目录

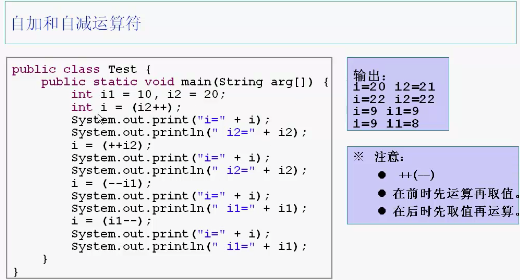
[运算符 2](#_Toc423780939)

[类关系图，面向对象基础，内存分析，this关键字，static关键字 3](#_Toc423780940)

[Package,访问权限,继承,super,方法重新，继承，casting,toString,equals 18](#_Toc423780941)

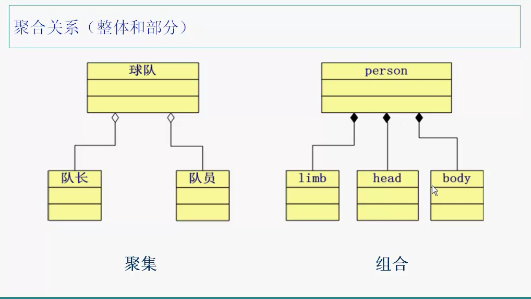
[多态，抽象类，final 26](#_Toc423780942)

## 运算符



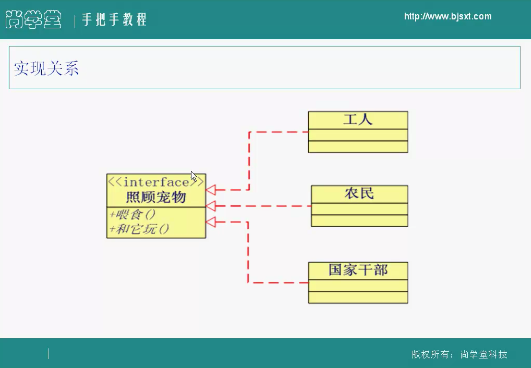
看int i = (i2++); ++在后面表示先进行表达式，再自加。所以是先把i2赋值给i，这是i等于20，然后再进行自加，i2=21此时

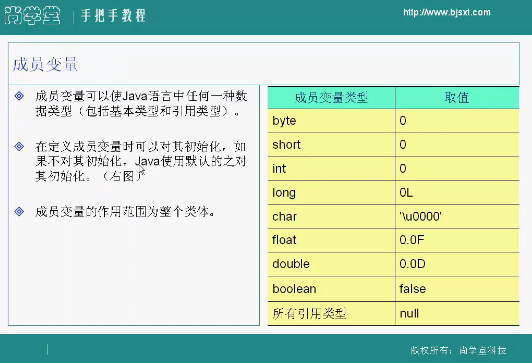
## 类关系图，面向对象基础，内存分析，this关键字，static关键字



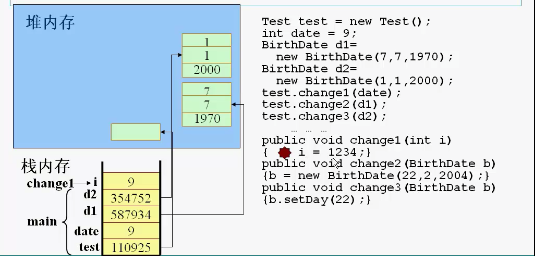
Aggregation: 聚集：一个队员可以在多个球队

组合：一个头只能在一个人身上



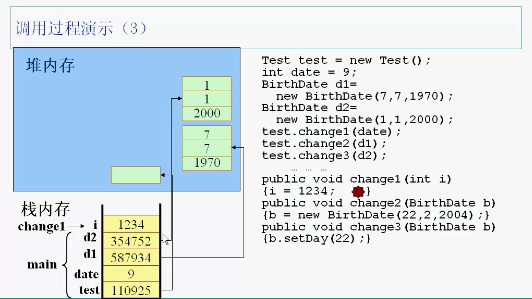


程序内存分析

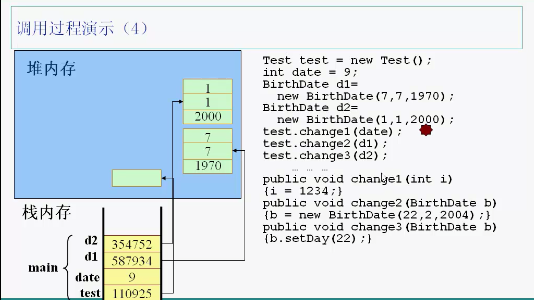


形参int i也要在main的stack里分配一个内存空间，根据值传递，date的值9被复制，然后传递给i，所以i=9现在

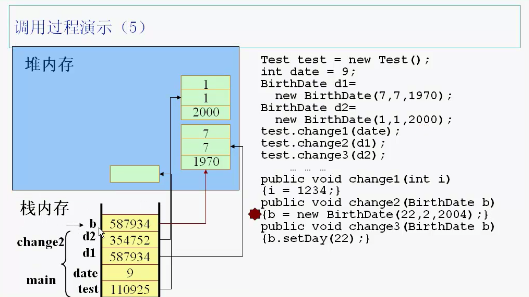
当执行到的时候，i被赋值为1234，现在的内存情况如下图所示



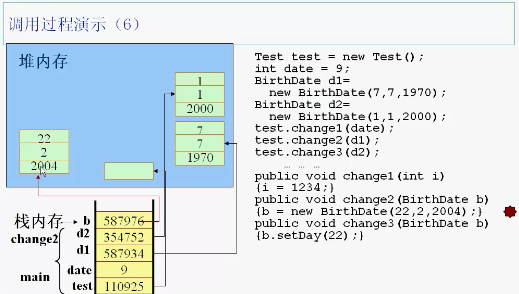
当方法test.change1(date)执行完后，为这个方法分配的局部变量内存全部消失，所以i消失了，而date还在，还是原值



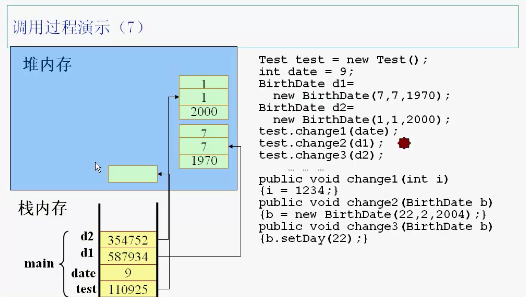
接下来执行test.change2(d1)方法，同样的，形参b被分配内存空间，其值等于被传入的di的地址值587934，此时b和di指向同一个对象



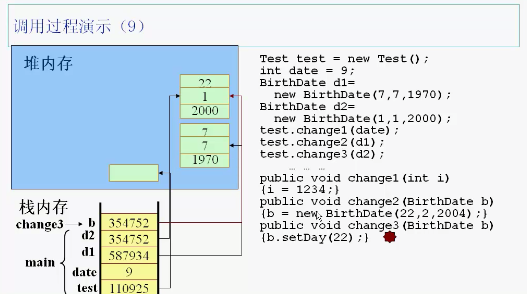
然后执行方法体，b=new BirthDate(22,2,2004); 此时会在堆内存里新创建一个对象，并且b指向这个对象



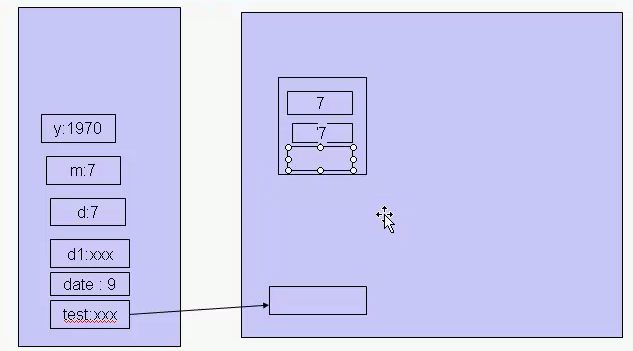
程序继续执行，当方法test.change2(d1)执行完毕后，同样的，为b分配的内存消失，b指向的BirthDate对象变为垃圾，被垃圾回收机制回收



程序继续执行，同样为形参b分配了一块内存空间，b的值等于传入的d2的值354752，指向同一个对象，此时调用该对象的setDay方法，则改变了b所指向对象的day的值，test.change3执行完后，为b分配的内存同样会消失，对象的值却永久的改变了



补充：



当在main中调用BirthDate b1=new BirthDate(7,7,1970)的时候，BirthDate的构造器BirthDate(int d, int m, int y)被调用，所以为3个形参分配了内存空间，构造器中为

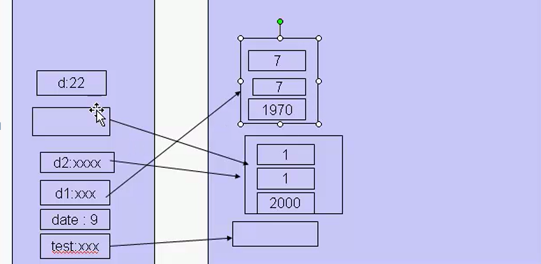
{

day=d; month=m; day=d;

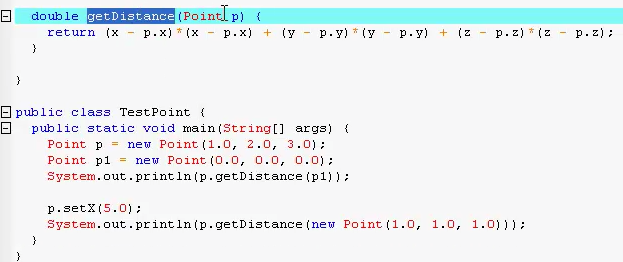
}，所以堆内存中的类的成员被赋值，当构造器调用完毕后，为三个形参分配的内存消失

注意在change3中，b调用了setDay(22)方法

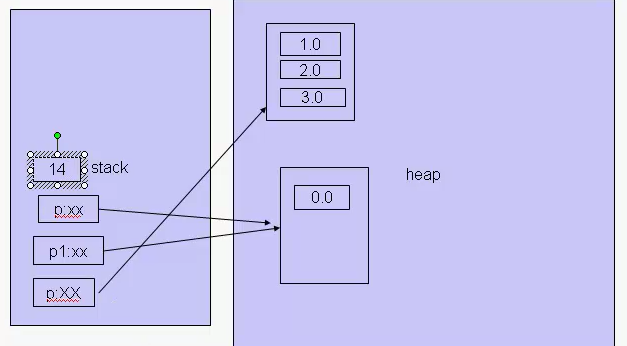
setDay(int d) { day = d;} 有一个int 形参，所以内存中会分配一个空间给d，值为22



新程序的内存分析重点

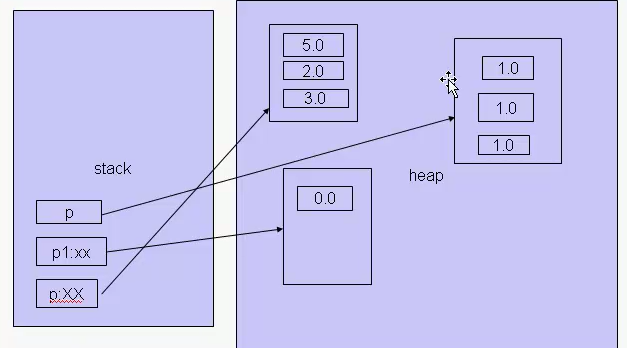


返回值的问题，p.getDistance(p1)将会有一个double的返回值，返回值没有名字，在stack里会创建一个临时的内存空间存储这个返回值，当方法结束后，这块空间也会消失，如果要保留这个返回值，可以用一个变量接收这个返回值

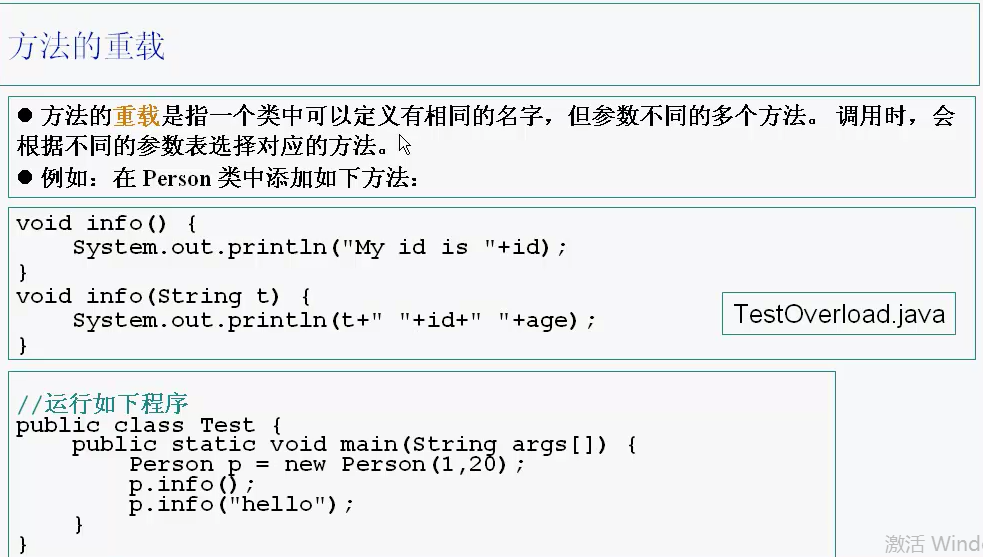


在看执行p.getDistance(new Point(1.0, 1.0, 1.0));

getDistance的形参p会指向这个新创建出来的对象

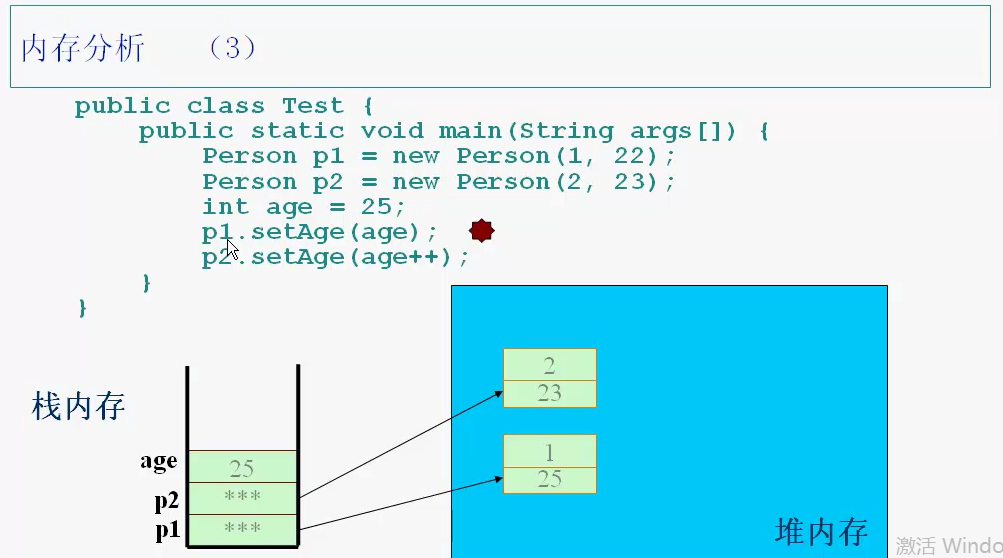


方法重载：返回值不一样不构成重载

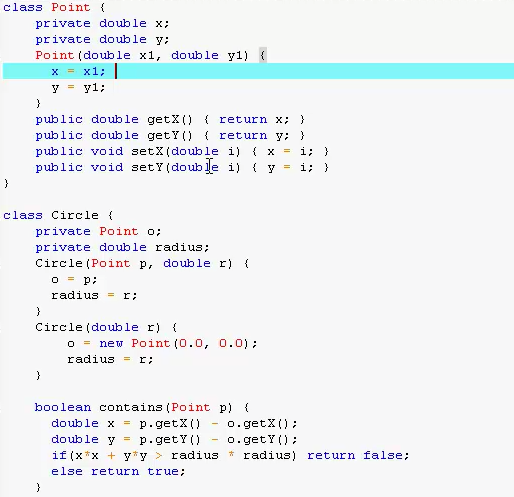


又一个内存分析：

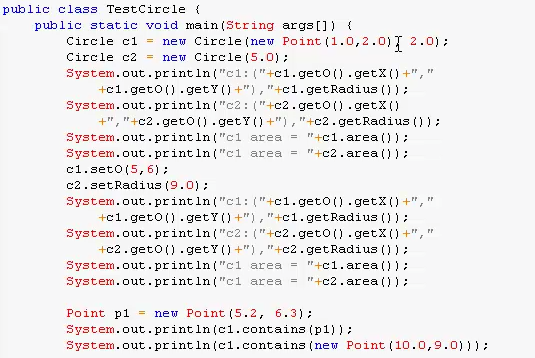
前面不需要分析，当程序执行到p2.setAge(age++)时，因为++在后面，所以先传值。形参\_age占一内存空间，值为25。所以p2的age也被设置为25，然后局部变量age自曾1变为26，此时如果打印出局部变量age的值，是26，注意此值是因为执行了自曾而增加的，并不是方法调用改变了局部变量的值



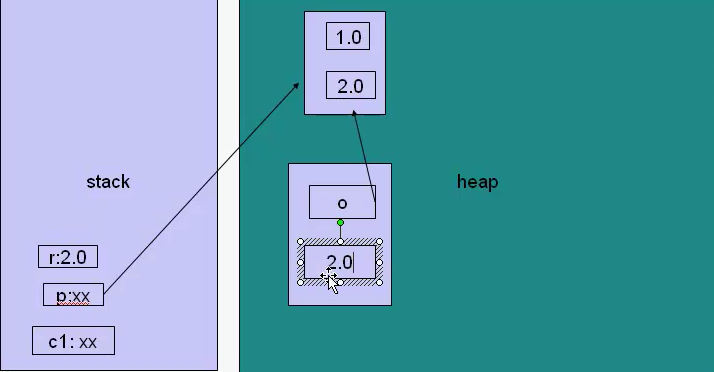
又一个内存分析



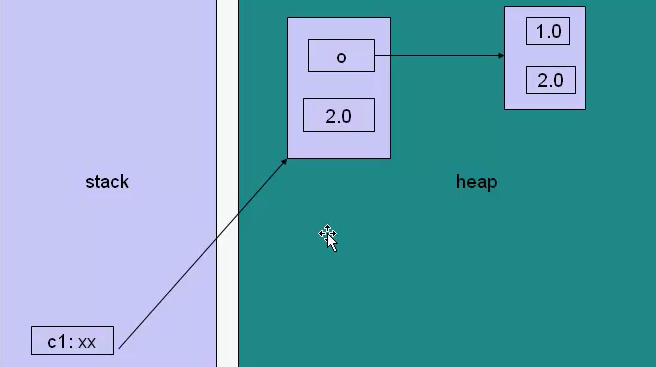




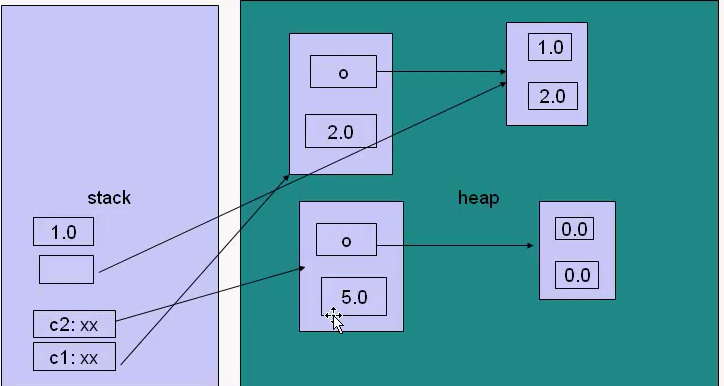
执行第一个语句后的内存空间如下图（省略了创建Point时的临时变量x1和y1的创建）



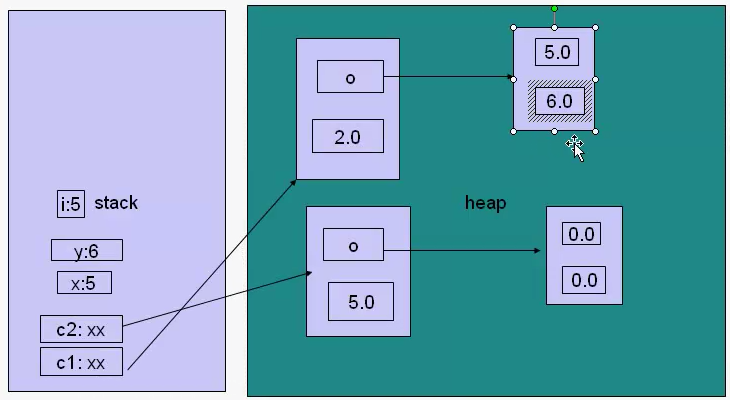
当第一个表达式执行完成后，内存中临时生成的变量被清除，内存就变为下图所示

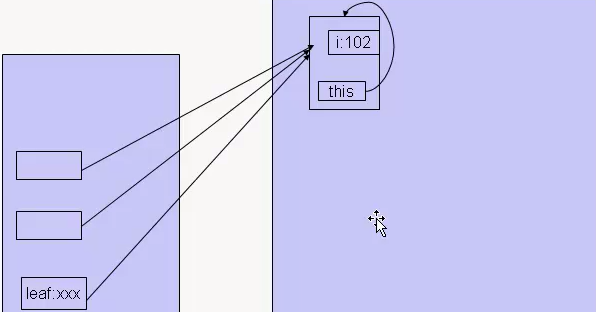
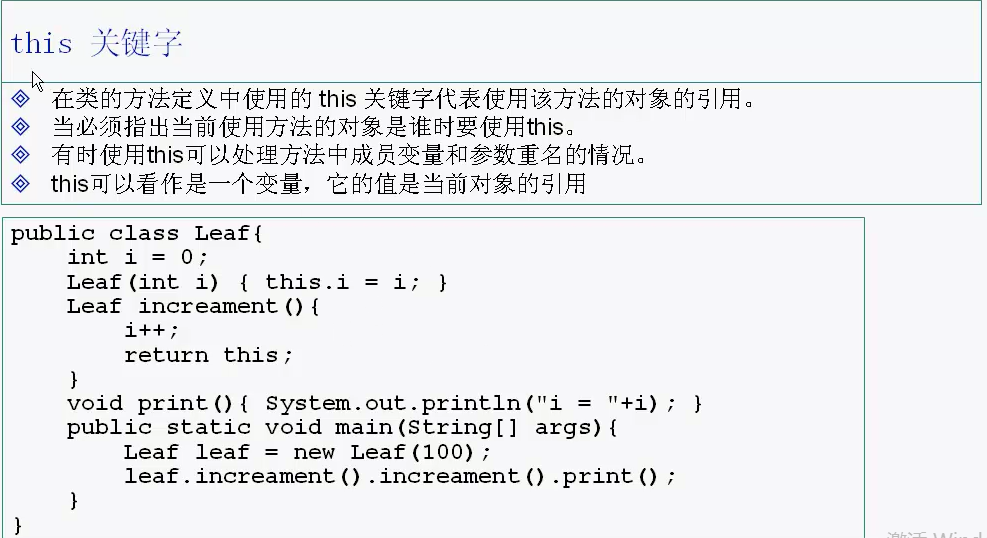


执行时，c1.getO( )会返回一个临时引用变量，指向和O指向的同一个对象，然后这个对象又调用 getX方法，返回一个临时值1.0

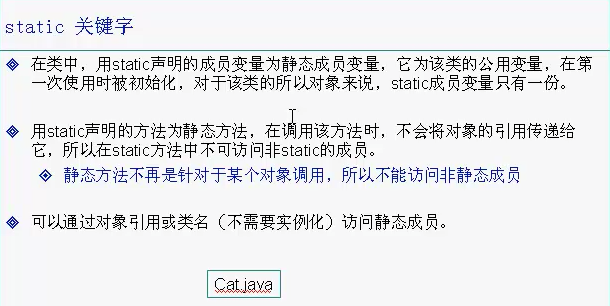


执行完后，内存如下图所示

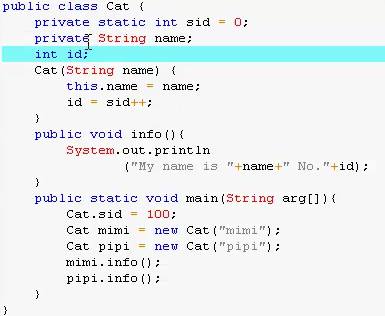




上图为内存分析，increment返回值和this指向同一个对象



看下面的程序

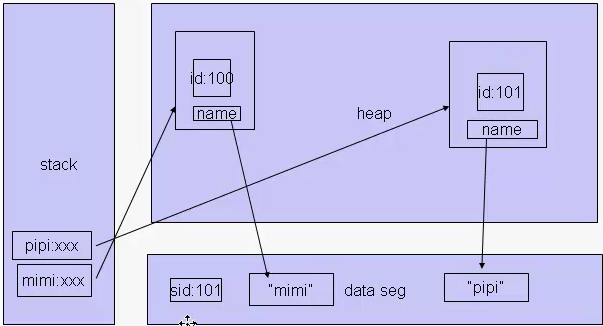


内存分析：

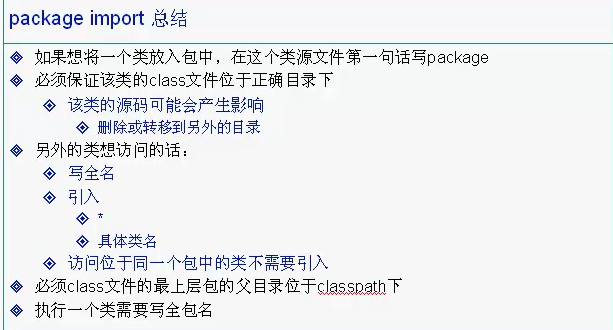
static成员在第一次使用类的时候创建，且创建在data segment中，同样，字符串常量也储存在data segment中



注意，在构造器中id=sid++; 第一只猫被创建时它的id被赋值为100，然后sid加1变为101，第二只猫创建时id被赋值为101，然后sid再加1，所以static成员可以当做计数器



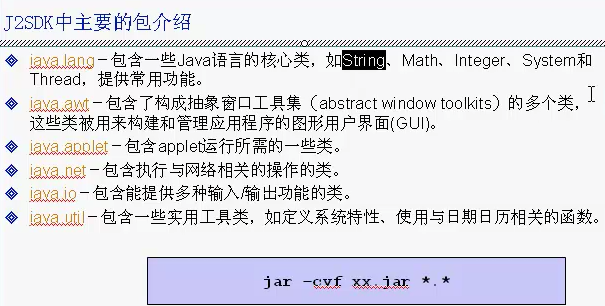
## Package,访问权限,继承,super,方法重新，继承，casting,toString,equals



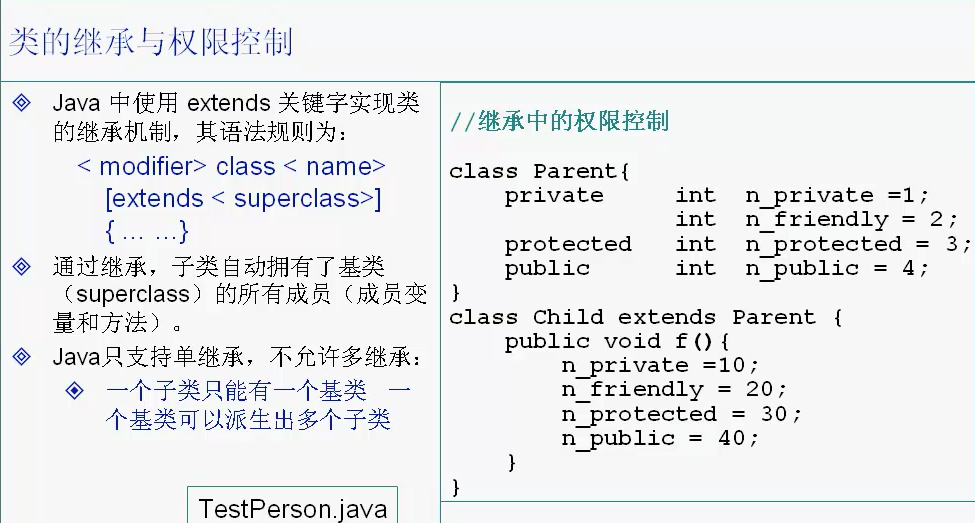
倒数第二句话，比如Cat.class位于com.sxt.java140这个包类，它的物理路径是d://java/testPackage/com/sxt/java140/Cat.class. 如果一个类在testPackage目录下，那么执行这个类的时候，由于它的classpath=. 所以会在本目录下找import com.sxt.java140，而com又在testPackage下，所以能找到。

相反，如果一个类在java目录下，执行的时候classpath必须要包含d://java/testPackage才能找到Cat类

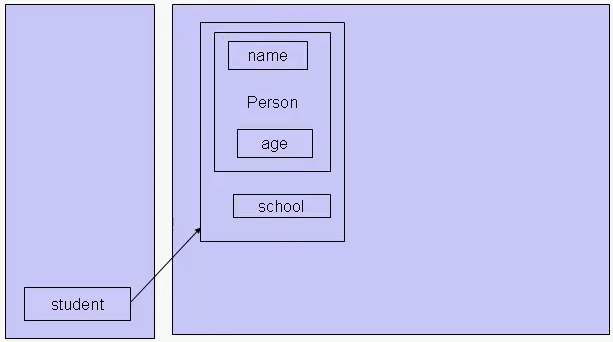
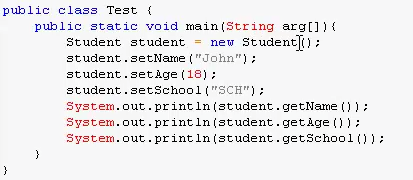
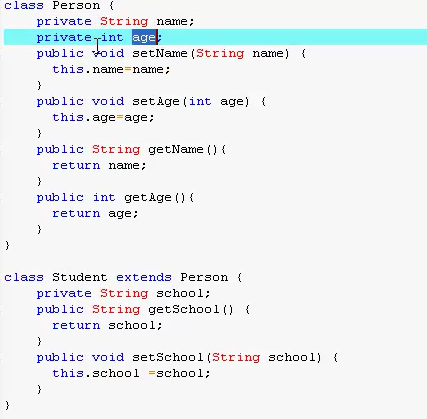
最后一句话，在testPackage执行Cat类时要加包全名: java com.sxt.java140.Cat



打包jar文件：进入testPack目录下，你的com在这个文件夹下。（定位到最上层包的上一层路径）。Xxx.jar为你的jar包名称，\*.\*表示这个目录下的所有子目录的类



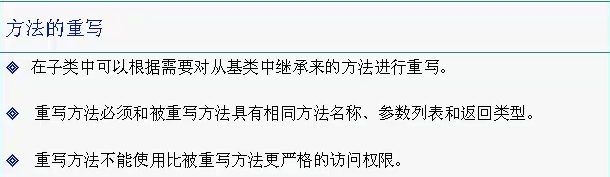
看下面的程序



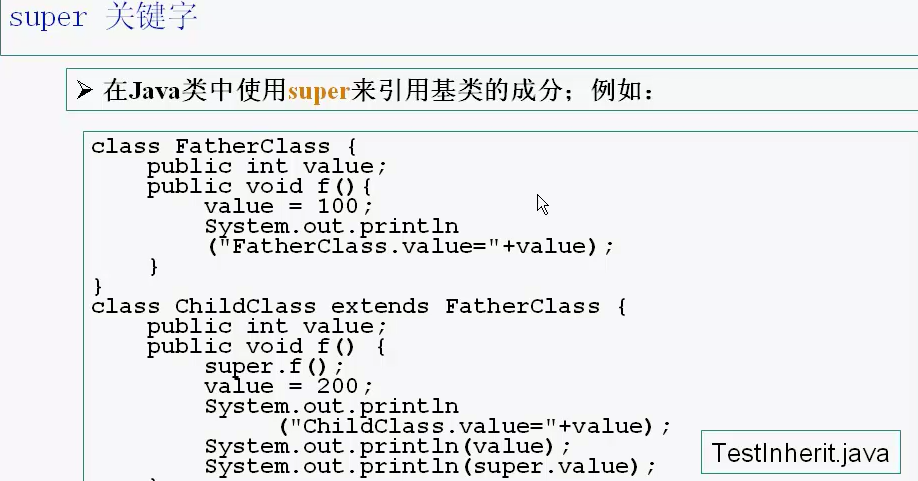
执行完后的内存，student对象也包含一个person对象，子类构造过程中，必须调用基类构造方法来构造基类对象。

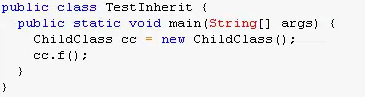


Defaut又叫包权限，同一个包的类可以访问



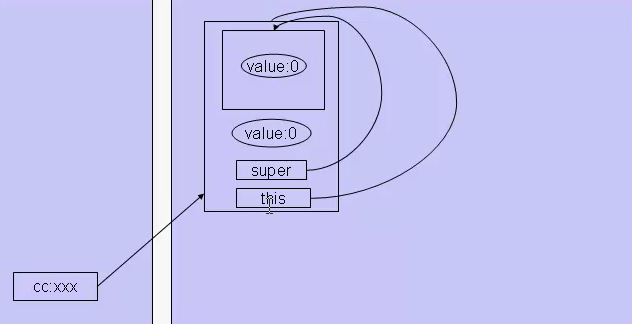
第三句话，如果父类中一个方法的权限是protected，那如果想在子类中重写这个方法，那么他的权限不能比protected严格，所以子类中的此方法的权限只能是protected或者public

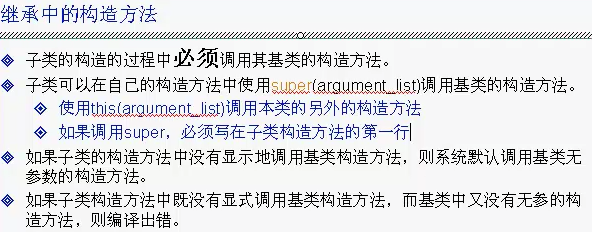




执行上面的程序，进行内存分析

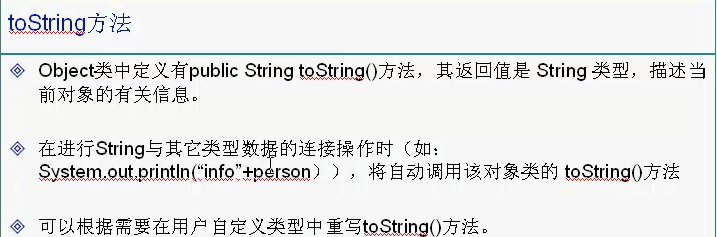
当执行完ChildClass cc = new ChildClass();时，内存如下图所示，注意，子类和父类中都有一个叫value的成员变量。如果创建的这个类是某个类的子类，那么它还会包括一个叫super的引用，这个引用指向它的父类对象，要调用父类中的方法或者成员变量，可以用super.方法名/成员变量，例如super.value就是访问父类中的value



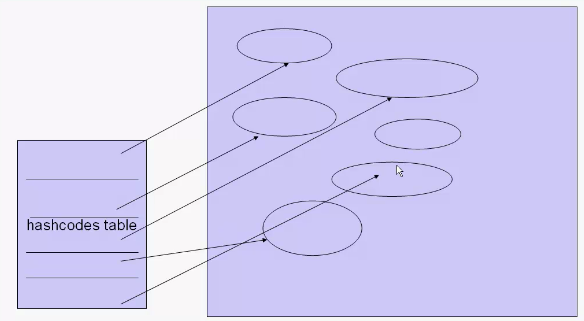


最后一句话，子类构造时如果没有显示调用父类构造器，那么就会调用父类的默认的无参数的构造器

另外，super必须写在子类构造方法的第一行，如果不这样，就像先把自己造出来，在把爸爸造出来，显然不合理



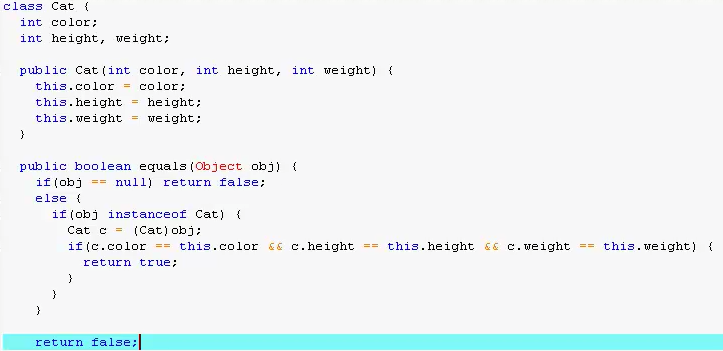
看第二句话，我们知道当一个字符串+其他数据时，其他数据自动转换为String类型，这里”info”+person，person是一个对象，这里会自动调用这个对象的toString方法来转换他

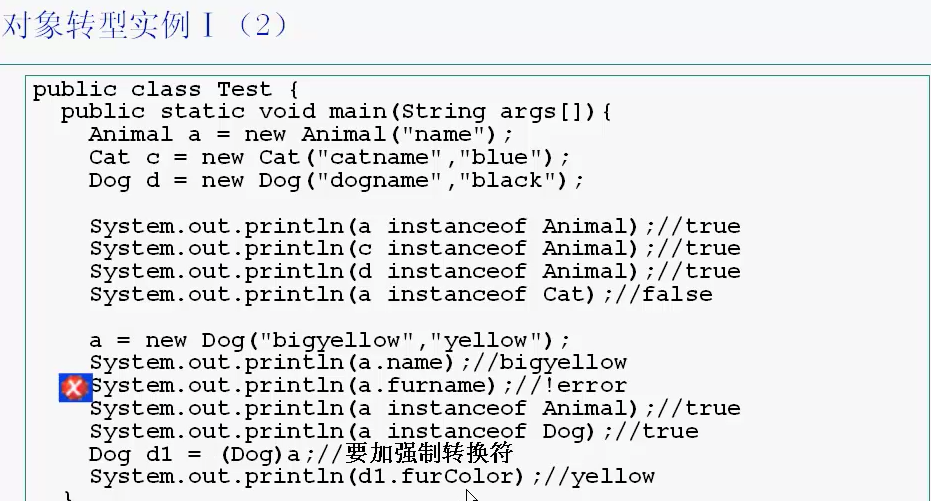
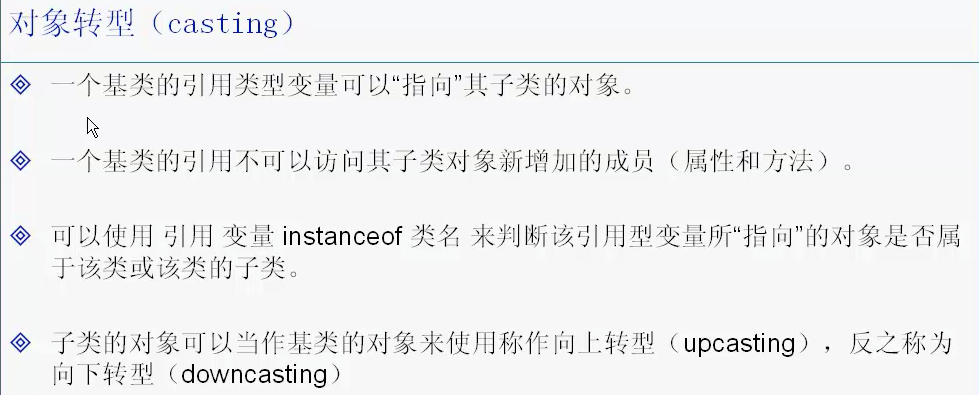


Hashcode被JVM用来记录每个对象在内存中的位置

equals()方法，Object基类提供这个方法，建议每个类重写这个方法，注意,instanceof操作符判断一个对象属不属于一个特定的类。注意，如果对象属于这个类，那么也必定属于这个类的父类，也就是说obj instanceof Object也是返回true。

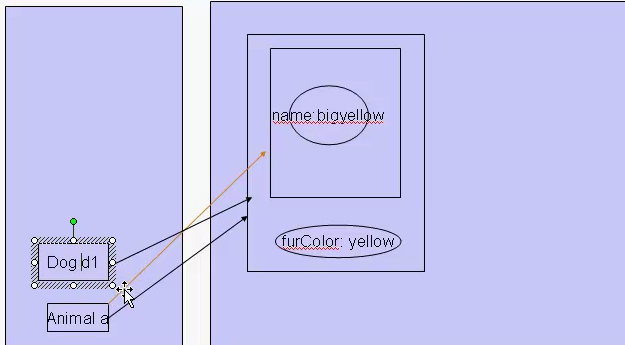
这个运算符常用于缩小范围，如一个方法的参数是Object, show(Object o)，传进去的实参必定是个Object或其子类，所以可以用instanceof来判断传进去的是不是你需要的特定子类





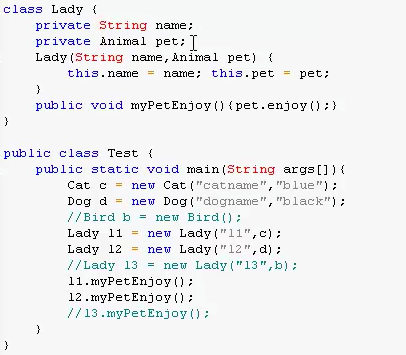
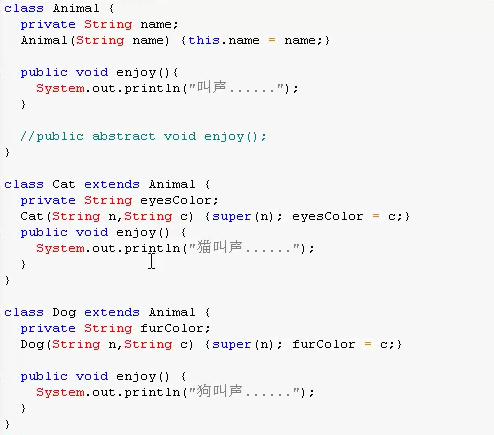
分析上面程序的内存：

当执行到时，原本Animal a的引用指向了一个新创建的它的子类Dog对象，然而这个引用a还是只把这个Dog对象当成Animal来看待，所以不能用a引用访问Dog的特有成员furColor，所以出错。注意，这里a虽然是被看成Animal，但是他实际是个Dog对象，所以返回的是true。要让a看到Dog的对象，就要进行强制转型，这句话创建了一个新引用d1，他指向a所指向的Dog对象，如果不加强制转换，此表达式将出错，因为无法将一个被看做是Animal的引用赋值给一个Dog引用，Animal不是一只Dog





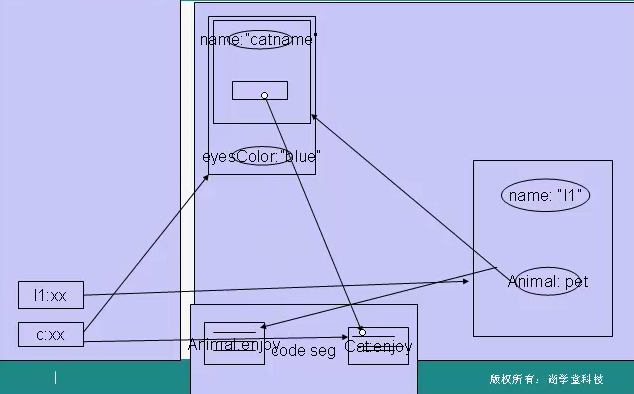
## 多态，抽象类，final，接口



看Lady的构造器，为什么定义成Animal呢？定义成Animal提供了最大程度的灵活性，可以养任何宠物

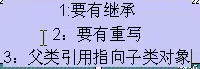
内存分析上面程序：注意，方法在内存中被存在一块单独的叫做code segment的区域，对象中有一个对方法的指针，指向方法所在的位置。当执行到时，形参pet的值等于传进去的Cat c的值，指向这只猫，然而由于形参是Animal类型的，所以此时pet成员变量只能看到Animal部分，所以不能访问Cat类的eyesColor对象。

然而执行时，调用的是pet.enjoy( )方法，可以看到方法的引用是在基类中的，所以可以访问。此时就是体现动态绑定的时刻，方法的调用并不是调用的Animal类的enjoy方法（引用pet看到的是Animal类型），而是调用的Cat类的enjoy方法，方法的调用是在运行时根据这个对象实际的类型决定的，并不是根据引用类型



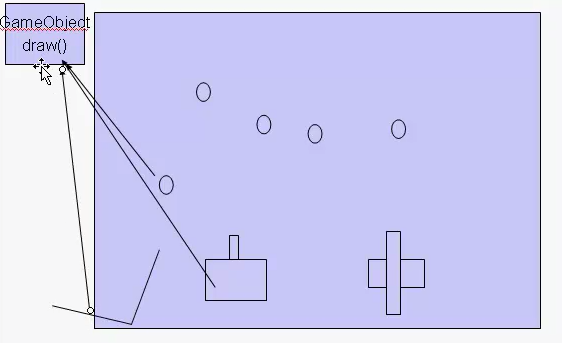
动态绑定（多态）把可扩展性发挥到了极致，我们可以非常简单的增加一个新类，而不去改变原来程序的任何结构

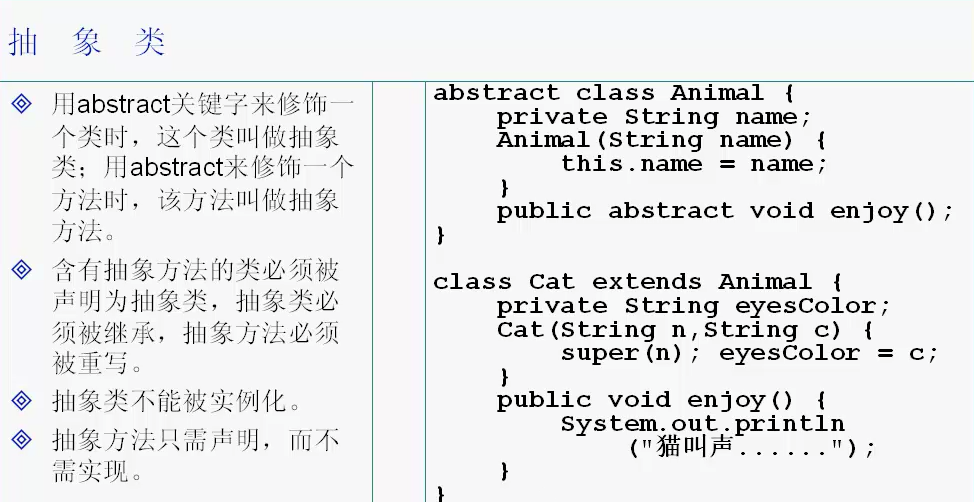


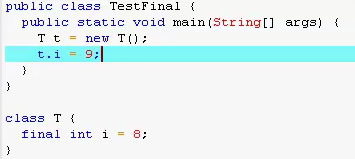
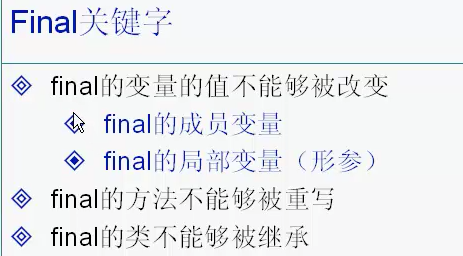
多态的三个必要条件

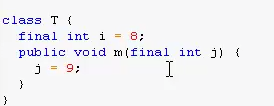
思考：雷电游戏的设计，设计一个基类，其他类都是它的子类，各自有不同的draw方法实现，利用多态可以让每个对象调用自己的方法，比如GameObject o = XXXX

o.draw( )。且提供了很大的扩展性，一旦有新的GameObject，也可以很容易的实现它的代码



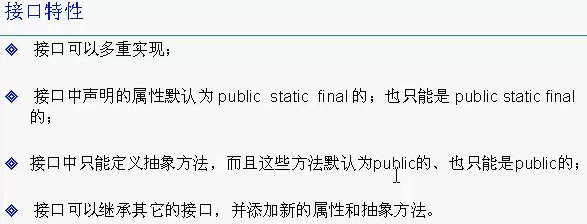
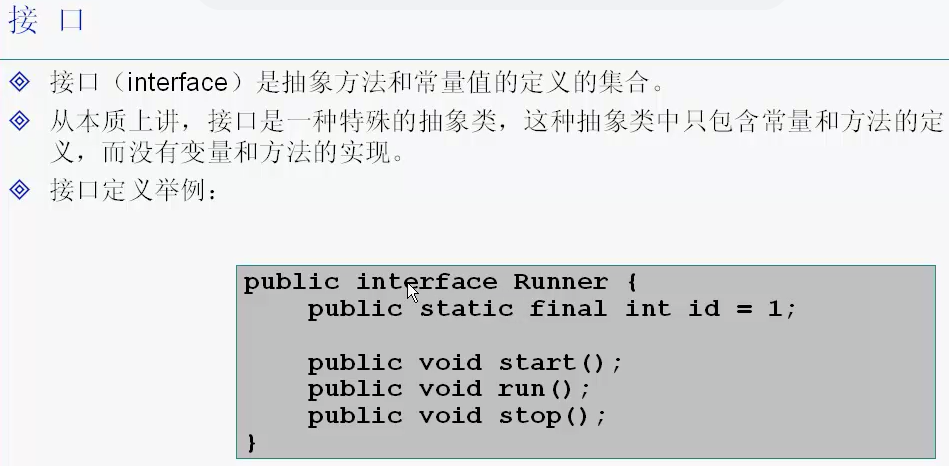
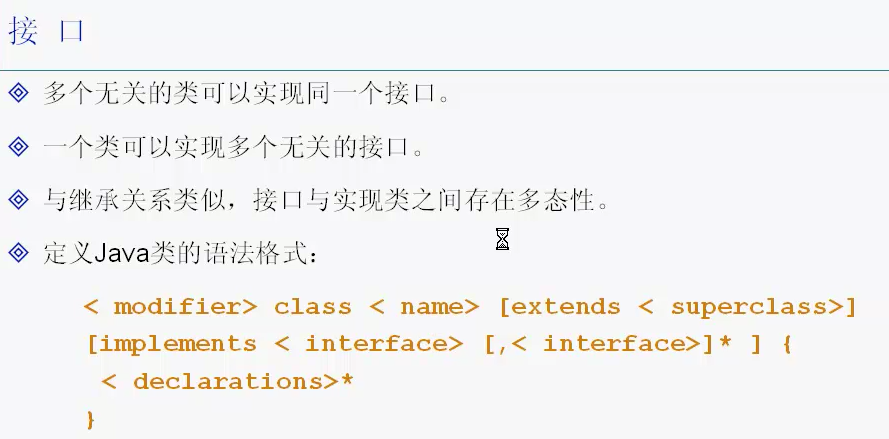


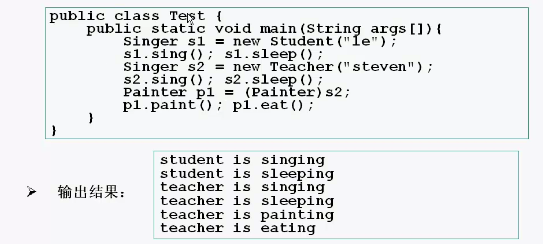
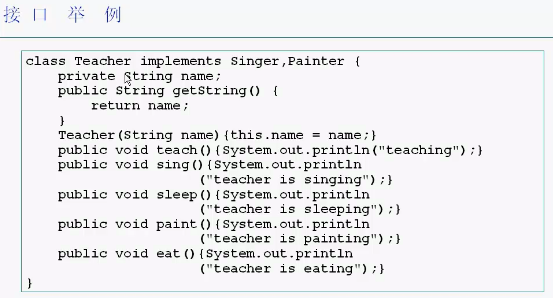
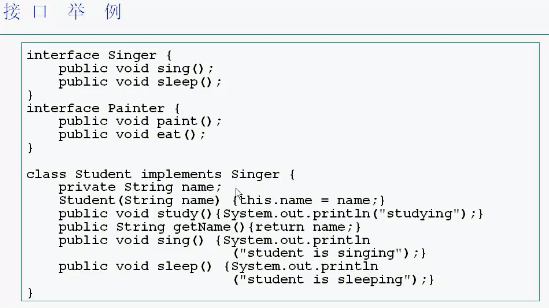
错误，不能改变i的值

错误，如果指定形参为final,那么不能在方法中改变这个参数。

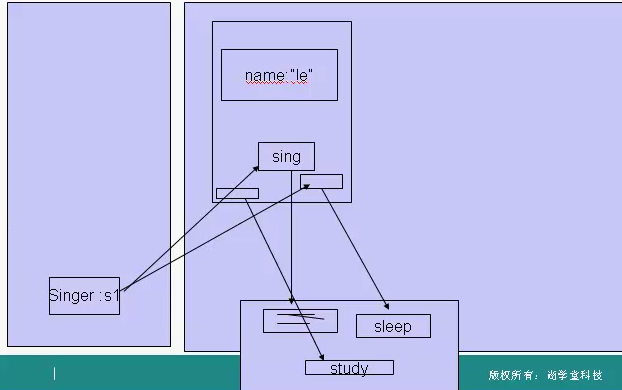
这样做的一个应用是，假设你有一个方法是, public void m( final Cat c )

{ c.doSotimes( ); }，此时形参是一个对象的引用，这样就可以避免在方法中把这个形参的引用指向别的地方





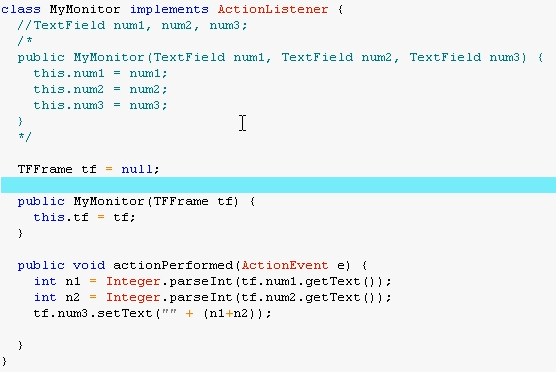
内存分析：Singer的引用s1指向一个Student对象，但是只能看到sing和sleep方法



同样的，Singer s2指向一个Teacher对象，但是只能看到sing和sleep方法，然后把s2强制转换为Painter，s2实际指向Teacher对象，Teacher类实现了Painter接口，所以可以转换，这时Painter p1也指向同一个Teacher对象，但是只能看到paint和eat方法



## 持有对方的引用，



在MyMonitor上需要访问TFFrame里TextField的值，一种方法是把这三个TextField的引用都传递给MyMonitor构造器，如MyMonitor(num1, num2, num3)，这样做不太优雅，可以考虑把让MyMonitor类持有一个TFFrame类的引用，这样就能访问TFFrame类向外暴露的成员了。