

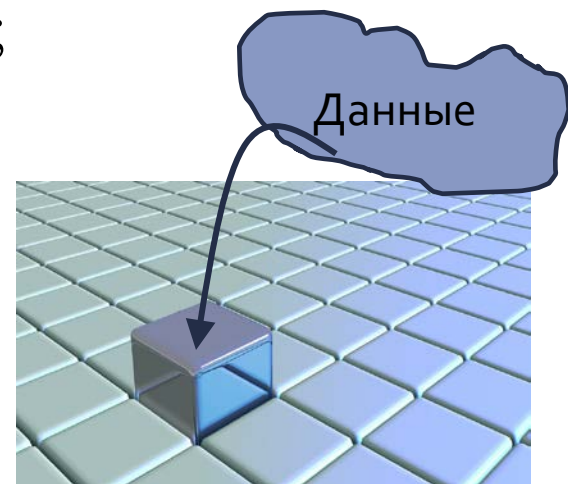


Способы инициализации полей

Способы инициализации полей

❑ Перед выполнением приложения Java специальный компонент виртуальной машины - *загрузчик классов* – загружает начальный класс этого приложения – класс с методом ***public static void main(String [] args)***, – и *верификатор Java* проверяет байт-код этого класса. *Затем этот класс инициализируется.*

- 1) автоматическая инициализация полей классов;
- 2) явные инициализаторы полей класса;
- 3) статические блоки инициализации;
- 4) логические блоки (не статические);
- 5) конструкторы



Способы инициализации полей

1) Первый вид инициализации класса – это автоматическая инициализация полей классов в значения по умолчанию.

Пример 1:

```
public class InitDemo1 {  
    private static char ch;  
    private static boolean bb;  
    private static byte by;  
    private static int ii;  
    private static float ff;  
    private static String str;  
    private static int[] array;  
    public static void main(String[] arg){  
        System.out.println("char: " + ch);  
        System.out.println("boolean: " + bb);  
        //...  
    }  
}
```

Вывод в консоли:

```
char:  
boolean: false  
byte: 0  
int: 0  
float: 0.0  
String: null  
Array: null
```

2) Второй вид инициализации класса – это явные инициализаторы полей класса в их начальные значения (каждое поле класса может явно быть проинициализировано некоторым значением и эту инициализацию можно записать в одну строку).

Пример 2:

```
public class InitDemo2 {  
    private static char ch = 'A';  
    private static boolean bb = true;  
    private static byte by = -56;  
    private static int ii = 1000;  
    private static float ff = 1.25e-2F;  
    private static String str = "Data";  
    private static int[] array = {0, 1, 2, 3};  
    // .....  
}
```

Вывод в консоли:

```
char: A  
boolean: true  
byte: -56  
int: 1000  
float: 1.25e-2  
String: Data  
Array: [0, 1, 2, 3]
```

Способы инициализации полей

- ❑ Компилятор Java автоматически генерирует метод инициализации класса (*внутренний метод с именем `<clinit>`*) для каждого класса.
- ✓ Метод гарантированно будет вызываться только один раз, когда класс впервые используется.
- ✓ Выражения инициализации полей класса вставляются в метод инициализации класса в порядке их появления в исходном коде (в выражении инициализации для поля класса можно использовать ранее объявленные поля класса).

Пример 3:

```
public class InitDemo3 {  
    //...  
    private static byte by = 17;  
    private static int ii = 24 * by;  
    //...  
}
```

Обратное не
допускается



Способы инициализации полей

- ✓ В выражении инициализации поля класса можно использовать обращение к статическому методу (преимущество – повторное использование, если вам нужно инициализировать поле класса).

Пример 4:

```
public class InitDemo4 {  
    private static int ii = initSt();  
    //...  
    private static int initSt() {  
        System.out.println("Init ii value");  
        return 1000;  
    }  
    public static void main(String[] arg) {  
        System.out.println("Main");  
        System.out.println("int: " + ii);  
    }  
}
```

Вывод в консоли:

```
Init ii value  
Main  
int: 1000
```

Если инициализируется класс с точкой входа в Java-программу, то метод *main()* выполняется после инициализации полей класса

3) Третий вид инициализации – это статические блоки инициализации, используются когда требуется некоторая логика (например, обработка ошибок или циклы для заполнения сложных наборов данных) и только один раз.

Ограничения:

- оператор **return** не может использоваться в пределах статического блока инициализации;
 - ключевое слово **this** не может использоваться в пределах статического блока инициализации;
 - на не статическую переменную (переменную экземпляра) нельзя ссылаться из статического блока инициализации.
- ❑ Компилятор Java вставляет код статического блока в метод инициализации класса (метод **<clinit>**) после инициализации полей класса выражениями.

Пример 5:

```
public class InitDemo5 {  
    private static char[] alph;  
  
    public static void main(String[] arg) {  
        System.out.print(Arrays.toString(alph));  
    }  
  
    static {  
        alph = new char[26];  
        int i = 0;  
        for (char c = 'a'; i < alph.length; c++, i++) {  
            alph[i] = c;  
        }  
    }  
}
```


Особенности

- ✓ Класс может иметь любое количество статических блоков инициализации;
- ✓ Они могут появляться в любом месте тела класса;
- ✓ Исполнительная система гарантирует, что статические блоки инициализации вызываются в том порядке, в котором они появляются в исходном коде;
- ✓ Такой блок выполняется только один раз, когда класс инициализируется или загружается.

- 4) Четвертый вид инициализации – не статические блоки инициализации, другими словами логические блоки, которые являются альтернативой конструкторам класса для инициализации полей экземпляра.

Выглядят:

```
{  
    // Любой код, необходимый для инициализации  
}
```

Примечание:



**Используются для разделения блока кода между несколькими конструкторами;
Компилятор Java копирует логические блоки инициализации в каждый конструктор наряду с инициализацией переменных экземпляра выражением в порядке их следования в исходном коде.**

Пример 6:

```
public class Student {  
    private static  
        int numOfStudents;  
        //...  
    public Student() {  
        //...  
        numOfStudents++;  
    }  
    public Student(String name) {  
        //...  
        numOfStudents++;  
    }  
}
```

Дублирование
кода

```
public class Student {  
    private static  
        int numOfStudents;  
        //...  
    {  
        numOfStudents++;  
    }  
    public Student() {  
        //...  
    }  
    public Student(String name){  
        //...  
    }  
}
```

Вынесение общего
кода в логический
блок

Порядок инициализации класса

Инициализация полей класса значения на умолчанию

Инициализация полей класса выражениями

Выполнение статических блоков инициализации

Если это класс с точкой входа, то выполнение метода *main()*

Порядок инициализации при создании экземпляра класса

Рекурсивный вызов и выполнение конструкторов суперклассов

Инициализация полей экземпляра значениями по умолчанию или начальными значениями

Выполнение логических блоков инициализации

Выполнение тела конструктора класса

Способы инициализации полей

Пример 7:

```
public class InitDemo6 {  
    private int a = 5;  
    private static int b = 100;  
    {  
        a = -5;  
        System.out.println("Logical block");  
    }  
    public InitDemo6() {  
        a = 10;  
        System.out.println("Constructor");  
    }  
    static {  
        b = -5;  
        System.out.println("Static block");  
    }  
    public static void main(String[] arg) {  
        System.out.println("Main");  
        InitDemo6 obj = new InitDemo6();  
        System.out.println("a=" + obj.a);  
    }  
}
```

Переменная экземпляра

Переменная класса

Вывод в консоли:

Static block
Main
Logical block
Constructor
a=10

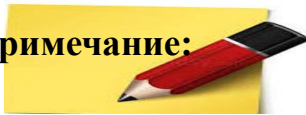


Инициализация поля типа final

Инициализация поля типа `final`

- должно быть инициализировано в той же строке, в которой и объявлено;
- или должно быть инициализировано в каждом конструкторе;
- или должно быть инициализировано в одном из логических блоков класса.

Примечание:



Потому что, переменная типа `final` может быть инициализирована только один раз.

Инициализация поля типа final

Пример 8:

```
public class InitDemo7 {  
    private final int xx = 50;  
    private final int zz;  
    private final int yy;  
    {  
        zz = 20;  
        System.out.println("Non-static block");  
    }  
    public InitDemo7() {  
        yy = 30;  
        System.out.println("Constructor");  
    }  
    public static void main(String[] arg) {  
        System.out.println("Main");  
        InitDemo7 obj = new InitDemo7();  
    }  
}
```

Явная инициализация
выражением

Инициализация в
логическом блоке

Инициализация в
конструкторе

Инициализация поля типа final

Пример 9:

```
public class InitDemo8 {  
    private final int xx = 50;  
    {  
        xx = 20;  
        System.out.println("Non-static block");  
    }  
    public InitDemo7() {  
        xx = 30;  
        System.out.println("Constructor");  
    }  
    public static void main(String[] arg) {  
        System.out.println("Main");  
        InitDemo8 obj = new InitDemo8();  
    }  
}
```

Ошибка компиляции:
нельзя изменить
переменную типа final

Ошибка компиляции:
нельзя изменить
переменную типа final

Инициализация поля типа final

- ❑ Поле типа final обязательно должно быть проинициализировано явно (значение по умолчанию не устанавливается):

Пример 10:

```
public class InitDemo9 {  
    private final int xx;  
    {  
        System.out.println("Non-static block");  
    }  
    public InitDemo7() {  
        System.out.println("Constructor");  
    }  
    public static void main(String[] arg) {  
        System.out.println("Main");  
        InitDemo9 obj = new InitDemo9();  
    }  
}
```

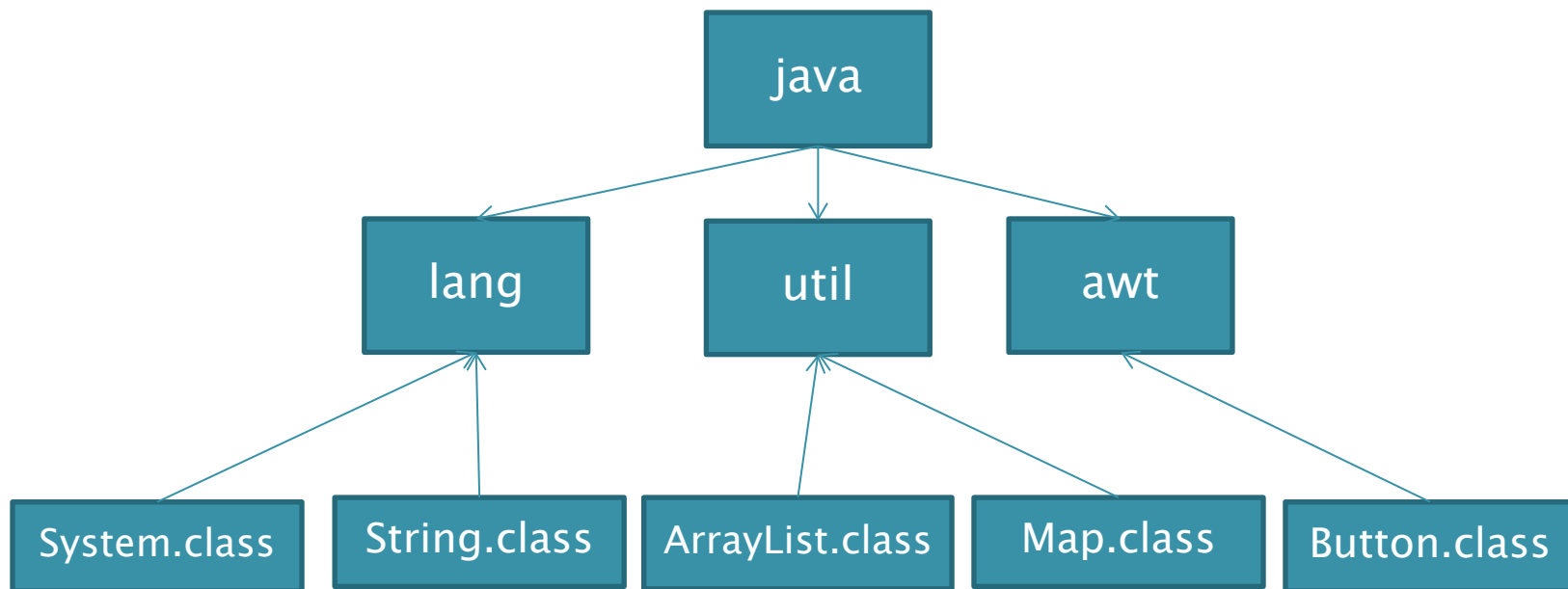
Ошибка компиляции:
переменная типа final
должна быть явно
инициализирована



Пакеты

❑ **Java пакеты** – это механизм группирования классов/интерфейсов Java, которые связаны друг с другом, в один "модуль" (пакет).

- Разделение классов на пакеты – улучшает читабельность и понимаемость исходного кода.
- Пакеты поддерживают иерархическую организацию и используются для организации больших программ в логические и управляемые единицы.



❑ **Java пакеты** – это пространства имен, которые позволяют разработчикам создавать закрытые (локальные) области, в которых можно объявлять классы/интерфейсы:

- имена этих классов не будут конфликтовать с идентичными именами классов в других пакетах, т.к. полное имя класса указывается с именами пакетов

Например, *java.lang.Math*.

Описание пакета: **package** <имя пакета>;

Иерархия пакетов:

package <pk>{.<pk>}₀^N;

Вложенный пакет

Доступ к классу из другого пакета:

com.mypack.Test;

Полное квалификационное
имя класса

Импортирование классов:

import <pk>{.<pk>}₀^N.<имя класса>|*;

Конкретный класс

Все классы пакета

Пример 11: использование класса из другого пакета

```
package com.knu.test;  
  
public class Main {  
    public static void main(String[] arg) {  
        com.knu.teststuds.Student myStud1 =  
            new com.knu.teststuds.Student("Peter");  
        System.out.println(myStud1.getFirstName());  
    }  
}
```

↑
Полное имя

Вывод в консоли:
Peter

Примечание:



Если классу Main необходимо использовать класс Student, то нужно сослаться на класс Student внутри класса Main.

Пример 12: использование импорта

```
package com.knu.test;  
  
import com.knu.teststuds.*;  
  
public class Main {  
    public static void main(String[] arg) {  
        Student myStud1 = new Student("Peter");  
        System.out.println(myStud1.getFirstName());  
    }  
}
```

Импорт всех классов
пакета, указанного
последним

Вывод в консоли:
Peter

Примечание:



Если в указанном последнем пакете кроме классов присутствуют вложенные пакеты, то они не импортируются

Статический импорт

- ❑ *Статический импорт* позволяет ссылаться на статические элементы одного класса в другом без указания полного пути (*т.е. удалить шаблонное повторение имени класса*).

Когда необходимо использовать статический импорт?

Когда вам требуется частый доступ к статическим членам из одного/двух классов.

Например,

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

или

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

Пример 13: статического импорта элементов класса

```
package com.knu.test;
```

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

```
public class Main {
```

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

```
        double r = 3;
```

```
        System.out.println(2 * PI * r);
```

```
        System.out.println( floor( cos(PI) ) );
```

```
    }
```

```
}
```

Статический импорт
всех статических
элементов класса **Math**

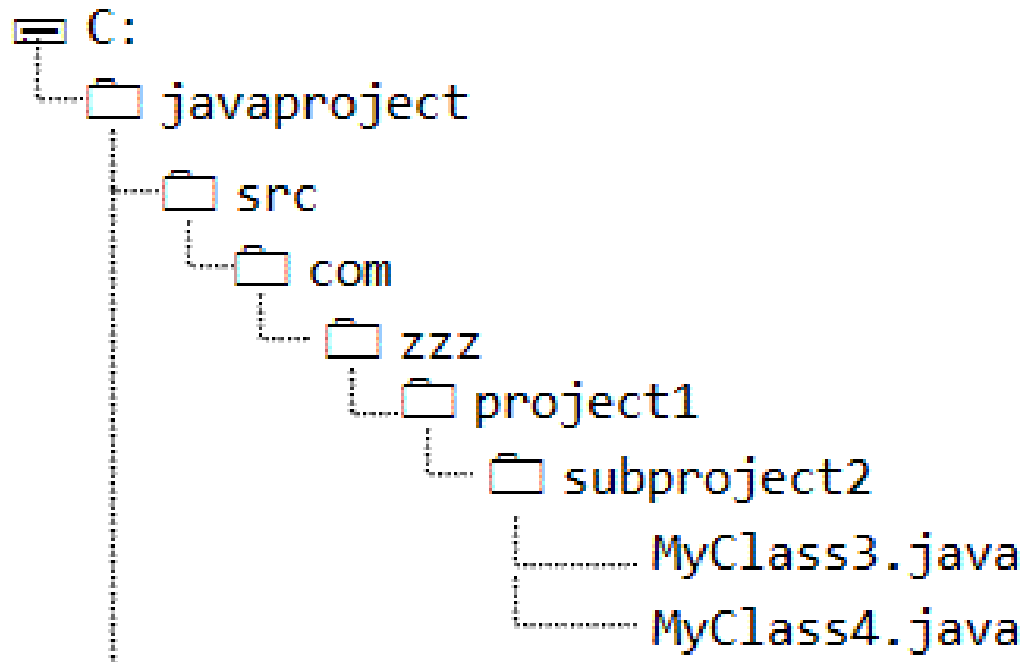
Без указания
принадлежности
классу

Организация пакетов

Для предотвращения конфликтов имен пакетов, есть соглашение об именовании пакетов

- ❑ Компании должны использовать свои зарезервированные доменные имена в Интернете, чтобы с них начинать имена пакетов. *Например, com.knu.test.*
- ✓ Если доменное имя в Интернет содержит недопустимый символ (*например, дефис*), то его нужно заменить символом подчеркивания;
- ✓ Если компонент доменного имени начинается с цифры или содержит зарезервированное ключевое слово Java, то нужно добавить подчеркивание в имя компонента.

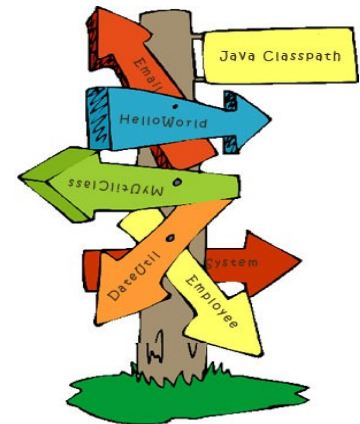
- ❑ Пакеты тесно связаны со структурой каталогов, используемой для хранения классов.



Например, класс **MyClass4** пакета **com.zzz.project1.subproject2** хранится в **"\$BASE_DIR\com\zzz\project1\subproject2\MyClass4.class"**, где **"\$BASE_DIR"** - обозначает базовый каталог пакетов.

Java пакеты

- ✓ Базовый каталог (**\$BASE_DIR**) может быть расположен в любом месте файловой системы.
- ✓ Компилятор и исполнительная система Java должны быть проинформированы о локализации (размещении) **\$BASE_DIR**, чтобы найти классы.
- ✓ Достигается это с помощью переменной среды окружения виртуальной машины называемой **CLASSPATH**.
- ✓ **CLASSPATH** похож на переменную **PATH** среды окружения, которая используется в операционной системе для поиска исполняемых программ




Java пакеты

- ❑ При создании пакета всегда следует руководствоваться простым правилом: называть его именем простым, но отражающим смысл, логику поведения и функциональность объединенных в нем классов.

Например,

```
com.knu.administration.constants  
com.knu.administration.dbhelpers  
com.knu.common.constants  
com.knu.common.dbhelpers.annboard  
com.knu.common.dbhelpers.courses  
com.knu.common.dbhelpers.guestbook  
com.knu.common.dbhelpers.learnres  
com.knu.common.dbhelpers.messages  
com.knu.common.dbhelpers.news  
com.knu.common.dbhelpers.prepinfo  
com.knu.common.dbhelpers.statistics  
com.knu.common.dbhelpers.subjectmark  
com.knu.common.dbhelpers.subjects
```

.....

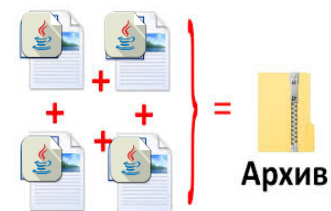


Упакованные архивы. Утилита jar

Java Archive (JAR) – это независимый от платформы формат файла, который позволяет сжимать и объединять несколько файлов, связанных с Java-приложением, в один файл.

Достоинства:

- Сжатие данных (с помощью алгоритма ZIP);
 - Легкость в распространении (передача одного большого файла по сети вместо множества мелких файлов идет быстрее и более эффективно);
 - Аутентификация: JAR-файл может быть подписан цифровой подписью автора.
- ❑ Исполнительная среда Java (JRE) или Java-приложения могут загружать классы из jar-файла непосредственно, без необходимости явного (предварительного) распаковывания jar-файла.



- ❑ Для поддержки таких функций JAR, как:
 - специфицирование точки входа в java-программу;
 - создание цифровой подписи;
 - контролирование версий программы
- используется файл, называемым манифестом.
-
- ❑ Манифест – это специальный файл (имя по умолчанию - **MANIFEST.MF**), находящийся в директории "META-INF" и содержащий информацию о файлах, находящихся в jar-файле.



Например,

1. Manifest-Version: 1.0
2. Created-By: 1.7.0_06 (Oracle Corporation)
3. Main-Class: MyPackage.Main
- 4.

Пустая строка
(обязательно)

Атрибут,
определяющий класс с
методом main()

- Если jar-файл имеет точку входа, он является *исполнительной jar-программой*, которую можно запустить из командной строки:

java -jar <имя файла>.jar

Создание jar-файла

- ❑ В состав JDK входит утилита *jar.exe*, с помощью которой и происходит упаковывание приложений Java:

jar <ключи> <имя jar-файла> [<имя главного класса>]
<список классов>

Ключи:

c – создать новый JAR-файл;

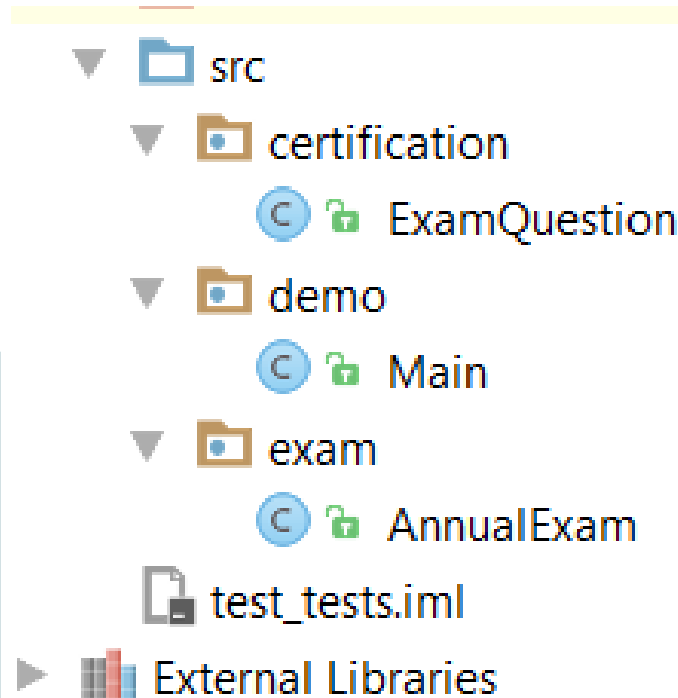
f – имя JAR-файла указывается первым в списке;

e – включить в манифест атрибут Main-Class, имя которого указывается вторым в списке;

v – выводить информацию о всех своих действиях.

Упакованные архивы. Утилита jar

Например,



Полные имена классов, начиная от базовой директории и с учетом разделителей операционной системы

Полное имя класса с точкой входа, пакеты и класс отделяются точкой

```
jar cvfe app.jar demo.Main
certification/ExamQuestion.class
exam/AnnualExam.class demo/Main.class
```



Документирование кода

Документирование кода

- ❑ В языке Java используются блочные и однострочные комментарии `/* */` и `//`, аналогичные комментариям, применяемым в C++.
- ❑ Введен также новый вид комментария `/** */`, который может содержать дескрипторы вида:
 - **@author** – задает сведения об авторе;
 - **@version** – задает номер версии класса;
 - **@exception** – задает имя класса исключения;
 - **@param** – описывает параметры, передаваемые методу;
 - **@return** – описывает тип, возвращаемый методом;
 - **@deprecated** – указывает, что метод устаревший и у него есть более совершенный аналог;
 - **@since** – с какой версии метод (член класса) присутствует;
 - **@throws** – описывает исключение, генерируемое методом;
 - **@see** – что следует посмотреть дополнительно.

Документирование кода

```
public class User {  
    /**  
     * personal user's code  
     */  
    private int numericCode;  
    /**  
     * user's password  
     */  
    private String password;  
    /**  
     * see also chapter #3 "Classes"  
     */  
    public User() {  
        password = "default";  
    }  
    /**  
     * @return the numericCode  
     * return the numericCode  
     */  
    public int getNumericCode() {  
        return numericCode;  
    }  
}
```

Пример 14:

```
/**  
 * @param numericCode the numericCode to set  
 * parameter numericCode to set  
 */  
public void setNumericCode(int  
numericCode) {  
    this.numericCode = numericCode;  
}  
/**  
 * @return the password  
 * return the password  
 */  
public String getPassword() {  
    return password;  
}  
/**  
 * @param password the password to set  
 * parameter password to set  
 */  
public void setPassword(String  
password) {  
    this.password = password;  
}  
}
```

Результат:

chapt02

class User

java.lang.Object
|__chapt02.User

```
public class User
extends java.lang.Object
```

Filed Summary

<code>private int</code>	<u>numericCode</u> personal user's code
<code>private java.lang.String</code>	<u>password</u> user's password

Constructor Summary

User()
see also chapter #3 "Classes"

Method Summary

<code>int</code>	<u>getNumericCode</u> ()
<code>java.lang.String</code>	<u>getPassword</u> ()



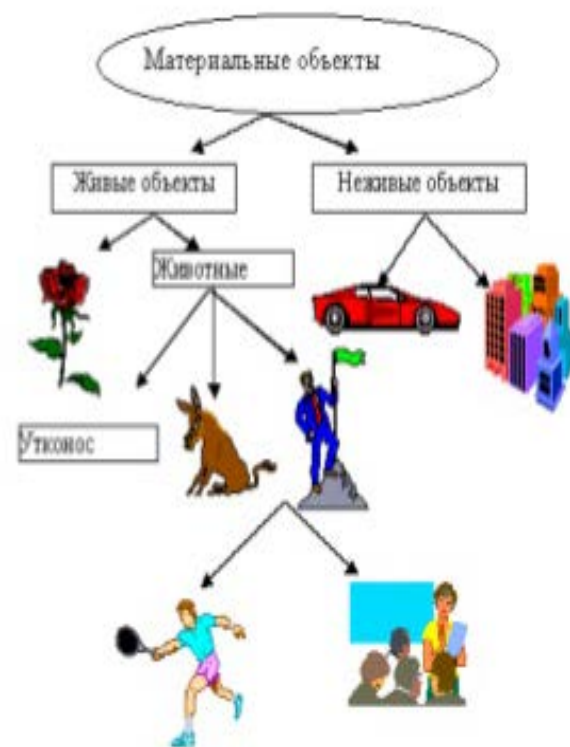
Реализация принципа ООП: наследование

Основы наследования

- ❑ **Наследование** — это возможность, позволяющая описать новый класс на основе уже существующего с частично или полностью заимствующимися свойствами.
- ❑ **Наследование** — это возможность создавать классы со свойствами, общими для набора связанных групп объектов.

Примечание:

- Класс, который является производным от другого класса, называется **подклассом** (а также расширением класса).
- Класс, от которого происходит подкласс, называется **суперклассом** (а также базовым классом или родительским классом).

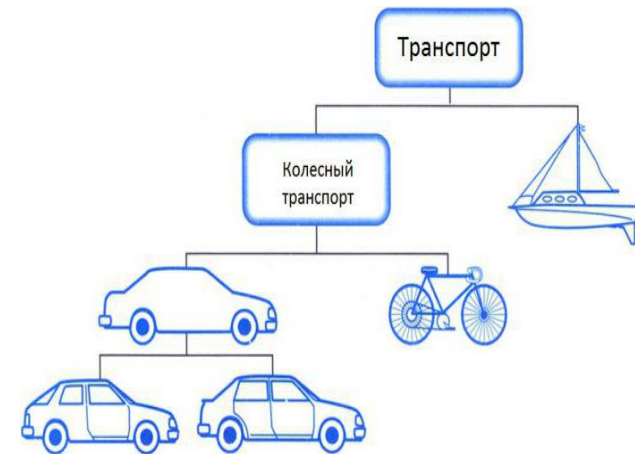


Основы наследования

Например,

Класс – новый автомобиль (Car);

Класс – подержанный автомобиль
(пробег, функция вычисления
стоимости с учетом пробега)



- ❑ Подкласс полностью удовлетворяет спецификации суперкласса, однако может иметь дополнительную функциональность.
 - *С точки зрения интерфейса (открытые методы класса), каждый подкласс полностью реализует интерфейс суперкласса. Обратное не верно.*

Основы наследования

□ Когда применяется наследование?

- если можно сказать, что объект Б – является разновидность объекта А.

водоем	←	озеро
осёл	→	животное
цветок	←	ромашка
автомобиль	← 	двигатель

Двигатель – это часть автомобиля, а не разновидность!

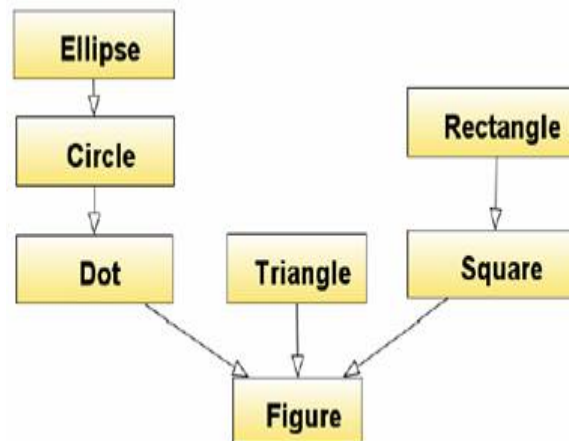
автомобиль	←	грузовик
косточка	→ 	ягода



Наследование позволяет приобретать свойства родителей

Преимущества наследования: *повторное использование.*

- ❑ После того, как поведение (метод) определяется в суперклассе, это поведение автоматически наследуется всеми подклассами (*таким образом, вы пишете метод только один раз и он может быть использован всеми подклассами*);
- ❑ После того, как набор данных (поля) определен в суперклассе, этот же набор свойств наследуется всеми подклассами (*класс и его дочерние классы разделяют общий набор характеристик*);
- ❑ Подклассу нужно только реализовать различия между собой и родителем.



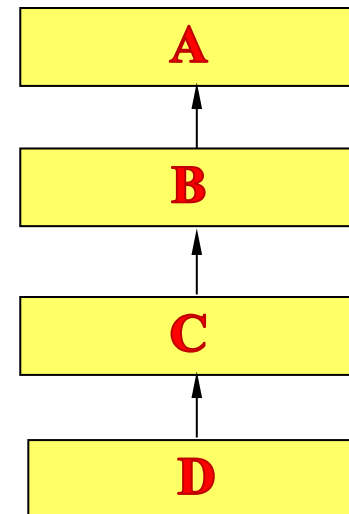
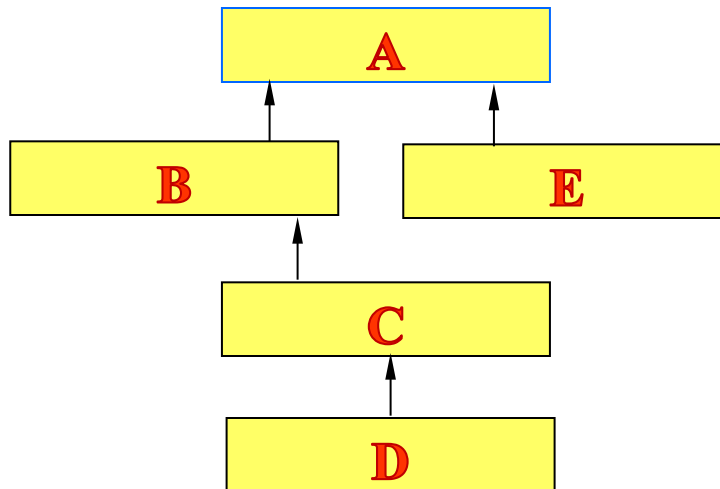
Основы наследования

Существуют две формы наследования:

- *Одинокое* (один родительский класс);
- *Множественное* (два и более родительских классов).



В Java реализовано только одинокое наследование.



Цепочки наследования