

Лекция 3

Пакеты и интерфейсы

Пакет (package) - это некий контейнер, который используется для того, чтобы изолировать имена классов (в C++ аналог пакета это пространство имен).

Для создания пакета используется ключевое слово **package**, которое должно стоять в начале файла (если такого слова нет, то классы в файле попадают в безымянное пространство имен).

Если объявить класс, как принадлежащий определенному пакету, например,
package java.awt.image;

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

Стоит отметить, что каталог, который транслятор Java будет рассматривать, как корневой для иерархии пакетов, можно задавать с помощью переменной окружения **CLASSPATH**.

С помощью этой переменной можно также задать несколько корневых каталогов для иерархии пакетов (через ; как в обычном PATH).

Если, например, написан класс Myclass.java и помещен в пакет test, тогда, после компиляции, этот класс можно запустить

java test.Myclass

Оператор import

После оператора package в файле обычно идут операторы import.

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.*;

Рассмотрим пример:

```
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);  
  }  
}
```

Класс Protection принадлежит пакету p1.

Кроме оператора `import` можно использовать также статический импорт.

Для того чтобы получить доступ к статическим членам классов, требуется указать ссылку на класс.

К примеру, необходимо указать имя класса `Math`:

```
double r = Math.cos(Math.PI * theta);
```

Использование статического импорта позволяет обойтись без ссылки на класс:

```
import static java.lang.Math.*;
```

.....

```
double r = cos(PI * theta);
```

Интерфейсы

Интерфейсы Java созданы для поддержки динамического выбора методов во время выполнения программы.

Интерфейсы похожи на классы, но в отличие от последних у интерфейсов нет переменных. Класс может иметь любое количество интерфейсов.

Интерфейс также отличается и от абстрактного класса.

В Java если класс реализует(наследует) интерфейс, то он должен реализовать все методы объявленные в интерфейсе, за исключением default методов.

В объявлении интерфейса используется ключевое слово `interface`. В интерфейсе можно также объявлять константы.

Т.о. определение интерфейса имеет вид:

```
interface имя {  
    тип_результата имя_метода1(список  
                                параметров);  
    тип имя_final-переменной = значение;  
}
```

(модификатор `final` можно не указывать, он будет добавлен автоматически).

Рассмотрим пример:

```
interface Callback {  
    void callback(int param);  
}
```

Интерфейсы допускают расширение.

Рассмотрим пример:

```
interface FloorWax{  
    double f();  
}
```

```
interface DessertTopping{  
    int Myconst=10;  
    double f1();  
}
```

```
interface Shimmer extends FloorWax, DessertTopping {  
    //расширение интерфейса  
    double amazingPrice();  
}
```

Java 8 позволяет добавлять неабстрактные реализации методов в интерфейс, используя ключевое слово `default`.

Пример:

```
interface A{  
    void g();  
    default void f(){  
        System.out.println("Method f");  
    }  
}  
class B implements A{  
    public void g(){  
        System.out.println("Method g");  
    }  
}  
public class JavaApplication104 {  
    public static void main(String[] args) {  
        A pa=new B();  
        pa.f(); //На экране Method f  
    }  
}
```

default методы можно и переопределять

```
interface A{  
    void g();  
    default void f(){  
        System.out.println("Method f");  
    }  
}  
class B implements A{  
    public void f(){  
        System.out.println("Method f_B");  
    }  
    public void g(){  
        System.out.println("Method g");  
    }  
}  
public class JavaApplication104 {  
    public static void main(String[] args) {  
        A pa=new B();  
        pa.f(); //На экране Method f_B  
    }  
}
```

```
interface A{  
    void f(int a);  
}  
interface B extends A{  
    default void f(int a){}  
}  
class C implements B{ //корректный код  
}  
public class Main {  
    public static void main(String[] args) {  
        C pc=new C();  
    }  
}
```

Однако следующий код вызовет ошибку компиляции

```
interface A{  
    void g();  
    default void f(){  
        System.out.println("Method f_A");  
    }  
}
```

```
interface C {  
    default void f(){  
        System.out.println("Method f_C");  
    }  
}
```

```
class B implements A,C{ //ошибка компиляции  
    public void g(){  
        System.out.println("Method g");  
    }  
}
```

Выше написанный код можно переписать следующим образом:

```
interface A{  
    void g();  
    default void f(){  
        System.out.println("Method f_A");  
    }  
}  
interface C {  
    default void f(){  
        System.out.println("Method f_C");  
    }  
}  
class B implements A,C{  
    public void f(){  
        System.out.println("Method f_B");  
    }  
    public void g(){  
        System.out.println("Method g");  
    }  
}
```

Также в интерфейсах можно определять статические методы:

```
interface A{  
    void f(int a);  
    static void g(){  
        System.out.println("Hello");  
    }  
}  
  
class B implements A{  
    public void f(int a){}  
  
public class Main {  
    public static void main(String[] args) {  
        A.g();  
    }  
}
```

Реализация интерфейса

Реализация интерфейса осуществляется при помощи ключевого слова **implements**.

Таким образом, класс реализующий интерфейс будет иметь вид

```
class имя_класса [extends суперкласс] [implements  
интерфейс0 [, интерфейс1...]] { тело класса }
```

Рассмотрим пример:

```
class Client implements Callback {  
    void callback(int p) {  
        System.out.println("callback called with " + p);  
    }  
}
```

Метод callback интерфейса, определенного ранее, вызывается через переменную - ссылку на интерфейс:

```
class Testiface {  
    public static void main(String args[]) {  
        Callback c = new Client();  
        c.callback(42);  
    }  
}
```

Конфликты имен

Если два метода отличаются только типом возбуждаемых исключений, метод класса реализующего интерфейс обязан соответствовать обоим объявлениям с одинаковыми сигнатурами (количеством и типом параметров), но может возбуждать свои исключения.

Однако методы в пределах класса не должны отличаться только составом возбуждаемых исключений.

Рассмотрим пример:

```
interface X {  
    void setup() throws SomeException;  
}  
interface Y {  
    void setup();  
}  
class Z implements X, Y {  
    public void setup() { ... }  
}
```

Класс Z может содержать единую реализацию, которая соответствует X.setup и Y.setup.

Метод может возбуждать меньше исключений, чем объявлено в его суперклассе, поэтому при объявлении Z.setup необязательно указывать, что в методе возбуждается исключение типа **SomeException**.

X.setup только разрешает использовать данное исключение.

Лямбда выражения

Синтаксис лямбда выражения имеет вид:

параметры->{тело функции}.

Типы параметров можно опускать. Рассмотрим примеры.

Пример:

```
p -> return p.getGender() == Person.Sex.MALE  
      && p.getAge() >= 18  
      && p.getAge() <= 25
```

```
interface MyInterface{  
    public void func(int a);  
}  
  
public class JavaApplication106 {  
    public static void f(MyInterface m){  
        m.func(20);  
    }  
    public static void main(String[] args) {  
        f(p->{int c=2*p; System.out.println(c);});  
    }  
}
```

В данном случае лямбда функция ничего не возвращает.

В данном случае return не нужен, т.к. лямбда функция ничего не возвращает.

Данный код эквивалентен следующему коду:

```
interface MyInterface{  
  public void func(int a);  
}  
class Impl implements MyInterface{  
  public void func(int a){  
    int c=2*a;  
    System.out.println(c);  
  }  
}  
public class JavaApplication106 {  
  public static void f(MyInterface m){  
    m.func(20);  
  }  
  public static void main(String[] args) {  
    MyInterface obj=new Impl();  
    f(obj);  
  }  
}
```

Однако следующий код вызовет ошибку компиляции:

```
interface MyInterface{  
  public void func(int a);  
  public void g();  
}  
public class Main {  
  public static void f(MyInterface m){  
    m.func(20);  
  }  
  public static void main(String[] args) {  
    f(p->{int c=2*p; System.out.println(c);});  
    //ошибка MyInterface не является  
    //функциональным интерфейсом  
  }  
}
```

Лямбда функция может принимать несколько параметров:

```
interface MyInterface{  
    public int func(int a,int b);  
}  
  
public class JavaApplication106 {  
    public static void f(MyInterface m){  
        System.out.println(m.func(20,30));  
    }  
    public static void main(String[] args) {  
        f((p,p1)->{int c=p+p1; return c;});  
    }  
}
```

Лямбда функции можно использовать для реализации функциональных интерфейсов:

```
class A {  
    int a;  
    public int getA(){  
        return a;  
    }  
}  
interface MyInterface{  
    public int func(A a);  
}  
public class Main {  
    public static void main(String[] args) {  
        MyInterface m=p->{return p.getA();};  
        A pa=new A();  
        pa.a=200;  
        System.out.println(m.func(pa));  
    }  
}
```

Замыкания в лямбда выражениях.

В лямбда выражениях можно использовать переменные из объемлющей области видимости:

```
class A {  
    int a;  
    public int getA(){  
        return a;  
    }  
}  
interface MyInterface{  
    public int func(A a);  
}  
public class Main {  
    static int b=100;  
    public static void main(String[] args) {  
        int c=30;  
        MyInterface m=p->{return p.getA()+b+c;}; //correct  
        A pa=new A();  
        pa.a=200;  
        System.out.println(m.func(pa));  
    }  
}
```

Ссылки на методы.

Ссылки на нестатические методы.

Рассмотрим пример:

```
class A {  
    int a;  
    public int getA(){  
        return a;  
    }  
}  
  
interface MyInterface{  
    public int func(A a);  
}  
  
class B{  
    public int mymethod(A pa){  
        return pa.a+200;  
    }  
}
```

```
public class Main {  
    static int b=100;  
    public static void main(String[] args) {  
        int c=30;  
        B pb=new B();  
        MyInterface m=pb::mymethod;  
        A pa=new A();  
        pa.a=200;  
        System.out.println(m.func(pa));  
    }  
}
```

Ссылки на статические методы.

Рассмотрим пример такой ссылки:

```
class A {  
    int a;  
    public int getA(){  
        return a;  
    }  
}  
  
interface MyInterface{  
    public int func(A a);  
}  
  
class B{  
    public static int mymethod(A pa){  
        return pa.a+200;  
    }  
}
```

```
public class JavaApplication106 {  
    static int b=100;  
    public static void main(String[] args) {  
        int c=30;  
        MyInterface m=B::mymethod;  
        A pa=new A();  
        pa.a=200;  
        System.out.println(m.func(pa));  
    }  
}
```

Ссылка на метод экземпляра из произвольного объекта определенного типа.

Рассмотрим пример:

```
String[] stringArray = { "Barbara", "James", "Mary", "John",  
                          "Patricia", "Robert", "Michael", "Linda" };
```

```
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

В данном случае вызов метода будет иметь вид:

a.compareToIgnoreCase(b).

Ссылка на конструктор.

В Java 8 можно использовать ссылки на конструкторы.

Рассмотрим пример:

```
interface MyInterface{
    public A func();
}

class A{
    int x;
    public A(){
        this.x=100;
    }
    public int getX(){
        return this.x;
    }
}

public class JavaApplication106 {
    public static void f(MyInterface m){
        A mm=m.func();
        System.out.println(mm.getX());
    }
    public static void main(String[] args) {
        f(A::new);
    }
}
```

ОПЕРАТОРЫ И ВЫРАЖЕНИЯ

Все программы на языке Java написаны в Unicode - 16-разрядном наборе символов. Первые 256 символов Unicode представляют собой набор Latin-1, а основная часть первых 128 символов Latin-1 соответствует 7-разрядному набору символов ASCII.

Идентификаторы

Идентификаторы Java, используемые для именования объявленных в программе величин (переменных и констант) и меток, должны начинаться с буквы, символа подчеркивания (_) или знака доллара (\$), за которыми следуют буквы или цифры в произвольном порядке.

Имена переменных в Java можно писать буквами национальных алфавитов.

Символы

Некоторые служебные символы в Java:

- `\n` переход на новую строку (`\u000A`)
- `\t` табуляция (`\u0009`)
- `\\` обратная косая черта (`\u005C`)
- `'` апостроф (`\u0027`)
- `"` кавычка (`\u0022`)
- `\ddd` символ в восьмеричном представлении, где каждое `d` соответствует цифре от 0 до 7

Восьмеричные символьные константы могут состоять из трех или менее цифр и не могут превышать значения `\377` (`\u00ff`). Символы, представленные в шестнадцатеричном виде, всегда должны состоять из четырех цифр.

Объявления переменных

В объявлении указывается тип, уровень доступа и другие атрибуты идентификатора.

Объявление состоит из трех частей: сначала приводится список модификаторов, за ним следует тип, и в завершение следует список идентификаторов(аналогично C++).

Локальные переменные могут объявляться без модификаторов, пример:

```
float[] x, y;
```

Поля с модификатором `final` должны инициализироваться при объявлении.

Массивы

Работа с массивами в Java аналогично C++.

Элементы массива могут иметь примитивный тип или являться ссылками на объекты, в том числе и ссылками на другие массивы.

Массив объявляется следующим образом:

```
int[] ia = new int[3];
```

в данном случае массив имеет три элемента.

Размер массива можно получить из поля `length`.

Рассмотрим пример:

```
for (int i =0; i < ia.length; i++)  
    System.out.println(i + ": " + ia[i]);
```

Массивы всегда являются неявным расширением класса `Object`.

Можно создавать массивы классов.

Рассмотрим пример:

```
class A{  
.....  
public A(int a, int b){....}  
.....  
}
```

```
A[] mas=new A[10];  
for(int i=0;i<mas.length;i++)  
    mas[i]=new A(3,2);
```

В Java можно использовать многомерные массивы:

```
float[][] mat = new float[4][4];
```

.....

```
for (int y = 0; y < mat.length; y++) {  
    for (int x = 0; x < mat[y].length; x++){  
        System.out.println(mat[x][y] + " ");  
    }  
    System.out.println();  
}
```

Первый (левый) размер массива должен задаваться при его создании.

Другие размеры могут указываться позже:

```
float[][] mat = new float[4][];  
for (int y = 0; y < mat.length; y++)  
    mat[y] = new float[4];
```

Инициализация массивов

Чтобы инициализировать массив, следует задать значения его элементов в фигурных скобках после его объявления.

Рассмотрим пример:

```
String[] dangers = { "Lions", "Tigers",  
                    "Bears" };
```

Для многомерных массивов имеем:

```
double[][] identityMatrix = {  
    { 1.0, 0.0, 0.0, 0.0 },  
    { 0.0, 1.0, 0.0, 0.0 },  
    { 0.0, 0.0, 1.0, 0.0 },  
    { 0.0, 0.0, 0.0, 1.0 }, };
```

Тип выражения

У каждого выражения имеется определенный тип.

Он задается типом компонентов выражения и семантикой операторов.

Тип всего выражения определяется максимальным типом его компонентов.

Неявное преобразование типов.

Возможны следующие неявные преобразования типов:

byte -> short -> int -> long -> float-> double

Явное преобразование типов

В Java возможны явные преобразования типов. Рассмотрим пример:

```
double d=5.6;
```

```
long l=(double) d;
```

Рассмотрим приведение типов в классах:

```
class A{....};
```

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

```
class test{
```

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

```
    A obj=new B();
```

```
    if(obj instanceof B)
```

```
        B bobj=(B) obj;
```

```
//Если приведение сработает, то bobj указывает
```

```
//туда же, куда и obj, в противном случае будет
```

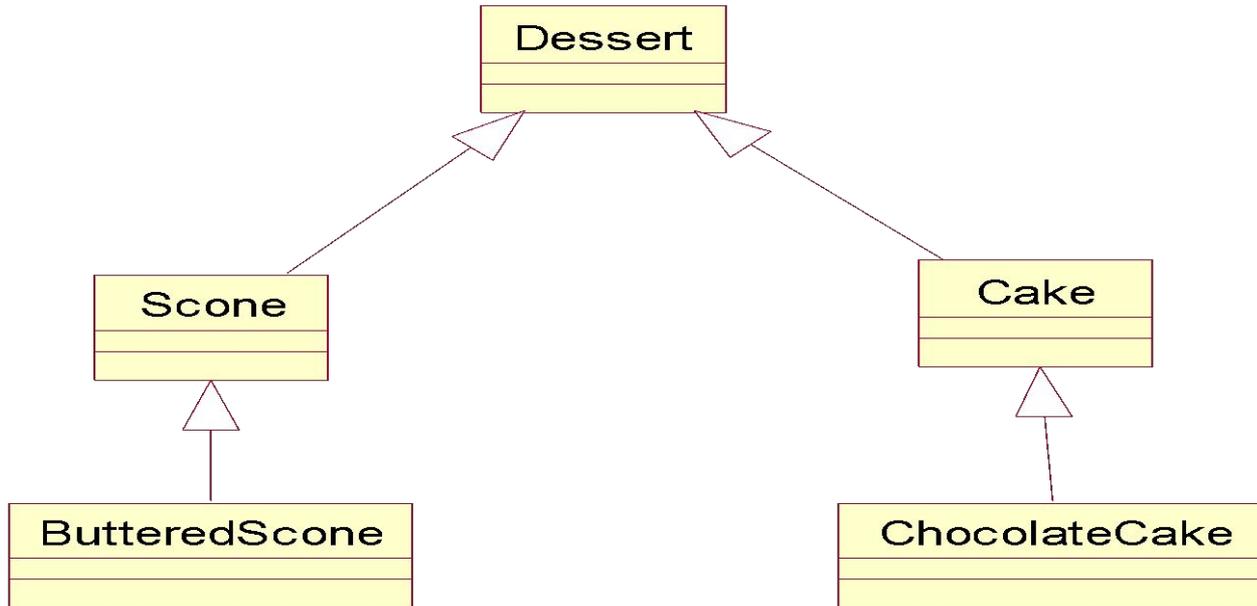
```
//выброшено исключение ClassCastException
```

```
.....}
```

```
}
```

Преобразования типов влияют на перегрузку методов.

Рассмотрим пример. Пусть имеется иерархия классов:



допустим имеется несколько перегруженных методов:

```
void f(Dessert d, Scone s){...};
```

```
void f(Cake c, Dessert d){...};
```

```
void f(ChocolateCake cc, Scone s){...};
```

Рассмотрим вызовы:

f(dessertRef,sconeRef); *//вызывается void*

f(Dessert d, Scone s);

f(chocolateCakeRef, dessertRef); *//вызывается*

void f(Cake c, Dessert d){...};

f(chocolateCakeRef, butteredsconeRef);

//вызывается void f(ChocolateCake cc, Scone s);

f(cakeRef,sconeRef); *//error*

Арифметические операторы

Арифметические операторы в Java совпадают с таковыми в C++.

Условный оператор

Условный оператор в Java совпадает с таковым в C++:

```
value=(x>0? x+2:x+3);
```

Побитовые операторы

& - побитовое И

| - побитовое ИЛИ

^ - операция XOR

<< - сдвиг битов влево с заполнением позиций справа нулями

>> - сдвиг битов вправо с заполнением позиций слева значением старшего(знакового) бита

>>> - сдвиг битов вправо с заполнением позиций слева нулями.