Лабораторная работа №4

Абстрактные классы и методы. Интерфейсы. Множественное наследование интерфейсов

Абстрактные классы

Абстрактным называется класс, на основе которого не могут создаваться объекты. При этом наследники класса могут быть не абстрактными, на их основе объекты создавать, соответственно, можно. Любой класс, содержащий методы abstract, также должен быть объявлен, как abstract. Поскольку у таких классов отсутствует полная реализация, их представителей нельзя создавать с помощью оператора new. Кроме того, нельзя объявлять абстрактными конструкторы и статические методы. Любой подкласс абстрактного класса либо обязан предоставить реализацию всех абстрактных методов своего суперкласса, либо сам должен быть объявлен абстрактным.

```
abstract class A {
abstract void callme();
void metoo() {
System.out.println("Inside A's metoo method");
} }
class B extends A {
void callme() {
System.out.println("Inside B's callme method");
} }

class my {
public static void main(String args[]) {
A a = new B();
a.callme();
a.metoo();
} }
```

Рассмотрим еще один пример абстрактного класса A:

```
abstract class A {
    int p1;
    A() {
        p1 = 1;
    }
    void print() {
        System.out.println(p1);
    }
}
class B extends A {
}
public class Main {
    public static void main(String[] args) {
        A ob1;
        // ошибка: ob1 = new A();
        B ob2 = new B(); // будет вызван конструктов по умолчанию из A
        ob2.print();
    }
}
```

Java разрешит описать конструкторы в классе A, но не разрешит ими воспользоваться (потому что запрещено создавать объекты абстрактного класса).
Обратите внимание на то, что объявление переменной ob1 как ссылки, на объект класса A тоже не запрещается.

Приведение классов

Зачем же может потребоваться ссылка ob1, какой объект с ней удастся связать? Ну, например, объект класса-потомка B. Дело в том, что класс A, как родитель, является более универсальным, чем потомок B. Это значит, что любой объект класса потомка может быть явно или даже автоматически приведён к классу родителю.
То есть следующее содержание метода main было бы вполне корректным:

```
A ob1;
B ob2 = new B();
ob1 = (A) ob2; // явное приведение
ob1.print();
```

Более того, приведение могло быть и неявным (автоматическим):

```
ob1 = ob2; // автоматическое приведение
```

Как для встроенных типов, так и для классов автоматическое приведение всегда возможно, когда переменную или объект мы пытаемся привести к более универсальному типу (целые числа к вещественным, объект потомка B к классу родителю A и пр.).

Абстрактные методы

Абстрактным называется метод, который не имеет реализации в данном классе. После круглых скобок, где перечислены его аргументы, ставится не открывающая фигурная скобка, чтобы начать блок описания метода, а точка с запятой. То есть описание у абстрактноно метода отсутствует. Перед именем метода указывается при этом модификатор abstract.
Какой смысл в создании метода без реализации? Ведь его нельзя будет использовать. Для объектов того класса, где метод описан – конечно же использовать нельзя, но вот если унаследовать класс и в потомках переопределить метод, задав там его описание, то для объектов классов потомков метод можно будет вызывать (и работать будут описанные в классах потомках реализации).
Чтобы исключить возможность использования абстрактного метода, в Java введено следующее требование: класс имеющий хоть один абстрактный метод обязан быть абстрактным классом.
Когда же уместно использовать абстрактные методы и классы? Сначала рассмотрим пример иерархии классов домашних животных, где нет ни абстрактных классов, ни абстрактных методов.

```
class Pet {
    String name;
    int age;
    boolean hungry;
    void voice() {
    }
    void food() {
        hungry = false;
```

```
    }
}
class Snake extends Pet {
    double length;
    void voice() {
        System.out.println("Шшш-ш-ш");
    }
}
class Dog extends Pet {
    void voice() {
        System.out.println("Гав-гав");
    }
}
class PatrolDog extends Dog {
    void voice() {
        System.out.println("Ррр-р-р");
    }
}
class Cat extends Pet {
    void voice() {
        System.out.println("Мяу-мяу");
    }
}
class Fish extends Pet {
}
public class Main {
    public static void main(String[] args) {
        Pet zorka = new Pet();
        zorka.food();
        Fish nemo = new Fish();
        nemo.voice();
    }
}
```

Поскольку нет какого-то общего звука, который издавали бы все домашние животные, то мы в классе Pet не стали задавать какую-то реализацию методу voice(), внутри метода не делается совсем ничего, но тем не менее у него есть тело, обособленное блоком из фигурных скобок. Метод voice() хороший претендент на то, чтобы стать абстрактным.

Кроме того, вряд ли можно завести себе домашнее животного неопределенного вида, то есть у вас дома вполне могли бы жить Cat, Dog или даже Snake, но вряд ли вы бы смогли завести животное Pet, являющееся непонятно кем.

Соответственно, в рамках реальной задачи вряд ли потребуется создавать объекты класса Pet, а значит его можно сделать абстрактным (после чего, правда, мы даже при делании не сможем создать объекты на его основе).

Обратите внимание, что при создании абстрактного класса Pet, во-первых, мы не можем создавать объекты этого класса , а, во-вторых, реализация метода voice() должна иметься во всех его потомках (хотя бы пустая реализация), не являющихся абстрактными классами.

Хотя, мы могли бы создать абстрактного потомка:

```
abstract class Fish extends Pet {
}
```

Но не могли бы создавать объектов класса Fish, нам пришлось бы расширять класс, чтоб в итоге получить не абстрактный и создавать на его основе объекты. Например:

```
class GoldenFish extends Fish {
    void voice() {
    }
}
```

Интерфейсы

Интерфейсы Java похожи на абстрактные классы. Они созданы для поддержки динамического выбора (resolution) методов во время выполнения программы. В отличие от классов у интерфейсов нет переменных представителей, а в объявлениях методов отсутствует реализация. Класс может иметь любое количество интерфейсов. Все, что нужно сделать — это реализовать в классе полный набор методов всех интерфейсов. Сигнатуры таких методов класса должны точно совпадать с сигнатурами методов реализуемого в этом классе интерфейса. Интерфейсы обладают своей собственной иерархией, не пересекающейся с классовой иерархией наследования. Это дает возможность реализовать один и тот же интерфейс в различных классах, никак не связанных по линии иерархии классового наследования. Именно в этом и проявляется главная сила интерфейсов. Интерфейсы являются аналогом механизма множественного наследования в C++, но использовать их намного легче.

Оператор interface

Определение интерфейса сходно с определением класса, отличие состоит в том, что в интерфейсе отсутствуют объявления данных и конструкторов. Общая форма интерфейса приведена ниже:

interface имя {
тип_результата имя_метода1(список параметров);
тип имя_final1-переменной = значение;
}

Обратите внимание — у объявляемых в интерфейсе методов отсутствуют операторы тела. Объявление методов завершается символом ; (точка с запятой). В интерфейсе можно объявлять и переменные, при этом они неявно объявляются final - переменными. Это означает, что класс реализации не может изменять их значения. Кроме того, при объявлении переменных в интерфейсе их обязательно нужно инициализировать константными значениями. Ниже приведен пример определения интерфейса, содержащего единственный метод с именем callback и одним параметром типа int.

interface Callback {
void callback(int param);
}

Оператор implements

Оператор implements — это дополнение к определению класса, реализующего некоторый интерфейс(ы).

class имя_класса [extends суперкласс]
[implements интерфейс0 [, интерфейс1...]] { тело класса }

Если в классе реализуется несколько интерфейсов, то их имена разделяются запятыми.

Ниже приведен пример класса, в котором реализуется определенный нами интерфейс:

class Client implements Callback {
//при реализации метода интерфейса он объявляется как public

```
public void callback(int p) {
System.out.println("callback вызван с аргументом " + p);
} }

public class interf {
public static void main(String args[]) {
Callback c = new Client();
c.callback(42);
}
}
```

Ниже приведен результат работы программы:
callback called with 42

Интерфейс это конструкция языка программирования Java, в рамках которой могут описываться только абстрактные публичные (abstract public) методы и статические константные свойства (final static). То есть также, как и на основе абстрактных классов, на основе интерфейсов нельзя порождать объекты.
Один интерфейс может быть наследником другого интерфейса.
Классы могут реализовывать интерфейсы (т. е. получать от интерфейса список методов и описывать реализацю каждого из них), притом, что особенно важно, один класс может реализовывать сразу несколько интерфейсов.
Перед описанием интерфейса указывается ключевое слово interface. Когда класс реализует интерфейс, то после его имени указывается ключевое слово implements и далее через запятую перечисляются имена тех интерфейсов, методы которых будут полностью описаны в классе.

Пример:

```
interface Instruments {
    final static String key = "До мажор";
    public void play();
}
class Drum implements Instruments {
    public void play() {
        System.out.println("бум бац бац бум бац бац");
    }
}
class Guitar implements Instruments {
    public void play() {
        System.out.println("до ми соль до ре до");
    }
}
```

Поскольку все свойства интерфейса должны быть константными и статическими, а все методы общедоступными, то соответствующие модификаторы перед свойствами и методами разрешается не указывать. То есть интерфейс можно было описать так:

```
interface Instruments {
    static public String key = "До мажор";
    void play();
}
```

Но когда метод play() будет описываться в реализующем интерфейс классе, перед ним всё равно необходимо будет явно указать модификатор public.

Переменные в интерфейсах

Интерфейсы можно использовать для импорта в различные классы совместно используемых констант. В том случае, когда вы реализуете в классе какой-либо интерфейс, все имена переменных этого интерфейса будут видимы в классе как константы. Это аналогично использованию файлов-заголовков для задания в C и C++ констант с помощью директив #define или ключевого слова const в Pascal / Delphi.
Если интерфейс не включает в себя методы, то любой класс, объявляемый реализацией этого интерфейса, может вообще ничего не реализовывать. Для импорта констант в пространство имен класса предпочтительнее использовать переменные с модификатором final. В приведенном ниже примере проиллюстрировано использование интерфейса для совместно используемых констант.

```
package interf;
import java.util.Random;

interface SharedConstants {
int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5; }

class Question implements SharedConstants {
Random rand = new Random();

int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30)
return NO; // 30%
else if (prob < 60)
return YES; // 30%
else if (prob < 75)
return LATER; // 15%
else if (prob < 98)
return SOON; // 13%
else
return NEVER; // 2%
}

void answer(int result){
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
```

```
System.out.println("Maybe");
break;
case LATER:
System.out.println("Later");
break;
case SOON:
System.out.println("Soon");
break;
case NEVER:
System.out.println("Never");
break;
} }
}

class MyInt implements SharedConstants{

      public static void main(String args[]) {
            Question q = new Question();
            q.answer(q.ask());
            q.answer(q.ask());
            q.answer(q.ask());
            q.answer(q.ask());
            }
}
```

Множественное наследование интерфейсов

Java не поддерживает множественное наследование классов. Это объясняется тем, что такое наследование порождает некоторые проблемы.
Чаще всего указываются неоднозначности, возникающие при так называемом «ромбовидном» наследовании, когда один класс «A» является наследником двух других классов «B» и «C», которые в свою очередь оба являются наследниками класса «D». (Благодаря очертаниям диаграммы наследования классов в этой ситуации, напоминающим очертания граненого алмаза проблема еще получила название «проблема Алмаза»). Проблема допустимости множественного наследования кроется в следующем. Предположим, что в родителе A определялся какой-то метод m1(). И этот же метод мы вызываем для объекта класса D. А что если m1() был переопределён в классах B и C. Какая реализация из трёх будет работать при вызове метода m1() для объекта класса D? От неоднозначности можно было бы избавиться потребовав в описанном случае при вызове уточнять при вызове, какой из методов требуется (так и сделано в некоторых языках), но в Java от множественного наследования классов решили отказаться.
Вместо множественного наследования классов в Java введено множественное наследование интерфейсов, которое частично решает проблемы (но, как будет показано в примере далее, к сожалению, не все).
Ниже показан пример реализации двух интерфейсов в классе My:

```
interface InterfaceA {

    public void doSomething1();
}
interface InterfaceB {

    public void doSomething2();
}


public class My implements InterfaceA, InterfaceB {

    public void doSomething1() {
        System.out.println("doSomething1 реализация реального класса ");
    }
    public void doSomething2() {
        System.out.println("doSomething2 реализация реального класса ");
    }
    public static void main(String[] args) {
        InterfaceA objA = new My();
        InterfaceB objB = new My();

        objA.doSomething1();
        objB.doSomething2();
    }
}
```

Часто всю открытую часть класса (т. е. общедоступные методы) предопределяют как раз в интерфейсе. Тогда взглянув на один лишь интерфейс можно понять какие же методы должны использоваться для взаимодействия с объектами данного класса. То есть интерфейсы вполне соответствуют принципам инкапсуляции. Как, впрочем, и принципу полиморфизма. Ведь в нескольких классах метод некоторого интерфейса может быть реализован по-разному, хотя и с одним и тем же именем.
Но интерфейсы, как говорилось выше, не являются совершенным инструментом лишенным всяких недостатков. Рассмотрим пример, когда у нас имеются два интерфейса, в каждом из которых есть свойства с одинаковыми именами (но, возможно, разными значениями) и методы с одинаковыми именами.
Унаследовав класс от пары этих интерфейсов мы не сможем обращаться к свойству его объектов напрямую, без указания того, какой из двух интерфейсов мы имели в виду. Это ограничение существует потому, что в интерфейсах свойствам может даваться разное начальное значение и, соответственно, программа не сможет определить какое же значение выбрать.
Также к свойству можно обратиться как к статическому свойству одного из интерфейсов (разумеется, это можно делать и если у свойств были бы разные имена).
Проблема исчезнет, если перед обращением к свойству мы приведём объект к одному из родительских интерфейсов (напомним, что любой объект можно явно привести к классу или интерфейса его родителя прямого или транзитивного).
К сожалению, создать отдельные реализации для двух одноимённых методов из разных интерфейсов в классе наследнике не получится (чтобы потом ими можно было пользоваться через то же приведение объектов к нужному интерфейсу). Если класс реализует несколько интерфейсов, в которых есть одноимённые методы, то в нём может задаваться лишь одна общая для всех реализация этих методов (и это уже ограничивает полиморфизм при множественном наследовании через интерфейсы в Java).
Итак, код примера:

```
interface Interface1 {
    int someField = 100;
    String someMethod();
}
interface Interface2 {
    int someField = 200;
    String someMethod();
}
class SomeClass implements Interface1, Interface2 {
```

```
    public String someMethod() {
        return "It Works";
    }
}
public class Main {
    public static void main(String[] args) {
        SomeClass a = new SomeClass();
        System.out.println( a.someMethod() ); // It works
        System.out.println( a.someField ); // ошибка
        System.out.println( ( (Interface1) a).someField ); // 100
        System.out.println( Interface1.someField ); // 100
    }
}
```

Задание 1.
На основании описаного выше класса Pet («домашние животные») создать абстрактный класс Pet, на основании которого создать подклассы для каждого вида домашних животных. В каждом подклассе реализовать метод voice() («голос»). Например, для собаки он будет иметь вид
void voice() {
System.out.println("Гав-гав");
}
Задание 2.
На основании созданного в **Задании 1** абстрактного класса Pet, создать интерфейс. Реализовать его в классах для нескольких видов домашних животных.

Задание 3.

Создать два интерфейса: ***CargoAuto*** (грузовой транспорт) и ***PassangersAuto*** (легковой транспорт), в каждом из которых имеется по одному методу. Создать класс Pickup (пикап), который должен обладать как возможностью перевозки грузов, так и пассажиров, поэтому он реализует сразу оба интерфейса: метод первого должен выдавать сообщение ***"Везу груз",*** а второго ***-"Везу пассажиров"***.