J2SE

Thursday, May 17, 2018

11:11 AM

**Java SE 第一讲 (2010.11)**

Java SE: Java Standard Edition -- J2SE

Java ME: Java Micro Edition -- J2ME

Java EE: Java Enterprise Edition --J2EE

Java是由Sun公司推出的(今年被Oracle公司以$7.4B 收购)

JDK: Java Development Kit (Java开发必备)

JRE:　Java Runtime Environment(Java执行环境)

JDK包含了JRE

JDK的版本: JDK1.4, JDK1.5(5.0):Tiger, JDK1.6(6.0):Mustang 野马

Download JDK:

[www.sun.com](http://www.sun.com) => Downloads => Java for Developer => Java SE87171 =>Download=> [jdk-8u171-windows-x64.exe](http://download.oracle.com/otn-pub/java/jdk/8u171-b11/512cd62ec5174c3487ac17c61aaa89e8/jdk-8u171-windows-x64.exe)

double-click to install, can be installed on Drive D:\Program Files\Java there will be two folders: JDK1.6.0 and JRE1.6.0

\jdk1.6.0\bin\java.exe and javac.exe 后者用于编译java程序, 编译后生成一个bytecode(字节码)文件, 改文件由java.exe执行

需要设置用户或者系统环境变量 $PATH=\jdk1.6.0\bin\;$PATH

运行command line: java -version 显示java版本信息

接下来就可以编写Java程序了  
可以直接使用Windows记事本来编写Java程序, 也可以使用Editplus, UltraEdit等高级文本编辑工具编写Java 源程序, 还可以  
使用专业的IDE(Integrated Development Environment)编写.

第一节课, 我们使用记事本来实现一个Hello World的java程序.

**所有的Java代码都以java后缀结束**.

Java程序的执行分为两部分:

1.编译

C:\>javac H:\Java\Java\_Excercise\Test.java //注意,文件名需要与类名一致, Java程序case-sensitive

//若编译成功, 会在同目录下生成同名的class文件.该文件  
 //为字节码(bytecode)文件, 可直接执行

2. 执行

C:\>java **-cp H:\java\Java\_Excercise** **Test** //执行时,无需加.class后缀

Hello World // -cp dirs file 没有-cp option 会在**$CLASSPATH**下找文件

Java是跨平台的语言, 真正执行的不是二进制代码(exe文件),而是字节码. 字节码由JVM(Java Virtual Machine)来执行.  
Java是跨平台的,而JVM不是跨平台的. JVM由c语言编写. Java之所以能够跨平台,本质原因在于JVM不是跨平台的.

**Jave SE 第二讲 原生数据类型(Primitive Data Type)**

Windows: notepad, editpulus, ultraedit, gvim

Linux: vi, vim, gedit

**Java中的数据类型分为两大类**

* 1. **原生数据类型(Primitive Data Type)**
  2. **引用类型(对象类型 Reference Type)**

**变量与常量**

所谓常量就是值不会变化的量. 变量就是值可以发生改变的量.

**如何定义变量?**

变量类型 变量名 ;

例: int a ;

a = 2 ; //给整型变量a赋值

a == 2 ; // 连续两个等号表示逻辑操作符相等

|  |  |
| --- | --- |
| **变量类型 变量名 = 值 ;** | **int a = 1 ;** |

**变量名: 在Java中, 变量名以下划线, 字母, $符号开头, 并且后跟下划线, 字母, $符或者数字. 变量名不能以数字开头.**

**Jave中的原生数据类型共有8种:**

* 1. **整型: INT** ( 4 bytes 32 bits) 2^31 -1, 大致9位数字
  2. **字节型: BYTE** (-128 - 127之间256个整数) 127
  3. **短整型: SHORT** (2 bytes - 16位) 32767 = 2^15 - 1
  4. **长整型: LONG** (8 bytes, 64 bits)
  5. **单精度浮点型: FLOAT** 浮点型指的是小数/实数 2表示为2.00000001
  6. **双精度浮点型: DOUBLE** 双精度表示的范围比单精度要大 2表示为2.000000000000001
  7. **字符型: CHAR(2 bytes)** 表示单个字母,或者一个中文字符,比如, 'A', '张', 用单引号括起来.可表示所有的UniCode字符
  8. **布尔类型: BOOLEAN** 表示TRUE或FALSE

**关于计算机系统中的数据表示**

位:bit(0或者1) 是计算机系统中的最小数据表示单位.

字节: byte, 1 byte = 8 bits

KB: 1KB = 1024 Bytes

MB: 1MB = 1024KB

GB: 1GB = 1024MB

Java中的注释

注释是给人看的, 不是给计算机看的. 编译器会自动忽略注释. Java中共有三种类型的注释

1) 单行注释: // 其后的所有内容均被忽略

2) 多行注释: /\* this is comment \*/ 多行注释来源于C++(C中的唯一注释是多行注释), **多行注释不允许嵌套**.

3) 另一种多行注释.用于产生JavaDoc帮助文档. /\*\* this is another comment \*/

**Jave SE 第三讲 原生数据类型使用陷阱(Pitfall of Primitive Data Type)**

* 1. **Java中的所有的浮点数默认都是DOUBLE**

float a = 1.2 ; //出错, 会损失精度, 因为1.2为Double类型

float a = 1.2f ; 或者 强制类型转换: (类型) 变量名

float a = (float)1.2 ;

* 1. **变量使用前必须声明类型且必须要初始化.变量只能定义一次. Java is a strong typed language.**

int a;

System.out.println(a); //出错.

int a=1;

int a=2;

System.out.println(a); //重复定义,出错.

* 1. **范围大的不能直接赋值给范围小的, 因为会损失精度. 反之则可以或者进行强制类型转换.**

int a=1;

short b=a;

System.out.println(b); //Error, 范围大的赋值给范围小的会损失精度

short a=1;

int b=a;

System.out.println(b); //Ok

**Jave SE 第四讲 Operator 运算符**

* 1. **当有若干变量参与运算时, 结果类型取决于这些变量中表示范围最大的那个变量类型.**

比如, 参与运算的变量为整型和双精度浮点型以及短整型, 最后结果的类型为Double

int a = 1;

int b = 2;

float c = (float)a / b ;

double(a)将a转化为一个匿名变量, 当a的类型不变, 仍为整型.

* 1. **取模(取余)运算符 %**

**取模的结果永远与被除数的符号相同, 且绝对值小于被除数**

5 % 3 = 2 ; -5 % 3 = -2 ; -1余-2

5 % (-3) = 2 ; +1余2

* 1. **关系运算符(比较运算符 comparasion operator)**

大于>, 小于<, 等于==, 不等于 != ,大于等于>=, 小于等于<= , 关系运算的结果是一个布尔值

关系运算符优先级高于赋值运算符

int a = 3;

int b = 2;

boolean c = a == b ;

System.out.println(c); //output false

* 1. **逻辑运算符 (logic operator)**

**重点介绍两个(对于语言的学习,刚开始不必面面俱到, 刚开始重点掌握常用的,了解其前因后果 )** 逻辑运算符本身也返回一个boolean值.逻辑运算符有两个操作数, 为双目运算符

逻辑与 &&: 只有当两个操作数都为真时才为真, 两个操作数均为布尔型

**关系运算符优先级高于逻辑运算符高于赋值运算符**

int a = 1;

int b = 2;

int c = 3;

int d = 4;

boolean **e = a < b && c < d** ;

System.out.println(e); //output true

逻辑或||: 双目运算符, 当两个操作数都为假时,结果才为假, 否则为真

**逻辑运算短路(short circuit)**特性:

**当第一个operator为假, && 不再计算第二个操作数值**

**同理, 当第一个operator为真, || 不在计算第二个操作数的值**.

int a = 1;

int b = 2;

int c = 3;

int d = 4;

int f = 5;

boolean e = a > b && (f = c) < d ;

System.out.println(e);

System.out.println(f); // output false 5 f没有被赋值

* 1. **变量的自增自减运算**

a++, ++a 等价于 a+1 , a += b 等价于 a = a + b ; a += 1 等价于 a = a + 1;

int a = 1

int b = a++ ; // b=1, a=2

int b = ++a ; // b=2, a=2

* 1. **条件运算符(三元表达式)**

type var = a?b:c; 如果a为真, 则b赋值给var, 否则将c赋值给var

i.e. int d = 1 < 2 ? 3 : 4 ; // d = 3

int d = 1 > 2 ? 3 : 4 ; // d = 4

**Jave SE 第六讲 流程控制语句 (Flow Control Statement)**

**顺序, 分支与循环**

**if语句**

if (布尔表达式)

{

//待执行代码

}

**If的第二种形式**

if (布尔表达式)

{

//待执行代码

}

else

{

//待执行的代码

}

**If的第二种形式**

if (布尔表达式)

{

//待执行代码

}

**else if** (布尔表达式

{

//待执行的代码

}

…

**else** //optional

{

//待执行的代码,

}

**Switch语句, 使用形式为**

switch (variable){ //此处变量类型只能为4种: byte, short, int, char

case 常量1:

//待执行代码

break;

case 常量2:

//待执行代码

break;

case 常量3:

//待执行代码

break;

default:

//待执行代码

}

**虽然case语句中的break是可选的, 但绝大多说情况下, 如果没有break, 程序的逻辑就会错误, 因此通常情况都  
要加上break**

**循环语句:　Java中共有三种循环语句 while, do … while 以及 for 循环**

**while循环, 形式为**

while (布尔表达式)

{

//待执行代码

}

**do … while 循环, 形式为:**

do{

//待执行代码

}

while(布尔表达式) ; //布尔表达式为真时, 执行do中的语句

while与do…while之间的区别: 如果布尔表达式第一次判断就为false, 那么while循环一次也不执行.而do…while  
至少会执行一次. 如果布尔表达式第一次判断为true, 则两者等价

**for循环,也是使用得最多一种循环**

for(变量初始化;条件判断;步进)

{

//待执行代码

}

* 1. 执行变量初始化
  2. 执行条件判断, 如果判断结果为假,那么退出循环, 执行循环后面的代码. 如果为真, 执行循环里面的代码
  3. 执行步进
  4. 重复步骤ii

**break语句， 经常用于循环语句中，用于跳出父循环， 执行循环后面的代码**

for(int i = 0; i < 10; i++){

System.out.println(i);

if(5 == i){

break;

}

} //output 0 1 2 3 4 5

**continue语句， 经常用于循环语句中，用于跳出本次循环， 开始下一次循环的执行**.

for (int i = 0; i < 10; i++ ){

if ( 5 == i)

{

continue ;

}

System.out.println(i);

} // output 0 1 2 3 4 6 7 8 9

**break和continue还可以搭配标签使用， 但在实际的开发中根本没有人会将break和continue搭配标签来使用。  
所以不建议使用。**

**Jave SE 第八讲 深入理解面向对象程序设计(Inside Object Oriented Programming)**

面向对象编程OOP， 面向对象设计OOD

什么是面向对象？在面向对象程序设计中， 有两个重要的概念：类（class）与对象（object）

类是一种抽象概念， 类中包含了数据与对数据的操作。类通常使用名词来表示，而对数据的操纵常用动词来表示。

比如人就是一个抽象的概念， 人具有姓名，年龄，身高等数据， 还有吃饭，跑步等操作数据的动作。

**对象**： 对象是一种具体的概念，是类的一种具体的表示方式。比如，人是 一个类， 而张三，李四，王五等具体的人  
 就是对象。**对象就是类的实例(Instance)**

类所包含的内容： 类一共包含两部分的内容：

a） 数据， 数据在类中成为属性（property 或**attribute**），或者成为成员变量（**Member Variable**）

b) 方法(**Method**)： 方法一般用动词来表示. 方法可以操作属性。

面向对象程序设计的三大特征：继承（**inheritance**）封装（**encapsulation**）和多态（**polymorphism**)

封装：类包含了数据与方法， 将数据和方法放在一个类中就构成了封装

如何定义类？

修饰符(modifier) class 类的名字

｛

//类的内容（包含了属性与方法）

｝

方法； 如何定义方法？

修饰符（modifier） 返回类型 方法名称([参数1， 参数2， 参数3， …])

{

//方法体

}

main 方法是整个Java程序的入口点。 如果类的定义中没有main方法， 则程序无法执行。

**方法定义不能嵌套， 不能在一个方法中定义另外一个方法 只能调用另外一个方法。**

**Java是面向对象的， 方法只能定义在类中。C++方法可以定义在类外面。**

**方法的调用，方法可以返回值也可以不返回值。是否返回值是由方法的定义决定的。**

**如何生成对象？**通过类来生成对象。 通常使用new关键字来生成对象**。**

**类名 变量名 ＝ new 类名() ;**

public class Person

{

};

Person person = new Person();

Person person2 = new Person（）；

Person person3 = new Person() ;

方法调用需要通过对象来完成。 方法调用的形式

对象.方法名（[参数名1，参数名2，参数名3 …])

关于方法的注意事项：

1）在方法定义中， 方法的返回类型要与return后面的变量或者常量的类型一致。

2） 在方法调用时， 给方法传递的参数需要与方法定义时的参数保持一致，且参数的个数也要一致。

3） 方法定义时的返回类型与接受返回值的变量的类型要保持一致。

public int add(int a, int b)

{

return a + b ;

}

方法定义时的参数成为形式参数（formal parameter), 比如上面的 int a, int b

int a = test.add(8,3) ; //方法调用时传入的参数称为实际参数（actual parameter），比如这里的8和3；

关键字void表示方法不返回值。

如果方法不返回值， 那么申明方法的时候使用void关键字， 在方法定义中可以有两种情况实现不返回值：

a）**不使用return语句**

b）**但是return后面没有任何值或者变量， 只有一个分号， 表示退出方法，返回到方法调用端。**

**Jave SE 第十讲**

* 1. 类中的属性又叫做成员变量（member variable）， 属性用英文表示为property或者attribute
  2. 对象（object） 又叫做实例（instance）。 生成一个对象的过程叫做类的实例化。
  3. 命名约定
     1. 类： 首字母大写，如果由多个单词构成， 每个单词的首字母都大写，中间不使用任何的连接符，比如

Person类， Member类

* 1. 方法： 首字母小写。如果一个方法名由多个单词构成，那么第一个单词的字母都小写， 从第二个单词开始，  
      每个单词的首字母大写。比如， addThreeInt
  2. 属性： 命名约定与方法完全一样。 比如， ageOfPerson
  3. 属性需要定义在类中，又叫做**成员变量**。 方法中的变量叫做**局部变量**。
  4. 如何定义属性？

public class Person

｛

modifier type attribute\_name ;

｝

使用属性与方法一样。 使用.运算符。 首先需要生成类的实例， 然后使用实例+"."的方式来使用

比如：

Person person = new Person();

person.age

* 1. 成员变量与局部变量的联系和区别
     1. 局部变量使用前必须要声明并赋初值才能使用；成员变量必须要声明但可以不赋初值。
     2. 两者都需要声明(定义)
     3. 如果没有初始化成员变量就开始使用, 对于每个类型的成员变量都有一个默认的初始值.
        1. byte, short, int, long 类型的默认初始值为0
        2. float, double类型的默认初始值为0.0
        3. char类型的默认初始值 '\u0000'
        4. boolean类型的默认初始值为false
  2. 引用类型(reference type): **引用类型用于指向对象. 一个对象可以被多个引用所指向, 但同一个时刻,每个引用  
      只能指向唯一的一个对象.如果一个对象被多个引用所指向,那么无论哪个引用对对象  
      的属性进行了修改,都会反映到其它的引用中**.

Person person = new Person(); // 这里, new Person()为对象实例, 而person为对对象实例的引用.

**注意: 在Java中, 你永远都无法直接操作内存中的对象,都是通过引用来间接操作**.

**Jave SE 第十一讲**

* 1. 如果一个类包含了属性与方法, 那么该类的每一个对象都具有自己的属性,但无论一个类有多少个对象, 这些对象共享  
     同一个方法..(方法放在内存中一个唯一的内存区)
  2. 对于方法参数传递的总结, **对于Java中的方法参数传递, 无论传递的是原生数据类型还是引用类型, 统一是传值**

**(pass by value**)

* 1. **类方法的参数应该避免与成员参数同名**, 比如如下代码:

class Person{

int age = 20;

public void change(Person person){

person = new Person() ;

person.age = 30;

}

// change2中的参数如果为age, 则下面的age会作为

// 局部变量而非成员变量.

// 因此, 成员方法的参数应该避免与成员变量同名

public void change2(int value1){

age = 40 ;

}

}

* 1. 什么类型的引用就能指向什么类型的对象, 比如People类型的引用就能指向People类的对象, 但不能指向Student  
     类型的对象. 比如  
       
      People people = new People() ; // 正确

People people = new Student() ; // 错误

* 1. 构造方法(Constructor): 用于完成对象属性的初始化工作, 构造方法的特点:
     1. 构造方法的名字必须与类名完全一致(包含大小写).
     2. 构造方法没有返回值, 连void也不能出现.
     3. 如果在定义一个类的时候, 没有为类声明构造方法, 那么Java编译器会自动为类添加**一个没有参数且方法体为空  
        的构造方法**, 我们称之为**默认的构造方法. (default constructor)**

Class Person {

…

public Person()

{

} // default constructor

}

* 1. 如果在定义一个类时, 为类声明了构造方法, 那么Java编译器就不会再为类添加构造方法了.
  2. 不能显示调用类的构造方法, 通常只能通过new关键字进行隐式调用(implicitly invoke)

* 1. New关键字在生成对象时需按如下顺序完成三件事情:
     1. **为对象开辟内存空间**
     2. **调用类的构造方法**
     3. **将生成对象的地址返回**.

* 1. 使用new来生成对象时侯, 后面的括号()表示构造方法的参数列表. 如果构造方法不接收参数, 那么小括号的内容为空.

如果构造方法接受参数, 那么小括号中的实际参数就需要与构造方法定义的形式参数一致(参数数量一致, 参数  
 类型一致, 按照顺序逐一赋值)

**Jave SE 第十四讲**

* 1. 在Java中要调用方法只能先生成类的实例, 然后通过该实例来调用方法.
  2. return; 表明方法调用结束, 控制返回到调用的程序

**Jave SE 第十五讲**

* 1. 如果一个Java源文件中定义了多个类, 那么最多只能有一个类是用public修饰的.
  2. 对象存储在堆heap中, 而原生变量类型都存储在栈(stack)中. 栈是后进先出(LIFO)的数据结构
  3. 对象方法的参数类型如果为一个类, 则可认为传入该方法的为一个指向该类对象的指针,比如  
      public void changePoint(Point point); //传入一个指向Point类对象的指针(地址的值)
  4. new方法相当于将对象变量指向一个新生成的对象.
  5. 对于对象类型的变量赋值相当与改变该对象的指针指向地址.

**Jave SE 第十六讲**

* 1. 构造方法(Constructor)

public class ConstructorTest{

public ConstructorTest()

{

// default constructor, if no other constructor,

// Java will add it automatically

}

public static void main(String[] args){

// how to invoke the construtor to create a new instance

// within the class definition

ConstructorTest test = new ConstructorTest() ;

}

}

* 1. 方法重载(Overload): 表示两个或多个方法名字相同, 但方法参数在**类型**, **顺序**或者**个数**上不同.

注意, 方法的返回值对是否重载没有影响.例子:

public class OverloadTest{

public int add(int a, int b){

return a + b ;

}

public int add(int a, int b, int c){

return a + b + c ;

}

public static void main(String[] args){

OverloadTest test = new OverloadTest();

int result = test.add(1,2) ;

int result2 = test.add(1,2,3);

System.out.println(result) ;

System.out.println(result2) ;

}

} // output 3 6

* 1. 构造方法的重载. 只需看参数即可. 对于重载的构造方法, 它们之间也可以互相调用. 如果希望在一个构造方法中调用另  
     外一个构造方法, 那么可以使用this()的方式调用. this()括号中的参数表示目标构造方法的参数.

this()必须要作为构造方法的第一条语句, 即, this()之前不能有任何可执行语句.

public class ConstructorOverload{

public ConstructorOverload(){

this(3) ; //this 表示调用当前类中参数为整数的构造方法

System.out.println("test") ;

// this 必须为方法中的第一条可执行代码, 如下语句会报错

// this(3) ;

}

public ConstructorOverload(int i){

System.out.println(++i) ;

}

public static void main(String[] args){

ConstructorOverload con = new ConstructorOverload() ;

}

}

* 1. 继承(Inheritance): Java是单继承的, 意味着一个类只能从另一个类继承. 被继承的类称为父类或者基类(superclass)

继承的类称为子类(subclass), Java中的继承使用extends关键字.

* 1. 当生成子类对象时, Java默认首先调用父类的默认构造方法. 从而生成父类的对象, 接下来再调用子类的构造  
     方法生成子类对象. 即, 要想生成子类的对象, 首先需要生成父类对象, 没有父类对象就没有子类对象.
  2. super关键字: super表示对父类对象的引用, this表示对自己的引用.

* 1. 如果子类使用super()显示的调用父类的某个构造方法, 那么执行的时候就会寻找super()所对应的构造方法而不再  
     使用父类默认的构造方法.而super也必须要作为构造方法的第一条执行语句, 前面不能有其它任何可执行语句.

* 1. 关于继承:
     1. 父类有的, 子类也有
     2. 父类没有的, 子类可以增加
     3. 父类没有的, 子类可以改变

* 1. 关于继承的注意事项:
     1. 构造方法不能继承
     2. 方法和属性都可以被继承
     3. 子类的构造方法隐式地调用父类的默认构造方法
     4. 当父类没有默认的构造方法时, 子类需要使用super来显式地调用父类的构造方法, super指的对父类的引用.
     5. super关键字必须是构造方法的第一行语句.

* 1. 方法重写(**Override**): 子类与父类的方法**返回类型**, **名称**以及**参数完全**一致. 这样我们说子类与父类的方法就构成了  
      重写关系. 例子:

public class InheritanceTest2{

public static void main(String[] arg){

Dog dog = new Dog() ;

dog.run(1);

}

}

class Animal{

public void run(){

System.out.println("animal is running") ;

}

}

class Dog extends Animal{

//子类的run方法与子类继承自父类的run方法构成了重载(Overload) 而非重写(Overwrite)

// 因为子类继承了父类的run()方法

public void run(int i){

System.out.println("dog is running") ;

}

}

* 1. 方法重写与方法重载之间的关系: 重载发生在同一个类内部的两个或多个方法, 而重写发生在父类与子类之间.  
      重载是一种平行的关系, 而重写一定是发生于继承中. 重写以后子类的方法就覆盖了父类的该方法.

* 1. 当两个方法形成重写关系时, 可以在子类方法中通过super.method的形式来调用父类的method方法. 其中super.method  
     不必放在第一行语句. 因为此时父类对象已经构造完成, 先调用父类方法还是子类方法是由程序的逻辑决定的.  
     只有在子类的Constructor中, super语句还必须放在其中的第一行.

* 1. 在定义一个类的时候, 如果没有显式的指定该类的父类, 那么该类就继承于 java.lang.Object类(JDK提供的一个类  
     Object类是Java中所有的类的直接或间接父类). 它是Java中所有类的根类(root class)

**Jave SE 第十六-二十讲 多态详解**

* 1. 多态(polymorphism): **父类的引用可以指向子类的对象. 反之, 子类的引用不能直接指向父类的对象, 需要通过  
      强制类型转换为子类型对象才可以**. **指向哪个对象就调用该对象的方法.**

**A subclass reference can be used when a superclass reference is expected.** In Java, method calls are always determined by the type of the actual object,  
 not the type of the variable containing the object reference. This is called  
 **dynamic method lookup**.

public class PolyTest{

public static void main(String[] args){

Flower rose = new Rose() ; //多态:父类引用可指向子类对象.

rose.sing() ;

}

}

class Flower{

public void sing(){

System.out.println("flower is singing") ;

}

}

class Rose extends Flower{

}

* 1. Parent p = new Child(); 当使用多态方式调用方法时, 首先检查父类中是否有sing()方法, 如果没有则  
     编译出错, 如果有, **则去调用子类的sing()方法**.

public class PolyTest2{

public static void main(String[] args){

// Parent parent = new Parent() ;

// parent.sing() ;

// Child child = new Child() ;

// child.sing() ;

Parent p = new Child() ;

p.sing() ;

}

}

class Parent{

/\*

public void sing(){

System.out.println("parent is singing") ;

}

\*/ //父类中必须要也有sing方法, 否则报错

}

class Child extends Parent{

public void sing(){

System.out.println("child is singing") ;

}

}

* 1. 一共有两种类型转换:
     1. 向上类型转换(upcast) : 由子类型对象引用转换为父类型对象引用, 比如将Cat转换为Animal类型. 对于向上类型  
         转换不需要显式指定 比如:

Cat cat = new Cat() ;

Animal animal = cat ; // 将子类型的引用转化为父类型的引用, 也可以写成(Animal)cat ;

animal.sing() ;

* 1. 向下类型转换(downcast):即由父类型对象转换为子类型对象,　比如将Animal类型转换为Cat类型,必须要显式  
     指定(即强制类型转换), 比如:

Animal a = new Cat() ; // is-a, a cat is an animal

Cat cat = a ; //报错, 子类对象引用不能直接指向父类对象, 需要强制转换  
Cat cat = (Cat)a ; // 父类对象可以通过强制类型转换变为子类对象

* 1. 使用强制类型转换将父类对象变为子类对象, 因为父类转换为子类后可以调用子类中新增的方法. 比如,  
     父类有三个方法, 子类有三个同名但重写的方法,另外有五个新增方法. 这样通过强制类型转换以后,  
     就可以调用这五个新增的方法. **即,当使用子类特有的方法时, 需要使用强制类型转换** .

* 1. 多态是一种运行期的行为,而不是一种编译期的行为. Java只有到运行时, 才知道多态对象引用具体指向谁.  
     这种行为称为晚绑定(late bounding)

public class PolyTest5{

public static void main(String[] args){

A a = null ;

if(args[0].equals("1")){

a = new B() ;

}

else if(args[0].equals("2")){

a = new C() ;

}

else if(args[0].equals("3")){

a = new D() ;

}

a.method() ;

}

}

class A{

public void method(){

System.out.println("A") ;

}

}

class B extends A{

public void method(){

System.out.println("B") ;

}

}

class C extends A{

public void method(){

System.out.println("C") ;

}

}

class D extends A{

public void method(){

System.out.println("D") ;

}

}

* 1. 多态一个重要作用, 利用多态可以定义一个公共的接口. 该接口的参数为基类(base class), 但当调用时可以传入  
     子类的引用(derived class)

public class PolyTest6{

public void run(Car car){ // 作为一个基类调用的公共接口可以接受子类参数, 比如如下的QQ

car.run() ;

}

public static void main(String[] args){

PolyTest6 test = new PolyTest6() ;

Car car = new BMW() ;

test.run(car) ;

QQ qq = new QQ() ;

**test.run(qq)** ; // qq upcast to car

}

} // output "BMW is running", "QQ is running"

class Car{

public void run(){

System.out.println("car is running") ;

}

}

class BMW extends Car{

public void run(){

System.out.println("BMW is running") ;

}

}

class QQ extends Car{

public void run(){

System.out.println("QQ is running") ;

}

}

**Jave SE 第二十一讲　抽象类(abstract class)**

* 1. 抽象类使用abstract 关键字定义, 抽象类无法实例化. 也就是说, 不能new出来一个抽象类的实例.
  2. 抽象方法（abstract method）是使用abstract关键字所修饰的方法称为抽象方法。**抽象方法只有声明没有实现**；  
      public abstract void method() {} ; // 不是抽象方法， 因为有花括号。

抽象方法必须定义在抽象类中， 不能在其它类中。

* 1. 如果某个类包含了抽象方法， 这个类一定是抽象类。
  2. 抽象类可以包含具体方法（有声明和实现的方法）。
  3. 如果一个类中包含了抽象方法， 那么这个类一定要声明为 abstract class
  4. 无论何种情况， 只要一个类是抽象类， 那么这个类就无法实例化。
  5. 在子类继承父类（父类为抽象类）的情况下， 那么该子类必须要实现父类中所定义的**所有的抽象方法**；否则，  
     该子类需要声明为一个抽象类。例子：

public class AbstractTest{

public static void main(String[] args){

Shape shape = new Triangle(10,6) ;

int area = shape.computeArea() ;

System.out.println("triangle:" + area) ;

shape = new Rectangle(10,10) ;

area = shape.computeArea() ;

System.out.println("rectangel:" + area) ;

}

}

**abstract class Shape**{

**public abstract int computeArea();** //计算形状面积

}

class Triangle extends Shape{

int width ;

int height ;

public Triangle(int width, int height){

this.width = width ;

this.height = height ;

}

public int computeArea(){

return width \* height / 2 ;

}

}

class Rectangle extends Shape{

int width ;

int height ;

public Rectangle(int width, int height){

this.width = width ;

this.height = height ;

}

public int computeArea(){

return width \* height ;

}

}

* 1. 为什么需要abstract class？   
     Sometimes it is desirable to force programmers to override a method.That happends when there's no  
     good default for the superclass and only the subclass programmer can know how to implement the   
     method properly.

**Jave SE 第二十二讲 接口类型，static， final关键字详解**

* 1. 接口类型（interface type）： 接口的地位类似于class。   
     **接口中的所有方法都是抽象方法**。（Java8中，还可为static或者default）在接口方法定义中，  
      可以省略abstract关键字。 比如： public abstract void output() ; 等同于 public void output() ;

* 1. A Java interface type declares the methods that can be applied to a variable of that type.  
      **接口类实际上定义了所有能使用接口类方法的变量类型**。

* 1. An interface type does not have **instance variable(成员变量）**and constructor.

* 1. All methods in an interface type are automatically **public** .但实现该接口的类方法必须定义为public

* 1. **接口类中不能有实例变量， 但可以包含常量**。 所有的常量自动定义为 **public final static**  
      public interface Named

{

**String NO\_NAME="(NONE)"** ;

}

* 1. 接口类中的方法默认为public抽象方法， 但Java8以后也可以为static或者default methods
     1. Static Methods in Interface

接口类中的静态方法并非直接操作某一个具体的类， 而是操作该接口类型。例子：

public interface Measurable{

double getMeasure() ; // An abstract method

static double average(Measurable[] objects) // A static method

{

… // Same implementation as in Data class

}

}

To call this method, provide the name of the interface and the method name:

double meanArea = Measurable.average(countries);

* 1. Default Methods in Interface

接口类中的默认方法为非静态方法，并且具有方法的实现。在一个实现了该接口的类中，如果没有  
 重写（override）该方法， 则该类会继承该方法。

public interface Measurable

{

double getMeasure(); // An abstract method

**default** boolean smallerThan(Measurable other)

{

return getMeasure() < other.getMeasure() ;

}

}

* 1. Conflicting Default Methods - 接口类默认方法冲突

如果一个类继承了父类的某个方法以及一个接口中的同名的默认方法。 又或者一个类同时继承了两个接口  
中的同名的默认方法。此时就产生了默认方法冲突。两条规则用来处理同名默认方法  
冲突：

* 1. Class Win(类优先)： 如果该类同时继承了父类中的方法以及接口中的默认方法时， 父类中的方法  
      将被继承，而接口的默认方法被忽略。
  2. Interface clash(接口冲突): 如果该类继承的是两个接口中的同名默认方法， 那么该类需要重写  
      （override）该方法。 例子：

public class Person

{

public String name(){

return firstName() + " " + lastName() ;

. . .

}

}

public interface Named

{

default String name(){

return "(NONE)";

}

}

public class User extends Person, implements Named

{

// Inherits Person.name()

. . .

}

**如果上面的person类为一个接口， 则User类需要重写name（）方法**。

* 1. **接口类引用可以指向任何对象，只要该对象的类实现了该接口**。  
     **换而言之， 也可以转换某类对象为接口类类型只要这个类实现了该接口**。对接口引用的方法调用是多态的  
     （polymorphic），即具体调用的是哪个类的方法是在运行时决定的。

An Interface reference can refer to an object of any class that implements the interface.  
Method calls on an interface reference are polymorphic. The appropriate method is determined  
at run time. **The JVM locates the correct method by first looking at the class of the actual  
object, and then calling the method with the given name in that class**.

* 1. **从接口引用类型转化为类引用类型。当接口型引用需要调用类方法时，需要强制类型转化为类类型的引用  
     （downcast）才可以调用类的方法**。比如：

Measurable max = large(us, uk) ; // Measurable 为一个接口类，并不包含getName()方法

Country maxCountry = (Country) max ; //首先将接口类类型强制转换为类类型引用

String name = maxCountry.getName() ;

* 1. 可以把接口看作一种特殊的抽象类。抽象类中可以包含concrete method， 但接口类中只能有abstract   
     method， 不能有concrete method

* 1. 同抽象类一样， 接口也无法实例化。

* 1. 类可以实现接口。 实现使用关键字implements表示某个类实现了某个接口。

* 1. **一个类实现了某个接口， 那么该类必须要实现接口中的所有方法**。否则，该类必须定义为抽象类。例子：

public class Test3{

public static void main(String[] args){

MyClass myClass = new MyClass() ;

myClass.output() ;

}

}

interface MyInterface{

public void output() ;

}

class MyClass implements MyInterface{

public void output(){

System.out.println("output") ;

}

}

* 1. Java是单继承， 也就是所某个类只能有一个父类。 但一个类可以实现多个接口，多个接口间用逗号分隔。

* 1. 某个类可以同时继承一个父类并实现多个接口。

class MyClass extends MyParent implements MyInterface,MyInterface2 { … }

* 1. 接口类型引用可以指向实现类的对象。

public class Test4{

public static void main(String[] args){

AA bb = new BB() ; // OK， 接口类引用可以指向实现类的对象

bb.output() ;

}

}

interface AA{

public void output() ;

}

class BB{

public void output(){

System.out.println("BB") ;

}

}

* 1. static关键字： 可用于属性， 方法以及类的声明中。

* 1. static修饰属性：无论该类生成多少实例。所有实例共同使用该静态成员变量。一个对象对该属性进行修改，其它  
      对象的静态成员变量的值也会随之修改。

public class StaticTest{

public static void main(String[] args){

MyStatic myStatic = new MyStatic() ;

MyStatic myStatic2 = new MyStatic() ;

myStatic.a = 10;

System.out.println(myStatic2.a) ;

}

}

class MyStatic{

static int a ;

} //output 10

* 1. 如果一个成员变量为static的， 那么可以通过类名.成员变量名的方式来使用它，这也是Java推荐的使用方式。  
     **静态成员变量可以看作类对象之间的一个公共变量**。

MyStatic myStatic = new MyStatic() ;

**MyStatic.a = 10;**

System.out.println(myStatic.a) ;

* 1. 在一个静态的成员方法中不能访问非静态的成员变量， 只能访问静态的成员变量。即，**静态只能访问静态的，  
     非静态的都可以 访问**。注意，**静态方法中也无法引用this关键字， 因为this也是非静态变量**。

public class StaticTest5{

public static void main(String[] args){

W.change() ;

}

}

class W{

int a = 10 ;

public static void change(){

a++ ; //错误， 不能在一个静态方法中去访问一个非静态成员变量

}

}

* 1. static修饰方法：static修饰的方法叫做静态方法。同静态成员变量一样， 静态方法可以直接通过类名.方法名  
     的方式来调用。 这也是Java所推荐的调用静态方法的方式。

public class StaticTest2{

public static void main(String[] args){

// MyStatic2 test = new MyStatic2();

// test.output();

**MyStatic2.output();**

}

}

class MyStatic2{

public **static** void output(){

System.out.println("output") ;

}

}

* 1. **静态方法指的是不需要从对象中调用而直接可以从类调用的方法**。调用方法为： **类名.方法名** ；  
     **子类可以继承父类的静态方法但不能重写该方法（override）只能说隐藏了该方法。（hide）**  
     **Subclass cannot override instance method but only hide calss method of superclass**   
     说隐藏（hide）**是因为如果从子类引用调用该方法， 则采用其子类方法的定义。而如果从父类  
     引用调用该方法， 则采用的还是父类方法的signature**。 这点同子类继承父类的普通方法是不同的。  
     当从父类继承普通方法时， 不能该引用是否为子类型的， 只要指向的对象为子类型的， 那么就会  
     采用子类的方法定义。另外， **main（）方法用于为静态方法。应该尽量少使用static方法，因为它  
     不是面向OOP的方法。**

public class StaticTest3{

public static void main(String[] args){

N n = new N();

n.output(); // print "N" 子类静态方法隐藏了父类静态方法

M m = new N(); //父类型引用指向子类型对象

m.output(); // print "M" 当从父类引用调用该方法，采用父类方法的定义。

}

}

class M {

public static void output(){

System.out.println("M") ;

}

}

class N extends M{

// @Override 提示方法不会Override父类的方法

public static void output(){

System.out.println("N") ;

}

}

* 1. **Static Imports：**   
     当引用类的时候，可以使用static关键字，这样就允许直接使用这些引用类中的静态变量和静态方法而无需  
     加上前缀的类名。 例如：

import static java.lang.System.\* ;

import static java.lang.Math.\* ;

public class RootTester

{

public static void main(String[] args){

double r = sqrt(PI) ; // Instead of Math.sqrt(Math.PI)

out.printin(r) ; // Instead of System.out

}

}

* 1. final关键字： final可以修饰属性，方法，类。
  2. final修饰类： 当一个类被final所修饰时，表示该类是一个**终态，即不能被继承**。
  3. final修饰方法：当一个方法被final所修饰时，　表示该方法是一个**终态方法，即不能重写（Override）**
  4. final修饰属性： 当一个属性被final所修饰时， 表示该属性的属性值不能被改变。**成员变量前加上final修饰符  
      相当于一个常量**
  5. final修饰引用类属性： **该引用的指向不能改变，即该对象不能再指向其它对象， 但引用指向的对象允许改变。**例子：

public class FinalTest2{

public static void main(String[] args){

People people = new People();

// people.address = new Address(); //error, 引用类型的成员变量也不能改变

people.address.name = "shanghai"; // ok, 并未改变引用地址， 改变引用地址

// 所指的对象。

}

}

class People{

**final Address address** = new Address(); //final 修饰一个引用类属性（对象属性）

}

class Address{

String name = "beijing" ;

}

* 1. final类型的成员变量必须赋初值：一般来说有两种赋值方式：
     1. 在声明final类型成员变量时就赋初值。
     2. **在声明final类型成员变量时不赋初值，那么在类的所有构造方法中都必须为该变量赋初值**。

public class FinalTest3{

**final** int a ; //会报错，因为对普通成员变量可以不赋初值，但final类型

// 成员变量必须赋初值。

public FinalTest3(){

a = 0 ; //或者，在构造方法中赋初值。但，两种方法只能在一个

//地方赋初值。否则也会报错。

}

public FinalTest3(int a){

// 会报错，“可能尚未初始化a”

this.a = a; //ok

}

}

* 1. 一个抽象类不能定义为 final。因为abstract定义了一个基类，目的是通过子类来重写该基类的方法，而final  
     表明该类无法被继承，这样就矛盾了。

* 1. static code block： **静态代码块**。静态代码块在类加载到JVM时执行， 而constructor是在类初始化时执行。  
     所以静态代码块先于构造方法执行。 并且静态代码块只执行（加载）一次， 而每生成一个新的对象  
     都要执行一次构造方法。生成对象的过程是首先将对象的类加载到JVM上， 然后由JVM来生成对象。  
     但每个类只会被JVM加载一次。

* 1. 如果继承体系中既有构造方法又有静态代码块， 那么首先执行最顶层的类的静态代码块， 一直执行到最底层的  
     静态代码块，然后再从最顶层的构造方法执行到最底层的构造方法。 注意， **静态代码块只会执行一次**。

public class StaticTest4{

public static void main(String[] args){

new S();

new S();

}

}

class P{

**static{**

**System.out.println("static block") ;**

**}**

public P(){

System.out.println("P constructor") ;

}

}

class Q extends P{

static {

System.out.println("Q static block") ;

}

public Q(){

System.out.println("Q constructor") ;

}

}

class S extends Q{

static{

System.out.println("S static block") ;

}

public S(){

System.out.println("S constructor") ;

}

}

//output :

--------------------

static block

Q static block

S static block

P constructor

Q constructor

S constructor

P constructor

Q constructor

S constructor

**Jave SE 第二十五讲 单例模式详解**

* 1. **设计模式（Design Pattern）**

设计模式大量依托于多态。主要经典的设计模式有23种。 讲解常见的15种。单例模式（singleton）表示一个类  
只会生成唯一的一个对象。

public class SingletonTest{

public static void main(String[] args){

Singleton singleton = Singleton.getInstance() ;

Singleton singleton2 = Singleton.getInstance() ;

System.out.println(singleton == singleton2) ;

}

}

class Singleton{

private static Singleton singleton = new Singleton() ;

private Singleton(){

}

public static Singleton getInstance(){

return singleton ;

}

}

**Jave SE 第二十六，二十七讲 String类**

* 1. Instanceof操作符：　用于判断某个对象是否是某个类的实例。　语法形式：

C:\B5AA96A5\E1A85534-0EE1-4208-8D2C-20098BE993D4_files\image001.png

People people = new Man() ;

System.out.println( people instanceof People) ; // 返回true， 因为Man是People的子类， 根据继承， 子类也是父类  
 // 因此Man 也可以看作是People的实例

* 1. **equals（）方法， 该方法定义在Object类中， 因此Java中的每个类都具有该方法， 对于Object类的equals（）方法来  
     说， 它是判断调用equals（）方法的引用与传入的引用是否一致， 即这两个引用是否指向同一个对象。 对于Object  
      类的equals（）方法来说， 等价于 ＝＝**

* 1. **对于String类的equals（）方法， 它是判断当前字符串与传入的字符串的内容是否一致或者两者是否为同一String对象。  
     所以** 对于String对象的相等性判断， 请使用 equals（）方法， 而不要使用 ＝＝， 因为 ＝＝ 只能判断两者是否为  
     同一对象。

* 1. String是常量， 其对象一旦创建完毕就无法改变。 当使用+拼接字符时， 会生成新的String对象， 而不是向原有的  
     String对象追加内容。

* 1. String Pool（字符串池）在栈中，不在堆中， new出的对象都在堆中  
      String s ＝ “aaa”； （采用字面值方式赋值）
     1. 查找String Pool中是否存在“aaa”这个对象， 如果不存在， 则在String Pool中创建一个“aaa”对象，  
        然后将String Pool中的这个“aaa”对象的地址返回来， 赋值给引用变量s， 这样s会指向String Pool中的  
        这个“aaa”字符串对象
     2. 如果存在， 则不创建任何对象， 直接将String Pool中的这个“aaa”对象地址返回来，赋给s引用。

* 1. String s ＝ new String（“aaa”）；
     1. 首先在String Pool中查找有没有“aaa”这个字符串对象， 如果有， 则不在String Pool中再去创建“aaa”  
        这个对象了， 直接在堆中（heap）中创建一个“aaa”字符串对象， 然后将堆中的这个“aaa”对象的地址  
        返回来， 赋值给s引用， 导致了s指向了堆中创建的这个“aaa”字符串对象。
     2. 如果没有， 则首先在String Pool中创建一个“aaa”对象， 然后再在堆中创建一个“aaa”对象， 然后将  
        堆中的这个“aaa”对象的地址返回来， 赋值给s引用， 导致了s指向了堆中所创建的这个“aaa”对象。
     3. 小结： 采用new String（）方法，**无论字符串池中是否有该对象** **堆中一定会创建新对象**。也就是说，  
         **在 new String(“aaa”) 语句执行后， 在String Pool和heap中都会存在一个 “aaa”对象**

* 1. 只有当子类重写过equals方法，才能用来比较对象的值， 否则只是继承Object的equals（）方法，用来比较  
     引用是否为同一个对象。比如下面的例子：

public class EqualsTest{

public static void main(String[] args){

Student s1 = new Student("zhangsan") ;

Student s2 = new Student("zhangsan") ;

System.out.println(s1 == s2) ;

System.out.println(s1.equals(s2)) ; // 从Object继承过来的equals方法仅用于比较

// 是否为同一对象。只有当子类重写过该方法

// 才能用于比较值。比如String类的equals()

}

}

class Student{

String name;

public Student(String name){

this.name = name ;

}

} // return false false

// 下面的例子通过override Student的euquals方法， 将同名的学生设为相等

public class EqualsTest{

public static void main(String[] args){

Student s1 = new Student("zhangsan") ;

Student s2 = new Student("zhangsan") ;

System.out.println(s1 == s2) ;

System.out.println(s1.equals(s2)) ; ) // return true now

}

}

class Student{

String name;

public Student(String name){

this.name = name ;

}

**public boolean equals(Object anObject)**{

if (this == anObject){

return true ;

}

if (anObject instanceof Student){

**Student anotherObject = (Student)anObject** ;

if(this.name.equals(anotherObject.name)){

return true;

}

}

return false ;

}

}

**Jave SE 第二十八讲 Object类, 第二十九讲 访问控制符**

* 1. 相等性比较（＝＝）
     1. 对于原生数据类型来说， 比较的是左右两边的值是否相等
     2. 对于引用类型来说， 比较左右两边的引用是否指向同一个对象，或者说左右两笔的引用地址是否相同。

* 1. java.lang.Object类可以看作java中所有类的基类。 java.lang包在使用时无需显式的导入，编译时由编译器自动导入。

* 1. API（Application Programming Interface) : 应用编程接口

* 1. 当打印引用时， 实际上会打印出引用所指向的对象的 toString（）方法的返回值， 因为每个类都会直接或者间接的  
     继承自Object， 而Object类中定义了toString（）， 因此每个类都有toString（）方法。

* 1. 包（package） 用于将完成不同功能的类分门别类， 放在不同的目录（包）下。 包的命名规则：  
     将公司域名或者邮件地址反转作为包名。 比如，www.shensiyuan.com 变为 com.shengsiyuan(包名）  
     对于包名， 每个字母都要小写。 如果定义类时没有使用package， 那么Java就认为我们所定义的类位于默认的包  
     中（default package）。

* 1. 编译带有package声明的Java源文件有两种方式：
     1. 直接编译， 然后根据类中所定义的包名，逐一手工建立目录结构， 最后将生成的class文件直接放到该目录  
        结构中（很少使用， 比较麻烦）
     2. 使用编译参数 -d, 方式为 javac -d **.** 源文件.java, 这样在编译后， 编译器会自动帮助我们建立好包所对应  
        的目录结构。

* 1. 有两个包名， 分别为 aa.bb.cc和 aa.bb.cc.dd， 那么我们称后者为前者的子包。

* 1. 导入（import）， 将使用package分离的各个类导入回来， 让编译器能够找到所需要的类。只有当引用的类不在同  
      一个包中才需要导入。
     1. import的语法： import com.shengsiyuan.PackageTest ;
     2. import com.shengsiyuan.\*, 表示导入 com.shengsiyuan包下的所有类。
     3. import aa.bb.\* 并不会导入aa.bb.cc包下的类， 此时需要这样写：

import aa.bb.\* ;

import aa.bb.cc.\* ;

* 1. 关于package， import， class顺序问题
     1. 首先需要定义包（package） 可选

package com.shengsiyuan // 使用 javac -d . B.java 自动创建子目录

public class B{

…

}

* 1. 接下来使用import进行导入， 可选
  2. 然后才是class或interface的定义。
  3. 如果两个类在同一个包下面， 那么则不需要导入， 直接使用即可。

* 1. 访问修饰符（access modifier）
     1. public（公共的）： 被public修饰的属性和方法可以被所有类访问。
     2. protected（受保护的）： 被protected修饰的属性和方法可以在类内部， 相同包中的类，以及该类的  
         子类所访问； **即使该子类在其它包中也可以访问**。
     3. private（私有的）： 被private所修饰属性和方法只能在该类内部使用
     4. 默认的（不加任何修饰符）。 在类内部以及相同包下的类可以访问， 实际上为**package access**
     5. 四种类型修饰符的区别：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Class | Package | Subclass (same pkg) | Subclass (diff pkg) | World |
| public | + | + | + | + | + |
| protected | + | + | + | + |  |
| no modifier | + | + | + |  |  |
| private | + |  |  |  |  |

+ : accessible  
blank : not accessible

**Jave SE 第三十一讲 Java数组**

* 1. 包装类（Wrapper Class）： 针对于原生数据类型的包装。所有的包装类（8个）都位于java.lang包下。Java中  
     的八个包装类分别为： byte， short，int， long， float， double， char， boolean

public class IntegerTest{

public static void main(String[] args){

int a = 10 ;

Integer integer = new Integer(a) ; //引用类型，非原生类

int b = integer.intValue() ;

System.out.println(b) ;

}

}

* 1. 数组（Array）： 相同类型的数据的集合。
  2. 数组定义： **type[] 变量名/数组名 ＝ new type[数组长度] ;** 例子：

int[] a = int[10] ;

* 1. 数组中的元素索引从0开始， 对于数组来说， 最大的索引为 （数组长度 － 1）
  2. 定义数组的第三种方式：
     1. **type[] 变量名 ＝ [new type[]]{逗号分隔的初始值列表}** ;
  3. Java中的每个数组都有一个名为length的属性， 代表数组的长度。**length**属性为 public final int, 数组长度一旦确定，  
     就无法改变了。
  4. 对于数组内容的比较不能使用 equals方法， 因为该方法没有被重写。
  5. int[] a = new int[10] ; 其中a是一个引用， 它指向了生成的数组对象的首地址， 数组中每个元素都是int类型， 其中仅存放数据  
     值本身。
  6. 如果数组的类型为对象， 数组中装的是对象的引用， 而非对象本身。比如，

Person[ ] p = new Person[3] ;

* 1. 二维数组是一种平面的二维结构，　本质上是数组的数组。