Программирование на Java
Тема 3.1 Отношения между классами и объектами

Наследование. Модификаторы видимости

Пример 1:

```
class Pet // домашние животные
  { protected int age; // возраст
    protected String name; // кличка
    Pet(){};
    public void voice()
     { System.out.println("голос домашнего животного");

     }
  }
class Cat extends Pet
{  private int mouseCatched; // число пойманных мышей
   public void voice()
     { System.out.println("мяу - мяу");

     }

}
```

Наследование. Модификаторы видимости

- Наследование устанавливает между классами отношение «обобщение / специализация». В подклассе (производном классе) структура и поведение исходного суперкласса (базового класса) дополняются и **переопределяются**.
- Для того, чтобы один класс был потомком другого, необходимо при его объявлении после имени класса указать ключевое слово **extends** и название суперкласса (базового класса).
- Например, производный класс **Cat** дополняет и переопределяет структуру и поведение базового класса **Pet** (т. е. наследует поля **age** и **name,** дополняет к ним поле **mouseCatched** и переопределяет метод **voice**()).
- С помощью ключевого слова **extends** в языке Java поддерживает простое наследование.
- Наследование называют простым, когда каждый класс-потомок имеет только один родительский класс ближайшего уровня.
- Доступ к любому члену класса — полю или методу — может быть ограничен. Для этого перед его объявлением ставится ключевое слово **private**. Оно означает, что к этому члену класса нельзя будет обратиться из методов других классов.
- Ключевое слово **public** может употребляться в тех же случаях, но имеет противоположный смысл. Оно означает, что данный член класса является доступным. Если это поле, его можно использовать в выражениях или изменять при помощи присваивания, а если метод, его можно вызывать.
- Ключевое слово **protected** означает, что доступ к полю или методу имеет сам класс и все его потомки.
- Если при объявлении члена класса не указан ни один из перечисленных модификаторов, то это означает, что доступ к члену класса имеют все классы, объявленные в том же пакете.

Конструкторы

Пример 2:

```
class Pet // домашние животные
  { protected int age; // возраст
    protected String name; // кличка
    Pet(){};
    Pet(String n, int a) { name = n; age = a; }
    public void voice()
     { System.out.println("голос домашнего животного");

     }
  }
class Cat extends Pet
{  private int mouseCatched; // число пойманных мышей
   Cat(String n, int a, int m)
    { super(n, a); mouseCatched = m; }

   public void voice()
     { System.out.println("мяу - мяу");

     }
}
```

Конструкторы

- Конструкторы, в отличие от других методов, не наследуются. Поэтому для того, чтобы выполнить инициализацию полей базового класса, унаследованных производным классом, необходимо вызывать конструктор базового класса.
- Ключевое слово **super** используется для вызова конструктора базового класса (суперкласса) в конструкторе производного класса (подкласса). Вызов конструктора суперкласса должен происходить в самом начале конструктора.
- Если в начале конструктора нет ни вызова super(), автоматически происходит обращение к конструктору суперкласса без аргументов.

Полиморфизм

```
class Main
 { public static void main (String args [])
     { Pet p = new Pet();
       p.voice();
       Cat Cat1 = new Cat("Мурзик", 2, 10);
       Cat1.voice();
       Pet Pet1 = new Cat("Пушок", 2, 10);
       Pet1.voice();
     }
}
```

```
Голос домашнего животного
мяу - мяу
мяу - мяу
```

Полиморфизм

- **Полиморфизм (polymorphism)** - положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов. Следовательно, любой объект, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций.
- Полиморфизм возникает там, где взаимодействуют наследование и динамическое связывание.
- В языке Java ссылки на базовый класс могут содержать не только адреса объектов базового класса, но и адреса объектов производных классов. В примере, ссылке **Pet1** на базовый класс **Pet** присваивается адрес объекта производного класса **Cat**.
- В языке Java все методы являются динамически связываемыми (виртуальными), т. е. могут переопределяться (замещаться) одноименными методами в производных классах. В примере, метод voice() базового класса **Pet** переопределяется (замещается) методом **voice**() производного класса **Cat**.

Абстрактные классы и методы

Пример 3:

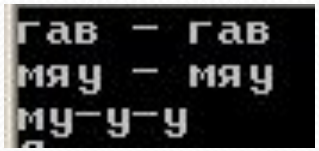
```
abstract class Pet // домашние животные
   { protected int age; // возраст
     protected String name; // кличка
     Pet(){};
     Pet(String n, int a) { name = n; age = a; }
    abstract void voice();
   }
class Cat extends Pet
{   int mouseCatched = 3; // число пойманных мышей
    Cat(String n, int a, int m)
     { super(n, a); mouseCatched = m; }

    public void voice()
      { System.out.println("мяу - мяу");

      }
}
```

Абстрактные классы и методы

```
class Dog extends Pet
{
    Dog(String n, int a)
     { super(n, a);}

    public void voice()
      { System.out.println("гав - гав");

      }
}
class Cow extends Pet
{
    Cow(String n, int a)
     { super(n, a);}

    public void voice()
      { System.out.println("му-у-у");

      }
}
```

Абстрактные классы и методы

```
class Main
 { public static void main (String args [])
     { Pet Ptr[] = new Pet[3];
     Ptr[0] = new Dog("Тузик", 2);
     Ptr[1] = new Cat("Мурзик", 3, 10);
     Ptr[2] = new Cow("Зорька", 5);
    for (int i = 0; i < Ptr.length; i++)
        Ptr[i].voice();
       //System.out.println(Ptr[1].mouseCatched); - Ошибка!!
     }
}
```



Абстрактные классы и методы

- В примере 3 животные поют своими голосами!
- Хотя массив ссылок **Ptr** [ ] имеет тип **Pet** , каждый его элемент ссылается на объект своего типа **Dog**, **Cat**, **Cow** . При выполнении программы вызывается метод конкретного объекта, а не метод класса, которым определялось имя ссылки. Так в Java реализуется полиморфизм.
- Класс **Pet** и метод **voice**() являются абстрактными. При описании класса **Pet** мы не можем задать в методе **voice** () никакой полезный алгоритм, поскольку у всех животных совершенно разные голоса.
- В таких случаях мы записываем только заголовок метода и ставим после закрывающей список параметров скобки точку с запятой. Этот метод будет *абстрактным* (abstract), что необходимо указать компилятору модификатором **abstract** .
- Если класс содержит хоть один абстрактный метод, то создать его экземпляры, а тем более использовать их, не удастся. Такой класс становится *абстрактным*, что обязательно надо указать модификатором **abstract** .
- Хотя элементы массива **Ptr**[] ссылаются на объекты подклассов **Dog**, **Cat**, **Cow** , но все-таки это переменные типа **Pet** и ссылаться они могут только на поля и методы, описанные в суперклассе **Pet** . Дополнительные поля подкласса для них недоступны. Попробуйте обратиться, например, к полю **mouseCatched** класса **Cat** , написав **Ptr[1].mouseCatched ,** компилятор "скажет", что он не может реализовать такую ссылку.

Модификатор final

Пример 4:

```
final class Cat extends Pet
{   int mouseCatched = 3; // число пойманных мышей
    Cat(String n, int a, int m)
     { super(n, a); mouseCatched = m; }

    public void voice()
      { System.out.println("мяу - мяу");

      }
}
```

Модификатор final

- Иногда наследование является нежелательным. Классы, которые нельзя расширить, называются *терминальными* (final). Для того чтобы отметить этот факт, в определении класса используется модификатор **final**.
- Например, чтобы предотвратить создание подклассов класса **Cat**, его нужно просто объявить с помощью модификатора **final**.
- Все методы терминального класса автоматически являются терминальными. Например, класс **String** является терминальным. Это означает, что определить подкласс этого класса невозможно.
- Отдельный метод класса также может быть терминальным. Такой метод не может замещаться никакими методами подклассов.
- Напомним, что с помощью модификатора **final** могут также объявляться константные поля. После создания объекта содержание константного поля изменить нельзя. Однако если класс объявлен терминальным, его поля от этого не становятся константными.

Внутренние классы (не статические вложенные классы)

- **Пример :**

```
public class Ship
{ // поля и конструкторы
  // abstract, final, private, protected - допустимы
  // определение внутреннего класса
  public class Engine
     {  // поля и методы
       public void launch()
        { System.out.println("Запуск двигателя");
     }
  public void init() { // метод внешнего класса
  // объявление объекта внутреннего класса
  Engine eng = new Engine();
  eng.launch();
  }
}
Ship.Engine obj = new Ship().new Engine();
Ship$Engine.class
```

Внутренние классы

- В Java можно определить (вложить) один класс внутри определения другого класса, что позволяет группировать классы, логически связанные друг с другом, и динамично управлять доступом к ним. Нестатические вложенные классы принято называть внутренними (inner) классами.
- Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса, в то же время внешний класс может получить доступ к содержимому внутреннего класса только после создания объекта внутреннего класса.
- В качестве примеров можно рассмотреть взаимосвязи классов «Корабль» и «Двигатель». Объект класса «Двигатель» расположен внутри (невидим извне) объекта «Корабль» и его деятельность приводит «Корабль» в движение. Оба этих объекта неразрывно связаны, то есть запустить «Двигатель» можно только посредством использования объекта «Корабль».
- Использование объекта внутреннего класса вне своего внешнего класса возможно только при наличии доступа (видимости). Внутренний класс может быть объявлен как private, что обеспечивает его полную невидимость вне класса-владельца и надежное сокрытие реализации. В этом случае ссылку obj, приведенную выше, объявить было бы нельзя. Создать объект такого класса можно только в методах и логических блоках внешнего класса. Использование protected позволяет получить доступ к внутреннему классу для класса в другом пакете, являющегося суперклассом внешнего класса.
- После компиляции объектный модуль, соответствующий внутреннему классу, получит имя Ship$Engine.class.
- Внутренние классы не могут содержать статические атрибуты и методы, кроме констант (final static). Внутренние классы имеют право наследовать другие классы, реализовывать интерфейсы и выступать в роли объектов наследования.

Статические вложенные классы

- **Пример :**

```
public class Ship
{ private int id;
  public static class LifeBoat
    { public static void down()
       { System.out.println("шлюпки на воду!");}
      public void swim()
       { System.out.println("отплытие шлюпки"); }
    }
}
```

Статические вложенные классы

```
public class RunnerShip
 { public static void main(String[] args)
    { // вызов статического метода
       Ship.LifeBoat.down();
      // создание объекта статического класса
      Ship.LifeBoat lf = new Ship.LifeBoat();
      // вызов обычного метода
      lf.swim();
    }
 }
```

Статические вложенные классы

- Статический вложенный класс логически связан с классом-владельцем, но может быть использован независимо от него.
- Такой класс способен наследовать другие классы, реализовывать интерфейсы и являться объектом наследования для любого класса, обладающего необходимыми правами доступа.
- Статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса, а напрямую имеет доступ только к статическим полям и методам внешнего класса.
- Для создания объекта вложенного класса объект внешнего класса создавать нет необходимости. Подкласс вложенного класса не способен унаследовать возможность доступа к членам внешнего класса, которыми наделен его суперкласс.
- Класс «Шлюпка» также является логической частью класса «Корабль», но его объекты могут быть использованы независимо от наличия объекта «Корабль».
- Статический метод вложенного класса «Шлюпка» вызывается при указании полного относительного пути к нему. Объект lf вложенного класса «Шлюпка» создается с использованием имени внешнего класса без вызова его конструктора.

Локальные и анонимные классы

- В Java можно создавать локальные и анонимные классы.
- Локальные классы объявляются внутри методов внешнего класса. Имеют доступ к членам внешнего класса. Имеют доступ как к локальным переменным, так и к параметрам метода при одном условии - переменные и параметры используемые локальным классом должны быть объявлены final. Не могут содержать определение (но могут наследовать) статических полей, методов и классов (кроме констант).
- Анонимный класс является дальнейшим развитием идеи локального класса. Вместо того чтобы объявлять локальный класс с помощью одного оператора Java и затем с помощью другого оператора создавать экземпляр данного класса, в анонимном классе эти этапы объединены в одну Java-конструкцию. Анонимный класс не имеет имени. К тому же, экземпляр его создается в том же выражении, в котором он объявляется, поэтому такой экземпляр может быть только один. В остальном анонимные классы по своим свойствам и принципам использования очень похожи на локальные. Конечно же, интерфейсы не могут быть анонимными.
- При разработке классов-адаптеров модели обработки событий выбор между именованным локальным классом и анонимным классом, как правило, определяется лишь стилем программирования и простотой чтения программы, а не какими-либо различиями в функциональности.

Пакеты

- **Пакет** — это группа взаимосвязанных классов, интерфейсов, подпакетов.

- Имя пакета одновременно является названием папки, в которой находятся файлы классов, входящие в пакет:

  java.util -> java\util

- Полное имя класса:

  <имя пакета>.<имя класса>

- Создание пакета (подпакета):

  package <имя пакета>;

  package <имя пакета>.<имя подпакета>;

Пакеты

- Все классы Java распределяются по пакетам. Кроме классов пакеты могут включать в себя интерфейсы и вложенные подпакеты. Образуется древовидная структура пакетов и подпакетов.
- Эта структура в точности отображается на структуру файловой системы. Каждый пакет имеет имя. Имя представляет собой обычный идентификатор Java. Это имя одновременно является названием папки, в которой хранятся файлы классов, входящие в пакет. А точка в имени преобразуется в разделитель имен файловой системы. То есть пакет с именем java.util будет представлен папкой util, находящейся внутри папки java.
- Каждый пакет образует одно пространство имен (namespace). Это означает, что все имена классов, интерфейсов и подпакетов в пакете должны быть уникальны. Имена в разных пакетах могут совпадать, но это будут разные программные единицы. Если надо использовать два класса с одинаковыми именами из разных пакетов, то имя класса уточняется именем пакета: пакет.класс . Такое уточненное имя называется полным именем класса.
- Пакетами пользуются еще и для того, чтобы добавить к уже имеющимся правам доступа к членам класса private, protected и public еще один, "пакетный" уровень доступа.
- Если член класса не отмечен ни одним из модификаторов private, protected, public, то, по умолчанию, к нему осуществляется пакетный доступ, а именно, к такому члену может обратиться любой метод любого класса из того же пакета. Пакеты ограничивают и доступ к классу целиком — если класс не помечен модификатором public , то все его члены, даже открытые, public , не будут видны из других пакетов.
- Чтобы создать пакет надо просто в первой строке Java-файла с исходным кодом записать строку package имя; , например: package mypack;
- Тем самым создается пакет с указанным именем mypack и все классы, записанные в этом файле, попадут в пакет mypack . Повторяя эту строку в начале каждого исходного файла, включаем в пакет новые классы. Имя подпакета уточняется именем пакета. Чтобы создать подпакет с именем, например, subpack , следует в первой строке исходного файла написать: package mypack.subpack;

Пакеты

- Рекомендуется использовать в качестве имени пакета доменное имя своего сайта, записанное в обратном порядке, например:

com.sun.developer.

- Импортирование классов и пакетов

```
import java.util.Vector;
import java.util.*;
```

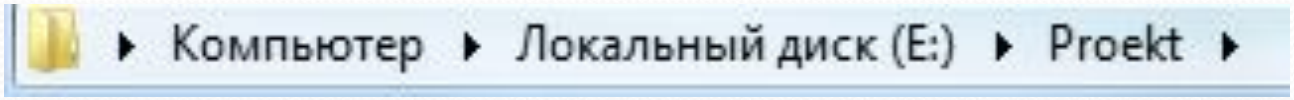
Пакеты

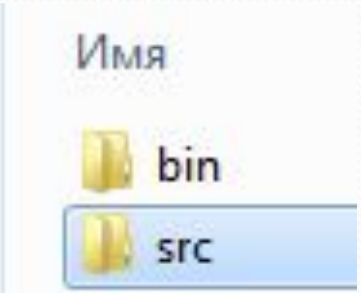
- Поскольку строка package имя; только одна и это обязательно первая строка файла, каждый класс попадает только в один пакет или подпакет.
- Компилятор Java может сам создать каталог с тем же именем mypack, а в нем подкаталог subpack, и разместить в них class-файлы с байт-кодами.
- Рекомендуется записывать имена пакетов строчными буквами, тогда они не будут совпадать с именами классов, которые, по соглашению, начинаются с прописной. Кроме того, рекомендуется использовать в качестве имени пакета или подпакета доменное имя своего сайта, записанное в обратном порядке, например: com.sun.developer.
- До сих пор мы ни разу не создавали пакет. Куда же попадали файлы с откомпилированными классами? Компилятор всегда создает для таких классов безымянный пакет, которому соответствует текущий каталог файловой системы. Вот поэтому class-файл всегда оказывался в том же каталоге, что и соответствующий Java-файл.
- Оператор import . Для чего он нужен? Дело в том, что компилятор будет искать классы только в - одном пакете, именно, в том, что указан в первой строке файла. Для классов из другого пакета надо указывать полные имена.
- Но если полные имена длинные, а используются классы часто, то стучать по клавишам, набирая полные имена, становится утомительно. Вот тут-то мы и пишем операторы import, указывая компилятору полные имена классов.
- Правила использования оператора import очень просты: пишется слово import и, через пробел, полное имя класса, завершенное точкой с запятой. Сколько классов надо указать, столько операторов import и пишется.
- Это тоже может стать утомительным и тогда используется вторая форма оператора import — указывается имя пакета или подпакета, а вместо короткого имени класса ставится звездочка *. Этой записью компилятору предписывается просмотреть весь пакет.

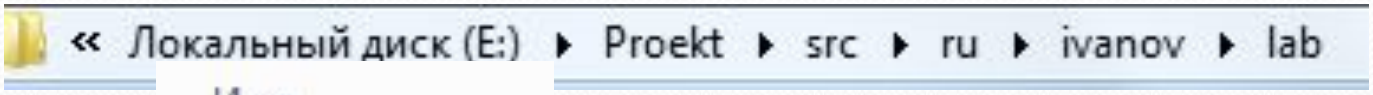
Структура Java-файла с исходным кодом

- В первой строке файла может быть необязательный оператор package .
- В следующих строках могут быть необязательные операторы import .
- Далее идут описания классов и интерфейсов.
- Среди классов файла может быть только один открытый public -класс.
- Имя файла должно совпадать с именем открытого класса.
- Все классы и интерфейсы, которые впоследствии предполагается использовать в других пакетах, должны быть объявлены открытыми (а значит, находиться в отдельных файлах).

Проектирование пакетов и их каталогов. Компиляция и сборка программы в командной строке. Пример1







Имя

Lab.java

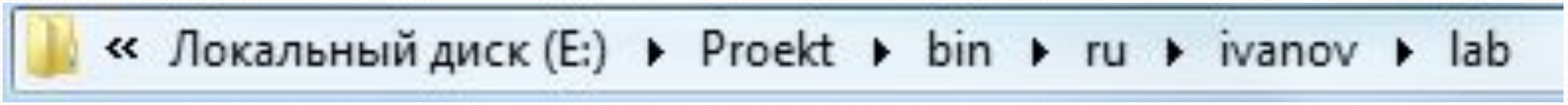
```
package ru.ivanov.lab;

public class Lab
{ public static void main (String args [])
    { Pet p = new Pet();
      p.voice();
      Cat Cat1 = new Cat("Мурзик", 2, 10);
      Cat1.voice();
      Pet Pet1 = new Cat("Пушок", 2, 10);
      Pet1.voice();
    }
}
```

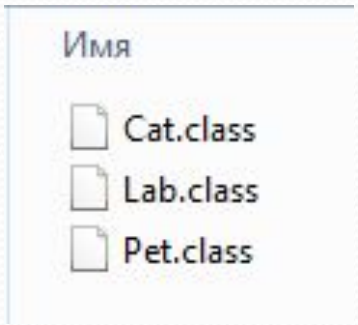
Проектирование пакетов и их каталогов. Компиляция и сборка программы в командной строке. Пример1

```
class Pet // домашние животные
  { protected int age; // возраст
    protected String name; // кличка
    Pet(){};
    Pet(String n, int a) { name = n; age = a; }
    public void voice()
     { System.out.println("голос домашнего животного");

     }
  }
class Cat extends Pet
{  private int mouseCatched; // число пойманных мышей
   Cat(String n, int a, int m)
    { super(n, a); mouseCatched = m; }

   public void voice()
     { System.out.println("мяу - мяу");

     }
}
```

Проектирование пакетов и их каталогов. Компиляция и сборка программы в командной строке. Пример1

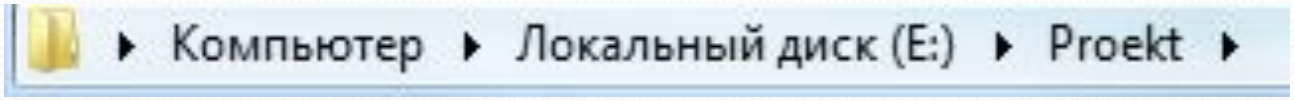
```
E:\Proekt>javac  -d bin src/ru/ivanov/lab/Lab.java
```

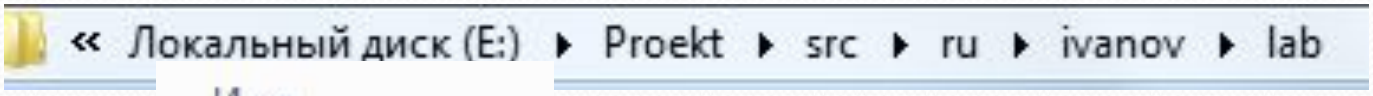





```
E:\Proekt>java -classpath bin ru.ivanov.lab.Lab
голос домашнего животного
мяу - мяу
мяу - мяу
```

Проектирование пакетов и их каталогов. Компиляция и сборка программы в командной строке. Пример2





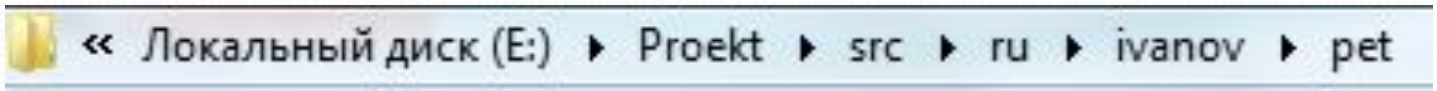
Имя

Lab.java

Lab.java | Pet.java | Cat.java

```
1  package ru.ivanov.lab;
2  import ru.ivanov.pet.*;
3
4  public class Lab
5   { public static void main (String args [])
6       {
7         Cat Cat1 = new Cat("Мурзик", 2, 10);
8         Cat1.voice();
9         Pet Pet1 = new Cat("Пушок", 2, 10);
10        Pet1.voice();
11        Pet p = new Pet();
12        p.voice();
13      }
14 }
```

Проектирование пакетов и их каталогов. Компиляция и сборка программы в командной строке. Пример2



Cat.java

Pet.java

Pet.java | Cat.java

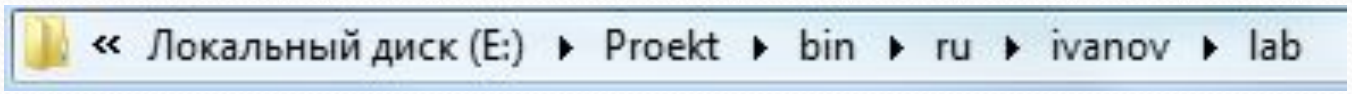
```
package ru.ivanov.pet;

public class Pet // домашние животные
  { protected int age; // возраст
    protected String name; // кличка
    public Pet(){}
    public Pet(String n, int a) { name = n; age = a; }
    public void voice()
     { System.out.println("голос домашнего животного");

     }
  }
```

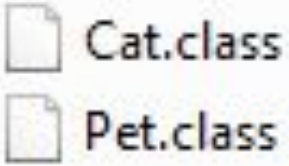
Pet.java | Cat.java

```
package ru.ivanov.pet;

public class Cat extends Pet
{  private int mouseCatched; // число пойманных мышей
   public Cat(String n, int a, int m)
    { super(n, a); mouseCatched = m; }

   public void voice()
     { System.out.println("мяу - мяу");

     }
}
```

Проектирование пакетов и их каталогов. Компиляция и сборка программы в командной строке. Пример2

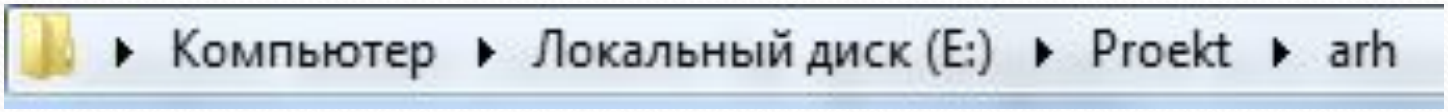
```
E:\Proekt>javac -sourcepath src -d bin src/ru/ivanov/lab/Lab.java
```



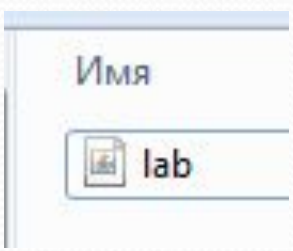


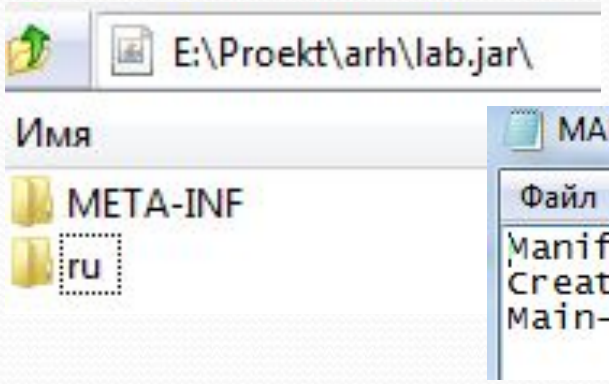

```
E:\Proekt>java -classpath bin ru.ivanov.lab.Lab
мяу - мяу
мяу - мяу
голос домашнего животного
```

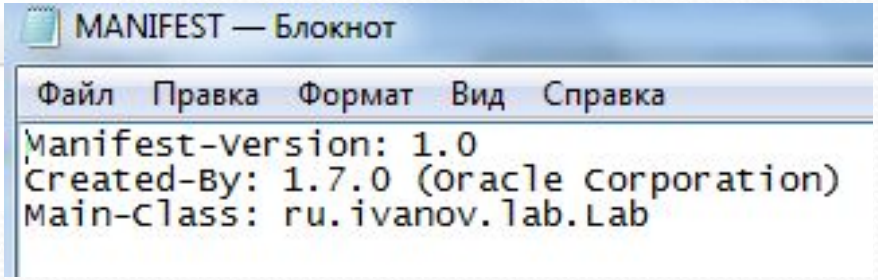
Создание архива jar. Пример3




```
E:\Proekt>jar cef ru.ivanov.lab.Lab arh/lab.jar -C bin .
```








```
E:\Proekt>java -jar arh/lab.jar
мяу - мяу
мяу - мяу
голос домашнего животного
```

Пакеты

- Jar-файл — это ZIP архив (то есть вы можете разархивировать его). Jar-файл должен в себе содержать набор классов и файл META-INF/MANIFEST.MF, в котором описаны характеристики данного jar-файла.
- Основной вариант создания Jar-файла: **jar cf jar-file input-file(s)**
- Jar – это утилита и набора утилит которые вы получаете при установке java.
- Программа jar принимает аргументы: вначале идут ключи потом аргументы программы, ключ с аргументом указывается последним, не указывать «-» перед аргументами, группировать короткие аргументы («cf» значит «-c -f »).
- 1. Опция c — говорит о том, что вы хотите создать (create) jar-файл.
- 2. Опция f — говорит о том, что вы хотите создать файл (file) с определённым именем (например, «jar-file.jar»).
- 3. Аргумент input-file(s) является разделенный пробелами список из одного или нескольких файлов, которые вы хотите включить в ваш JAR-файл. input-file(s) аргумент может содержать символ «*». Если любой из входных является каталогом, содержимое этих каталогов добавляются в архив JAR рекурсивно.
- Когда вы создаете JAR-файл, он автоматически получает файл манифеста по умолчанию (если вы его не указали во входных файлах – он будет создан автоматически). В jar-файле может быть только один файл манифеста с указанным путём: META-INF/MANIFEST.MF
- Элементы манифеста имеют форму "заголовок: значение" пар. Имя заголовка отделяется от ее значения двоеточием.
- Чтобы создать jar-файл с манифестом: **jar cfm jar-file manifest-addition input-file(s)**
- Ключ «f» и «m» оба требуют аргументов, поэтому мы вначале указываем ключи, а потом в том же порядке указываем (если это необходимо) недостающее аргументы. В начале мы указали аргумент «f», а потом «m», поэтому первый аргумент будет имя выходного файла, а второй это имя (и путь) к манифесту.

Пакеты

- Если вы разрабатываете приложение, которое поставляется в JAR-файл, необходимо каким-то образом указать, какой класс в JAR-файле является входной точкой приложения (который содержит функцию main). Вы предоставляете эту информацию с Main-Class заголовка в манифесте, который имеет общий вид: **Main-Class: имя класса** .
- Значение имени класса является именем класса, который является входной точкой приложения.
- После того как вы установите Main-Class заголовка в манифесте, вы запустите файл JAR с помощью следующей формы Java команду: **java -jar JAR-file**
- Не указав главного класса в манифесте вам придется выполнять вашу программу так: **java -cp JAR-file.jar MainClass**
- Если вы хотите указать лишь главный класс в манифесте, то вам не нужно создавать весь манифест, вы можете указать, необходимы параметр при вызове jar: **jar cfe app.jar MyApp MyApp.class**
- Опция e — говорит о точки входа в программу (entrypoint).
- Вам придется ссылаться на классы в другие файлы JAR из JAR-файла (если вы используете сторонние библиотеки в своем приложении). Для этого вам необходимо включить следующие поля в манифест: **Class-Path: jar1-name jar2-name directory-name/jar3-name**
- Данный путь указывается относительно расположению выполняемого jar файла. К примеру, Netbeans складывает все библиотеки в папку lib, которую помещает рядом с собранным приложением, и соответственно указывает путь к библиотекам.
- Рассмотрим команду: **jar cef ru.ivanov.lab.Lab arh/lab.jar –C bin .** :
- **ru.ivanov.lab.Lab –** имя главного класса и путь к нему;
- **arh/lab.jar** – имя архивного файла и каталог, в котором он находится;
- **- C bin .** - часть этой команды **- C bin** велит утилите Jar перейти в каталог bin, а ., следующая за -C bin, велит утилите Jar архивировать все содержимое каталога.

Контрольные вопросы

1. Понятие наследование в ООП. Как реализуется наследование в Java.
2. Понятие полиморфизма в ООП. Как реализуется полиморфизм в Java.
3. Абстрактные классы и абстрактные методы: назначение, синтаксис описания и примеры использования.
4. Внутренние классы: характеристика и примеры использования.
5. Статические вложенные классы: характеристика и примеры использования. Понятие локальных и анонимных классов.
6. Пакеты в Java: понятие и назначение. Проектирование пакетов и их каталогов. Компиляция и сборка программы в командной строке.
7. Понятие, назначение и состав jar-файлов. Примеры создания jar-файлов.