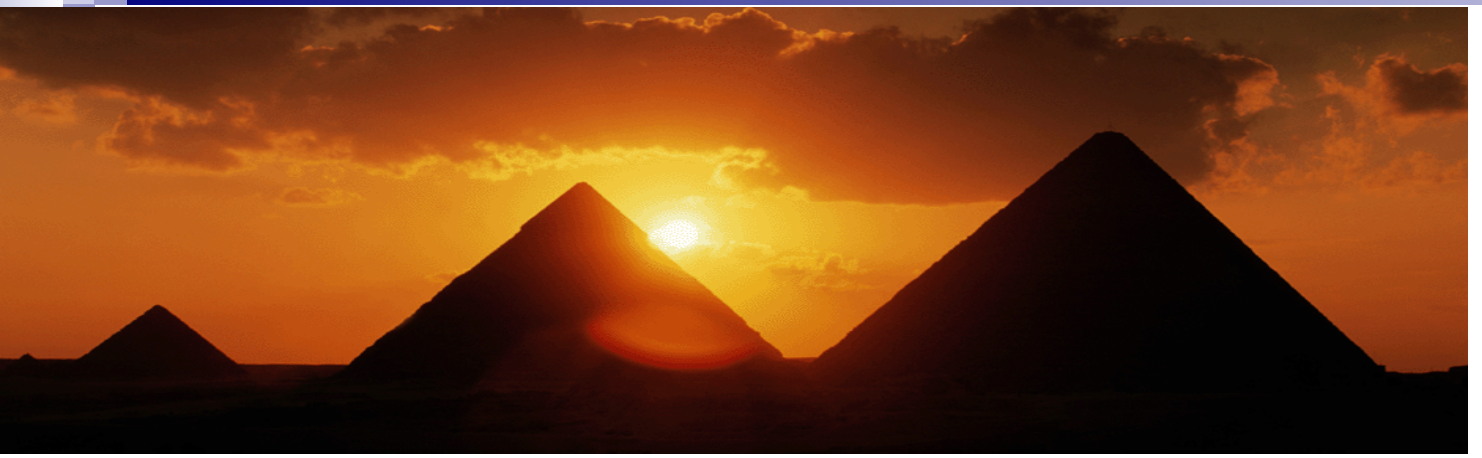
Типы данных и основы ООП

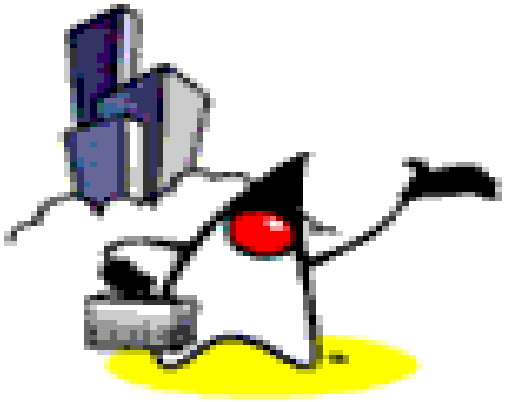
Андрей Дмитриев
andrei-dmitriev@yandex.ru
http://in4mix2006.narod.ru/
2008



Что такое тип данных?

- Спектр значений
- Набор допустимых операций

Программа



- Примитивные и ссылочные типы
- Размеры данных
- Преобразование и приведение типов
- Массивы
- Основы ООП

Примитивные и ссылочные типы

- Язык Java является объектно-ориентированным, но существуют типы данных (простые/примитивные), не являющиеся объектами
  - Фактор производительности
- Простые типы делятся на 4 группы:
  - целые: int, byte, short, long
  - числа с плавающей точкой: float, double
  - символы: char
  - логические: boolean
- Введение в синтаксис языка классов позволяет создавать свои типы, получившие название ссылочных

Примитивные типы

Примитивный тип	Размер(бит)	Минимальное значение	Максимальное значение	Класс-оболочка
boolean	**-**	**-**	**-**	**Boolean**
char	**16**	**Unicode 0**	**U2^16-1**	**Character**
byte	**8**	**-128**	**127**	**Byte**
short	**16**	**-2^15**	**2^15-1**	**Short**
int	**32**	**-2^31**	**2^31-1**	**Integer**
long	**64**	**-2^63**	**2^63-1**	**Long**
float	**32**	**IEEE754**	**IEEE754**	**Float**
double	**64**	**IEEE754**	**IEEE754**	**Double**
void	**-**	**-**	**-**	**Void**

Размер типа данных

- Размер одинаков для всех платформ; за счет этого становится возможной переносимость кода.
- Размер boolean не определен. Указано, что он может принимать значения true или false.

Значения по умолчанию

Неинициализированная явно переменная примитивного типа принимает значение в момент создания:

Примитивный тип	Значение по умолчанию
boolean	**false**
char	**'\u0000' (null)**
byte	**(byte)0**
short	**(short)0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

Использование значения по умолчанию

Значение по умолчанию может быть неразрешенным в конкретной программе. Рекомендуется присвоение значения в момент создания переменной:

```
int i = 0;
```

Классы-оболочки

- Часто переменные примитивных типов должны быть использованы к качестве объектов.
- Для каждого примитивного типа Java в пакете java.lang существует класс-оболочка: Long, Integer, Float,…
- Класс-оболочка – немодифицируемый класс

```
char symbol = 'x';
Character symbol = new Character(c);
```

Числа с высокой точностью

- Предназначены для проведения расчетов без потери точности
- Не имеют примитивных аналогов
- BigInteger – для представления длинных целых чисел
- BigDecimal – для представления больших чисел с плавающей точкой

Приведение типов

- Приведение – это изменение типа (автоматическое или явное)
- Явное приведение позволяет :
  - сделать тип преобразования более точным
  - форсировать преобразование, если оно не может быть выполнено автоматически

Расширяющее преобразование

Результирующий тип имеет больший диапазон значений, чем исходный тип:

```
int x = 200;
long y = (long)x;
long value = (long)200; //необязательно,
т.к. компилятор делает это автоматически
```

Сужающее преобразование

Результирующий тип имеет больший диапазон значений, чем исходный тип.

```
long value = 1000L;
int value2 = (int)value; //обязательно.
Иногда это единственный способ сделать
код компилируемым
```

Приведение ссылочных типов

- Java позволяет проводить приведение от любого примитивного типа к любому примитивному типу, за исключением boolean
- Для классов действуют несколько другие правила приведения

Литералы

Литерал - это конкретное значение, присвоенное или использованное в выражении:

```
//использование при присвоении
int i1 = 0;
char c = '2';
//использование при присвоении
if (c == '1') {
  //использование в бинарной операции
    i1 = i0 + 10000;
}
```

Литералы (cont.)

- Для числовых типов можно указывать систему счисления и тип литерала
- По умолчанию тип литерала – int

```
// восьмеричное представление (ведущий ноль)
int i3 = 0177;
// суффикс для long
long n1 = 200L;
//рекомендуется пользоваться заглавной L
long n2 = 200l;
```

Литералы (cont.)

Для литералов чисел с плавающей точкой можно указывать тип литерала и задавать экспоненциальную запись числа.

```
float f1 = 1;
float f2 = 1F;      // суффикс для float
float f3 = 1f;      // суффикс для float
float f4 = 1e-45f;  // 10 - основание степени
double d1 = 1d;    // суффикс для double
double d2 = 1D;    // суффикс для double
double d3 = 47e47d; //10 - основание степени
```

Символы и кодировки

- Печатные символы можно записать в апострофах: ' a ', ' N ', ' ? '.
- Код любого символа с десятичной кодировкой от 0 до 255 можно задать, записав его не более чем тремя цифрами в восьмеричной системе счисления в апострофах после обратной наклонной черты:
  - ' \123 ' — буква S ,
  - ' \346 ' — буква Ж в кодировке CP1251.
  - Наибольший код ' \377 ' — десятичное число 255.
- Код любого символа в кодировке Unicode набирается в апострофах после обратной наклонной черты и латинской буквы u ровно четырьмя шестнадцатеричными цифрами:
  - ' \u0053 ' — буква S ,
  - ' \u0416 ' — буква Ж .

Символы и кодировки (cont.)

- Прописные русские буквы в кодировке Unicode занимают диапазон:
  - от ' \u0410 ' — заглавная буква А ,
  - до ' \u042F ' — заглавная Я.
- Строчные буквы:
  - от ' \u0430 ' — а ,
  - до ' \044F ' — я .
- В какой бы форме ни записывались символы, компилятор переводит их в Unicode, включая и исходный текст программы.
- Компилятор и исполняющая система Java работают только с кодировкой Unicode.

Управляющие символы

Запись символа	Код ASCII	Значение
' \n '	10	символ перевода строки newline
' \r '	13	символ возврата каретки CR
' \f '	12	символ перевода страницы FF
' \b '	8	символ возврата на шаг BS
' \t '	9	символ горизонтальной табуляции HT
' \\ '	-	обратная наклонная черта
' \" '	-	кавычка
' \' '	-	апостроф

Ссылочные типы данных

```
String s; //создание ссылки
s = “Hello”; //присвоение значения
```

Для большинства классов используется оператор new:

```
String s1 = new String(“World”);
```

Программист имеет возможность создавать свои ссылочные типы.

Константы

Переменные с неизменяемыми значениями называются константами.

```
final int MAXIMUM_SPEED = 250;
```

Операции

Выделяют четыре типа:

- арифметические
- поразрядные
- отношений
- логические

Арифметические операции

- Операнды должны иметь числовой тип
- Можно использовать операции на типе char

+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Остаток от деления
++	Инкремент.
--	Декремент.

Поразрядные (bitwise) операции

- Действуют на индивидуальные биты
- Применяются для числовых операндов

~	Отрицание
&	И
|	ИЛИ
^	Исключающее ИЛИ
>>	Сдвиг вправо
<<	Сдвиг влево

Операции отношений

Определяют отношения, которые один операнд имеет к другому

==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Логические операции

Работают только с типом boolean

&&	И
^	Исключающее ИЛИ
||	ИЛИ
!	Отрицание
==	Равно
!=	Не равно
?:	Троичная условная операция

Размещение сущностей

- Переменная может быть размещена в одном из двух хранилищ:
  - Стек – расположен в памяти VM и имеет поддержку в виде указателя стека
  - Куча (heap) – область памяти VM, в которой хранятся все ссылочные типы
- Переменной простого типа выделяется память в стеке
- Ссылки располагаются в куче

Массивы

Для хранения нескольких однотипных значений используется ссылочный тип – массив:

```
//примитивный тип, размер массива задан явно
int price[] = new int[10];
//неявное  задание размера
int rooms[] = new int[]{1, 2, 3};
//содержит ссылочные переменные
Item [] items = new Item[10];
Item [] undefinedItems = new Item[]{
      new Item(1),
      new Item(2),
      new Item(3)};
```

Доступ к элементам массива

- Доступ осуществляется по индексу
- Размер хранится в немодифицируемом поле массива length

```
for (int i = 0;
     i < undefinedItems.length;
     i++){
    // должен быть представим в виде
    // строки (переопределен метод
    // toString())
   System.out.println(undefinedItems[i]);
}
```

Многомерный массив

- Является массивом массивов.
- Концептуально представляет собой многомерную матрицу

```
int twoDim [][] = new int[4][5];
```

Определение: `int twoD[] [] = new int [4][5]`

Массив массивов

Каждый из массивов может иметь отличную от других длину

```
int twoDim [][] = new int[4][];
twoDim[0] = new int [10];
twoDim[1] = new int [20];
twoDim[2] = new int [30];
twoDim[3] = new int [100];
```

Ошибки времени выполнения

Обращение к несуществующему индексу массива отслеживается виртуально машиной во время исполнения кода:

```
public class Main {
    public static void main(String[] args) {
        int array [] = new int[]{1, 2, 3};
        System.out.println(array[3]);
    }
}
//попытка обратиться к несуществующему
//индексу:
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 3
        at Main.main(Main.java:27)
```

Ошибки времени выполнения (cont.)

Попытка поместить в массив неподходящий элемент пресекается виртуальной машиной:

```
Object x[] = new String[3];
//попытка поместить в массив содержимое
//несоответствующего типа
x[0] = new Integer(0);

Exception in thread "main"
java.lang.ArrayStoreException: java.lang.Integer
        at Main.main(Main.java:22)
```

Строки

- Класс java.lang.String представляет собой хранилище символов и функциональность для работы с ними
- Строка имеет фиксированную длину и не завершается специальным символом
- Возможности:
  - обращение к символу по его номеру,
  - поиск,
  - выделение подстроки,
  - изменение регистра,
  - и т.п.
- Объект класса String – неизменяем

Основы ООП

- *Инкапсуляция* (сокрытие данных) - объединение в одной сущности данных и методов работы с ними.
- *Наследование* – возможность класса-наследника приобретать признаки класса-предка.
- *Полиморфизм* – способность наследников по-другому реализовывать возможности предков. Объект способен проявлять признаки своего предка.

Инкапсуляция

- Состояние объекта определяется значениями его полей
- Сокрытие данных осуществляется с помощью установки модификаторов, влияющих на видимость членов класса
- В языке Java используется несколько уровней сокрытия

Модификаторы доступа полей

- Нет модификатора – данное поле доступно отовсюду в данном пакете
- public – поле доступно отовсюду
- private - поле доступно только в данном классе
- protected - поле доступно в данном классе и во всех его наследниках

Полиморфизм

Экземпляр класса может выступать как экземпляр любого класса-предка данного класса

```
class Animal {
   void talk(){…};
}
class Dog extends Animal {
    void talk()
{ System.out.println("Woof!"); }
}
class Cat extends Animal {
    void talk()
{ System.out.println("Meow!"); }
}
```

Полиморфизм (cont.)

Экземпляр класса Dog или Cat одновременно является экземпляром класса Animal:

```
Animal myDog = new Dog();
Animal myCat = new Cat();
```

Отношения между классами

- Ассоциация
- Наследование
- Агрегация
- Использование
- Инстанцирование
- Метакласс

Ассоциация

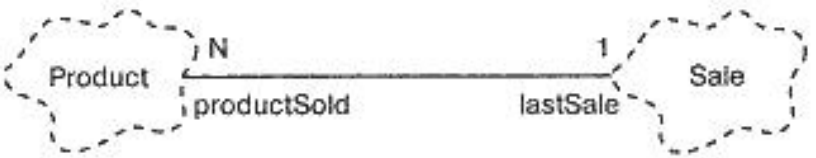
- Ассоциация – это смысловая связь.
- Мощность ассоциации (количество участников):
  - "один-к-одному"
  - "один-ко-многим"
  - "многие-ко-многим"

Пример: товары и продажи

- Класс **Product -** это то, что мы продали в некоторой сделке, а класс **Sale -** сама сделка, в которой продано несколько товаров.
- Данная ассоциация работает в обе стороны.
- Задавшись товаром, можно выйти на сделку, в которой он был продан, а пойдя от сделки, найти, что было продано.

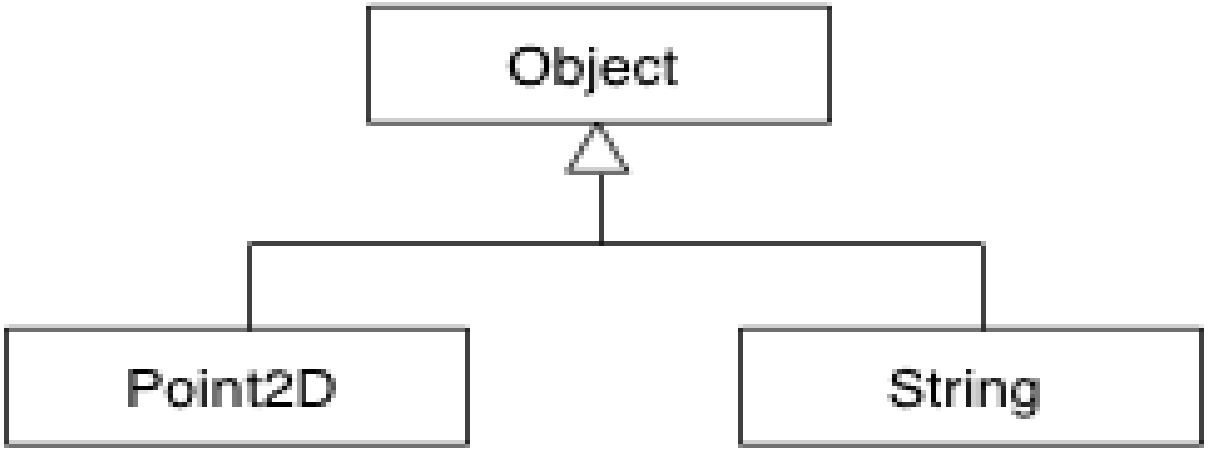
```
class Product {

   protected Sale lastSale;
}
class Sale {
   protected Product productSold;
}
```



Наследование

Наследование - это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других классов (множественное наследование)



Ромбическое наследование

Класс не может приобретать свойства нескольких других классов через механизм наследования:

```
abstract class Animal {
    abstract void talk();
}
class Frog extends Animal {
     void jump() { //прыгать}
}
class Bird extends Animal {
    void fly() {//летать }
}
//ошибка, разрешено наследоваться только от одного класса.
class FlyingFrog extends Frog, Bird{…}
```

Ромбическое наследование (cont.)

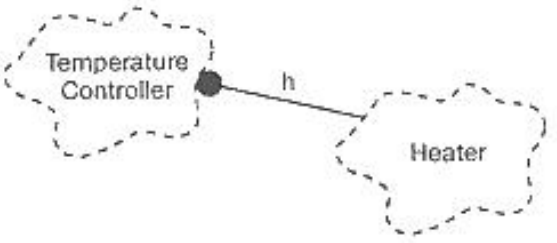
Для передачи свойств сразу от нескольких сущностей был введен новый тип данных – интерфейс. Использую дополнительную сущность, можно добиться того же результата как и при множественном наследовании:

```
interface Flying {
    void fly();
}
class Frog {
     void jump() { //прыгать}
}
class Bird implements Flying{
    void fly() {//летать }
}
class FlyingFrog extends Frog implements Flying{…}
//разрешено
```

Агрегация

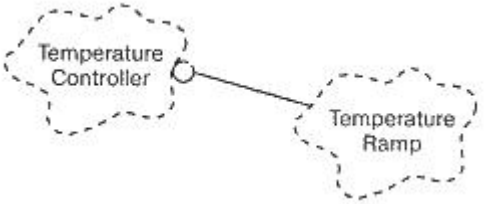
- Физическое включение
  - Включение по значению (объект класса **Heater** не существует отдельно от объемлющего экземпляра класса **TemperatureController)**.
  - Включение по ссылке (объекты живут отдельно друг от друга: мы можем создавать и уничтожать экземпляры классов независимо).
- Агрегация без физического включения
  - Например, акционер владеет акциями, но они не являются его физической частью.

```
class TemperatureController {
   public TemperatureController(){};
   private Heater header;
}
```



Использование

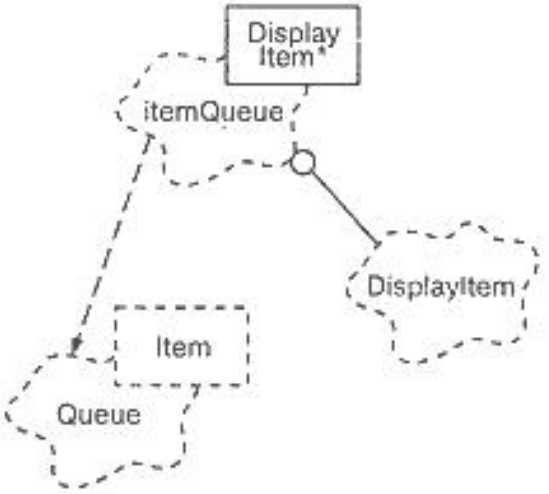
- Класс TemperatureRamp упомянут как часть сигнатуры метода process; это дает нам основания сказать, что класс TemperatureController пользуется услугами класса TemperatureRamp.
- Клиенты и серверы. Отношение использования между классами соответствует равноправной связи между их экземплярами.
- Это то, во что превращается ассоциация, если оказывается, что одна из ее сторон (клиент) пользуется услугами другой (сервера)




```
class TemperatureController {
   public TemperatureController() {}
   public MeasureResult process(TemperatureRamp ramp){
  //Измеряем температуру и возвращаем результат
};
```

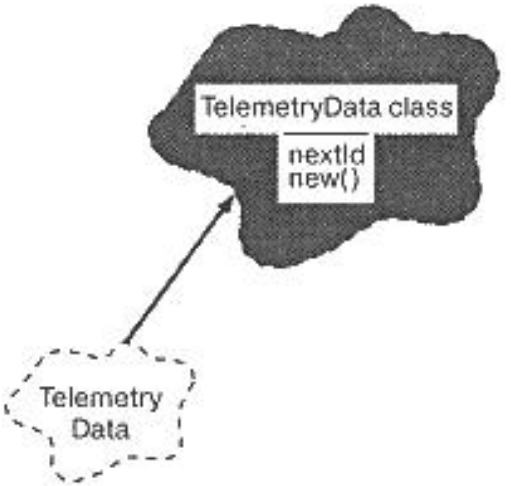
Инстанцирование

- Объекты **intQueue** и **itemQueue -** это экземпляры различных классов.
- Тем не менее, они получены из одного параметризованного класса **Que**
- Параметризованный класс не мо
экземпляров, пока он не будет и




```
Queue<int> intQueue;
Queue<DisplayItem> itemQueue;
```

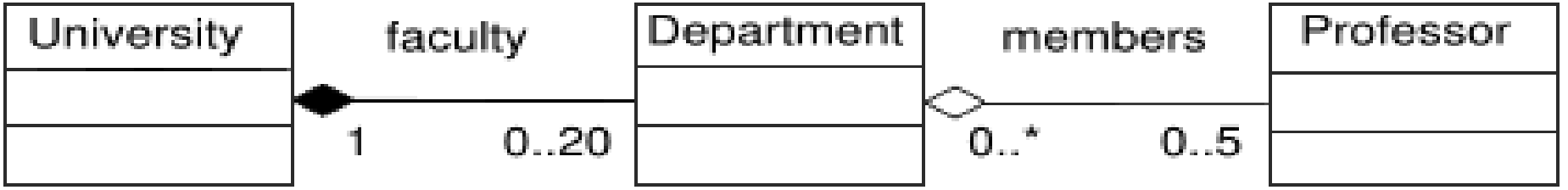
Метакласс



- Любой объект является экземпляром какого-либо класса
- Что будет, если мы попробуем и с самими классами обращаться как с объектами?
- Для этого нам надо ответить на вопрос, что же такое класс класса?
- Ответ - это метакласс. Иными словами, метакласс - это класс, экземпляры которого суть классы
- Метаклассы венчают объектную модель в чисто объектно-ориентированных языках

Почему композиция?

Композиция объектов в некоторых случаях позволяет незначительными усилиями добиться результата, аналогичного тому, что можно получить при наследовании



Композиция (cont.)

Предположим, что класс Fax выполняет некоторое действие неправильно (метод Fax.doBad())

```
class Fax {
    void send(){
        doGood();
        doBad();
    }
    void doBad(){…}
}
```

Композиция (cont.)

Если данный метод doBad()реализован в классе Fax, то можно сделать так

```
class NewFax extends Fax {
 void send(){
       doGood();
       doBad();
   }
   void doBad(){
     //делаем то же самое, но правильно;
   }
}
```

Композиция (cont.)

Того же можно добиться, если поместить класс Fax внутрь другого класса

```
class BetterFax {
  private Fax fax = …;
  void send(){
     fax.doGood();
     doBad_Fixed();
  }
  …
}
```

Правда ли что…

- Нет специальных обозначений для бинарного представления литералов, например :

  int t = 1010101B;
- Литерал “47e47d” имеет значение 47*10^47?
- char может хранить символы в Unicode формате?
- Композиция объектов лучше, чем наследование?

Ссылки

- Ken Arnold, James Gosling, David Holmes, The Java Programming Language, Fourth Edition.
- Спецификация языка Java:
  - http://java.sun.com/docs/books/jls/

Q&A

Спасибо!

Типы данных и основы ООП

Андрей Дмитриев
andrei-dmitriev@yandex.ru
http://in4mix2006.narod.ru/
2008