Основы синтаксиса языка Java

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

Программа

- Зарезервированные слова
- Операторы
- Рекомендации по оформлению
- Время жизни и области видимости
- Модификаторы доступа
- Вызовы методов
- Конструктор
- Статический блок
- Повторное использование имен
- Внутренние классы
- Интерфейсы
- Абстрактные классы
- Константы

Основы: программа DragonWorld

```
package heroes;

public class HelloDragonWorld {
    public static void main(String []args){
        System.out.println("Hello DragonWorld!");
     }
}
```

Основы: пакет

Пакет – это совокупность классов и подпакетов, объединенных общим именем

```
package mydragons;

public class Dragon{//реализация }

  //использование

import mydragons.Dragon;

Dragon red = new Dragon(Dragon.RED);

Dragon black = new mydragons.Dragon(Dragon.BLACK);
```

Основы: класс

- Класс – это базовая сущность ООП, обладающая определенными свойствами
- Любая программа на языке Java представляет собой класс

```
package animals.slowanimals;

public class Reptile {
    public void eat(Bird b){
        b.eaten = true;
    }
}
```

Основы: поле

- Поле – это именованное свойство класса или объекта
- Поле может относиться как к каждому объекту, так и к классу в целом

```
package animals.slowanimals;

public class Reptile {
    private int length;
}
```

Основы: объект

- Объект – это переменная, типом которой является соответствующий класс
- Объект также называют экземпляром класса

```
package animals.slowanimals;
//класс:
public class Reptile {
    private int length;
}
…
//объект:
Reptile gecko = new Reptile();
```

Основы: метод

- Метод – это программная функция, относящаяся к определенному объекту или классу
- Области, откуда метод может быть доступен, определяются модификаторами метода

```
package animals.slowanimals;

public class Reptile {
    private int length;
    public void eat(Bird b){
        length++;
    }
}
```

Основы: наследование

Класс может заимствовать методы другого класса. Язык Java поддерживает операцию наследования

```
// наследование производится с помощью
// ключевого слова extends
public class Dragon extends Reptile {
    //внутреннее поле класса
    private String magic = “fire”;
    public String getMagic(){
        //возврат результата
         return magic;
    }
}
```

Область видимости переменной

Каждая переменная может быть использована только в ее области видимости

```
int i = 10;
//область видимости ограничится
//ближайшими фигурными скобками.
System.out.println(i);
for (int j = 0; j < 100; j++){
// j видна внутри блока for
    System.out.println(j);
}
//i видна после блока for
System.out.println(i);
//j: переменная вне  области видимости - ошибка
System.out.println(j);
```

Модификаторы

- Модификаторы доступа являются реализацией принципа инкапсуляции в языке Java
- Изменяя модификаторы, можно контролировать область видимости
  - полей
  - методов
  - классов

Модификаторы полей

private	Поле не может быть использовано нигде кроме данного класса или его экземпляра
protected	Поле не может быть использовано нигде кроме данного класса и всех его наследников
public	Поле доступно отовсюду
отсутствие модификатора	Поле доступно только из текущего пакета

Модификаторы полей (cont.)

volatile	Значение этого поля будет обновляться каждый раз при обращении к нему. Обычно используется при параллельном исполнении программы
static	Поле принадлежит структуре класса. Одно значение присуще всем экземплярам
final	Поле не может быть изменено
transient	Поле не участвует в процессе сериализации (сохранение состояния объекта во внешнюю память) по умолчанию

Модификаторы методов

private	Метод не может быть использован откуда либо кроме данного класса (его объекта)
protected	Метод не может быть использован ниоткуда кроме данного класса (его объекта) и всех его наследников (их объектов)
public	Метод доступен из любого пакета (публичный API)
отсутствие модификатора	Метод доступен только из данного пакета

Модификаторы методов (cont.)

final	Метод не может быть переопределен в наследнике
static	Метод принадлежит классу
abstract	Метод не имеет реализации
synchronized	Запрещено одновременное выполнение метода на разных потоках
native	Метод имеет реализацию на языке C или C++

Модификаторы классов

отсутствие модификатора	Класс доступен только в текущем пакете
public	Класс доступен из любого пакета (публичный API)
final	Класс не может иметь наследников
abstract	Класс является абстрактным, нельзя создать объект этого класса
static	Допустимо только для вложенных классов. Внутренний класс является статическим членом внешнего класса

Конструктор

- Конструктор – это метод, создающий экземпляр класса
- Не имеет заданного возвращаемого значения
- Имеет то же имя, что и класс

```
public class Dragon{
     private String color = gold;
      public Dragon(String newColor){
            color = newColor;
      }
}
```

Конструктор по умолчанию

В классе всегда присутствует конструктор по умолчанию, если явно конструктор не задан

```
public class Dragon {
      private String color;
      public String getColor () {
         return color;
     }
}
//вызывается конструктор по умолчанию
Dragon dragon = new Dragon();
```

Вызов метода

- Вызов метода – это обращение к члену класса по его имени
- Результат вызова метода – выполненные операторы и возвращаемое значение (если указано)

```
public class Dragon{
    private String name = "Kesha";
    public String getName (){
        return "I am"+name;
    }
}
Dragon dragon = new Dragon(); //конструктор
System.out.println(dragon.getName()); //метод
```

Возвращаемое значение

- Метод может возвращать одно значение (может быть простой тип, ссылочный тип, массив) в точку вызова
- Если метод не возвращает никакого значения, то его возвращаемый тип – void

```
public void dumpValue (){
    System.out.println("value = " + value);
}
```

Виртуальный метод

- Виртуальным называется метод, который замещает собой соответствующий метод предка, если метод вызывается для потомка
  - Метод класса может быть переопределен в наследнике
  - Конкретная реализация метода для вызова будет определяться во время исполнения
  - Поиск того метода, который следует вызывать называется диспатчем (dispatch)
- Виртуальность – связывание класса с его методами на этапе создания объекта

Повторное использование имен (переопределение)

Методы предка и наследника могут быть одноименными

```
class Reptile{
    public void move(){/*ползти*/ }
}

class Dragon extends Reptile{
    public void move(){/*лететь*/}
}
Dragon d = new Dragon();
d.move(); //обращение к методу экземпляра Dragon
Reptile r = new Reptile();
r.move(); //обращение к методу экземпляра Reptile
```

Повторное использование имен (сокрытие)

Статические методы принадлежат классу

```
class Reptile{
    public static void move(){}
}
class Dragon extends Reptile{
    public static void move(){}
}
Dragon d = new Dragon();
Reptile r = new Reptile();
d.move(); //обращение к методу Dragon
r.move(); //обращение к методу Reptile

//Рекомендуется использовать вызовы класса:
Reptile.move();
Dragon.move();

Reptile r1 = new Dragon();
//что выведет r1.move();?
```

Повторное использование (перегрузка)

Методы выполняют схожую функцию над разными типами данных

```
class HungryDragon {
    public void eat(int foodWeight){…}
    public void eat(String foodWeight){
        //разбор строки на значимое целое число
        try {
            int i = Integer.parseInt(foodWeight);
        } catch (NumberFormatException e){…}
        … //что бы вы поставили сюда?
    }
}
HungryDragon hd = new HungryDragon();
hd.eat(10);
hd.eat("10");
```

Повторное использование имен (затенение)

- Локальная переменная делает одноименную глобальную переменную невидимой в локальной области
- Так делать не рекомендуется

```
public class Dragon {
    static String type = "Just Dragon";
    public static void main(String [] s){
        String type = "Black Dragon";
        //выведет "Black Dragon"
        System.out.println(type);
    }
}
```

Повторное использование имен (перекрытие)

- Использование имен существующих методов и полей вносит в программу путаницу
- Использование существующих имен классов недопустимо

```
public class BadExample {
    static String System;
    public static void main(String [] s){
        System.out.println(“A string");
    }
}
```

Результат работы программы:

BadExample.java:4: cannot resolve symbol symbol : variable out

location: class java.lang.String

System.out.println("A string");

Правила оформления

- Основная цель хорошего оформления - программа должна выглядеть понятной
- Среди требований можно отметить:
  - Отступы
  - Мнемоничность имен
  - Разделение операторов
  - Использование фигурных скобок
  - И т.д.
- Рекомендации от компании Sun Microsystems:
  - http://java.sun.com/docs/codeconv/

Пример оформления

```
/*
 * Discipline.java
 * Дисциплина - курс с определенным названием,
  лектором и годом (семестром).
 * Created on 7 Апрель 2007 г., 15:12
*/

package disc;

import java.util.*;
/**
 * @author Mark
 */
public class Discipline {
```

Пример оформления (cont.)

```
    private int id;                          //уникальный ID
    private String name = "?";               //название курса
    private String lector = "?";             //лектор, ведущий курс
    private String year = "?";               //год (семестр), когда
                                                        читался
    private String annotation;                //краткая аннотация

    private int numberOfLectures;              //количество лекций
    private int numberOfPractices;             //количество практик
    private int numberOfTeorWorks;             //количество
                                          теоретических работ
                                  //список заданий на практики
    private ArrayList pwList = new ArrayList();
```

Пример оформления (cont.)

```
/** Creates a new instance of Course */
    public Discipline() {
    }

    public Discipline(String name, String lector,
                      String year) {
        this.name = name;
        this.lector = lector;
        this.year = year;
    }

    //далее методы get и set
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
```

Пример оформления (cont.)

```
/** Получаем дисциплину из списка по ее ID; если в
   * списке нет, возвращаем "пустую"*/
   public Discipline disciplineByID(int id){
       for (int i=0; i < this.getSize(); i++) {
            if (get(i).getId() == id) {
                 return get(i);
            }
       }

       Discipline nullDisc = new Discipline();
       return nullDisc;
   }
```

Пример неправильного оформления

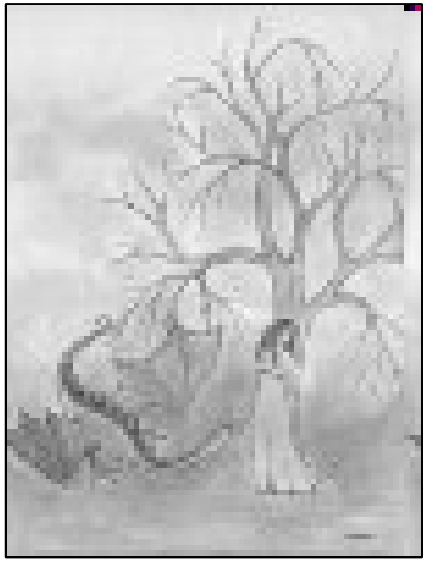
```
private void intvlTo(String v, int t) {
    if (v==null)
      throw new NullPointerException("Er");
array.insertElementAt(v, t);
      ChoicePeer peer = (ChoicePeer)this.peer;
      if (peer != null) {
          peer.addItem(v);
      }
      if (selectedIndex < 0
      // no selection or selection shifted up
  || selectedIndex >= t) {
          select(0); }
```

Пример неправильного оформления (cont.)

```
private void removeNoInvalidate(int position){
    pItems.removeElementAt(position);
    ChoicePeer peer = (ChoicePeer)this.peer;
    /*Similar line in method getValue() does the
same. */
    if (pItems.size() == 0){
    selectedIndex = -1;}
    if ((mask & ITEM_EVENT_MASK)!= 0 ||
        itemListener != null) {
        return true;
    }
```

Передача параметров

- Метод класса может получать до 255 параметров
- Фактический параметр считается локальной переменной метода



```
class Dragon {
    public void eat(Object obj){}
    public void fly(String direction){}
}

Dragon d = new Dragon();
d.eat(new Girl());
d.fly(Direction.WEST);
```

Поле this

- Каждый объект имеет ссылку на самого себя
- Может использоваться для формирования ссылки на перегруженный конструктор и на поля объекта

```
class Dragon {
    private int weight;
    public Dragon(int weight){
        //затенение
        this.weight = weight;
    }
}
```

Поле super

- Каждый объект имеет ссылку на объект-предок
- Позволяет организовать восходящие вызовы конструкторов

```
class NamedDragon extends Dragon {
    private String name;
    public NamedDragon(int weight, int name){
//обращение к конструктору класса Dragon,
// который умеет инициализировать объект
// его весом
        super(weight);
        this.name = name;
    }
}
```

Поле super (cont.)

С помощью данного поля можно вызывать методы предка

```
class Dragon extends Reptile{
    int flyingSpeed;
    public void attack(Object obj){
  //обращение к предку за выполнением
  //базовых действий
      super.attack(obj);
      burn(obj); //метод класса Dragon
    }
   // Dragon умеет атаковать, как Reptile,
   // а заодно сжигать жертву
   public void burn(Object obj){ //сжечь объект
   }
}
```

Статический блок

Класс может иметь в себе участок кода, выполняющийся при инициализации класса

```
class Dragon {
   //статический блок выполняется до того, как создался
     первый экземпляр класса
     static {
       System.out.println("Dragons are alive!");
     }
     //конструктор может не выполниться ни разу,
     //в то время как статический инициализатор
     //выполнится при загрузке
     public Dragon(){
         System.out.println("New dragon has born");
     }
}
```

Порядок инициализации

Инициализация членов класса и выполнение статического инициализатора происходит в порядке их описания в классе

```
class Dragon {
    static int dragonCount = 10;
    static {
   //ошибка - переменная dragonEnemy еще не
     проинициализирована
       System.out.println(dragonEnemy);
   //ОК - переменная dragonCount уже проинициализирована
       System.out.println(dragonCount);
     }
     static String dragonEnemy = "Phoenix";
}
```

Внутренние классы

- Класс, описанный внутри другого класса называется внутренним
- Может быть использован для
  - удобства
  - ограничения области его видимости
  - и для сокрытия реализации

http://java.sun.com/docs/books/tutorial/java/javaOO/innerclasses.html

Внутренние классы (cont.)

Тело внутреннего класса содержится в теле другого класса

```
public class DragonFlying {
    class Speed { //внутренний класс
        private int i = 100;
        public int value(){ return i; }
    }
    class Destination {//внутренний класс
        private String label;
        Destination(String whereTo){
            label = whereTo;
        }
        String readLabel(){ return label;}
    }
}
```

Внутренние классы (cont.)

Внешний класс имеет доступ ко внутренним и может создавать его экземпляры

```
public void ship(String dest){
    Speed s = new Speed();
    Destination d = new Destination(dest);
    System.out.println(d.readLabel());
}
public static void main(String[] args) {
    DragonFlying fd = new DragonFlying();
    f1.ship("Tanzania");
}
```

Внутренние классы (cont.)

Внутренний класс может быть доступен и снаружи

```
//может быть возвращен внутренний класс
  public Destination to(String s) {
    return new  Destination(s);
  }
  public Speed speed() {
    return new Speed();
  }

 DragonFlying f = new DragonFlying();
//используется префикс содержащего класса
 DragonFlying.Speed s = f.speed();
 DragonFlying.Destination d = f.to("Borneo");
```

Типичное использование внутреннего класса

Внутренний класс реализует некий публичный интерфейс

```
public interface Destination {
    String readLabel();
}
public interface Speed {
    int value();
}
```

Типичное использование внутреннего класса (cont.)

Реализация остается закрытой

```
public class DragonFlying {
 private class DFSpeed implements Speed {…}

 protected class DFDestination implements
 Destination {
   private String label;
   private DFDestination(String whereTo) {
     label = whereTo;
   }
   public String readLabel() { return label; }
 }
```

Типичное использование внутреннего класса (cont.)

Наружу доступен только публично известный интерфейс

```
public Destination dest(String s){
    return new DFDestination(s);
}
public Speed speed() {
    return new DFSpeed();
}
```

Типичное использование внутреннего класса (cont.)

Клиент может взаимодействовать только с публичным интерфейсом

```
class Test {
    public static void main(String[] args) {
        DragonFlying df = new DragonFlying();
        Speed s = df.speed();
        Destination d = df.dest("Tanzania");
        // Незаконно - нельзя получить доступ
        // к private классу:
        //! DragonFlying.DFSpeed dfs = df.new DFSpeed();
    }
}
```

Внутренний класс в случайном контексте

Класс может быть определен даже в теле метода

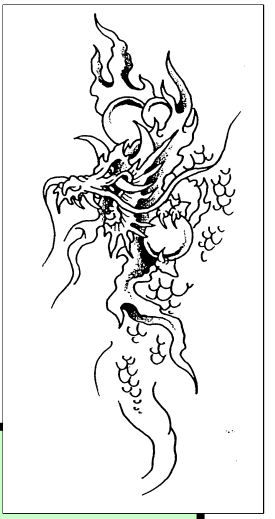
```
private void internalTracking(boolean b) {
   if(b) {
     class TrackingSlip {
       private String id;
       TrackingSlip(String s) { id = s; }
       String getSlip() { return id; }
     }
     TrackingSlip ts = new TrackingSlip("slip");
     String s = ts.getSlip();
   }
   // Нельзя его здесь использовать! Вне контекста:
   //! TrackingSlip ts = new TrackingSlip("x");
 }
```

Внутренний класс в JDK

Анонимный внутренний класс может быть определен в теле метода

```
private void doOnSeparateThread() {
  EventQueue.invokeLater(new Runnable(){
     public void run(){
          System.out.println("Executing on Event
                Dispatching Thread");
     }
  });
}
```

Константы



Константа - это именованное значение, неизменяемое стандартными средствами языка Java

```
class Dragon {
    final static int headCount = 1;
}
//ошибка – попытка присвоить значение константе. Дракон – не Змей
   Горыныч!
Dragon.headCount  = 3;

class Gorinich {
 //MutableFloat – это класс-хранилище дробного числа и  позволяющий
изменять его
    final static MutableFloat headCount = new MutableFloat(1);
}
//ОК, т.к. значение указателя не меняется, меняется только
//содержимое
Gorinich.headVount.setValue(3);
```

Константы в статическом блоке

- Инициализацию константы можно отложить, но только до времени выполнения статического блока класса
- Допустимо произвести только одно присваивание константе

```
class Dragon {
    //нет инициализации
    final static int headCount;
      static {
         //ОК - 1-я инициализация
          headCount = 1;
         //Ошибка - константа уже присвоена
          headCount = 3;
      }
   }
}
```

Абстрактный класс

- Класс является абстрактным, если имеет модификатор abstract
- Класс должен быть помечен этим модификатором, если у него хоть один абстрактный метод (помечен словом abstract и не имеет реализации)

```
abstract class FlyingThing {
  protected String name;
  abstract public void fly();
  public String  getName(){
      return name;
   }
}
//ошибка,абстрактный класс не  может  иметь реализаций
FlyingThing aThing = new FlyingThing();
```

Наследование от абстрактного класса

Как правило, абстрактный класс служит для создания базы дерева наследования классов

```
class Dragon extends FlyingSomething{
  public fly(){
      flySomewhere();//реализуем полет куда-нибудь
  }
}
//ОК - создавать экземпляры можно
Dragon d = new Dragon();
//ОК - создание ссылки на абстрактный класс и
  инициализация конкретным классом
FlyingSomething fs = new Dragon();
```

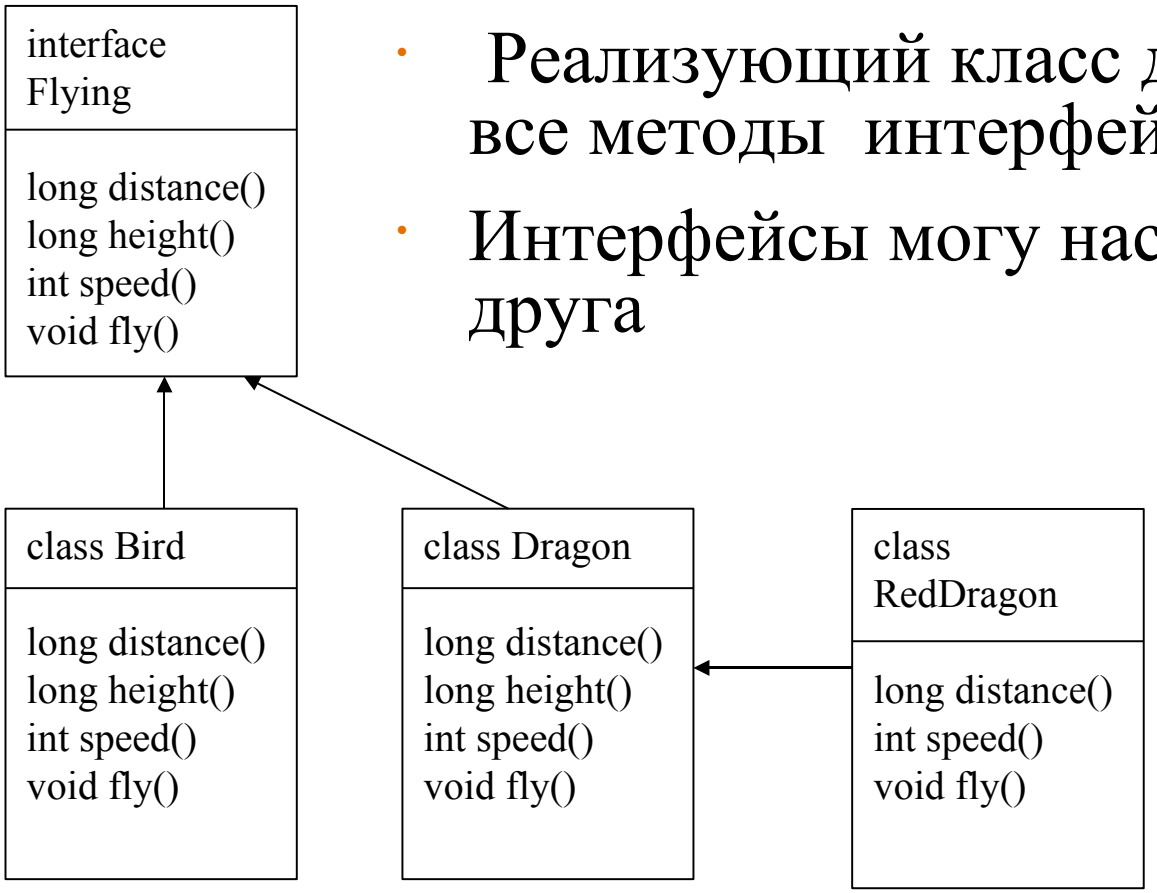
Реализация интерфейса

Интерфейс – это сущность, предназначен для формирования структуры реализующего его класса или для наследования другим интерфейсом

```
public interface Flying{
// класс, реализующий данный интерфейс,
// должен предоставить реализацию для
// этого метода
      int speed();
}
```

Реализация интерфейса (cont.)

- Класс может реализовывать множество интерфейсов
- Реализующий класс должен реализовать все методы интерфейса
- Интерфейсы могу наследоваться друг от друга



Реализация интерфейса (cont.)

Реализация позволяет снабдить класс дополнительными свойствами

```
public class Dragon implements Flying {
     protected int speed;
     public int speed(){
          return speed;
     }
}
public class RedDragon extends Dragon{
     public int speed(){
        return 2*speed;
     }
     public long distance(){…}
     public long burn(Object obj){…}
}
```

Правда ли что…

- Перед определением любого класса нужно указать пакет?
- Все методы в Java – виртуальные?
- Невозможно создать экземпляр абстрактного класса?
- Интерфейсы могут наследоваться друг от друга?

Ссылки

- Рекомендации по оформлению программ:
  - http://java.sun.com/docs/codeconv/
- Учебник (раздел про внутренние классы):
  - http://java.sun.com/docs/books/tutorial/java/javaOO/innerclasses.html

Q&A

Спасибо!

Основы синтаксиса языка Java

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

2008