# Лабораторная работа N 3

## *Разработка объектно-ориентированной программы на Java*

### 1. Введение

Все современные языки программирования можно считать в том или ином смысле ориентированными на объекты. В языке Java продолжается традиция SmallTalk считать все используемые конструкции объектами. Другие объектно-ориентированные языки не придерживаются этого принципа столь строго, но в Java объектами считаются и константы, и типы, и записи, и структуры данных.

**Целью лабораторной работы** является изучение основ объектно-ориентированного программирования в Java. Изучить способы создания иерархии классов, перезагрузки функций в Java, механизмы сокрытия информации, ознакомиться с пакетами и интерфейсами JDK.

### 2. Общие сведения

### 2.1 ООП в Java.

Язык Java является объектно-ориентированным языком. В основе языка Java лежит понятие класса как некого единства данных и функций (методов), которые обрабатывают эти данные. Если в традиционном процедурном программировании данные и функции отделены друг от друга, то в объектно-ориентированном они объединяются под общей "крышей" с названием класс. Класс можно определить как тип объекта. Каждый объект принадлежит некоторому классу, определяющему совокупность данных и методов, характерных для этого объекта. Объектно-ориентированное программирование основано на трех концепциях:

- **Инкапсуляция** — объединение в объекте данных и функций для обработки этих данных.

- **Наследование** — механизм, посредством которого один объект может наследовать свойства (данные и функции) другого объекта и добавлять к ним черты, характерные для него.

- **Полиморфизм** — это свойство, которое позволяет одно и то же имя использовать для решения разных задач.

Сочетание наследования и полиморфизма позволяет легко создать серию подобных, но вместе с тем уникальных объектов.

Благодаря наследованию такие объекты имеют много схожих характеристик, а благодаря полиморфизму, каждый из них может обладать собственным поведением.

Класс объявляется с помощью ключевого слова `class`.

Синтаксис объявления класса:

```
class имя_класса {
    функции и переменные класса
}
```

### 2.1.1 Инкапсуляция

Инкапсуляция заключается в объединении данных и функций, которые обрабатывают эти данные, в единое целое, называемое классом.

Имеется возможность задать ограничения для объектов других классов на доступ к функциям (*методам*) и переменным (*данным*) этого класса. Для этого существуют так называемые модификаторы доступа **public** (публичный), **protected** (защищенный), **private** (приватный), **final**(конечный). Модификаторы доступа обеспечивают защиту данных и методов класса от внешнего вмешательства и неправильного использования.

Переменные и методы должны быть "видны" другим объектам только в том случае, когда правила доступа разрешают это. Хорошее конструирование диктует правило, что должно быть видимо только то, что необходимо, и не более.

- Если модификатор перед объявлением свойства (переменная/метод класса) не указывается, то свойство будет "видно" только классам в том же пакете.
- Модификатор **public** указывает, что свойства (методы/переменные) данного класса будут видны объектам других классов. Это так называемая широко открытая видимость (wide-open visibility).
- Модификатор **private** указывает, что доступ к свойствам и методам класса может осуществляться только посредством методов объектов данного класса.
- Модификатор **protected** указывает, что данные и методы данного класса могут быть доступны как из объектов данного класса так из объектов подклассов данного класса и классов из того же пакета.
- Модификатор **private protected** представляет собой специальную комбинацию модификаторов доступа **private** и **protected.** Комбинация **private protected** разрешает доступ к свойству класса только из подклассов данного класса**. Данный модификатор не используется в версиях языка Java выше версии 1.0.**

Если функция класса не является **public**, то апплет не может вызвать данную функцию используя оператор ".". Единственный способ вашего апплета получить доступ к членам с меткой **protected** состоит в использовании интерфейсных функций.

Если перед определением переменной поместить модификатор **final**, то это фактически превращает ее в константу. Метод, помеченный как **final**, в подклассах нельзя переопределить.

Использовать метод **final** можно с разными целями. Во-первых, *для повышения безопасности*. Можно позволить создавать подкласс и наследовать в нем все свойства суперкласса, запрещая, однако, применение собственных версий методов при наследовании. Во-вторых, *для уменьшения времени выполнения программы*. Если метод помечен как **final**, других его версий не существует. Следовательно, в процессе выполнения программы нет необходимости обращаться к процедурам динамического связывания и ожидать результатов их работы. Компилятор может оптимизировать код программы, поэтому модификатор **final** должно применять как можно чаще.

### 2.1.2 Наследование. Иерархия классов.

*Наследование(inharitance)* представляет собой одну из самых мощных концепций объектно-ориентированного программирования.

Свойства класса передаются "по наследству". Скажем, можно создать класс "автомобили", содержащий методы и данные, характерные для всех автомобилей, и на его основе другой класс — "легковые автомобили", наследующий (inherit) все свойства родительского класса "автомобили" и обладающий некоторыми другими дополнительными собственными свойствами.

Наследование облегчает работу программиста, позволяя записывать общие части программы только один раз (в родительском классе).

Такая стратегия, например, упрощает передачу кода программы по сети. В Java апплет — это объект некоторого класса, наследующего свойства общего класса Applet. Большая часть программного обеспечения храниться локально в браузере, что существенно сокращает объем пересылаемого по сети кода.

Чтобы воспользоваться наследованием Java, надо объявить новый класс, который является расширением другого класса. Новый класс называется *подклассом*, а исходный — *суперклассом*.

Подкласс наследует все свойства родительского класса (суперкласса). Подкласс получает все свойства суперкласса, но, кроме того, может обладать дополнительными свойствами. У класса может быть только один суперкласс. Такая подсхема носит название *единичного наследования*.

Для того чтобы объявить класс `A` подклассом суперкласса `B` используется ключевое слово `extends`.

```
class A extends B {
    ...
}
```

Если суперкласс не указывается явно при объявлении класса с помощью ключевого слова `extends` то роль суперкласса выполняет класс `Object`. Т.е. фактически все классы в Java являются подклассами класса `Object`.

Объявление класса может содержать модификатор `public`.Этот модификатор делает класс доступным всем остальным классам.

Если вы не указываете модификатор доступа `public` объявление считается дружественным. Все объявления такого типа являются общедоступными в пределах своего модуля компиляции (файл исходного кода) или пакета. Классы не могут быть объявлены как `private` или `protected`.

### 2.1.3 Полиморфизм

*Термин полиморфизм* происходит *от двух греческих слов* поли*, что означает много, и* морф*, что означает форма. Следовательно, полиморфизм имеет отношение ко многим формам чего-то.*

Полиморфизм предоставляет возможность производным классам (*подклассам*) менять то, что делают методы, унаследованные ими от базовых классов (*суперклассов*). Другим видом полиморфизма является *перезагрузка методов* (overloading) - использование одного и того же имени для задания общих для класса действий. Выполнение каждого конкретного действия при этом определяется типом данных. Т.е. класс может содержать несколько версий метода, которые отличаются количеством и/или типом аргументов.

Например, для языка С, в котором полиморфизм поддерживается недостаточно, нахождение абсолютной величины числа требует различных функций: `abs(), labs()` и `fabs()`.Эти функции подсчитывают и возвращает абсолютную величину целых, длинных целых и чисел с плавающей точкой соответственно. В Java это все может быть выполнено с помощью одной функции `abs()`.

Пример определения двух методов с одинаковыми именами:

```
...
class Car{
    Tire leftFront=new Tire();
    Tire rightFront=new Tire();
    Tire leftRear=new Tire();
    Tire rightRear=new Tire();
        ...
    void SwapTires (Tire a, Tire b){
        Tire temp;
        temp=a;
```

```
        a=b;
        b=temp;
    }
    public void RotateTires( ){
        SwapTires(this.leftFront, this.rightRear);
        SwapTires(this.leftRear, this.rightFront);
    }
    public void RotateTires( Tire t1, Tire t2, Tire t3, Tire t4,){
        SwapTires(t1, t2);
        SwapTires(t2, t3);
        SwapTores(t3,t4);
    }
}
```

В примере созданы две версии метода RotateTires. Если происходит вызов RotateTires() без параметров, используется первая версия, которая переставляет шины по диагонали. Для вызова второй версии метода следует написать:

```
RotateTires(    this.leftFront,
                this.rightRear,
                this.leftRear,
                this.rightFront);
```

### 2.1.4 Использование super.

При разработке программ может понадобиться добавление в подкласс дополнительных свойств, которых не было в суперклассе. Например, в базовом классе определена сложная подпрограмма, которую нужно перенести в подкласс с небольшими дополнениями. Переопределение метода в подклассе потребует дублирования значительного объема кода. Для того, чтобы не переписывать код метода суперкласса можно использовать ключевое слово `super`. Ключевое слово `super` указывает компилятору, что необходимо вызвать метод суперкласса.

Пример:

```
class Transportation{
    int MaxLoad;
    int PeopleCapacity;
    void Horn () {
        System.out.println("Honk.");
    }
}
class Sedan extends Transportation {
    int StereoWattage;
    SparkPlug p1,p2,p3,p4,p5,p6;
    OilFilter o1;
    AirFilter a1;
    Sedan(int StereoSize,Int doors){
        E
    }
    void TuneUp(){
        p1=new SharkPlug();
        p2=new SharkPlug();
        p3=new SharkPlug();
        p4=new SharkPlug();
        p5=new SharkPlug();
        p6=new SharkPlug();
        o1=new OilFilter();
        a1=new AirFilter();
        //etc.
    }
}
class LowRider extends Sedan{
    AirShocks s1,s2,s3,s4;
```

```
    void TuneUp(){
        super.TuneUp();
        s1=new AirShocks();
        s2=new AirShocks();
        s3=new AirShocks();
        s4=new AirShocks();
    }
}
```

Оператор `new` служит для создания нового объекта. Оператор состоит из трех частей: ключевого слова `new` имени класса объекта и набора аргументов, передаваемых в конструктор.

### 2.1.5 Замещение методов

Новый подкласс Point3D класса Point наследует реализацию метода distance своего суперкласса (пример PointDist.java). Проблема заключается в том, что в классе Point уже определена версия метода distance(mt x, int y), которая возвращает обычное расстояние между точками на плоскости. Мы должны заместить (override) это определение метода новым, пригодным для случая трехмерного пространства. В следующем примере проиллюстрировано и совмещение (overloading), и замещение (overriding) метода distance.

```
class Point
{
  int x, y;
  Point(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
  double distance(int x, int y)
  {
    int dx = this.x - x;
    int dy = this.y - y:
    return Math,sqrt(dx*dx + dy*dy);
  }
  double distance(Point p)
  {
    return distance(p.x, p.y);
  }
}
class  Point3D extends Point
{
  int z;
  Point3D(int x, int y, int z)
  {
    super(x, y);
    this.z = z;
  }
  double distance(int x, int y,  int z)
  {
    int dx = this.x - x;
    int dy = this.y - y;
    int dz = this.z -  z;
    return Math.sqrt(dx*dx + dy*dy + dz*dz);
  }
  double distance(Point3D other)
  {
    return distance(other.x, other.y, other.z);
  }
  double distance(int x, int y)
  {
    double dx = (this.x / z) - x;
```

```
    double dy = (this.y / z) - y;
    return Math.sqrt(dx*dx + dy*dy);
  }
}
class Point3DDist
{
  public static void main(String args[])
  {
    Point3D p1 = new Point3D(30, 40, 10);
    Point3D p2 = new Point3D(0, 0, 0);
    Point p = new Point(4, 6);
    System.out.println("p1 = " + p1.x +  ", " + p1.y + ", " + p1.z);
    System.out.println("p2 = " + p2.x +  ", " + p2.y + ", " + p2.z);
    System.out.println("p = " + p.x + ", " + p.y);
    System.out.println("p1.distance(p2) = " + p1.distance(p2));
    System.out.println("p1.distance(4, 6) = " + p1.distance(4, 6));
    System.out.println("p1.distance(p) = " + p1.distance(p));
  }
}

Ниже приводится результат работы этой программы:
C:\> Java Point3DDist

p1 = 30, 40, 10
p2 = 0, 0, 0
p = 4, 6
p1.distance(p2) = 50.9902
p1.distance(4, 6) = 2.23607
p1.distance(p) = 2.23607
```

Обратите внимание - мы получили ожидаемое расстояние между трехмерными точками и между парой двумерных точек. В примере используется механизм, который называется динамическим назначением методов (dynamic method dispatch).

### 2.1.6 Динамическое назначение методов

Давайте в качестве примера рассмотрим два класса, у которых имеют простое родство подкласс / суперкласс, причем единственный метод суперкласса замещен в подклассе.

```
class A
{
  void callme()
  {
    System.out.println("Inside A's callrne method");
  }
}
class B extends A
  {
  void callme()
  {
    System.out.println("Inside B's callme method");
  }
}
class Dispatch
{
  public static void main(String args[])
  {
    A a = new B();
    a.callme();
  }
}
```

Обратите внимание - внутри метода main мы объявили переменную a класса A, а проинициализировали ее ссылкой на объект класса B. В следующей строке мы вызвали метод callme. При этом транслятор проверил наличие метода callme у класса A, а исполняющая система, увидев, что на самом деле в переменной хранится представитель класса B, вызвала не метод класса A, а callme класса B. Ниже приведен результат работы этой программы:

```
C:\> Java

Inside B's calime method

СОВЕТ
```

Программистам Delphi / C++ следует отметить, что все Java по умолчанию являются виртуальными функциями (ключевое слово virtual).

Рассмотренная форма динамического полиморфизма времени выполнения представляет собой один из наиболее мощных механизмов объектно-ориентированного программирования, позволяющих писать надежный, многократно используемый код.

## 2.2 Конструкторы объектов

Для каждого создаваемого вами объекта требуется какого-то вида инициализация. Обычно код инициализации помещают в тело функции-конструктора класса. Функция конструктор является методом класса, который имеет то же имя, что и сам класс (имя функции-конструктора совпадает с именем класса). Например, если вы используете класс Car, функция конструктор будет называться тоже Car. Конструктор класса вызывается каждый раз при создании объекта этого класса. Функция-конструктор не может возвращать значений и при этом в ее определении не пишется ключевое слово `void`. Тип возвращаемого этой функцией значения просто не нужно указывать.

В рассмотренном выше примере для создания объектов использовался обычный конструктор, который создается неявно при определении класса (`Tire()`). Этот конструктор выделяет в памяти блок требуемого размера и инициализирует начальные значения переменных.

### 2.2.1 Ключевое слово this

Ключевое слово `this` интерпретатор Java рассматривает как указатель на текущий объект. Оно особенно полезно при связывании объектов.

Пример использования ключевого слова `this`:

```
class Speaker{
    String massage="";
    String quitMassage="";
    int boredomFactor=3;
    void yourTurn(Speaker whoIsNext){
        if(boredomFactor--<0)
        System.out.println(quitMassage);
        else{
            System.out.println(massage);
             whoIsNext.yourTurn(this);
        }
    }
}
public class Talk{
    public static void main(String str[]){
        Speaker Bob, Tom;
        Tom=new Speaker();
        Tom.massage="Bob, you do it";
        Tom.boredomFactor=4;
        Tom.quitMassage="Fine";
```

```
        Bob=new Speaker();
        Bob.massage="Tom, you do it";
        Bob.quitMassage="Ok";
        Bob.yourTurn(Tom);
    }
}
```

### 2.3 Абстрактные классы и интерфейсы

Абстрактным классом в Java называется класс, который содержит абстрактные методы — методы, пред объявлением которых указывается ключевое слово `abstract` и определение которых дается только в подклассах класса.

```
public abstract class Road{
    abstract void DrawRoadt();
    abstract int FindMedian(int []x) ;
    //...Other methods.
}
public class Superhighway extends Road{
    void DrawRoad(){
//...
}
    int FindMedian(int []x){
//...
}
}
```

В классе Road определение методов завершается символом "точка с запятой" и не содержит не одной строки кода. Определение этих методов должно быть дано позже, в одном из подклассов Road. При выполнении программы компилятор Java не допустит создания объектов класса Road, пока не станут доступны все объявленные методы, и до этого момента нельзя использовать операторы типа Roud roud66=new Road(). Перед этим должен быть задать подкласс, содержащий код для всех абстрактных методов в соответствии с иерархией наследования.

Можно определять объекты абстрактного класса, но нельзя инициализировать их значения оператором `new`. Следует обратиться к конструктору подкласса, в котором окончательно определяются все части абстрактного метода.

#### 2.3.1 Интерфейсы. Слабое множественное наследование.

Некоторые объектно-ориентированные языки программирования (например, C++) допускают наследование свойств нескольких различных классов - *множественное наследование*. Это дает большую свободу при разработке программы, но на этапах компиляции и отладки свойства класса могут перекрываться, и возникают ошибки, которые трудно обнаружить. Java допускает наследование свойств только одного класса и имеет дополнительный механизм для наследования свойств других классов, называемый "выполняемым наследованием" (implementation).

Множественное наследование в Java реализуется с использованием интерфейсов. Интерфейс (Interface class) является по существу абстрактным классом. **Все** методы интерфейса являются абстрактными, при этом не требуется использовать ключевое слово `abstract`.

Определение интерфейса начинается с зарезервированного слова `interface`:

```
interface InterfaceName{
//методы и переменные
}
```

Переменные интерфейса по умолчанию являются `static final`.

```
interface Noisy{
    void makeNoise ();
}
class Pet{
            String name;
            int Birthdate;
            // In days since 1/1/1980. int LastDateFed;
            // In days since 1/1/1980 int FeedingInterval;
            int myX, myY, myZ;
            //Location.
            void Move (int x, int y, int z){
                        // Code to move location.
                        myX+=x;
                        myY+=Y;
                        myZ+=Z;
            }
}
class Dog extends Pet implements Noisy{
            void makeNoise (){
                        System.out.printin( "Bark.");
            }
}
class Cat extends Pet implements Noisy{
            void makeNoise (){
                        System.out.printin( "Meow.");
                        }
}
class Fish extends Pet{
            void Move (int x, int y, int z){
                //Assume water level is at z=0. So myZ<=0 to live.
                myX+=x;
                myY+=Y;
                myZ+=Z;
                if (myZ>0)
                        myZ = 0;
        }
}
```

В данной программе создается интерфейс Noisy, суперкласс Pet и три подкласса:Dog, Cat, Fish, наследующие свойства Pet. Если в подклассе нет метода move(), то используется его версия из суперкласса Pet. В подклассе Fish имеется своя версия этого метода.

Два из трех подклассов обеспечивают интерфейс Noisy, предоставляя метод makeNoise, который не имеет параметров и не возвращает данных. Если бы подклассы не содержали описания методов, то при компиляции была бы зафиксирована ошибка.

Теперь для всех классов реализующих интерфейс Noisy можно быть уверенным, что реализация метода makeNoisy существует.

### 2.3.2 Различия интерфейсов и абстрактных классов.

Интерфейсы и абстрактные классы похожи, но у них есть отличия.

Абстрактный класс помимо абстрактных методов может иметь и неабстрактные методы. В интерфейсе это невозможно. Подкласс может наследовать свойства нескольких интерфейсов, но не несколько абстрактных классов.

В интерфейсах `extends` можно использовать для того чтобы создать один интерфейс как подкласс другого:

```
interface MathFact{
    double pi=3.141592;
    double e=2.718;
}
```

```
interface ChemistryFacts extends MathFacts{
    double AvogadrosNum=6.023e23;
}
```

Если где-то в программе обеспечивается интерфейс `ChemistryFacts`, то будут доступны переменные `pi,e` и `AvogadrosNum`. В примере в интерфейсе определяются переменные, которые являются константами. Это обычная практика. Большинство программистов располагают определения глобальных констант в одном месте, там, где их легко найти.

Класс может одновременно обеспечить несколько различных интерфейсов:

```
public class Foo extends Bar implements Moo, Boo, Hoo {
//E
}
```

Методы и переменные интерфейса по умолчанию имеют уровень доступа `public` и явного модификатора `public` не требуют. Модификаторы же `private` и `protected` в интерфейсах использовать нельзя

Библиотека классов Java заполнена интерфейсами, которые содержат шаблоны и протоколы для достижения самых разных целей. Вы можете создавать собственные интерфейсы, но в большинстве случаев написания приложений вы, скорее всего, будете импортировать и реализовывать существующие объявления интерфейсов. Вы можете использовать интерфейсы, реализованные различными классами в библиотеке Java.

## 2.4 Пакеты

Пакет (package) - это некий контейнер, который используется для того, чтобы изолировать имена классов. Например, вы можете создать класс List, заключить его в пакет и не думать после этого о возможных конфликтах, которые могли бы возникнуть, если бы кто-нибудь еще создал класс с именем List.

Все идентификаторы, которые мы до сих пор использовали в наших примерах, располагались в одном и том же пространстве имен (name space). Это означает, что нам во избежание конфликтных ситуаций приходилось заботиться о том, чтобы у каждого класса было свое уникальное имя. Пакеты - это механизм, который служит как для работы с пространством имен, так и для ограничения видимости. У каждого файла .java есть 4 одинаковых внутренних части, из которых мы до сих пор в наших примерах использовали только одну. Ниже приведена общая форма исходного файла Java.

**одиночный оператор package (необязателен)**
**любое количество операторов import (необязательны)**
**одиночное объявление открытого (public) класса**
**любое количество закрытых (private) классов пакета(необязательны)**

### 2.4.1 Оператор package

Первое, что может появиться в исходном файле Java - это оператор package, который сообщает транслятору, в каком пакете должны определяться содержащиеся в данном файле классы. Пакеты задают набор раздельных пространств имен, в которых хранятся имена классов. Если оператор package не указан, классы попадают в безымянное пространство имен, используемое по умолчанию. Если вы объявляете класс, как принадлежащий определенному пакету, например,

```
package java.awt.image;
```

то и исходный код этого класса должен храниться в каталоге java/awt/image.

ЗАМЕЧАНИЕ

Каталог, который транслятор Java будет рассматривать, как корневой для иерархии пакетов, можно задавать с помощью переменной окружения CLASSPATH. С помощью этой переменной можно задать несколько корневых каталогов для иерархии пакетов (через ; как в обычном PATH).

### 2.4.2 Трансляция классов в пакетах

При попытке поместить класс в пакет, вы сразу натолкнетесь на жесткое требование точного совпадения иерархии каталогов с иерархией пакетов. Вы не можете переименовать пакет, не переименовав каталог, в котором хранятся его классы. Эта трудность видна сразу, но есть и менее очевидная проблема.

Представьте себе, что вы написали класс с именем PackTest в пакете test. Вы создаете каталог test, помещаете в этот каталог файл PackTest.Java и транслируете. Пока - все в порядке. Однако при попытке запустить его вы получаете от интерпретатора сообщение "can't find class PackTest" ("Не могу найти класс PackTest"). Ваш новый класс теперь хранится в пакете с именем test, так что теперь надо указывать всю иерархию пакетов, разделяя их имена точками - test.PackTest. Кроме того Вам надо либо подняться на уровень выше в иерархии каталогов и снова набрать "java test.PackTest", либо внести в переменную CLASSPATH каталог, который является вершиной иерархии разрабатываемых вами классов.

### 2.4.3 Оператор import

После оператора package, но до любого определения классов в исходном Java-файле, может присутствовать список операторов import. Пакеты являются хорошим механизмом для отделения классов друг от друга, поэтому все встроенные в Java классы хранятся в пакетах. Общая форма оператора import такова:

```
import пакет1 [.пакет2].(имякласса|*);
```

Здесь пакет1 - имя пакета верхнего уровня, пакет2 - это необязательное имя пакета, вложенного в первый пакет и отделенное точкой. И, наконец, после указания пути в иерархии пакетов, указывается либо имя класса, либо метасимвол звездочка. Звездочка означает, что, если Java-транслятору потребуется какой-либо класс, для которого пакет не указан явно, он должен просмотреть все содержимое пакета со звездочкой вместо имени класса. В приведенном ниже фрагменте кода показаны обе формы использования оператора import:

```
import java.util.Date
import java.io.*;
```

ЗАМЕЧАНИЕ

Но использовать без нужды форму записи оператора import с использованием звездочки не рекомендуется, т.к. это может значительно увеличить время трансляции кода (на скорость работы и размер программы.

Все встроенные в Java классы, которые входят в комплект поставки, хранятся в пакете с именем java. Базовые функции языка хранятся во вложенном пакете java.lang. Весь этот пакет автоматически импортируется транслятором во все программы. Это эквивалентно размещению в начале каждой программы оператора

```
import java.lang.*;
```

Если в двух пакетах, подключаемых с помощью формы оператора import со звездочкой, есть классы с одинаковыми именами, однако вы их не используете, транслятор не отреагирует. А вот при попытке использовать такой класс, вы сразу получите сообщение об ошибке, и вам придется переписать операторы import, чтобы явно указать, класс какого пакета вы имеете ввиду.

```
class MyDate extends Java.util.Date
{
}
```

## 2.5 Ограничение доступа

Java предоставляет несколько уровней защиты, обеспечивающих возможность тонкой настройки области видимости данных и методов. Из-за наличия пакетов Java должна уметь работать еще с четырьмя категориями видимости между элементами классов:

Подклассы в том же пакете.

Не подклассы в том же пакете.

Подклассы в различных пакетах.

Классы, которые не являются подклассами и не входят в тот же пакет.

В языке Java имеется три уровня доступа, определяемых ключевыми словами: private (закрытый), public (открытый) и protected (защищенный), которые употребляются в различных комбинациях. Содержимое ячеек таблицы определяет доступность переменной с данной комбинацией модификаторов (столбец) из указанного места (строка).

| | private | модификатор отсутствует | private protected | protected | public |
|---|---|---|---|---|---|
| тот же класс | да | да | да | да | да |
| подкласс в том же пакете | нет | да | да | да | да |
| независимый класс в том же пакете | нет | да | нет | да | да |
| подкласс в другом пакете | нет | нет | да | да | да |
| независимый класс в другом пакете | нет | нет | да | да | да |

На первый взгляд все это может показаться чрезмерно сложным, но есть несколько правил, которые помогут вам разобраться. Элемент, объявленный public, доступен из любого места. Все, что объявлено private, доступно только внутри класса, и нигде больше. Если у элемента вообще не указан модификатор уровня доступа, то такой элемент будет виден из подклассов и классов того же пакета. Именно такой уровень доступа используется в языке Java по умолчанию. Если же вы хотите, чтобы элемент был доступен извне пакета, но только подклассам того класса, которому он принадлежит, вам нужно объявить такой элемент protected. И наконец, если вы хотите, чтобы элемент был доступен только подклассам, причем независимо от того, находятся ли они в данном пакете или нет - используйте комбинацию private protected.

Ниже приведен довольно длинный пример, в котором представлены все допустимые комбинации модификаторов уровня доступа. В исходном коде первого пакета определяется три класса: Protection, Derived и SamePackage. В первом из этих классов определено пять целых переменных - по одной на каждую из возможных комбинаций уровня доступа. Переменной n приписан уровень доступа по умолчанию, n_pri - уровень private, n_pro - protected, n_pripro - private protected и n_pub - public. Во всех остальных

классах мы пытаемся использовать переменные первого класса. Те строки кода, которые из-за ограничения доступа привели бы к ошибкам при трансляции, закомментированы с помощью однострочных комментариев (//) - перед каждой указано, откуда доступ при такой комбинации модификаторов был бы возможен. Второй класс - Derived - является подклассом класса Protection и расположен в том же пакете p1. Поэтому ему доступны все перечисленные переменные за исключением n_pri. Третий класс, SamePackage, расположен в том же пакете, но при этом не является подклассом Protection. По этой причине для него недоступна не только переменная n_pri, но и n_pripro, уровень доступа которой - private protected.

```
package p1;
public class Protection
{
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  private protected int n_pripro = 4;
  public int n_pub = 5;
  public Protection()
  {
    System.out.println("base constructor");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = "  + n_pro);
    System.out.println("n_pripro = " + n_pripro);
    System.out.println("n_pub = " + n_pub);
  }
}
class Derived extends Protection
{
  Derived()
  {
    System.out.println("derived constructor");
    System.out.println("n = " + n);
    // только в классе
    // System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = "    + n_pro);
    System.out.println("n_pripro = " + n_pripro);
    System.out.println("n_pub = "    + n_pub);
  }
}
class  SamePackage
{
  SamePackage()
  {
    Protection p = new Protection();
    System.out.println("same package constructor");
    System.out.println("n = " + p.n);
    // только в классе
    // System.out.println("n_pri = " + p.n_pri);
    System.out.println("n_pro = " + p.n_pro);
    // только в классе и подклассе
    // System.out.println("n_pripro = " + p.n_pripro):
    System.out.println("n_pub = " + p.n_pub):
  }
}
```

## 3. Дополнительная литература

**Java**


1. **Мультимедийный Обучающий Курс TeachPro Java Для Начинающих**
2. Картузов А.В. **Программирование на языке JAVA**

3.Герберт Шилдт, Джеймс Холмс **Искусство программирования на JAVA**.
4. Патрик Нотон, Герберт Шилдт **Полный справочник по Java**.
5. Вязовик Н.А. **Программирование на JAVA. Курс лекций на intuit.ru**

## 4. Порядок выполнения работы

В соответствии с **вариантом** выполните следующее основное задание:

1. Создайте новый проект в среде Eclipse.
2. Создайте **базовый абстрактный класс**, добавив в него соответствующие **поля** и **методы** (в том числе методы-акцессоры, конструкторы).
3. Отнаследовать **классы-потомки** от базового класса, добавив в них соответствующие **поля** и **методы** (в том числе методы-акцессоры, конструкторы).
4. В разработанных классах должны быть использованы ключевые слова static, super final, this.
5. В базовом классе создать метод **print**, который печатает содержимое полей и обеспечить его замещение в классах потомках.
6. Переопределить методы **finalize()**, который будет выводит сообщение о том,что объект класса уничтожен и метод **toString()**, дающий строковое описание объекта.
7. Создать **интерфейс** и обеспечить реализацию его методов классами –потомками.
8. Поместить созданные классы и интерфейс в **пакет**.
9. Создайте вне пакета **управляющий класс**, предусмотрев в нем точку входа (main), который будет подключать пакет и осуществлять тестирование разработанных классов и интерфейса.
10. Подготовьте отчет о выполнении лабораторной работы:

Для успешной сдачи лабораторной работы необходимо:

- представить преподавателю отлаженный код программы для указанного варианта задания;
- подготовить отчет по работе.

## 5. Порядок оформления отчета

Отчет о выполнении лабораторной работы должен содержать:

1) титульный лист;
2) задание;
3) текст программы;
4) результаты работы программы.

## 6. Варианты заданий

| *№ вар-та* | *Базовый класс* | *Приватные поля базового класса* | *Порожденные классы* | *Интерфейс* |
|---|---|---|---|---|
| 1 | **Студент** | ФИО, курс и идентификационный номер | **студент-дипломник**, имеющий тему диплома,<br>**абитуриент**, имеющий балл по ЕГЭ | **Учащийся**. Определить функции переназначения курса и идентификационного номера |
| 2 | **Животное** | Классификация (вид, семейство), число конечностей, число потомков | **домашнее животное**, имеющее кличку<br>**дикое животное**, имеющее место обитания | **Млекопитающие**. Определить функции переназначения и числа потомков и вывода способа кормления. |
| 3 | **Машина** | марка, число цилиндров, мощность | **грузовик**, имеющий грузоподъемность кузова<br>**легковая**, имеющая объем багажника. | **Автомобиль,** Определить функции переназначения марки и мощности |
| 4 | **Точка** | координаты | **Окружность**, имеющая радиус<br>**эллипс,** имеющий два радиуса | **Геометрическая фигура**. Определить функции, вычисления площади и длины окружности |
| 5 | **Точка** | координаты | **прямоугольник**, имеющий высоту и ширину.<br>**прямоугольный треугольник,** имеющий высоту и основание. | **Геометрическая фигура**. Определить функции, вычисления площади и периметра |
| 6 | **Жидкость** | название, плотность | **спиртной напиток**, имеющий крепость<br>**газированный напиток**, имеющий вкус | **Напитки.** Определить функции переназначения плотности и крепости. |
| 7 | **Человек** | имя (указатель на строку), возраст, вес | **школьник**, имеющий класс<br>**абитуриент**, имеющий ВУЗ | **Ученик**. Определить функции переназначения возраста и класса. |
| 8 | **Окно** | координаты верхнего левого и нижнего правого угла, цвет фона | **окно с меню**, имеющий строку меню<br>**окно редактора**, имеющий строку с текстом | **Оконный интерфейс.** Определить функции переназначения цвета фона и строки меню. |
| 9 | **Человек** | имя (указатель на строку), возраст, вес | **несовершеннолетний**, имеющий номер свидетельства о рождении<br>**совершеннолетний**, имеющий номер паспорта | **Гражданин**. Определить функции переназначения возраста и номера паспорта. |
| 10 | **Живые существа**. | Классификация (вид, семейство), число конечностей, число | **Рыбы**, имеющий кол-во плавников<br>**Птицы**, имеющий размах крыла | **Животные**. Определить функции переназначения числа конечностей и потомков. |

| | | | | |
|---|---|---|---|---|
| | | потомков | | |
| 11 | **Двигатель** | Мощность, объем | **Бензиновый двигатель**, имеющий марку бензина<br>**Гибридный двигатель**, имеющий емкость аккумулятора. | **Ходовая часть.** Определить функции переназначения мощности и марки бензина |
| 12 | **Оборудование для ПК** | Энергопотребление,<br>Габариты (высота ширина, длина) | **Жесткий диск**, имеющий объем<br>**Процессор**, имеющий тактовую частоту | **Устройство.** Определить функции переназначения энергопотребления и габаритов |
| 13 | **Компьютер** | Быстродействие<br>Объем ОЗУ | **Персональный компьютер**, имеющий тип монитора<br>**Ноутбук**, имеющий длительности работы батареи. | **ЭВМ**. Определить функции переназначения объема ОЗУ и быстродействия |
| 14 | | | | |
| 15 | | | | |