JDK(Java Development Kits):java开发工具包，编写java程序使用到的工具包，为开发者提供封装好的java类库。

SDK(Software Development Kits)：软件开发工具包，是一个笔JDK范围更广的概念。比如开发安卓应用，你需要安卓开发工具包，叫 android sdk，比如你开发java程序，需要用到java sdk，所以一般使用sdk这个概念，你需要在前面加上限定词。

public static void main(string args[]):

static： 表明方法是静态的，不依赖类的对象的，是属于类的，在类加载的时候 main() 方法也随着加载到内存中去。

.Java：class 字节码文件

多行注释无法嵌套使用

整型常量默认为int型的，浮点型常量默认为double型的

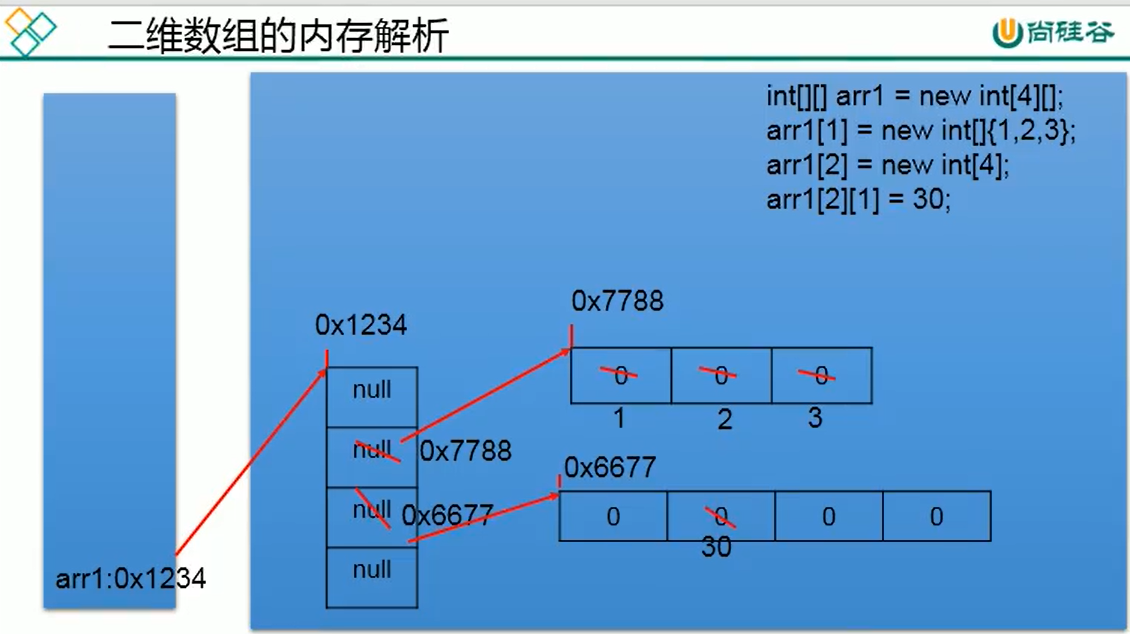
short、byte、char相互运算，结果自动转化为int

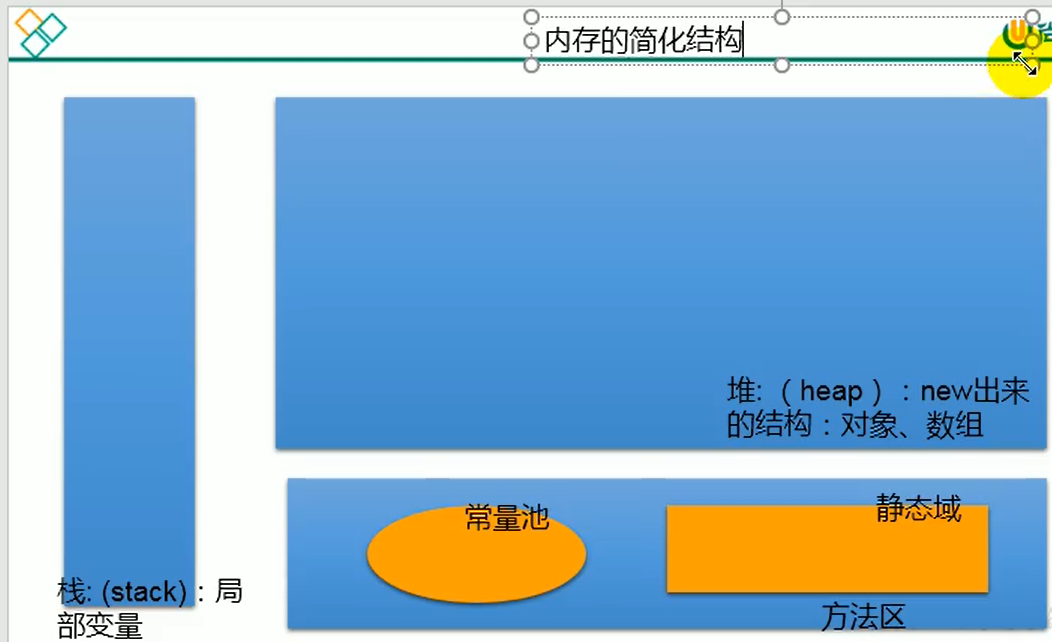
byte b = 9;

byte a = b+1；//出错，右侧运算出来为int型数据

取模运算的结果与被模数的符号相同

switch 的表达式包含：byte 、short、char、int、string和枚举6种数据类型





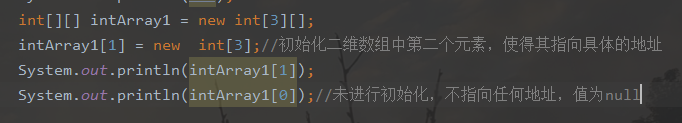
Int[] intArray = new int[3][];//未初始化

IntArray[1] = new int[5];//将第二个数组元素初始化为5元数组

intArray中存储的值是这个三元数组的首地址

int[1]中存储的值是5元数组的首地址

相当于二维数组的第一维（IntArray[1]）用于存储每个实际存储数据的数组的首地址



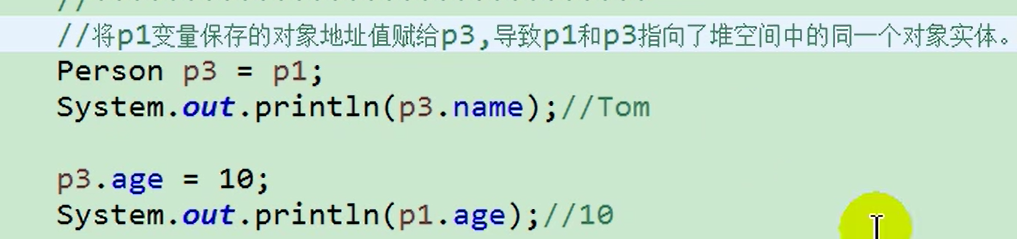


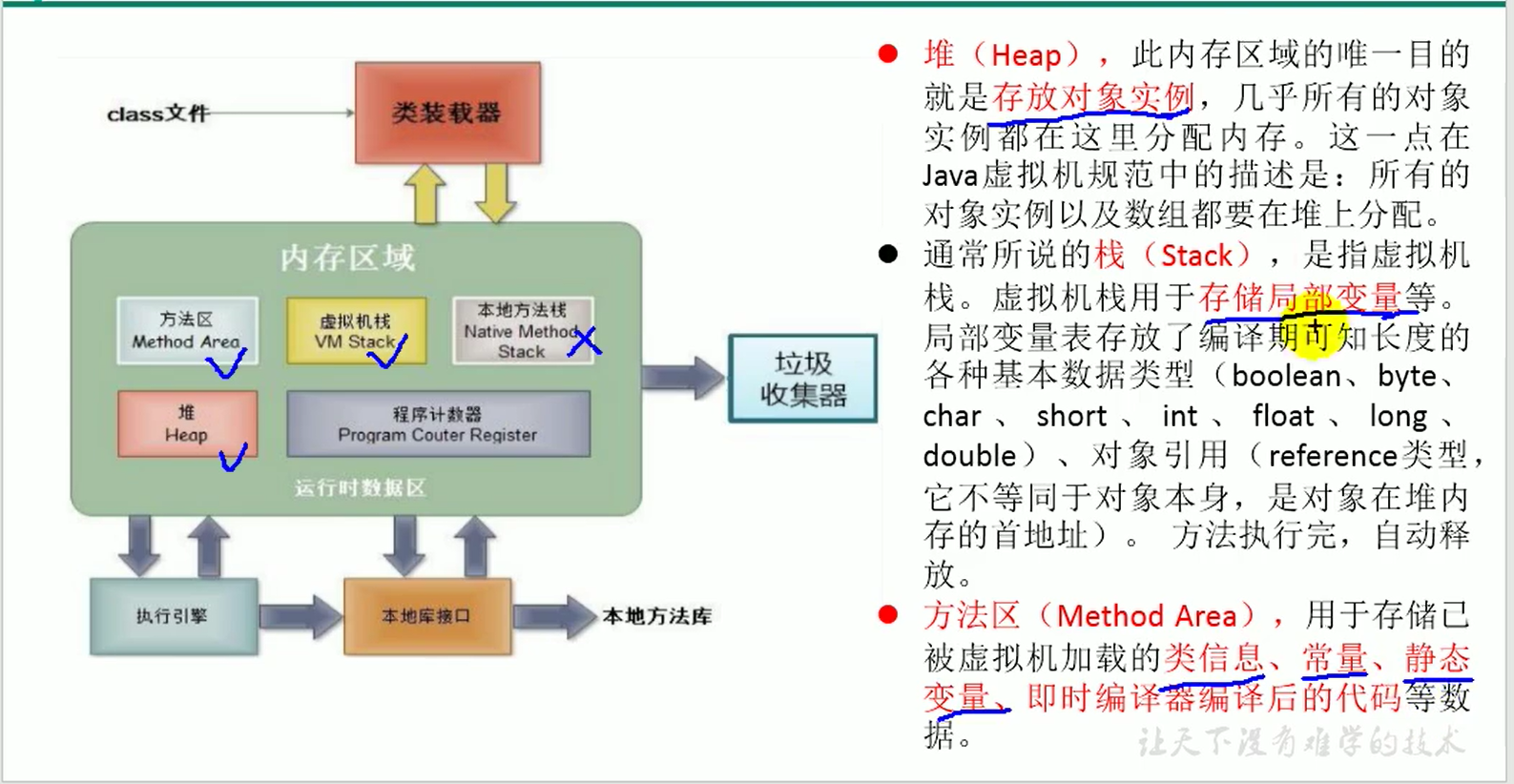
求[a,b]之间的随机数：

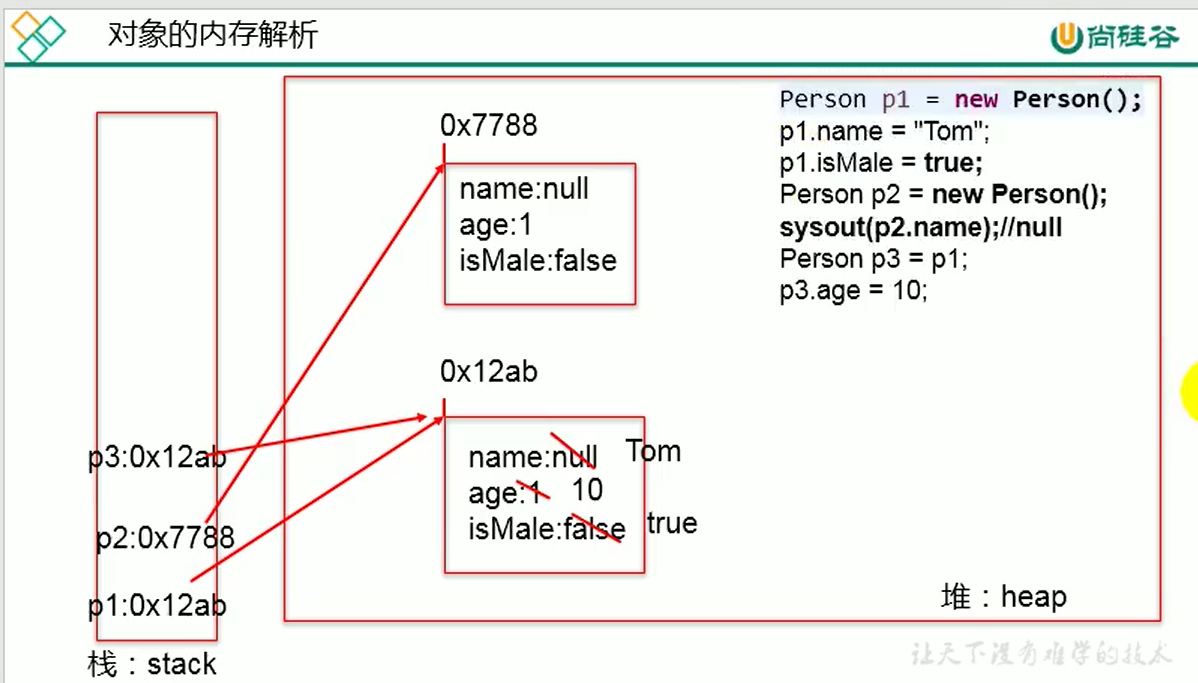
Math.random()\*(b-a+1)+a;

Math.random():生成的是double型的数据

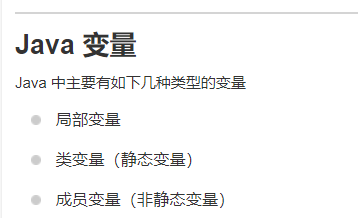








P3不算是创建对象，而是声明一个变量，并将p1指向的地址值赋给p3



数组在未初始化的情况下，会给数组中的每一个元素赋上默认值。

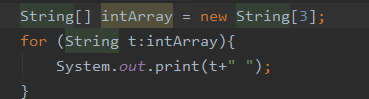
Byte、short、Int：0

Char：ASCII值的0;’’或者’\u0000’

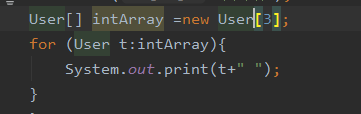
String、或者对象默认值为null。

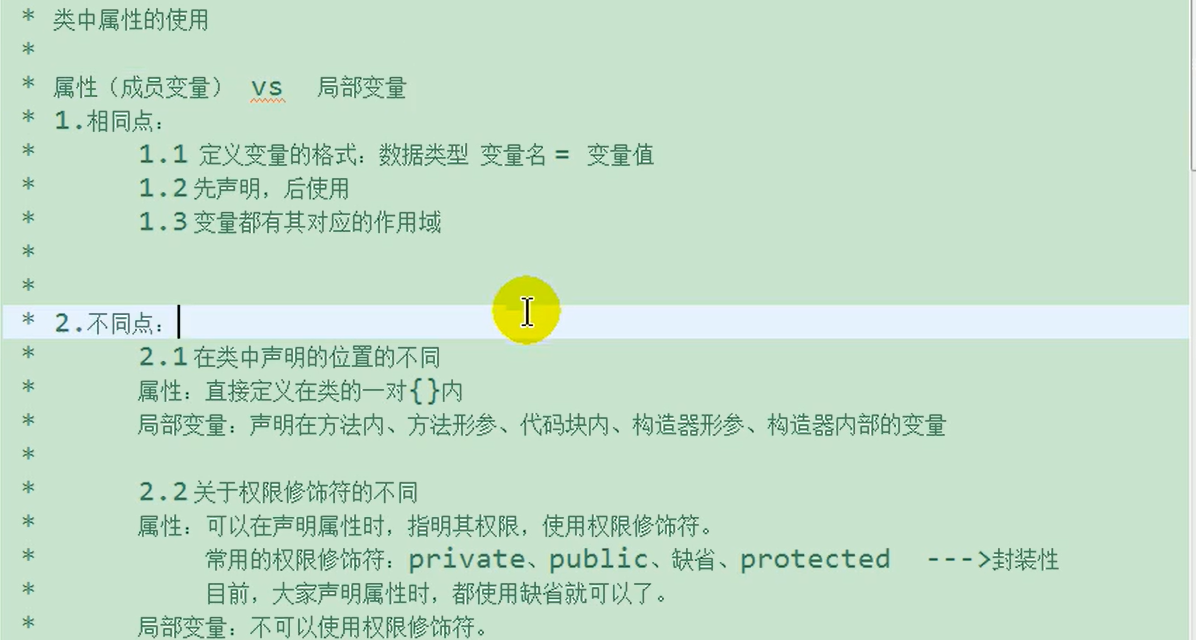
int[] intArray = new int[3];  
for (int t:intArray){  
 System.*out*.print(t+" ");  
 }  
 }

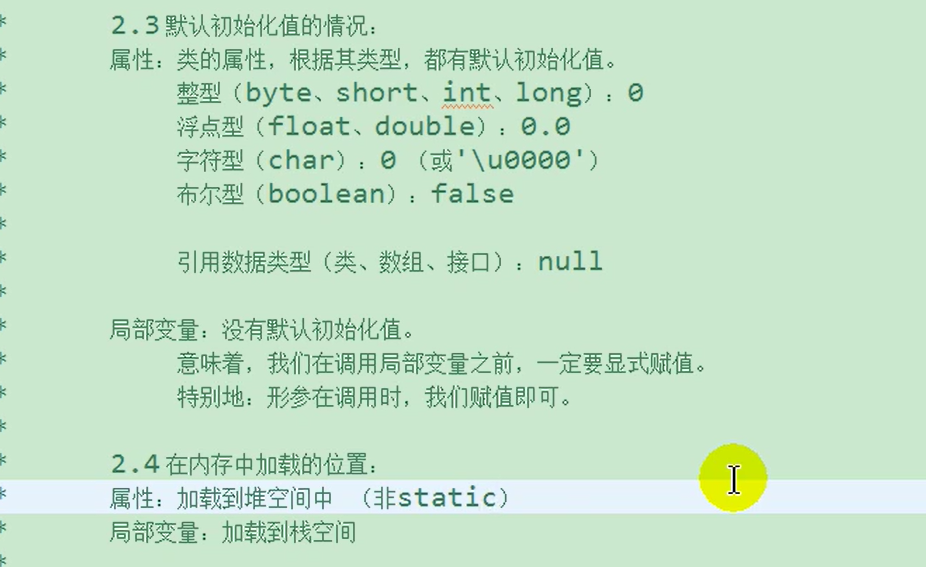


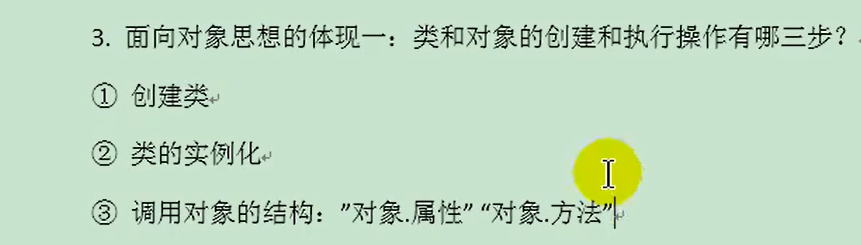


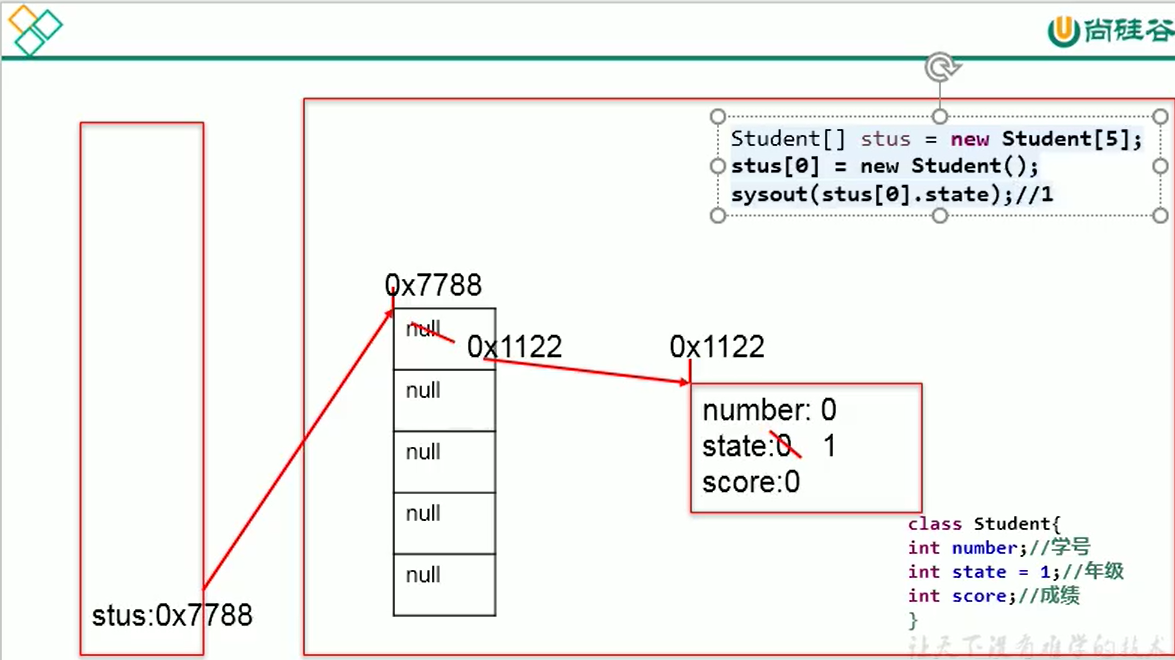












**匿名对象：**

创建的对象没有显示的赋给具体的变量，即未将对象的地址值给任何变量

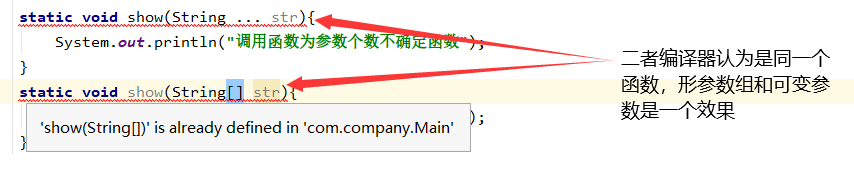
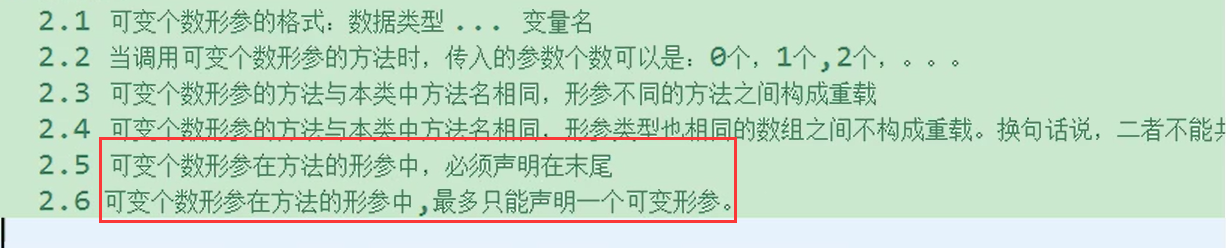
匿名对象只能使用一次

**new** Phone().**price**=1999;*//匿名对象的使用***new** Phone().showPrice();*//上面的匿名对象与这个匿名对象是两个不同的匿名对象*

虽然第一个匿名对象将手机的价格设置为了1999，但是第二行的匿名对象是完全不同的对象，所以它的price属性仍为默认值。

数组的翻转（reverse）：

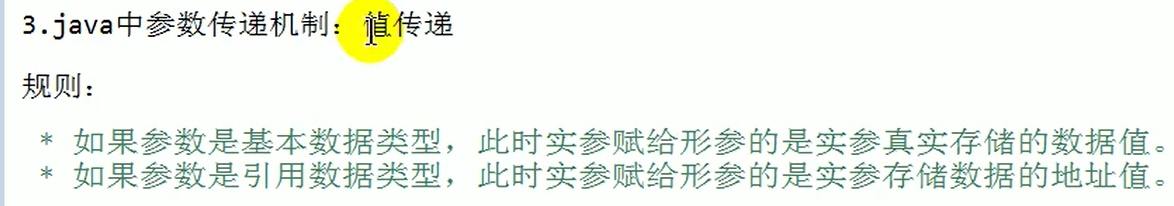
**int**[] intArray = {12,45,88,1,33,77,51,777};  
Arrays.*sort*(intArray);  
**for**(**int** i=0, j=intArray.**length**-1;i<=j;i++,j--)  
{  
 **int** temp = intArray[i];  
 intArray[i] = intArray[j];  
 intArray[j] = temp;  
}



String[] str = {**"hello"**,**"hi"**};  
**char**[] charArray = {**'a'**,**'b'**,**'c'**};  
System.***out***.println(str);  
System.***out***.println(charArray);

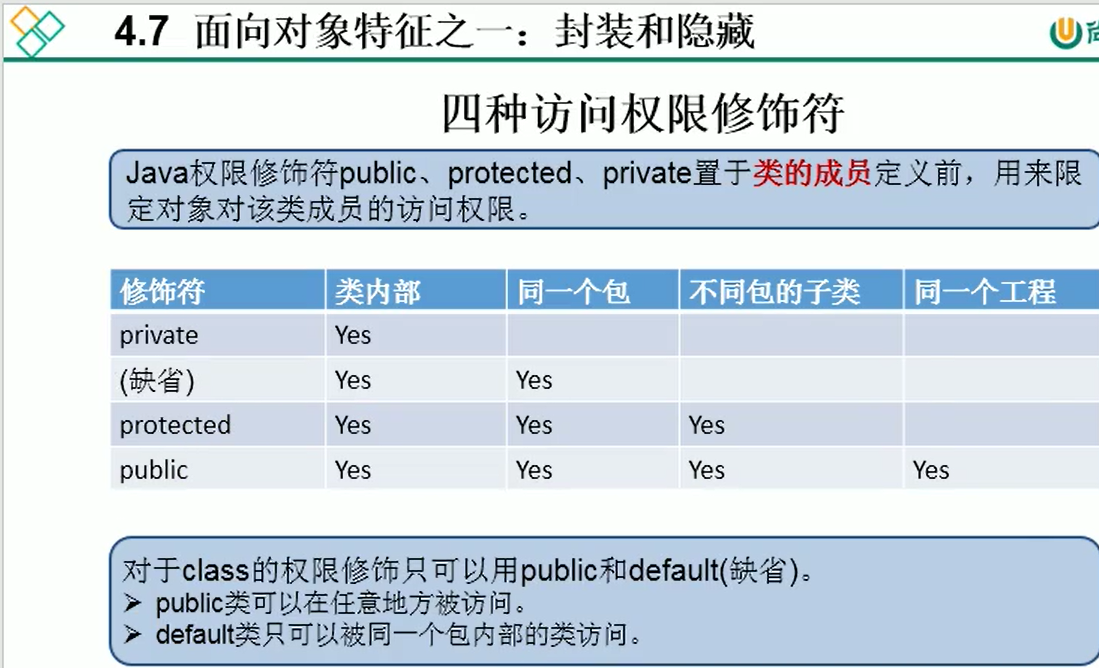


Char类型的数组打印数组名，输出的是遍历数组元素的结果。其他类型的为地址值



## 封装性：

四种访问权限：private、缺省、protected、public为



封装性的具体体现：

体现一:将类的属性xxx私化(private),同时提供公共的(public)方法来获取(getXxx)和设置( setXxx)此属性的值

体现二:不对外暴露的私有的方法

体现三:单例模式(将构造器私有化)

体现四:如果不希望类在包外被调用，可以将类设置为缺省的。

构造器的作用：

1、创建对象

2、初始化对象的属性（成员变量）

一旦我们显示的定义了构造器，系统就不再给我们提供空参构造器

系统给的默认的构造器的访问权限与该类的访问权限相同

**javaBean:**

JavaBean是一种特殊的符合以下标准的java类：

1、类是公共的

2、有一个无参的公共的构造器

3、有属性，且包含属性的get、set方法

**This关键字**

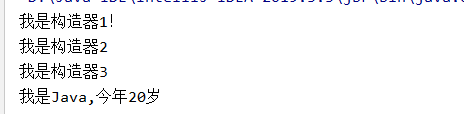
This关键字指向的是当前的对象，利用关键字this可以在构造器中引用该类的其他构造器，只能引用一个构造器且只能在构造器的第一行进行引用，不能在其他行进行引用

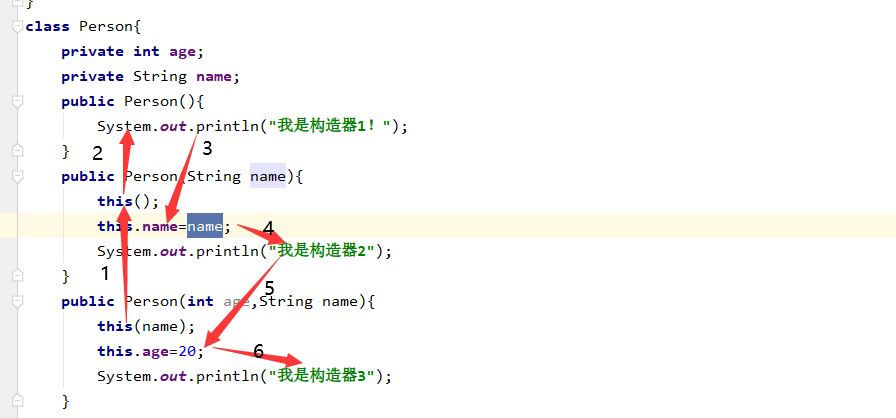
This这个使用相当于递归

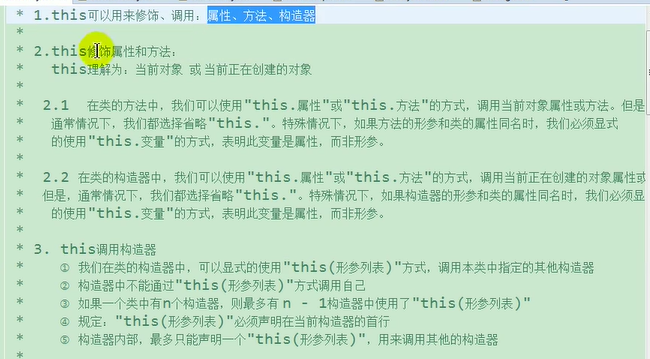
**class** Person{  
 **private int age**;  
 **private** String **name**;  
 **public** Person(){  
 System.***out***.println(**"我是构造器1！"**);  
 }  
 **public** Person(String name){  
 **this**();  
 **this**.**name**=name;  
 System.***out***.println(**"我是构造器2"**);  
 }  
 **public** Person(**int** age,String name){  
 **this**(name);  
 **this**.**age**=20;  
 System.***out***.println(**"我是构造器3"**);  
 }  
  
 **public int** getAge() {  
 **return age**;  
 }  
  
 **public void** setAge(**int** age) {  
 **this**.**age** = age;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
}

Person person = **new** Person(20,**"Java"**);  
System.***out***.println(**"我是"**+person.getName()+**","**+**"今年"**+person.getAge()+**"岁"**);

运行截图：







**package包关键字的使用：**

1、为了更好地管理项目中的类

2、使用package声明类或者接口所属的的包，声明在源文件首行

3、包名属于标识符，也遵守相应的命名规则。包名字母全为小写

4、每“.”一次，就代表一层文件目录

5、同一包下，不允许有两个名字相同的类

**Import（导入）关键字：**

1.在源文件中使用import显式的导入指定包下的类或接口

2.声明在包的声明和类的声明之间。

3.如果需要导入多个类或接口，那么就并列显式多个import语句即可

4.举例:可以使用java.util.\*的方式，一次性导入util包下所有的类或接口。通配符 "\*"

5.如果导入的类或接口是java.lang包下的，或者是当前包下的，则可以省略此import语句。

6.如果在代码中使用不同包下的同名的类。那么就需要使用类的全类名的方式指明调用的是哪个类。

7.如果已经导入java.a包下的类。那么如果需要使用a包的子包下的类的话，仍然需要导入。

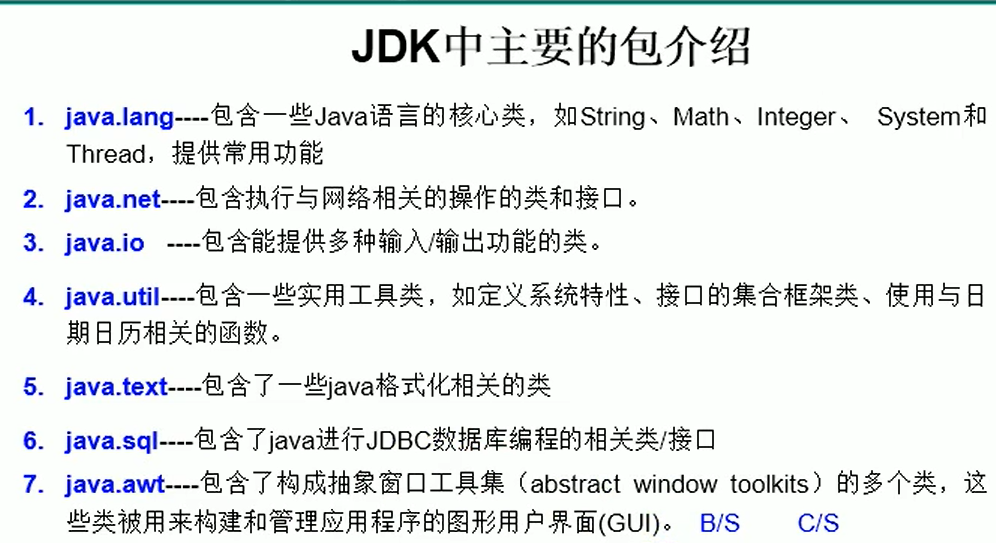
8. import static组合的使用:调用指定类或接口下的静态的属性或方法

Demo：

**import static** java.lang.System.\*;*//导入的是SYStem类下的静态方法或者属性*

***out***.println(**"out还可以这样写！"**);





## 继承：

一、继承性的好处:

1、减少了代码的瓦余

2、提高了代码的复用性\*便于功能的扩展

3、国为之后多态性的使用，提供了前提

二、继承性的格式:class A extends B{}

A:子类：派生类、subclass

B:父类：超类、基类、superclass

2.1

一旦子类A继承父类B以后，子类A中就获取了父类B中声明的所有的属性和方法。

特别的，父类中声明为private的属性或方法，子类继承父类以后，仍然认为获取了父类中私有的结构。只有因为封装性的影响,使得子类不能直接调用父类的结构而已。

2.2子类继承父类以后,还可以声明自己特有的属性或方法:实现功能的拓展。

## 方法的重写

3．重写的规定:

方法的声明:权限修饰符 返回值类型 方法名(形参列表) throwS 异常的类型

{

//方法体

}

约定俗称:子类中的叫重写的方法,父类中的叫被重写的方法

1、子类重写的方法的方法名和形参列表与父类被重写的方法的方法名和形参列表相同

2、子类重写的方法的权限修饰符不小于父类被重写的方法的权限修饰符

>特殊情况:子类不能重写父类中声明为private权限的方法返回值类型:

3、返回值类型

>父类被重写的方法的返回值类型是void,则子类重写的方法的返回值类型只能是void

>父类被重写的方法的返回值类型是A类型，则子类重写的方法的返回值类型可以是A类或A类的子类（如父类返回值类型是Object，子类可以为String）

>父类被重写的方法的返回值类型是基本数据类型(比如:double)，则子类重写的方法的返回值类型必须是相同的类类型

4、重写的方法抛出的异常类型不大于父类被重写的方法抛出的异常类型(具体放到异常处理时候讲)

子类和父类中的同名同参数的方法要么都声明为非static的(能够重写)，要么都声明为static的(不是重写)。只有非static的方法才能被重写。

## super关键字的使用

1、super理解为:父类的

2、super可以用来调用:属性、方法、构造器

3、super的使用：调用属性和方法

3.1、我们可以在子类的方法或构造器中。通过使用"super.属性或"super.方法"的方式，显式的调用父类中声明的属性或方法。但是，通常情况下，我们习惯省略"super."

3.2、特殊情况:当子类和父类中定义了同名的属性时，我们要想在子类中调用父类中声明的属性，则必须显式的使用""super.属性"的方式,表明调用的是父类中声明的属性。

3.3、特殊情况。当子类重写了父类中的方法以后，我们想在子类的方法中调用父类中被重写的方法时，则必须显式的使用""super.方法"的方式,表明调用的是父类中被重写的方法。

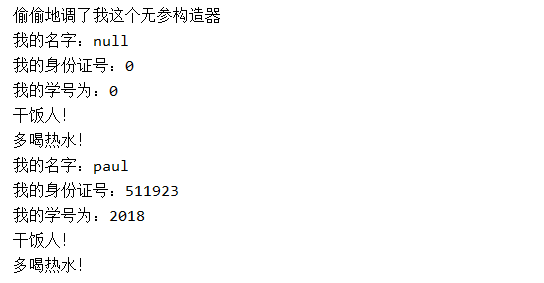
Super示例：

**public class** Person {  
 **private** String **name**;*//姓名* **private int id**;*//身份证号* **public** Person(){  
 System.***out***.println(**"偷偷地调了我这个无参构造器"**);  
 }  
 **public** Person(String name,**int** id){  
 **this**.**name** = name;  
 **this**.**id** = id;  
 }  
 **public void** eat(){  
 System.***out***.println(**"干饭人！"**);  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public int** getId() {  
 **return id**;  
 }  
  
}

**package** com.study;  
  
**public class** Student **extends** Person {  
 **private int id**;*//学生学号* **public** Student(){  
 }  
 **public** Student(String name,**int** id1, **int** id){  
 **super**(name, id1);  
 **this**.**id** = id;  
 }  
 **public void** eat(){  
 **super**.eat();  
 System.***out***.println(**"多喝热水！"**);  
 }  
 **public void** printIfor(){  
 System.***out***.println(**"我的名字："**+**this**.getName());*//首先先该子类中寻找，找不到再去父类中寻找* System.***out***.println(**"我的身份证号："**+**super**.getId());*//直接去父类中寻找，直到找到为止* System.***out***.println(**"我的学号为："**+**this**.**id**);  
 }  
}

Student james = **new** Student();*//调用无参构造器*Student paul = **new** Student(**"paul"**,511923,2018);*//调用有参构造器*james.printIfor();  
james.eat();  
paul.printIfor();  
paul.eat();

运行截图：



## 面向对象特征之三:多态性

多态是同一个行为具有多个不同表现形式或形态的能力。

多态就是同一个接口，使用不同的实例而执行不同操作

1.理解多态性,可以理解为一个事物的多种形态。

2何为多态性:

对象的多态性:父类的引用指向子类的对象（或子类的对象赋给父类的引用)

3．多态的使用:虚拟方法调用

有了对象的多态性以后，我们在编译期，只能调用父类中声明的方法，但在运行期，我们实际执行的是子类重写父类的方法。总结:编译,看左边,运行,看右边。

4.多态性的使用前提:类的继承关系、方法的重写

5.对象的多态性，只适用于方法,不适用于属性（编译和运行都看左边)

**虚拟方法调用**(多态情况下)

子类中定义了与父类同名同参数的方法，在多态情况下，将此时父类的方法称为虚拟方法，父类根据赋给它的不同子类对象，动态调用属于子类的该方法。这样的方法调用在编译期是无法确定的。

Salary s = new Salary("员工 A", "北京", 3, 3600.00);

Employee e = new Salary("员工 B", "上海", 2, 2400.00);

Employee是父类Salary是子类

* 实例中，实例化了两个 Salary 对象：一个使用 Salary 引用 s，另一个使用 Employee 引用 e。
* 当调用 s.mailCheck() 时，编译器在编译时会在 Salary 类中找到 mailCheck()，执行过程 JVM 就调用 Salary 类的 mailCheck()。
* e 是 Employee 的引用，但引用 e 最终运行的是 Salary 类的 mailCheck() 方法。
* 在编译的时候，编译器使用 Employee 类中的 mailCheck() 方法验证该语句， 但是在运行的时候，Java虚拟机(JVM)调用的是 Salary 类中的 mailCheck() 方法。

以上整个过程被称为虚拟方法调用，该方法被称为虚拟方法。

Java中所有的方法都能以这种方式表现，因此，重写的方法能在运行时调用，不管编译的时候源代码中引用变量是什么数据类型。

重载，是指允许存在多个同名方法，而这些方法的参数不同。编译器根据方法不同的参数表，对同名方法的名称做修饰。对于编译器而言，这些同名方法就成了不同的方法。它们的调用地址在编译期就绑定了。Java的重载是可以包括父类和子类的，即子类可以重载父类的同名不同参数的方法。

所以:对于重载而言，在方法调用之前，编译器就已经确定了所要调用的方法，这称为“早绑定”或“静态绑定”;

而对于多态，只有等到方法调用的那一刻，编译器才会确定所要调用的具体方法，这称为“晚绑定”或“动态绑定”。

引用一句Bruce Eckel的话:“不要犯傻，如果它不是晚绑定，它就不是多态。”

多态的使用:

1、当调用子父类同名同参数的方法时，实际执行的是子类重写父类的方法---虚拟方法调用

2、不能调用子类所特有的方法和属性:编译时，p2是Person类型。

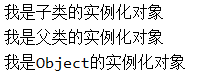
3、有了对象的多态性以后，内存中实际上是加载了子类特有的属性和方法的，但是由于变量声明为父类类型，导致编译时，只能调用父类中声明的属性和方法。子类特有的属性和方法不能调用。

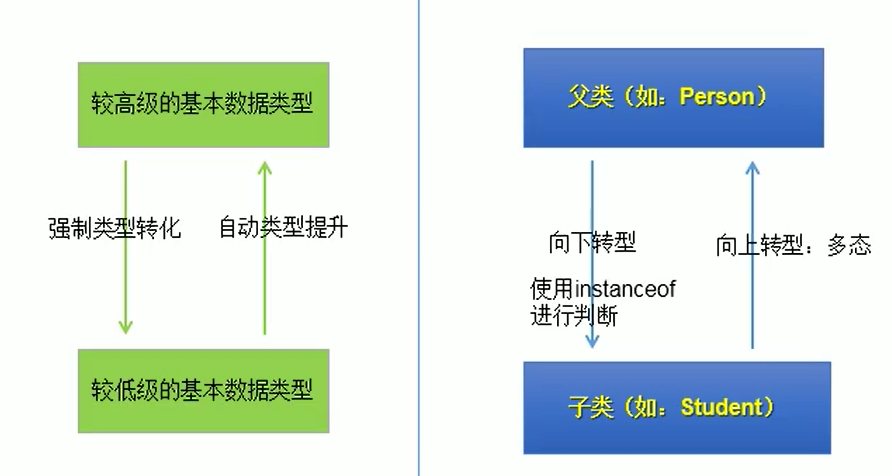
如果a instanceofA返回true,则a instanceofB也返回true.其中,类B是类A的父类。

Base jack = **new** Sub(22,**"jack"**);*//向上转型，即多态*

**if**(jack **instanceof** Sub){  
 System.***out***.println(**"我是子类的实例化对象"**);  
}  
**if**(jack **instanceof** Base){  
 System.***out***.println(**"我是父类的实例化对象"**);  
}

**if**(jack **instanceof** Object){  
 System.***out***.println(**"我是Object的实例化对象"**);  
}





**== :运算符**

1．可以使用在基本数据类型变量和引用数据类型变量中

2．如果比较的是基本数据类型变量:比较两个变量保存的数据是否相等。(不一定类型要相同)

3、如果比较的是引用数据类型变量:比较两个对象的地址值是否相同.即两个引用是否指向同一个对象实体

补充:==符号使用时，必须保证符号左右两边的变量类型一致。

二、equals(()方法的使用:

1．是一个方法,而非运算符

2.只能适用于引用数据类型

3. 0bject类中equals()的定义,

public boolean equals(object obi) {

return (this == obi);

}

说明: object类中定义的equals()和==的作用是相同的:比较两个对象的地址值是否相同.即两个引用是否指向相同的地址

4．像String、Date、File、包装类等都重写了0bject类中的equals()方法。重写以后，比较的不是两个引用的地址是否相同，而是比较两个对象的"实体内容"是否相同。

5．通常情况下，我们自定义的类如果使用equals()的话，也通常是比较两个对象的"实体内容""是否相同。那么，我们就需要对Object类中的equals(()进行重写.

重写的原则:比较两个对象的实体内容是否相同.

Object i = **true**?**new** Integer(1):**new** Double(2.0);*//三元运算符要求:左右两边数据类型相同，在编译时会自动类型提升*System.***out***.println(i);



object类中toString()的使用:

1．当我们输出一个对象的引用时，实际上就是调用当前对象的toString()

2. 0bject类中toString()的定义:

public string tostring( {

return getclass().getName() +"@” + Integer.toHexString(hashcode());

3．像String.Date.File、包装类等都重写了Object类中的toString()方法。使得在调用对象的toString(O)时,返回"实体内容"信息

4．自定义类也可以重写toString()方法，当调用此方法时，返回对象的"实体内容"

### Static关键字

static关键字的使用

1.static:静态的

2.static可以用来修饰:属性、方法、代码块、内部类

3.使用static修饰属性:静态变量

3.1属性,按是否使用static修饰，又分为:静态属性vs非静态属性(实例变量)

实例变量:我们创建了类的多个对象，每个对象都独立的拥有一套类中的非静态属性。当修改其中一个对象中的非静态属性时,不会导致其他对象中同样的属性值的修改。

静态变量:我们创建了类的多个对象，多个对象共享同一个静态变量。当通过某一个对象修改静态变量时，会导致其他对象调用此静态变量时,是修改过了的。

3.2 static修饰属性的其他说明:

静态变量随着类的加载而加载。可以通过"类.静态变量"的方式进行调用。静态变量的加载要早于对象的创建。

由于类只会加载一次，则静态变量在内存中也只会存在一份:存在方法区的静态域中。

类变量 实例变量

类 yes no

对象 yes yes

4.使用static修饰方法:静态方法

随着类的加就而加载,可以通过"类.静态方法""的方式进行调用

静态方法 非静态方法

类 yes no

对象 yes yes

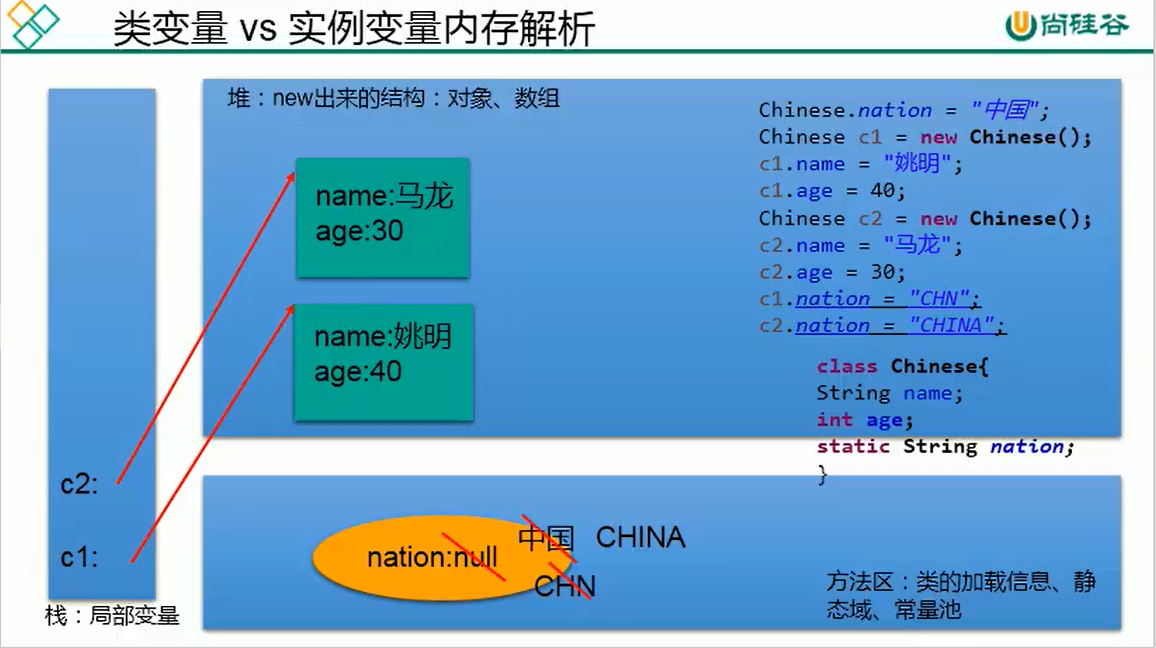
静态方法中,只能调用静态的方法或属性

非静态方法中,可以调用非静态的方法或属性,也可以调用静态的方法或属性

5. static注意点:

5.1 在静态的方法内，不能使用this关键字、super关键字

5.2 关于静态属性和静态方法的使用,大家都从生命周期的角度去理解。



## 单例模式

单例(Singleton)设计模式

设计模式是在大量的实践中总结和理论化之后优选的代码结构、编程风格、以及解决问题的思考方式。设计模免去我们自己再思考和摸索。模式就像是经典的棋谱，不同的棋局，我们用不同的棋谱，”套路”

所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法。如果我们要让类在一个虚拟机中只能产生一个对象，我们首先必须将类的构造器的访问权限设置为private，这样，就不能用new操作符在类的外部产生类的对象了，但在类内部仍可以产生该类的对象。因为在类的外部开始还无法得到类的对象，只能调用该类的某个静态方法以返回类内部创建的对象，静态方法只能访问类中的静态成员变量，所以，指向类内部产生的该类对象的变量也必须定义成静态的。

单例设计模式:

1．所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例。

2．如何实现?

饿汉式VS懒汉式

3．区分饿汉式和懒汉式饿汉式:

饿汉式：

坏处：对象加载时间过长，类加载时对象便加载到内存

好处：饿汉式是线程安全的

懒汉式:

好处:延迟对象的创建。

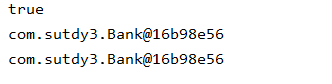
目前的写法坏处:线程不安全。--->到多线程内容时,再修改

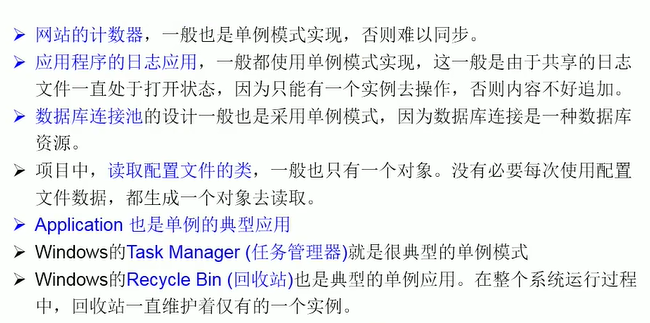
两种单例模式的创建方法：

*//饿汉式单例模式* **class** Bank {  
 **private** Bank(){  
 }  
 **private static** Bank *instance*= **new** Bank();*//直接在类创建时就将对象实例化并加载到内存* **public static** Bank getInstance(){  
 **return** *instance*;  
 }  
}  
 **class** Order{  
 **private** Order(){  
  
 }  
 *//懒汉式单例模式* **private static** Order *instance* = **null**;*//当需要创建对象才进行实例化，但存在线程不安全的情况* **public static** Order getInstance(){  
 **if**(*instance*==**null**)  
 *instance* = **new** Order();  
 **return** *instance*;  
 }

调用与运行结果：

Bank banker1 = Bank.*getInstance*();  
Bank banker2 = Bank.*getInstance*();  
Order order1 = Order.*getInstance*();  
Order order2 = Order.*getInstance*();  
System.***out***.println(order1==order2);  
System.***out***.println(banker1);  
System.***out***.println(banker2);

  
单例模式的使用场景：



## 代码块：

4．静态代码块

>内部可以有输出语句

>随着类的加载而执行,而且只执行一次>作用:初始化类的信息

>如果一个类中定义了多个静态代码块，则按照声明的先后顺序执行>静态代码块的执行要优先于非静态代码块的执行

>静态代码块内只能调用静态的属性、静态的方法，不能调用非静态的结构

5．非静态代码块

>内部可以有输出语句>随着对象的创建而执行

>每创建一个对象，就执行一次非静态代码块

>作用:可以在创建对象时，对对象的属性等进行初始化

>如果一个类中定义了多个非静态代码块，则按照声明的先后顺序执行

>非静态代码块内可以调用静态的属性、静态的方法，或非静态的属性、非静态的方法

对属性可以赋值的位置:

I

1、默认初始化

2、显式初始化

3、构造器中初始化

4、有了对象以后，可以通过"对象.属性"或"对象.方法"的方式，进行赋值s

5、在代码块中赋值

在实现类的多态时，如：

CodeBlock instance = **new** SubCodeBlock(**"Jan"**);

父类、子类的构造器，静态、非静态的代码块加载顺序：

1、首先将父类、子类两个类（包括两者的static相关属性、方法、代码块）加载到内存中。

2、再加载父类的非静态代码块、然后是父类的构造器

3、最后加载子类的非静态代码块、然后加载子类的构造器

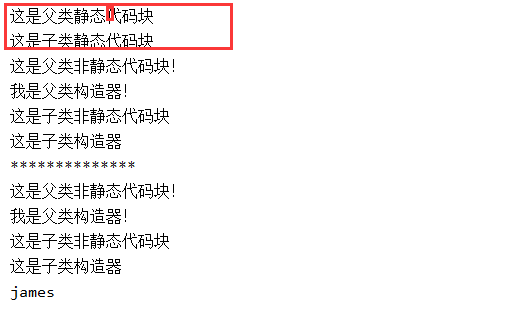
不管是父类还是子类的静态代码块只在类加载的时候加载一次，往后就不再加载。

**public class** CodeBlock {  
 **private** String **name**;  
 **public** CodeBlock(String name){  
 **this**.**name**=name;  
 System.***out***.println(**"我是父类构造器！"**);  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
 *//静态代码块* **static**{  
 System.***out***.println(**"这是父类静态代码块"**);  
  
 }  
 *//非静态代码块* {  
 System.***out***.println(**"这是父类非静态代码块！"**);  
 }  
}

**public class** SubCodeBlock **extends** CodeBlock {  
 **public** SubCodeBlock(String name){  
 **super**(**"james"**);  
 System.***out***.println(**"这是子类构造器"**);  
 }  
 **static**{  
 System.***out***.println(**"这是子类静态代码块"**);  
 }  
 {  
 System.***out***.println(**"这是子类非静态代码块"**);  
 }  
  
}

**public static void** main(String[] args) {  
 CodeBlock instance = **new** SubCodeBlock(**"Jan"**);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 CodeBlock instance1 = **new** SubCodeBlock(**"Paul"**);  
 System.***out***.println(instance.getName());

运行结果：



## Fina关键字

1、 final可以用来修饰的结构：类、方法、变量

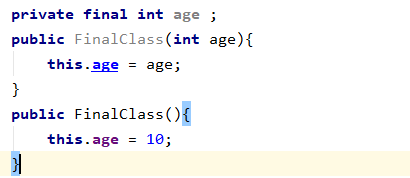
2.、final 用来修饰一个类:此类能被其他类所继承。

比如:String类、System类、StringBuffer类

3、final用来修饰方法:表明此方法不可以被重写

比如:object类中getclass();

4.final用来修饰变量:此时的"变量"就称为是一个常量

4.1 final修饰属性:可以考虑赋值的位置有:显式初始化、代码块中初始化、构造器中初始化（若有多个构造器，每个构造器都要对该属性进行初始化）

4.2在方法中不能为final修饰的关键字进行赋值，因为无法确定是否会调用该方法 。

4.3 final修饰局部变量:

尤其是使用final修饰形参时，表明此形参是一个常量。当我们调用此方法时，给常量形参赋一个实参。一旦赋值以后,就只能在方法体内使用此形参，但不能进行重新赋值。

static final用来修饰属性:全局常量