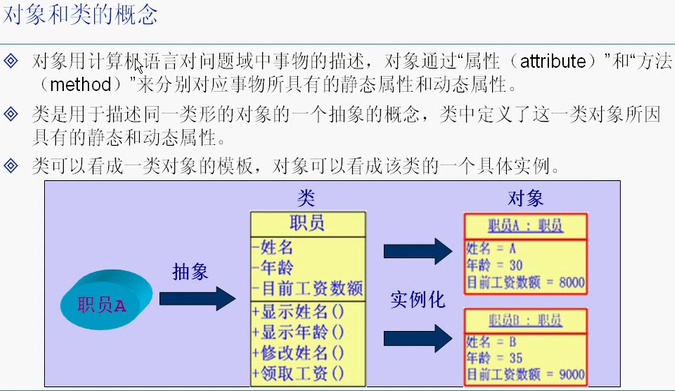
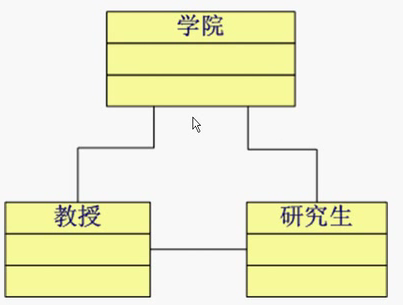
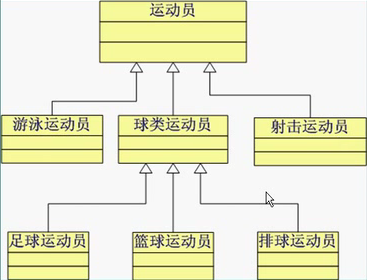
1. 重要的面向对象思想：拿到一个问题是，不要考虑我首先要干什么，其次要干什么，最后要干什么等等，而是应该首先考虑在这个问题中应该有哪几个类，哪几个对象；其次考虑在这几个类中应该有哪几个属性，哪几个方法；第三、再考虑类和类之间具有什么关系。
   1. 例子：我开车去新疆，这句话中的类和方法是什么
      1. 类：Driver(name,sex;drive())，Car(grand,size;go(),start(),stop()…)…
2. 对象和类的概念：



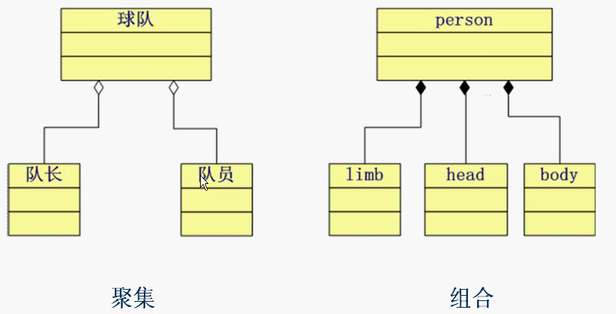
1. 类(对象)之间的关系
   1. 关联关系



* 1. 继承关系(一般和特殊，只要“xx是一种xx”能说通，就可以考虑用继承关系)

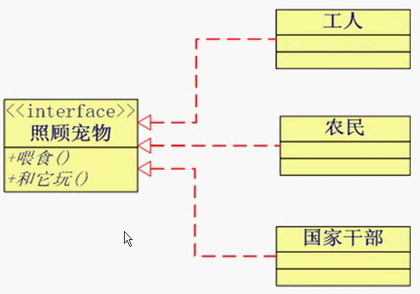


* 1. 聚合关系(整体和部分，只要“xx是xx的一部分”能说通，就可以考虑用聚合关系)

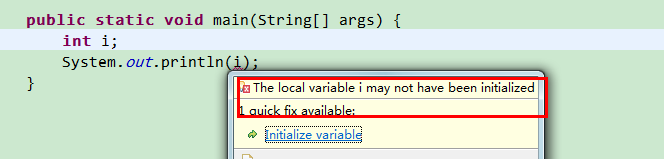


说明：聚集松耦合；组合紧耦合，密不可分。聚集和组合都叫聚合。

* 1. 实现关系

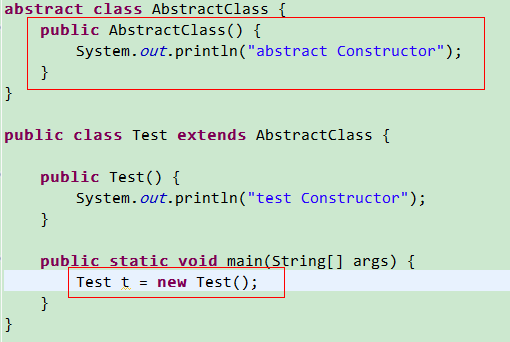


1. Java类中，在定义**成员变量**时可以对其初始化，如果不对其初始化，Java使用默认的值对其初始化。成员变量的作用范围是整个类体。(**局部变量**Java不会对其初始化，使用局部变量必须为其初始化；**有final修饰符的成员变量没有默认值，必须在构造器结束之前被赋予一个明确的值**)

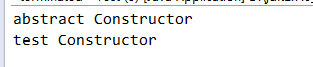


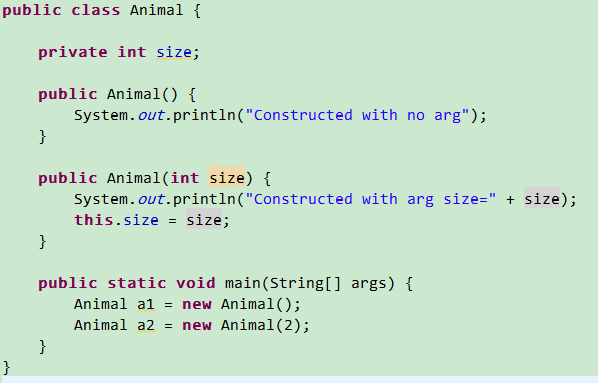


1. 构造函数(构造函数**与类同名**且**没有返回值**:void也不行)，采用new来创建对象时，会调用该类的构造方法。构造函数不会被继承。每个类，至少都会有一个构造函数，就算是***抽象类，也有构造函数***。



输出：

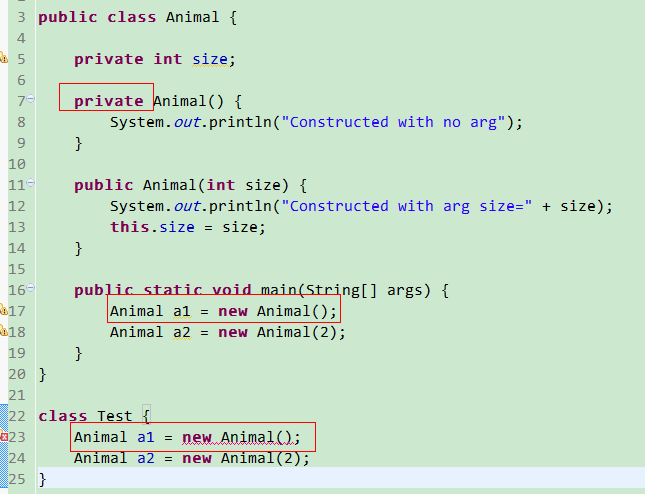




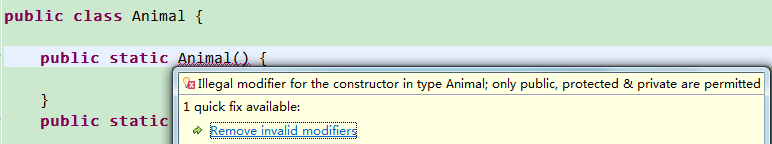
输出：



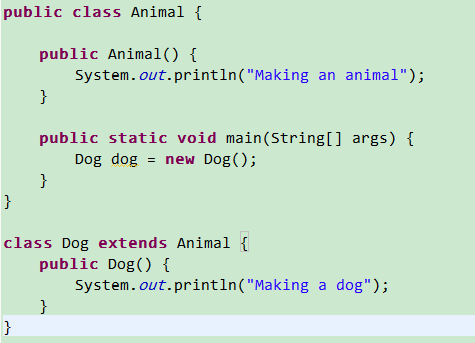
* 1. 当没有指定构造方法时，编译器会自动为其添加空的构造方法：public 类名() {}。
  2. 一旦定义了构造方法，编译器就不会为其添加构造方法(编译器只会在你完全没有设定构造函数时才会自动添加空的构造方法)。
  3. 在自己定义了构造方法后，最好能写上无参的构造函数让人可以选择使用默认值。
  4. 构造函数可以是私有的，意味着该类以外不能存取，即不能在类外通过new创建该对象。



* 1. 构造方法不能声明为static的，只能被public、protected、private三种修饰



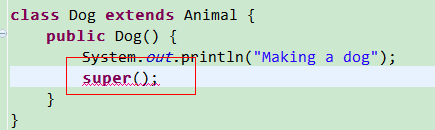
1. 继承中的构造方法：
   1. 子类的构造的过程**必须**调用其基类的构造方法(启动构造函数的连锁反应，即构造函数链，一直执行到Object类的构造函数为止。这就像小孩不能在父母出生之前出生一样)！

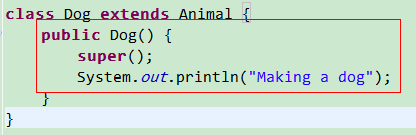


输出:



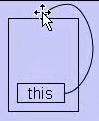
* 1. 子类可以在自己的构造方法中使用super(arg)调用基类的构造方法，使用this(arg)调用本类的另外的构造方法。如果使用了super，必须写在子类构造方法的第一行(子类的构造函数会立即调用父类的构造函数)。调用父类构造函数的唯一方法就是调用super()。如果我们没有写super()，***编译器会自动为我们加上没有参数的super()***。



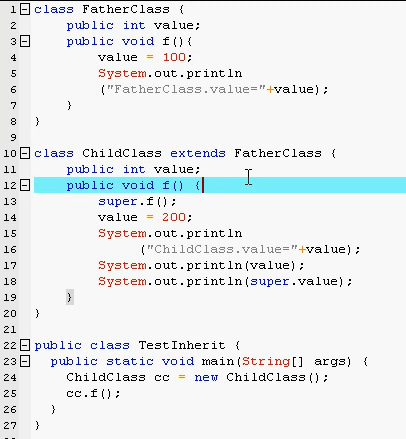


* 1. 如果子类的构造方法中没有显式地调用基类的构造方法，系统会默认调用基类**无参的**构造方法(编译器自动加上的super()方法)。
  2. 构造函数中使用super(arg)来调用父类的构造函数，使用this(arg)来调用自己的某个构造函数，且super(arg)和this(arg)必须放在构造函数的第一行。这就意味着**super(arg)和this(arg)不能同时使用**。

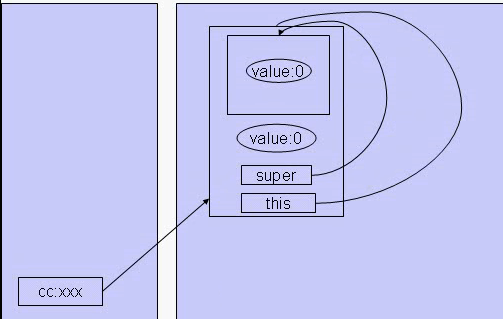
1. 方法的重载：一个类中可以定义有相同的名字，但是参数不同的多个方法。调用时，会根据不同的参数列表选择相对应的方法。(构造方法也可以重载)
   1. 参数不同：1、参数个数不一样；2、参数类型不一样。(满足其一即可)
   2. 注：方法的返回类型不同不构成重载。
2. 方法的覆盖：子类中覆盖父类中的方法要遵循以下规则：
   1. 参数必须要一样，且返回类型必须要兼容
   2. 不能降低方法的存取权限
3. this关键字：在类方法定义中使用的this关键字代表**使用**该方法的对象的引用。
   1. this可以看作是一个变量，它的值是当前对象的引用。
   2. 你对哪个对象调的这个方法，this指的就是谁。

指向自身的引用

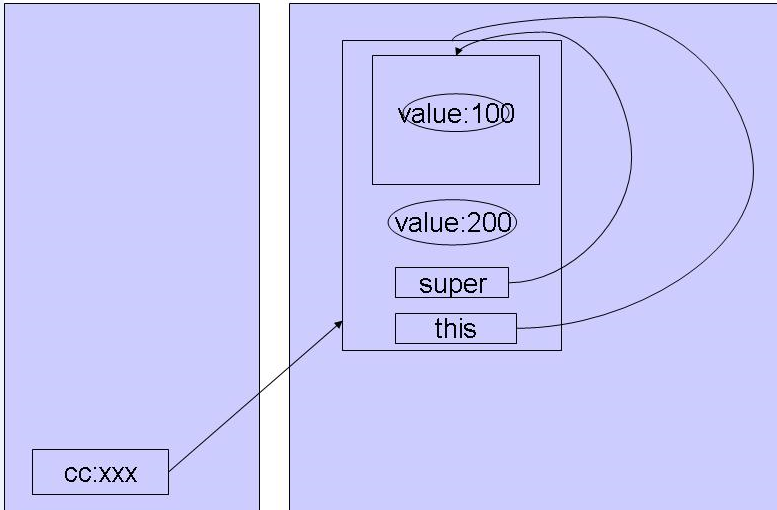
1. super关键字：在Java类中使用super来引用基类的成分



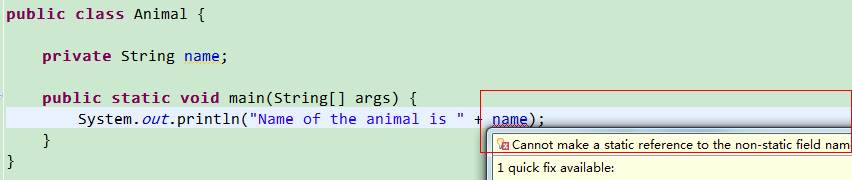
ChildClass cc = new ChildClass()执行完成后内存布局：

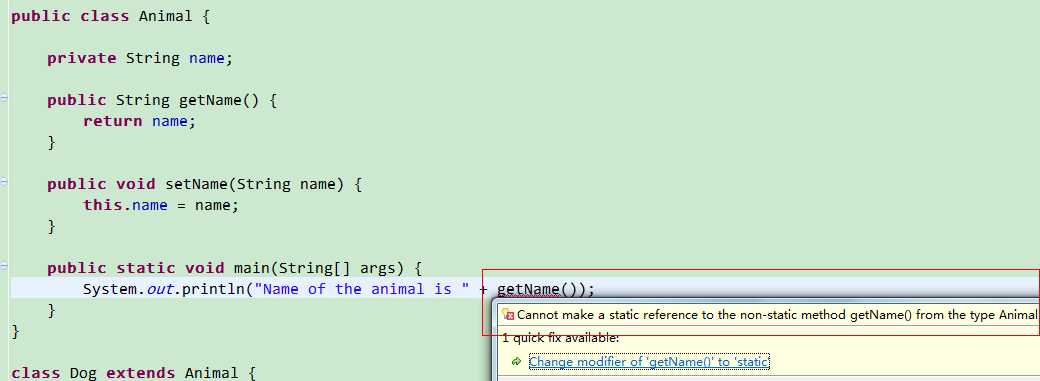


**cc.f()调用完毕后:**

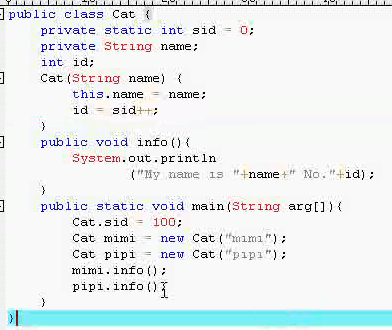


1. static关键字：(被static声明的成员变量以及字符串常量存放在data seg内存中)
   1. 在类中，用static声明的成员变量(即静态变量)，它为该类的公用变量(属于类，而不是属于某一个对象，所有对象共享该static变量)，在第一次使用时被初始化，对于该类的所有对象来说，static成员变量只有一份，即**都相同**。
      1. 实例变量：每个实例一个
      2. 静态变量：每个类一个
   2. **静态变量**是在**类被加载**(第一次创建该类的实例，或者使用该类的静态变量或静态方法)时初始化的(包括**静态语句块**)。
      1. 静态变量会在该类的任何对象创建之前就完成初始化
      2. 静态变量会在该类的任何静态方法执行之前就初始化。
   3. 用static声明的方法为静态方法(标记出不需要类实例的方法，即不用new对象，直接使用Class.method()调用即可)，在调用该方法时，不会将对象的引用传递给它，所以在**static方法中不可以访问非static的成员**。静态方法不再是针对某个对象调用，而是在无关特定类的实例情况下执行、通过类名直接调用的，在此情况下，静态的方法也不会知道可以使用哪个实例变量值，所以**不能访问非static成员(包括非静态方法和非静态变量)**。(静态的方法是不知道堆上有哪些实例的)

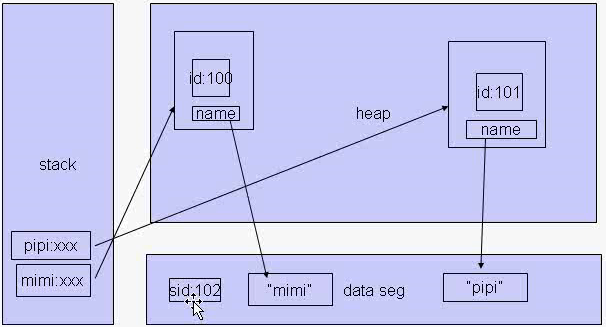




* + 1. 以类的名称调用静态的方法：Math.min(88, 83)
    2. 以引用变量的名称调用非静态的方法：Song s = new Song(); s.play();
  1. 可以通过对象引用或类名(不需要实例化)访问静态成员(通过对象引用来访问，该静态成员依然还是静态的)。

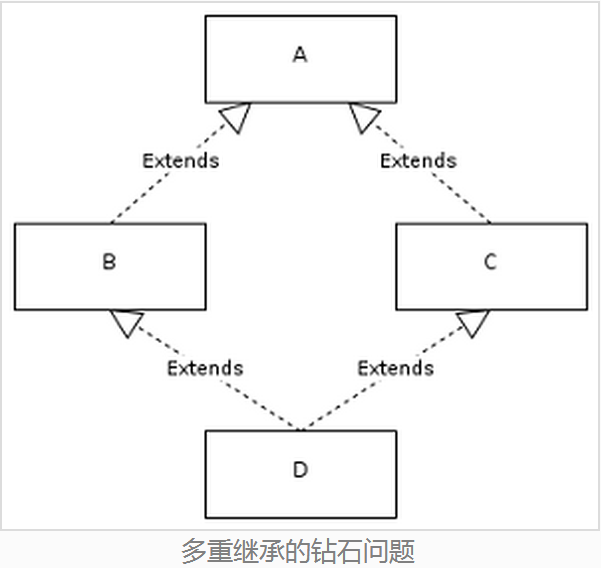


内存分析：



* 1. Java中static方法不能被覆盖，因为方法覆盖是基于运行时动态绑定的，而static方法是编译时静态绑定的。static方法跟类的任何实例都不相关，所以概念上不适用。

1. 类的继承与权限控制
   1. Java中使用extends关键字实现类的继承机制。通过继承，子类自动拥有了基类的所有成员(成员变量和方法)。class Child extends Parent {}
      1. Java只支持单继承，不允许多继承：一个子类只能有一个基类。一个基类可以派生出多个子类(注：**interface可以继承多个interfaces**)。Java不支持多重继承的主要原因有两个：
         1. 简单：多重继承的“菱形继承”问题



有两个类B和C继承自A。假设B和C都继承了A的方法并且进行了覆盖，编写了自己的实现。假设D通过多重继承继承了B和C，那么D应该继承B和C的重载方法，那么它应该继承哪个的呢？是B的还是C的呢？

C++中经常会掉入这个陷阱，虽然它也提出了替代的方法来解决这个问题。我们在Java中就不会出现这个问题。就算两个接口拥有同样的方法，实现的类只会有一个方法，这个方法由实现的类编写。动态的加载类会让多重继承的实现变得困难。

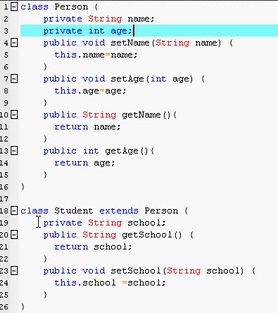
* + - 1. 很少使用：

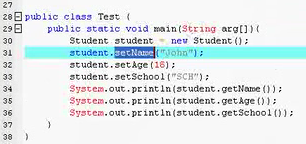
我们使用Java已经很长时间了，我们有多少次因为缺少多重继承而面临困难呢？我个人的经验是一次都没有。因为多重继承很少有机会被用到，所以更安全的做法是去掉它而保持简单性。

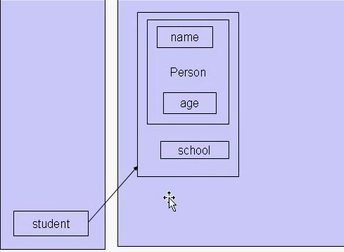
就算是碰到需要多重继承的情景，我们也可以找到替代方法。

我的观点是，去掉对多重继承的支持不是Java的缺陷，对开发者来说是件好事。

* + 1. 内存分析:

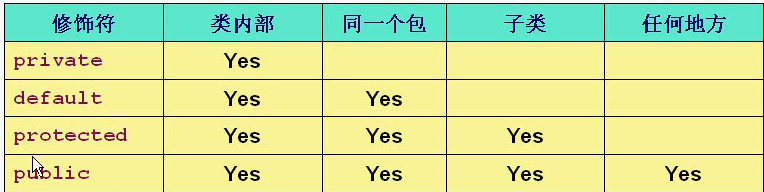






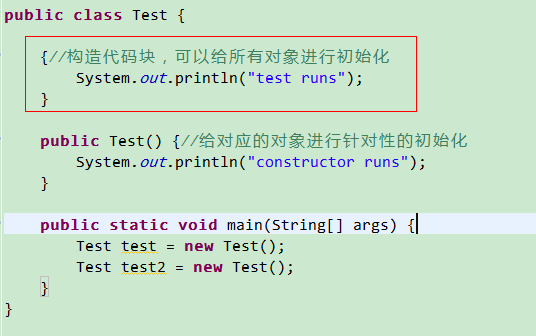
* 1. Java权限修饰符public、protected、default(包权限)、private。

**什么都不写的话**就是**default**权限：int i = 0；



* 1. 用于class的权限修饰符只可以用public和default(public类可以在任意地方被访问；default类只可以被同一个包内部的类访问)

1. 构造代码块：定义在类里面的写在{}里面的



输出：

