4 Основы языка Java

4.1 Основы: Программа DragonWorld

```
package heroes;

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

4.2 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);
```

4.3 Основы: класс

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

```
package animals.slowanimals;

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

4.4 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;
}
```

4.5 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();
```

4.6 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++;
   }
}
```

4.7 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 fire;
    }
}
```

4.8 Область видимости переменной

Каждая переменная может быть использована только в ее области видимости.

```
int i = 10;
System.out.println(i); //область видимости ограничится ближайшими фигурными
скобками.
```

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

4.9 Модификаторы

1. Модификаторы доступа являются реализацией принципа инкапсуляции в языке Java.
2. Изменяя модификаторы, можно контролировать область видимости
   - Полей
   - Методов
   - классов

4.9.1 Модификаторы полей

- Отсутствие модификатора – поле доступно только из текущего пакета.
- static – поле принадлежит структуре класса. Одно значение присуще всем экземплярам.
- final – поле не может быть изменено.
- transient – не участвует в процессе сериализации по умолчанию.
- 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.
- volatile – значение этого поля будет обновляться каждый раз при обращении к нему.

4.9.2 Модификаторы методов

- Отсутствие модификатора – доступен только из данного пакета.
- public – метод доступен из любого пакета (публичный API).
- final – 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.
- private – метод не может быть использован ниоткуда кроме данного класса (его объекта).
- protected – метод не может быть использован ниоткуда кроме данного класса (его объекта) и всех его наследников (их объектов).

4.9.3 Модификаторы классов

- Отсутствие модификатора – доступен в текущем пакете.
- public – класс доступен из любого пакета (публичный API).
- final – не может расширяться методом наследования от него.
- abstract – класс является абстрактным, нельзя создать объект этого класса.
- static – допустимо только для вложенных классов.

4.10 Конструктор

- Конструктор – это метод, создающий экземпляр класса.
- Не имеет заданного возвращаемого значения.
- Имеет то же имя, что и класс.
- В классе всегда присутствует конструктор по умолчанию, если явно конструктор не задан.

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

4.11 Вызов метода

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

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

4.12 Виртуальный метод

- Метод класса может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.
- Программисту необязательно знать точный тип объекта для работы с ним через виртуальные методы: достаточно лишь знать, что объект принадлежит классу или наследнику класса, в котором метод объявлен.
- Метод, который, будучи описан в потомках, замещает собой соответствующий метод везде, даже в методах, описанных для предка, если он вызывается для потомка.

4.13 Повторное использование имен

4.13.1 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
```

4.13.2 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(); //обращение к методу Reptile
```

4.13.3 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
        eat(Integer.parseInt(foodWeight));
    }
}
HungryDragon hd = new HungryDragon();
hd.eat(10);
hd.eat("10");
```

4.13.4 Затенение

1) Локальная переменная делает одноименную глобальную переменную невидимой в локальной области.
2) 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);
    }
}
```

4.13.5 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");
    }
}
```

4.14 Правила оформления

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

4.15 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);
```

4.16 Поле this

Каждый объект имеет ссылку на самого себя, которая 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;
    }
}
```

4.17 Поле super

Каждый объект имеет ссылку на объект-предок, которая позволяет организовать восходящие вызовы конструкторов.

```
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
    }
}
```

4.18 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");
  }
}
```

4.19 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);
//OK - переменная dragonCount уже проинициализирована
        System.out.println(dragonCount);
  }
  static String dragonEnemy = "Phoenix";
}
```

4.20 Константы

Константа - это именованное значение, неизменяемое средствами языка Java.

```
  class Dragon {
   final static int headCount = 1;
}
Dragon.headCount  = 3;//ошибка – попытка присвоить значение константе. Дракон
– не Змей Горыныч!
   class Gorinich {
    //MutableFloat – это класс-хранилище дробного числа и позволяющий изменять
его
   final static MutableFloat headCount = new MutableFloat(1);
}
Gorinich.headVount.setValue(3);//OK, т.к. значение указателя не меняется,
меняется только содержимое
```

4.21 Константы в статическом блоке

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

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

4.22 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();
```

4.23 Наследование от абстрактного класса

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

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

4.24 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();
}
```

Класс может реализовывать множество интерфейсов. Реализующий класс должен реализовать все методы интерфейса. Интерфейсы могу наследоваться друг от друга.
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){…}
}
```

5 Введение в ООП

5.1 Инкапсуляция

Инкапсуляция (сокрытие данных) - объединение в одной сущности данных и методов работы с ними.

- 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.

5.2 Наследование

Наследование – возможность класса-наследника приобретать признаки класса-предка.

```
abstract class Animal {
    abstract void talk();
}
class Frog extends Animal {
     void jump() { //прыгать}
}
class Bird extends Animal {
    void fly() {//летать }
}
```

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

```
//ошибка, разрешено наследоваться только от одного класса.
class FlyingFrog extends Frog, Bird{…}
```

5.3 Полиморфизм

Полиморфизм – способность наследников по-другому реализовывать возможности предков. Объект способен проявлять признаки своего предка.

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!"); }
}
```

Экземпляр класса Dog или Cat одновременно является экземпляром класса Animal:

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

5.4 Ромбическое наследование

Для передачи свойств сразу от нескольких сущностей был введен новый тип данных – интерфейс. Используя 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
```

5.5 Примеры

5.5.1 Наследование и инкапсуляция

Класс Pixel наследует от класса Point. При этом он расширяет базовый класс, инкапсулируя поле color (цвет), а также переопределяет метод предка clear, добавляя обнуление данного поля.

```
public class Pixel extends Point {
    Color color;
    public void clear() {
        super.clear();
        color = null;
    }
}
```

5.5.2 Полиморфизм (точка и пиксель)

В классе-предке (Point) и классе-наследнике (Pixel) определен метод show. При вызове данного метода выполняется разный код – в зависимомти от того, к объекту какого класса идет обращение.

```
public class PolyDemo {
    public static void main(String args[]) {

        Point list[] = new Point[3];
        list[0] = new Point(0, 0);
        list[2] = new Point(2, 2);
        list[1] = new Pixel(1, 1);
        for (int i = 0; i < list.length; i++) {
            list[i].show();
        }
    }
}
class Point {
    int x, y;
    Point(int x, int y){
        this.x = x;
        this.y = y;
    }
    void show(){
        System.out.println("I'm a Point["+x+","+y+"]");
    }
}
class Pixel extends Point{
    int color;
    final static int defaultColor = 0;
    Pixel(int x, int y){
        super(x, y);
        this.color = defaultColor;
    }

    Pixel(int x, int y, int c){
        super(x, y);
        this.color = c;
    }
    void show(){
        System.out.println("I'm a Pixel["+x+","+y+"]"+color);
    }
}
```

5.5.3 Интерфейсы

В интерфейсе KeyboardListener определены 2 метода, принимающие код нажатой клавиши. Класс CustomComponent реализует интерфейс – а значит, обязан раелизовать оба метода интерфейса.

```
interface KeyboardListener {
    public void altKeyPressed(int keyCode);
    public char getChar(int keyCode);
}
class  CustomComponent implements KeyboardListener{
    public void altKeyPressed(int keyCode){
        // обработка нажатия клавиши
    }
    public char getChar(int keyCode){
        //преобразование кода клавиши в символ
    }
}
```

```
CustomComponent customComp = new CustomComponent();
customComp.altKeyPressed(27);
customComp.getChar(KeyEvent.VK_Z);
```

5.5.4 Использование интерфейса в качестве типа

В сигнатуре метода можно указать интерфейс в качестве типа параметра. Тогда при обращении к методу параметром может выступать объект любого класса, реализующего данный интерфейс.

```
void handleKeyboard(KeyboardListener kbl, int code) {
    System.out.println(kbl.getChar(code));
}
```