Лабораторная работа №3

Классы

Базовым элементом объектно-ориентированного программирования в языке Java являет-ся класс. Для того, чтобы создать класс, достаточно иметь исходный файл, в котором будет присутствовать ключевое слово class, и вслед за ним — допустимый идентификатор и пара фигурных скобок для его тела.

```
class Point {
}
```

Имя исходного файла Java должно соответствовать имени хранящегося в нем класса. Регистр букв важен и в имени класса, и в имени файла.

Класс — это шаблон для создания объекта. Класс определяет структуру объекта и его *методы,* образующие функциональный интерфейс. В процессе выполнения Java-программы система использует определения классов для создания представителей классов. Представители являются реальными *объектами.* Термины «представитель», «экземпляр» и «объект» взаимозаменяемы. Ниже приведена общая форма определения класса.

```
class имя_класса extends имя_суперкласса { type переменная1_объекта:
type переменная2_объекта:
type переменнаяN_объекта:
type имяметода1(список_параметров) { тело метода;
}
type имяметода2(список_параметров) { тело метода;
}
type имя методаM(список_параметров) { тело метода;
}
}
```

Ключевое слово extends указывает на то, что *«имя_класса»* — это подкласс класса *«имя_суперкласса».* Во главе классовой иерархии Java стоит единственный ее встроенный класс — Object. Если вы хотите создать подкласс непосредственно этого класса, ключевое слово extends и следующее за ним имя суперкласса можно опустить — транслятор включит их в ваше определение автоматически.

Переменные представителей (instance variables)

Данные инкапсулируются в класс путем объявления переменных между открывающей и закрывающей фигурными скобками, выделяющими в определении класса его тело. Эти переменные объявляются точно так же, как объявлялись локальные переменные в предыдущих примерах. Единственное отличие состоит в том, что их надо объявлять вне методов, в том числе вне метода main. Ниже приведен фрагмент кода, в котором объявлен класс Point с двумя переменными типа int.

```
class Point { int x, y;
}
```

В качестве типа для переменных объектов можно использовать как любой из простых типов, так и классовые типы.

Оператор new

Оператор new создает экземпляр указанного класса и возвращает ссылку на вновь созданный объект. Ниже приведен пример создания и присваивание переменной p экземпляра класса Point.

```
Point p = new Point();
```

Вы можете создать несколько ссылок на один и тот же объект. Приведенная ниже программа создает два различных объекта класса Point и в каждый из них заносит свои собственные значения. Оператор точка используется для доступа к переменным и методам объекта.

```
class TwoPoints {
public static void main(String args[]) {
Point p1 = new Point();
Point p2 = new Point();
p1.x = 10;
p1.y = 20;
p2.x = 42;
p2.y = 99;
System.out.println("x = " + p1.x + " y = " + p1.y);
System.out.println("x = " + p2.x + " y = " + p2.y);
} }
```

В этом примере снова использовался класс Point, было создано два объекта этого класса, и их переменным x и y присвоены различные значения. Таким образом мы продемонстрировали, что переменные различных объектов независимы. Ниже приведен результат, полученный при выполнении этой программы.

```
x = 10 y = 20
x = 42 y = 99
```

Добавим в класс Point метод main и, тем самым, получим законченную программу.

```
class Point { int x, y;
public static void main(String args[]) {
Point p = new Point();
p.x = 10;
p.y = 20;
System.out.println("x = " + p.x + " y = " + p.y);
} }
```

Объявление методов

Методы - это подпрограммы, присоединенные к конкретным определениям классов. Они описываются внутри определения класса на том же уровне, что и переменные объектов. При объявлении метода задаются тип возвращаемого им результата и список параметров. Общая форма объявления метода такова:

```
тип имя_метода (список формальных параметров) {
тело метода:
}
```

Тип результата, который должен возвращать метод может быть любым, в том числе и типом void - в тех случаях, когда возвращать результат не требуется. Список формальных параметров - это последовательность пар тип-идентификатор, разделенных запятыми. Если у метода параметры отсутствуют, то после имени метода должны стоять пустые круглые скобки.

```
class Point { int x, y;
void init(int a, int b) {
x = a;
У = b;
} }
```

В Java отсутствует возможность передачи параметров *по ссылке* на примитивный тип. В Java все параметры примитивных типов передаются *по значению,* а это означает, что у метода нет доступа к исходной переменной, использованной в качестве параметра. Заметим, что все объекты передаются по ссылке, можно изменять содержимое того объекта, на который ссылается данная переменная. Передать переменные примитивных типов по ссылке можно через обрамляющие классы-оболочки.

Скрытие переменных представителей

В языке Java не допускается использование в одной или во вложенных областях видимости двух локальных переменных с одинаковыми именами. Интересно отметить, что при этом не запрещается объявлять формальные параметры методов, чьи имена совпадают с именами переменных представителей. Давайте рассмотрим в качестве примера иную версию метода init, в которой формальным параметрам даны имена x и y, а для доступа к одноименным переменным текущего объекта используется ссылка **this**.

```
class Point { int x, y;
void init(int x, int y) {
this.x = x;
```

```
this.y = y } }
class TwoPointsInit {
public static void main(String args[]) {
Point p1 = new Point();
Point p2 = new Point();
p1.init(10,20);
p2.init(42,99);
System.out.println(“x = “ + p1.x + “ y =” + p1.y);
System.out.println(“x =” + p2.x +” y =” + p2.y);
} }
```

Так как имена параметров и имена полей класса в данном случае у нас совпадают, то мы используем ключевое слово this. Это ключевое слово представляет ссылку на текущий объект. Поэтому в выражении `this.x = x;` первая часть `this.x` означает, что x - это поле текущего класса, а не название параметра x. Если бы у нас параметры и поля назывались по-разному, то использовать слово `this` было бы необязательно.

Конструкторы

Инициализировать все переменные класса всякий раз, когда создается его очередной представитель — довольно утомительное дело даже в том случае, когда в классе имеются функции, подобные методу init. Для этого в Java предусмотрены специальные методы, называемые конструкторами. Конструктор — это метод класса, который инициализирует новый объект после его создания. Имя конструктора всегда *совпадает* с именем класса, в котором он расположен (также, как и в C++). У конструкторов нет типа возвращаемого результата - никакого, даже void. Заменим метод init из предыдущего примера конструктором.

```
class Point { int x, y;
Point(int x, int y) {
this.x = x;
this.y = y;
} }
class PointCreate {
public static void main(String args[]) {
Point p = new Point(10,20);
System.out.println("x = " + p.x + " y = " + p.y);
} }
```

Программисты на Pascal (Delphi) для обозначения конструктора используют ключевое слово **constructor**.

Совмещение методов

Язык Java позволяет создавать несколько методов с одинаковыми именами, но с разными списками параметров. Такая техника называется *совмещением методов* (**method overloading**). В качестве примера приведена версия класса Point, в которой совмещение методов использовано для определения альтернативного конструктора, который инициализирует координаты x и y значениями по умолчанию (-1).

```
class Point { int x, y;
Point(int x, int y) {
this.x = x;
this.y = y;
}
Point() {
x = -1;
y = -1;
} }
class PointCreateAlt {
public static void main(String args[]) {
Point p = new Point();
System.out.println("x = " + p.x + " y = " + p.y);
} }
```

В этом примере объект класса Point создается не при вызове первого конструктора, как это было раньше, а с помощью второго конструктора без параметров. Вот результат работы этой программы:

```
x = -1 y = -1
```

ЗАМЕЧАНИЕ

Решение о том, какой конструктор нужно вызвать в том или ином случае, принимается в соответствии с количеством и типом параметров, указанных в операторе new. Недопустимо объявлять в классе методы с одинаковыми именами и *сигнатурами*. В сигнатуре метода не учитываются имена формальных параметров учитываются лишь их типы и количество.

this в конструкторах

Очередной вариант класса Point показывает, как, используя this и совмещение методов, можно строить одни конструкторы на основе других.

```
class Point { int x, y;
Point(int x, int y) {
this.x = x;
this.y = y;
}
Point() {
this(-1, -1);
} }
```

В этом примере второй конструктор для завершения инициализации объекта обращается к первому конструктору.

Наследование

Вторым фундаментальным свойством объектно-ориентированного под-хода является наследование (первый – инкапсуляция). Классы-потомки имеют возможность не только создавать свои собственные переменные и методы, но и наследовать переменные и методы классов-предков. Классы-потомки принято называть подклассами. Непосредственного предка данного класса называют его суперклассом. В очередном примере показано, как расширить класс Point таким образом, чтобы включить в него третью координату z.

```
class Point3D extends Point { int z;
Point3D(int x, int y, int z) {
this.x = x;
this.y = y;
this.z = z; }
Point3D() {
this(-1,-1,-1);
} }
```

В этом примере ключевое слово extends используется для того, чтобы сообщить транслятору о намерении создать подкласс класса Point. Как видите, в этом классе не понадобилось объявлять переменные x и y, поскольку Point3D унаследовал их от своего суперкласса Point.

super

В примере с классом Point3D частично повторялся код, уже имевшийся в суперклассе. Вспомните, как во втором конструкторе мы использовали **this** для вызова первого конструктора того же класса. Аналогичным образом ключевое слово **super** позволяет обратиться непосредственно к конструктору суперкласса (в Delphi / C++ для этого используется ключевое слово **inherited**).

```
class Point3D extends Point { int z;
Point3D(int x, int y, int z) {
super(x, y); // Здесь мы вызываем конструктор суперкласса
this.z=z;
public static void main(String args[]) {
Point3D p = new Point3D(10, 20, 30);
System.out.println( " x = " + p.x + " y = " + p.y + " z = " + p.z);
} }
```

Вот результат работы этой программы:

x = 10 y = 20 z = 30

Замещение методов

Новый подкласс Point3D класса Point наследует реализацию метода distance своего суперкласса. 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.

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:

Inside B's callme method

final

Все методы и переменные объектов могут быть замещены по умолчанию. Если же вы хотите объявить, что подклассы не имеют права замещать какие-либо переменные и методы вашего класса, вам нужно объявить их как final (в Delphi / C++ не писать слово **virtual**).

```
final int FILE_NEW = 1;
```

По общепринятому соглашению при выборе имен переменных типа final — используются только символы верхнего регистра (т.е. используются как аналог препроцессорных констант C++). Использование final-методов порой приводит к выигрышу в скорости выполнения кода — поскольку они не могут быть замещены, транслятору ничто не мешает заменять их вызовы *встроенным* (in-line) кодом (байт-код копируется непосредственно в код вызывающего метода).

finalize

В Java существует возможность объявлять методы с именем **finalize.** Методы finalize аналогичны деструкторам в C++ (ключевой знак **~)** и Delphi (ключевое слово **destructor**). Исполняющая среда Java будет вызывать его каждый раз, когда сборщик мусора соберется уничтожить объект этого класса.

static

Иногда требуется создать метод, который можно было бы использовать вне контекста какого-либо объекта его класса. Так же, как в случае main, все, что требуется для создания такого метода — указать при его объявлении модификатор типа **static**. Статические методы могут непосредственно обращаться только к другим статическим методам, в них ни в каком виде не допускается использование ссылок this и super. Переменные также могут иметь тип static, они подобны глобальным переменным, то есть доступны из любого места кода. Внутри статических методов недопустимы ссылки на переменные представителей. Ниже приведен пример класса, у которого есть статические переменные, статический метод и статический блок инициализации.

```
class Static {
static int a = 3;
static int b;
static void method(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("static block initialized");
b = a * 4;
}
public static void main(String args[]) {
method(42);
} }
```

Ниже приведен результат запуска этой программы.

```
x = 42
a = 3
b = 12
```

Модификаторы доступа.

Модификаторы доступа используются для управления доступностью элементов класса из других частей программы (в других классах).

Элемент, объявленный с ключевым словом public (открытый), доступен во всех классах, как в своем пакете, так и во всех классах в любом другом пакете. Этот модификатор можно использовать при объявлении класса. Тогда этот класс доступен для всех классов других пакетов. В каждом файле должен содержаться только один public класс и имя файла должно совпадать с именем такого класса.

Элемент, объявленный с модификатором protected (защищенный) в некоем классе A, доступен во всех классах, являющихся подклассами класса A.

Модификатор private (закрытый) сильнее всего ограничивает доступность элемента. Он его делает невидимым за пределами данного класса. Даже подклассы данного класса не смогут обращаться к элементу, объявленному как private.

Если тип доступа к элементу не указан (доступ по умолчанию), то он доступен из всех классов, входящих в данный пакет.

Пример 1

```
class A          {
    private int n;
    A() { k=2; n=11; }
    int summa() { return k+n; }
    public int getN() { return n; }
    public void setN(int nn) { n=nn; }
}

class TestModificators {
    public static void main(String args[]){
        A obj=new A();              // создание объекта класса A
        // получить значение переменных
        int kk=obj.k;
        System.out.println("k="+kk);
        int nn=obj.getN();
        System.out.println("n="+nn);
        obj.k=10;
        obj.setN(15);
        int s=obj.summa();
        System.out.println("summa="+s);
    }
}
```

Задание 1.
Создать класс MyType с двумя конструкторами: один без параметров (он выводит на экран сообщение «Конструктор без параметров»), а второй – с параметром – целой переменной V, который выводит сообщение «Конструктор с одним параметром V» и присваивает значение параметра V целой переменной myData, которая является открытой переменной класса. Класс MyType имеет открытый метод myMethod(), который выводит на экран сообщение "myMethod!" и значение переменной V. В myData первичном классе (содержащем метод main) NewClass создать два объекта класса MyType на основе двух конструкторов и в каждом из объектов вызвать метод myMethod() .

Задание 2.
Создать класс Person, описывающий отдельного человека. Создать подкласс Person, описывающий сотрудника предприятия - класс Employee. Класс Employee кроме имени и фамилии человека (поля name и surname) определяет дополнительное поле для хранения названия компании, в которой работает сотрудник.
Поля name и surname в базовом классе Person объявлены с модификатором private. В классе Employee переопределить метод displayInfo() базового класса (выводит имя и фамилию человека). В нем с помощью ключевого super обратиться к методу displayInfo() базового класса, и вывести дополнительную информацию, относящуюся только к Employee.