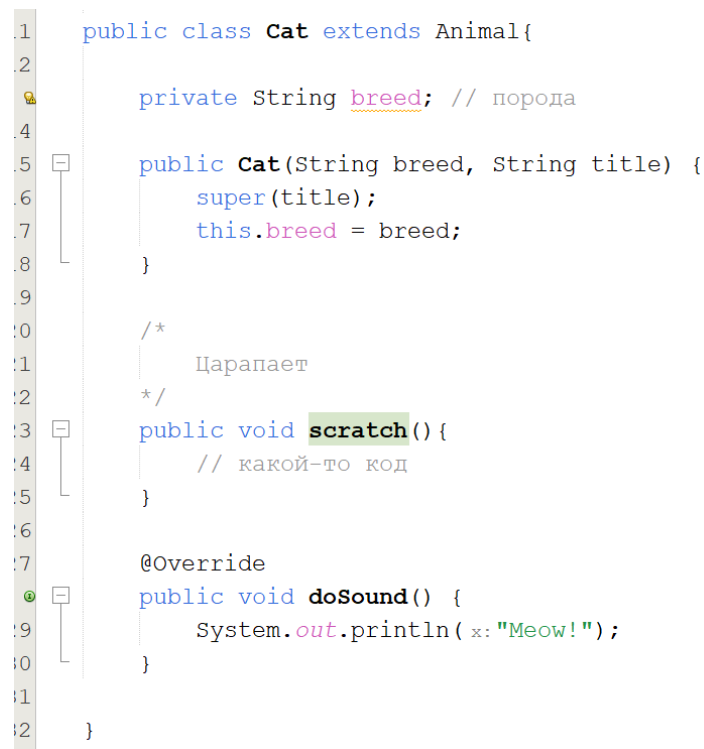
Очевидно, что Cat должен наследоваться от класса Animal. Пометим это:



```
7  /**
8   Cat is not abstract and does not override abstract method doSound() in Animal
9   ----
10  (Alt-Enter shows hints)
11
12  public class Cat extends Animal{
13
14      private String breed; // порода
15
16      /**
17       Царапает
18      */
19      public void scratch() {
20          // какой-то код
21      }
22  }
23
```

Рисунок 2

IDE «подсвечивает» ошибку при таком наследовании. Необходимо обязательно переопределить метод **doSound**, так как он абстрактный у нашего родительского класса. Также создадим конструктор в классе Cat.



```
1  public class Cat extends Animal{
2
3      private String breed; // порода
4
5      public Cat(String breed, String title) {
6          super(title);
7          this.breed = breed;
8      }
9
10     /**
11      Царапает
12     */
13     public void scratch() {
14         // какой-то код
15     }
16
17     @Override
18     public void doSound() {
19         System.out.println( x: "Meow!");
20     }
21
22 }
```

Рисунок 3

В главном классе создадим объект класса Cat и вызовем метод doSound.

```
public class Practical3 {

    public static void main(String[] args) {
        System.out.println("Practical task 1.6");

        Cat cat = new Cat(breed: "Британский", title: "кошка");
        Animal anim = new Cat(breed: "Шотландский", title: "кошка");
        // Animal anim2 = new Animal(); // НЕЛЬЗЯ СОЗДАТЬ ОБЪЕКТ АБСТРАКТНОГО КЛАССА!!!

        anim.doSound();
    }
}

com.mirea.kt.practical3.Practical3 > main >
Output X

Run (Practical2) X  Debugger Console X  Run (Test1) X  Run (Practical3) X

cd C:\Users\User\Documents\NetBeansProjects\Practical3; "JAVA_HOME=C:\Program Files\Java\
Running NetBeans Compile On Save execution. Phase execution is skipped and output directori
Scanning for projects...

-----< com.mirea.kt:Practical3 >-----
Building Practical3 1.0
-----[ jar ]-----
--- exec-maven-plugin:3.0.0:exec (default-cli) @ Practical3 ---
Practical task 1.6
Meow!

BUILD SUCCESS
```

Рисунок 4

**Полиморфизм** позволяет нам хранить ссылку на объект Cat в ссылочных переменных Animal и Cat. Но нельзя создать объект Animal!

В java для реализации абстракции также существуют интерфейсы. Для работы с ними создадим в этом же проекте знакомую иерархию классов Person-Student (или скопируйте эти классы из прошлых занятий).

```
public class Student extends Person {

    private String group; // название группы
    private int number; // порядковый номер
    boolean isFullTime; // тип обучения - очно или заочно

    public Student(String group, int number, boolean isFullTime, String name) {
        super(name);
        this.group = group;
        this.number = number;
        this.isFullTime = isFullTime;
    }

    public void goToUniversity(String univerName){
        System.out.println("Student go to " + univerName);
    }
}
```

Классы Person и Student должны содержать переменные, методы (в том числе сеттеры и геттеры), а также конструкторы.

Создадим в проекте интерфейс – **SausageLover** (любитель сосисок). Нажать правой кнопкой мыши на пакет нашего проекта, далее New -> Java Interface.

```
7  /**
8   *
9   * @author User
10  */
11  public interface SausageLover {
12
13  }
14
15  }
```

Рисунок 6

Опишем в интерфейсе один метод – eatSausage:

```
public interface SausageLover {

    void eatSausage(int count);

}
```

Рисунок 7

У метода нет тела. А по умолчанию он публичный и абстрактный (ключевые слова писать не нужно).

Реализуем данный интерфейс в наших классах – Cat (наверно, все коты любят сосиски) и Student (какой студент откажется от хорошего хот-дога?).

Несмотря на то, что сущности «студент» и «кошка» из разных иерархий наследования, они реализуют один и тот же интерфейс **SausageLover**! Это одно из главных отличий интерфейса от абстрактного класса.

Абстрактный класс связывает между собой и объединяет классы, имеющие **очень близкую связь** (отношение IS-A). В то же время, один и тот же интерфейс могут реализовать классы, у которых вообще нет ничего общего.

Так как метод в интерфейсе абстрактный, то в классах, реализующих интерфейс, необходимо его переопределить.

```

public class Cat extends Animal implements SausageLover{

    private String breed; // порода

    public Cat(String breed, String title) {
        super(title);
        this.breed = breed;
    }

    /*
        Царапает
    */
    public void scratch() {
        // какой-то код
    }

    @Override
    public void doSound() {
        System.out.println( x: "Meow!");
    }

    @Override
    public void eatSausage(int count) {
        System.out.println("Cat eats " + count + " sausages");
        // еще какой-то специфичный код
    }

}

```

Рисунок 8

Теперь для ссылочной переменной типа Cat мы можем вызвать метод **eatSausage**.

## 2 упражнение

Создайте новое приложение, содержащее класс Person, или продолжите работать с java-приложением, созданным на практическом занятии 1.4.

Внесем некоторые изменения в класс Person. Необходимо создать переменные класса, содержащие имя, фамилию (раньше было только имя) и возраст. Также создадим конструктор, геттеры и сеттеры.

```

    * @author User
    */
public class Person {
    private String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public void doSleep(int hours){
        System.out.println("Person sleep " + hours + " hours");
    }
}

```

Рисунок 9

Добавим в класс Person новый метод – talk (человек ведь умеет говорить), в котором «человек» здоровается.

```

public void talk(){
    System.out.println("Hello! I am an ordinary person!");
}

```

Рисунок 10

Создадим в главном классе объект класса Person и проверим его работоспособность, вызвав метод **talk**.

```
1  * @author User
2  */
3  public class Practical3 {
4
5      public static void main(String[] args) {
6          System.out.println("Practical task 1.6");
7          Person person = new Person( firstName: "Михаил", lastName: "Салтыков");
8          person.setAge( age:21);
9          person.talk();
10     }
11 }

com.mirea.kt.practical3.Practical3 >

Output X

Run (Practical2) X  Debugger Console X  Run (Practical3) X

cd C:\Users\User\Documents\NetBeansProjects\Practical3; "JAVA_HOME=C:\\Prog
Running NetBeans Compile On Save execution. Phase execution is skipped and
Scanning for projects...

-----< com.mirea.kt:Practical3 >-----
Building Practical3 1.0
-----[ jar ]-----
--- exec-maven-plugin:3.0.0:exec (default-cli) @ Practical3 ---
Practical task 1.6
Hello! I am an ordinary person!
-----
BUILD SUCCESS
```

Рисунок 11

Создадим класс Student (если его нет в проекте), добавим переменные класса и конструктор. Класс Student расширяет класс Person.

```
public class Student extends Person {

    private String group; // название группы
    private int number; // порядковый номер
    boolean isFullTime; // тип обучения - очно или заочно

    public Student(String group, int number, boolean isFullTime, String firstName, String lastName) {
        super(firstName, lastName);
        this.group = group;
        this.number = number;
        this.isFullTime = isFullTime;
    }

    public void goToUniversity(String univerName){
        System.out.println("Student go to " + univerName);
    }
}
```

Рисунок 12

Для класса Student необходимо переопределить два метода:

- 1) Метод **toString** (метод наследуется от класса Object). Этот метод преобразует объект класса к строке. Необходимо реализовать в этом методе «собственный» функционал:

```

51
52     @Override
53     public String toString() {
54         return "Student{firstName = " + getFirstName() +
55             ", lastName = " + getLastName() +
56             ", age = " + getAge() +
57             ", group = " + this.group +
58             ", number = " + this.number +
59             ", isFullTime = " + this.isFullTime + "}";
60     }
61 }

```

Рисунок 13

- 2) Метод **talk** (метод наследуется от класса **Person**). У сущности «студент» собственная реализация данного метода:

```

@Override
public void talk() {
    System.out.println("Hello! I'm not an ordinary person!"
        + " I am a student!");
}

```

Рисунок 14

Необходимо обратить внимание на аннотацию **@Override**. Она означает, что метод переопределяется от родительского класса.

Создадим новый класс – **MFC** (МФЦ). Данный класс будет работать с сущностью «человек» и предоставлять различные государственные услуги – всё как в жизни! Для класса **MFC** определим переменные, которые отражают адрес многофункционального центра и количество доступных сотрудников. Также необходимо реализовать конструктор, сеттеры (с проверкой на некорректные значения) и геттеры.



```

10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
public class MFC {
    private String location;
    private int operatorsCount;

    public MFC(String location) {
        this.location = location;
    }

    public void setOperatorsCount(int operatorsCount) {
        if(operatorsCount <= 0){
            this.operatorsCount = 1;
        }else{
            this.operatorsCount = operatorsCount;
        }
    }

    public int getOperatorsCount() {
        return operatorsCount;
    }
}

```

Рисунок 15

Определим в классе MFC метод, который будет изменять имя и фамилию человека (одна из услуг МФЦ). В качестве аргумента метод принимает ссылку (ссылочную переменную) класса Person. Выберем название для метода – **changePersonParams**. При наличии свободных операторов в МФЦ, «человек» здороваются (в методе talk рассказывает о себе), затем предоставляет свои данные (имя и фамилию) для изменения:

```

public void changePersonParams(Person person){
    if(person != null && this.operatorsCount > 0){
        person.talk();
        String fName = person.getFirstName();
        fName = fName.toUpperCase();
        String lName = person.getLastName();
        lName = lName + "-Щедрин";
        lName = lName.toUpperCase();
        person.setFirstName( firstName: fName);
        person.setLastName( lastName: lName);
    }
}

```

Рисунок 16

Как метод работает с этими данными? Он преобразует символы имени и фамилии объекта класса **Person** в прописные (заглавные), а также добавляет к фамилии «-Щедрин». Убедимся, что метод работает – вызовем его из главного класса и изучим значения переменных в режиме отладки:

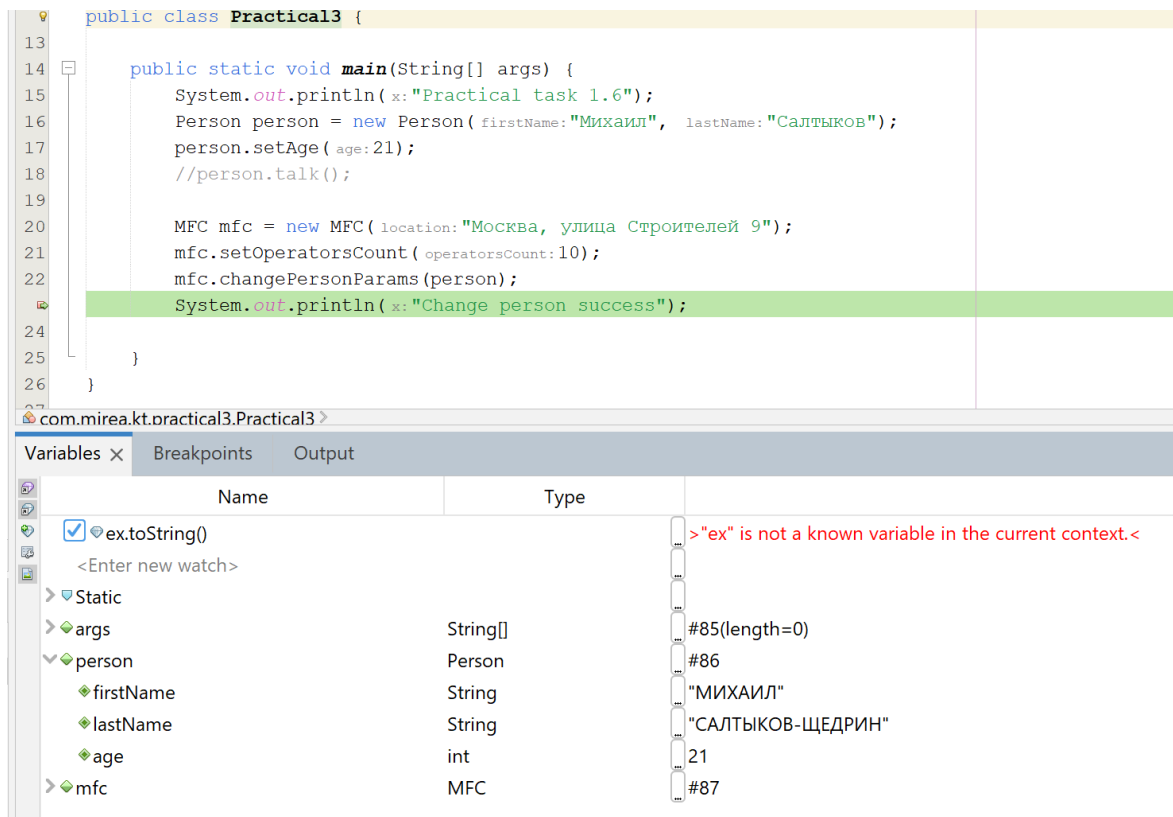
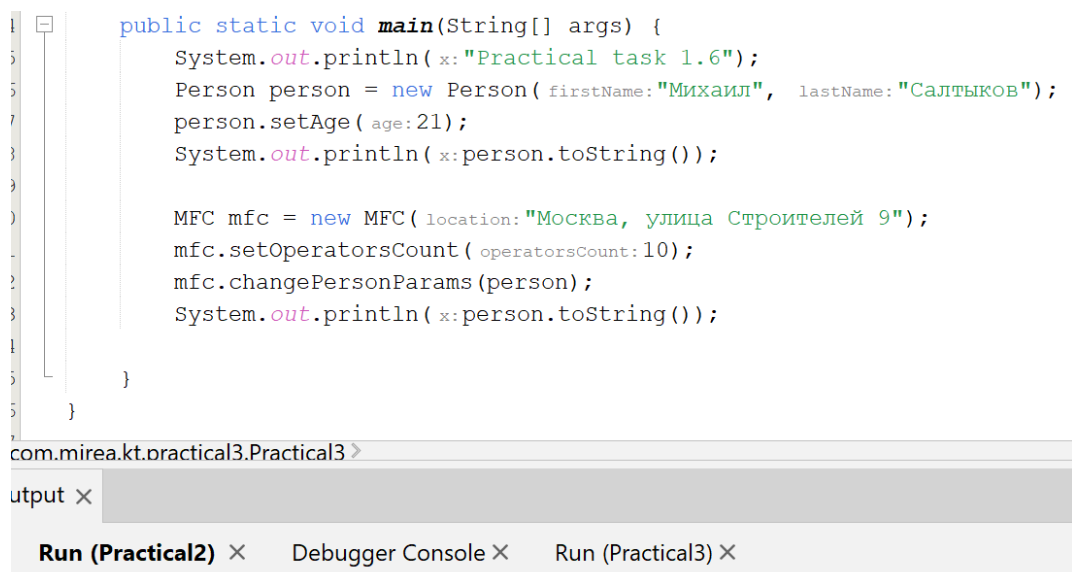


Рисунок 17

Как можно заметить, значения переменных **firstName** и **lastName** объекта класса **Person** были изменены методом **changePersonParams** класса **MFC**. Переопределим метод **toString** в классе **Person** (так же как для «студента») и выведем значения переменных до преобразования и после.



```

--- exec-maven-plugin:3.0.0:exec (default-cli) @ Practical3 ---
Practical task 1.6
Person{firstName = Михаил, lastName = Салтыков, age = 21}
Hello! I am an ordinary person!
Person{firstName = МИХАИЛ, lastName = САЛТЫКОВ-ЩЕДРИН, age = 21}
-----
BUILD SUCCESS
-----
Total time: 0.715 s
Finished at: 2023-02-23T16:15:32+03:00
-----

```

Рисунок 18

Реализуем такое же преобразование имени и фамилии для экземпляра класса **Student**. Для этого необходимо создать объект класса **Student** (см. практическое занятие 1.4) и передать ссылочную переменную в метод **changePersonParams** класса **MFC**. Но метод **changePersonParams** в качестве аргумента принимает переменную типа **Person**. Как передать в этот метод переменную типа **Student**? Как нужно изменить код метода?

```

public MFC(String location) {
    this.location = location;
}

public void changePersonParams(Person person) {
    if (person != null && this.operatorsCount > 0) {
        person.talk();
        String fName = person.getFirstName();
        fName = fName.toUpperCase();
        studentName = person.getLastName();
    }
}

```

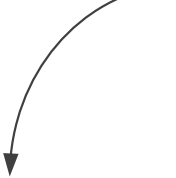


Рисунок 19

Не нужно ничего менять! ООП позволяет нам передавать дочерний тип ссылки вместо родительского в качестве аргумента. Более того, метод **talk** будет вызван именно тот, который реализован в классе **Student**. Это пример использования **полиморфизма**.

Это можно проверить:

```
13 public class Practical3 {
14
15     public static void main(String[] args) {
16         System.out.println(x:"Practical task 1.6");
17         //Person person = new Person("Михаил", "Салтыков");
18         Student student = new Student(group:"РИБО-00", number:5, isFullTime:true, firstName:"Михаил", lastName:"Салтыков");
19         student.setAge(age:21);
20         System.out.println(x:student.toString());
21
22         MFC mfc = new MFC(location:"Москва, улица Строителей 9");
23         mfc.setOperatorsCount(operatorsCount:10);
24         mfc.changePersonParams(person:student);
25         System.out.println(x:student.toString());
26     }
27 }
```

com.mirea.kt.practical3.Practical3

Output ×

Run (Practical2) ×    Debugger Console ×    Run (Practical3) ×

```
cd C:\Users\User\Documents\NetBeansProjects\Practical3; "JAVA_HOME=C:\Program Files\Java\jdk1.8.0_241" cmd /c "\"C:\'
Running NetBeans Compile On Save execution. Phase execution is skipped and output directories of dependency projects (t
Scanning for projects...

-----< com.mirea.kt:Practical3 >-----
Building Practical3 1.0
-----[ jar ]-----

--- exec-maven-plugin:3.0.0:exec (default-cli) @ Practical3 ---
Practical task 1.6
Student{firstName = Михаил, lastName = Салтыков, age = 21, group = РИБО-00, number = 5, isFullTime = true}
Hello! I'm not an ordinary person! I am a student!
Student{firstName = МИХАИЛ, lastName = САЛТЫКОВ-ЩЕДРИН, age = 21, group = РИБО-00, number = 5, isFullTime = true}

BUILD SUCCESS
```

Рисунок 20

Содержимое консоли доказывает, что вызывался метод **talk**, переопределенный в классе **Student**. Несмотря на то, что **changePersonParams** «работает» с ссылкой типа **Person**.

## Индивидуальное задание

**Задание:** модернизировать (модифицировать) Java-программу из практического задания №1.4 в соответствии с вариантом.

### Требования:

- 1) На сдачу практического задания отводится 7 дней (или до следующего практического занятия).
- 2) Вариант определяется согласно порядковому номеру студента в журнале.
- 3) В случае дистанционного выполнения практического задания код программы необходимо выложить на Github и предоставить ссылку на него.

- 4) Итоговая программа должна компилироваться без ошибок, полностью выполнять требуемый функционал и при старте выводить в консоль номер варианта и ФИО студента.
- 5) Необходимо использовать принципы ООП. Названия переменных и методов должны отражать суть и нести смысловую нагрузку.
- 6) Необходимо придерживаться стилистике по написанию Java-кода.

#### Дополнительная информация:

- 1) Перед выполнением задания рекомендуется ознакомиться с Лекциями №1.3, №1.5, а также выполнить Практическое задание №1.4.
- 2) Для работы со строками (тип String) рекомендуется ознакомиться со справочным материалом из официальной документации Java SE: <https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>
- 3) Для вывода на экран значений полей класса рекомендуется переопределить метод toString() и использовать его.

Вариант 1. Для работы с сущностью **АВТОМОБИЛЬ (Car)** разработать класс **АВТОСЕРВИС**, в котором метод **modify** будет изменять название (марку) автомобиля по следующему алгоритму:

- Все символы «а» должны быть заменены на «о»
- Все символы «і» должны быть заменены на «е»
- Все символы должны быть прописными

Начальные значения полей для сущности **АВТОМОБИЛЬ** вводит пользователь с клавиатуры после старта программы. В результате работы программы необходимо вывести на экран все значения полей (в том числе измененное) через запятую.

Вариант 2. Для работы с сущностью **ТЕЛЕФОННЫЙ АППАРАТ (Telephone)** разработать класс **ПРОДАВЕЦ**, в котором метод **modify** будет изменять название производителя телефонного аппарата по следующему алгоритму:

- Все символы «о» должны быть заменены на «а».
- Все символы названия должны быть строчные, кроме названий, начинающихся на «N» (или «n») – такие оставлять как есть.

Начальные значения полей сущности **ТЕЛЕФОННЫЙ АППАРАТ** вводит пользователь с клавиатуры после старта программы. В результате работы программы необходимо вывести на экран все значения полей (в том числе измененное) через запятую.

Вариант 3. Для **двух** вариантов дочерних классов сущности **ВРАЧ (Doctor)** переопределить (для разных дочерних классов) метод **writeRecipe** (если у вас этого метода в родительском классе нет – создайте его), который на вход в качестве аргумента принимает переменную типа String с текстом рецепта:

- В первом случае текст рецепта необходимо развернуть (например, «поставь градусник» -> «кинсударг ьватсоп»).
- Во втором случае необходимо выполнить любое другое преобразование строки (на усмотрение студента).

Значения полей сущности **ВРАЧ**, выбор конкретной реализации сущности (дочернего класса), а также текст рецепта вводит пользователь последовательно с клавиатуры с после старта программы. В результате работы программы необходимо вывести на экран преобразованный текст рецепта.

Вариант 4. Для работы с сущностью **РАСТЕНИЕ (Plant)** разработать класс **САДОВНИК**, в котором метод **filter** будет изменять название растения по следующему алгоритму:

- В названии должны отсутствовать гласные буквы.
- К концу получившегося названия растения необходимо дописать строку «VGTBL».

Начальные значения полей для сущности **РАСТЕНИЕ** вводит пользователь с клавиатуры после старта программы. В результате работы программы необходимо вывести на экран все значения полей (в том числе измененное) через запятую.

Вариант 5. Для работы с сущностью **МАГАЗИН (Store)** разработать класс **РЕВИЗОР**, в котором необходимо реализовать два метода:

- метод **closeStore** будет присваивать значение null полю name класса **Store** (если у вас не было такого поля – создайте его), если до этого оно имело значение «IKEA» (если нет, то оставить как было).
- метод **rebrand** будет изменять название магазина (поле name) на «Вкусно и точка», если до этого оно имело значение «MacDonalds»; если же поле имело любое другое значение, то необходимо убрать только первый символ названия.

Начальные значения полей для сущности **МАГАЗИН** вводит пользователь с клавиатуры после старта программы. В результате работы программы необходимо вывести на экран все значения полей (в том числе измененное) через запятую.