**3.1、static关键字（重点）**

这个关键字一直在见到，从未被真正使用过，而在Java中，static关键字可以用于定义属性及方法。

**3.1.1、使用static定义属性**

在讲解static定义属性操作之前，首先编写如下的一道程序。

**范例：**现在定义一个表示北京人的操作类，所有人所在的城市都是北京

|  |
| --- |
| class Person {  private String name ;  private int age ;  String country = "北京" ; // 此处暂时不封装  public Person(String name,int age) {  this.name = name ;  this.age = age ;  }  public String getInfo() {  return "姓名：" + this.name + "，年龄：" + this.age + "，城市：" + this.country ;  }  }  public class TestDemo {  public static void main(String args[]) {  Person per1 = new Person("张三",20) ;  Person per2 = new Person("李四",21) ;  Person per3 = new Person("王五",22) ;  System.out.println(per1.getInfo()) ;  System.out.println(per2.getInfo()) ;  System.out.println(per3.getInfo()) ;  }  } |

那么，以上的程序，现在的内存分配图如下。

很明显，现在不管有多少个Person对象，那么所有的country属性都应该是相同的内容，那么现在请思考一下，如果有一天时光穿越了，北京变为燕京了，这个时候如果Person类已经有了5000W个对象产生了，你认为此时的代价是什么？

5000W个Person对象都需要修改country的内容，那么如果按照修改一个对象的country属性内容要花费1ms来计算，那么5000W个对象的修改时间是：50000S = 很多分钟，那么现在就可以发现问题：

· country属性应该是一个公共的内容，但是以上代码是将其分配给了每一个对象；

· 对country属性维护的时候要考虑的对象太多了，也不方便。

**所以，如果要想将country与普通属性相区分，并且表示公共的概念，则应该使用static关键字完成。**

|  |
| --- |
| class Person {  private String name ;  private int age ;  **static String country = "北京" ;** // 此处暂时不封装  public Person(String name,int age) {  this.name = name ;  this.age = age ;  }  public String getInfo() {  return "姓名：" + this.name + "，年龄：" + this.age + "，城市：" + this.country ;  }  }  public class TestDemo {  public static void main(String args[]) {  Person per1 = new Person("张三",20) ;  Person per2 = new Person("李四",21) ;  Person per3 = new Person("王五",22) ;  per1.country = "燕京" ;  System.out.println(per1.getInfo()) ;  System.out.println(per2.getInfo()) ;  System.out.println(per3.getInfo()) ;  }  } |

那么，现在一个对象对country属性所做的修改，一定会影响到其他对象，但是这里有一个问题了，既然使用了static定义的属性表示的是公共属性，那么如果现在由某一个对象修改是不合适的，应该由所有对象的集合的最大代表，就是类了，即：static属性最好的调用，是通过“类名称.static属性”的方式来完成。

|  |
| --- |
| Person.country = "燕京" ; // 类.static属性 |

通过本程序，就应该清楚以下几点：

· 使用static定义的属性不在堆内存之中保存，保存在全局数据区；

· 使用static定义的属性表示类属性，类属性可以由类名称直接进行调用；

· static属性虽然定义在了类之中，但是其可以在没有实例化对象的时候进行调用（普通属性保存在堆内存里，而static属性保存在全局数据区之中）；

**提示：在以后的开发之中，首先想到的不是static属性，就是普通属性。**

**3.1.2、使用static定义方法**

除了使用static定义属性之外，方法上也可以使用static进行定义，那么很明显，使用static定义的方法也可以在没有实例化对象产生的情况下由类名称直接进行调用。

|  |
| --- |
| class Person {  private String name ;  private int age ;  private static String country = "北京" ; // 此处暂时不封装  public Person(String name,int age) {  this.name = name ;  this.age = age ;  }  **public static void setCountry(String c) {**  **country = c ;**  **}**  **public static String getCountry() {**  **return country ;**  **}**  public String getInfo() {  return "姓名：" + this.name + "，年龄：" + this.age + "，城市：" + this.country ;  }  }  public class TestDemo {  public static void main(String args[]) {  **Person.setCountry("燕京") ; // 类.static方法()**  **System.out.println(Person.getCountry()) ;**  Person per1 = new Person("张三",20) ;  System.out.println(per1.getInfo()) ;  }  } |

以上的代码如果单独观察应该都不难理解，主要都是由于static定义的特殊性所决定的，但是现在的类之中就比较热闹了，因为分成了两派：**static派、非static派，而对于方法的操作有如下两个定义：**

· static定义的方法不能调用非static的方法或属性；

· 非static定义的方法可以调用static的属性或方法。

**讨论：**为什么要有这样的限制呢？

· 使用static定义的属性和方法，可以在没有实例化对象的时候使用；

· 非static定义的属性和方法，必须实例化对象之后才可以进行调用。

**3.1.3、主方法**

在讲解主方法之前，首先来思考一下小小的区别：在之前讲解Java方法定义的格式给过以下格式：如果一个方法在主类之中定义，并且由主方法直接调用的时候，那么前面必须有public static。

|  |
| --- |
| **public static** 返回值类型 方法名称 (参数列表) {  [return [返回值] ;]  } |

**范例：**观察代码

|  |
| --- |
| public class TestDemo {  public **static** void main(String args[]) {  print() ; // 直接调用  }  public **static** void print() {  System.out.println("Hello World .") ;  }  } |

按照之前所学习的概念来讲，此时，表示的是一个static方法调用其他的static方法，但是如果这个时候的print()方法上没有static呢？所有的非static方法几乎都有一个特点：**方法要由实例化对象调用**。

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  new TestDemo().print() ; // 对象调用  }  public void print() { // 非static方法  System.out.println("Hello World .") ;  }  } |

那么现在继续来观察主方法的组成，可以发现里面有很多的关键字：

· public：是一种访问权限，表示公共；

· static：此方法由类名称直接调用，执行类：java 类名称；

· void：主方法是一切的开始，开弓没有回头箭；

· main：系统规定的一个方法名称，执行类的时候默认找到此名称；

· String args[]：表示的是一些运行时参数，通过字符串接收。

**范例：**接收参数

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  for (int x = 0 ; x < args.length ; x ++) {  System.out.println(args[x]) ;  }  }  } |

所有的属性在执行类的时候使用空格分隔，例如：java TestDemo 参数 参数 参数 …

但是，由于在默认情况下空格作为参数的分隔符，那么如果说现在要输入的参数本身就存在空格呢？那么这些参数上可以使用“"”声明：java TestDemo "hello world" "hello mldn"。

**3.1.4、static关键字的使用**

在实际的工作之中，使用static的原因有二：

· 希望可以在没有实例化对象的时候可以轻松的执行类的某些操作；

· 现在希望表示出数据共享的概念。

**范例：**统计一个类产生的实例化对象个数

一个类肯定可以有多个实例化对象产生，那么现在希望增加一个统计操作，可以统计出一个类之中的所产生的实例化对象的个数。如果现在要产生新的实例化对象，则一定会调用类中的构造方法，所以可以在构造方法中增加统计，而且这个统计的变量，应该是所有对象共享的，那么应该将其定义为static属性。

|  |
| --- |
| class Person {  private static int count = 0 ;  public Person() { // 构造方法一定会调用  System.out.println("对象个数：" + ++ count) ;  }  }  public class TestDemo {  public static void main(String args[]) {  new Person() ;new Person() ;new Person() ;  }  } |

**范例：**为类的属性自动命名

例如，在一个类之中可能会有多个构造方法，那么假设类之中存在了一个name属性，要求这个属性一定要有内容，即：如果用户调用了无参构造方法，那么name也需要有一个具体的内容出现，所以这个时候就可以利用static完成。

|  |
| --- |
| class Book {  private static int count = 0 ;  private String title ;  public Book() { // 自动命名  this("NOTITLE - " + count ++) ;  }  public Book(String title) {  this.title = title ;  }  public String getTitle() {  return this.title ;  }  }  public class TestDemo {  public static void main(String args[]) {  System.out.println(new Book().getTitle()) ;  System.out.println(new Book("Java SE").getTitle()) ;  System.out.println(new Book().getTitle()) ;  }} |

以上的两个代码只是为了说明static声明变量的特点，实际之中基本上很少去这样使用，不过如果出现了类似的应用，应该能够反应出来其实现的基本原理。

**3.2、代码块（了解）**

代码块是在程序之中使用“{}”定义起来的一段程序，而根据代码块声明位置以及声明关键字的不同，代码块一共分为四种：普通代码块、构造块、静态块、同步块（暂时不讲解）。

**3.2.1、普通代码块**

普通代码块是定义在方法之中的代码块，在讲解此操作之前，首先来观察以下代码。

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  if (true) {  int x = 10 ; // 局部变量  System.out.println("x = " + x) ;  }  int x = 100 ; // 全局变量  System.out.println("x = " + x) ;  }  } |

普通代码块就是if取消。

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  { // 普通代码块  int x = 10 ; // 局部变量  System.out.println("x = " + x) ;  }  int x = 100 ; // 全局变量  System.out.println("x = " + x) ;  }  } |

几乎用不到的，知道就行了。

**3.2.2、构造块**

普通代码块是定义在方法之中的，而构造块是定义在类之中的代码块。

|  |
| --- |
| class Person {  public Person() {  System.out.println("构造方法。") ;  }  { // 构造块  System.out.println("构造块。") ;  }  }  public class TestDemo {  public static void main(String args[]) {  new Person() ; new Person() ; new Person() ;  }  } |

可以发现，构造块优先于构造方法执行，而且每当有一个新的实例化对象产生的时候，就会出现构造块的执行。

**3.2.3、静态块**

静态块也是定义在类之中的，如果一个构造块上使用了static关键字进行定义的话，那么就表示静态块，但是静态块要考虑两种情况：

**情况一：**在非主类之中定义的静态块

|  |
| --- |
| class Person {  public Person() {  System.out.println("构造方法。") ;  }  { // 构造块  System.out.println("构造块。") ;  }  static {  System.out.println("静态块。") ;  }  }  public class TestDemo {  public static void main(String args[]) {  new Person() ; new Person() ; new Person() ;  }  } |

可以发现，静态块优先于构造块执行，而且不管有多少个实例化对象产生，静态块只调用一次。

**情况二：**在主类中定义的静态块

|  |
| --- |
| public class TestDemo {  static {  System.out.println("静态块。") ;  }  public static void main(String args[]) {  System.out.println("Hello World .") ;  }  } |

在主类之中的静态块优先于主方法执行。那么既然有静态块有这个特点，实际上就可以简单的修改一下。

**有意思的操作：**不使用主方法输出“Hello World.”，彻底终结了

|  |
| --- |
| public class TestDemo {  static {  System.out.println("Hello World .") ;  System.exit(1) ;  }  } |

以上代码在JDK 1.7之前都可以正常使用，**JDK 1.7**之后完蛋了，bug终于解决。

**3.3、思考题**

要求完成一个系统的登陆程序，通过初始化参数输入用户名和密码，如果用户名是“mldn”，密码是“hello”，则显示欢迎光临的信息，如果不是则显示“登陆失败”的信息。

设计的时候多考虑一下类的组成结构，以及可能出现的问题。

如果说现在什么都不考虑，代码编写如下：

|  |
| --- |
| public class LoginDemo {  public static void main(String args[]) {  if ("mldn".equals(args[0]) && "hello".equals(args[1])) {  System.out.println("用户登录成功！") ;  } else {  System.out.println("用户登录失败！") ;  }  }  } |

这个时候的功能实际上是很容易实现的，但是本程序如果这样实现有两个问题：

· 如果用户没有输入参数，args这个接收初始化参数的数组，没有数据，会出现数组越界的异常，那么现在可以通过增加一些验证来完成；

|  |
| --- |
| public class LoginDemo {  public static void main(String args[]) {  if (args.length != 2) { // 参数不足两个  System.out.println("输入的参数错误！") ;  System.exit(1) ; // 程序退出  }  if ("mldn".equals(args[0]) && "hello".equals(args[1])) {  System.out.println("用户登录成功！") ;  } else {  System.out.println("用户登录失败！") ;  }  }  } |

· 考虑程序结构，主方法之中代码太多了，不应该如此复杂，主方法不需要关注细节，只关注使用。

指纹打卡过程之中：将手指头放到扫描器上，关注是否打卡成功。所以在主方法之中应该也关心数据的输入，和登3录信息的返回。打卡流程：**客户端（信息） 🡺 接收信息并验证 🡺 指纹库**。

**范例：**定义一个只负责数据验证的类

|  |
| --- |
| class LoginValidate { // 登录验证  private String username ;  private String password ;  public LoginValidate(String username,String password) {  this.username = username ;  this.password = password ;  }  public boolean isValidate() { // 验证数据  if ("mldn".equals(this.username) && "hello".equals(this.password)) {  return true ;  }  return false ;  }  } |

**范例：**定义采集信息类

· 接收输入数据，验证输入数据，调用验证类。

|  |
| --- |
| class Operate { // 操作类  private String data[] ; // 输入数据  public Operate(String data[]) {  this.data = data ;  this.exit() ; // 构造方法自动完成验证  }  public void exit() {  if (this.data.length != 2) {  System.out.println("输入的参数错误！") ;  System.exit(1) ; // 程序退出  }  }  public String getInfo() {  if (new LoginValidate(this.data[0],this.data[1]).isValidate()){  return "用户登录成功！" ;  } else {  return "用户登录失败！" ;  }  }  } |

**范例：**客户端，只负责放入数据和接收信息

|  |
| --- |
| public class LoginDemo {  public static void main(String args[]) {  System.out.println(new Operate(args).getInfo()) ;  }  } |

类设计的原则（没有原则）：程序是离不开生活的，在生活之中的可以说的通的结构，程序之中都可以说得通。而且一个功能类只完成本功能类有关的操作。

**3.4、答疑：三个引用传递**

上一次的三个引用程序，分析原则：

· 引用传递：一块堆内存同时被多个栈指向称为引用传递，方法接收对象，和直接赋予一个对象；

· 字符串：字符串的内容一旦声明则不可改变，如果要改变字符串对象的内容，就相当于修改指向，为了简化理解：将字符串理解为基本数据类型那样，不受内存控制。

**范例：**第一道引用范例

|  |
| --- |
| class Demo {  private int data ;  public Demo(int data) {  this.data = data ;  }  public void setData(int data) {  this.data = data ;  }  public int getData() {  return this.data ;  }  }  class Util {  public static void fun(Demo temp) {  temp.setData(30) ;  }  }  public class TestDemo {  public static void main(String args[]) {  Demo demo = new Demo(100) ;  Util.fun(demo) ;  System.out.println(demo.getData()) ;  }  } |

**范例：**第二道引用范例，字符串，目的：验证字符串内容一旦声明不可改变，字符串对象改变的是内存地址的指向

|  |
| --- |
| class Util {  public static void fun(String temp) {  temp = "World" ;  }  }  public class TestDemo {  public static void main(String args[]) {  String str = "Hello" ; // 直接赋值  Util.fun(str) ;  System.out.println(str) ;  }  } |

**范例：**第三道引用

|  |
| --- |
| class Demo {  private String data ;  public Demo(String data) {  this.data = data ;  }  public void setData(String data) {  this.data = data ;  }  public String getData() {  return this.data ;  }  }  class Util {  public static void fun(Demo temp) {  temp.setData("World") ;  }  }  public class TestDemo {  public static void main(String args[]) {  Demo demo = new Demo("Hello") ;  Util.fun(demo) ;  System.out.println(demo.getData()) ;  }  } |

不是你们疯绝对就是我疯，但是我决定了，在我疯之前，把你们弄疯了先。

**3.5、内部类（重点）**

如果现在的时间是在2002 ~ 2005年之前，内部类都不作为重点，都是一个可有可无的内容，但是到了2005年之后，框架技术开始发展（Spring + Hibernate + Struts），内部类在J2EE的开发之中，开始使用广泛，到了今天，2009年之后，Android开发开始火了，所以内部类也大量的使用在Android开发之中。

强调：对于内部类今天先学会语法，至于用法，还需要结合实际的问题分析。如果从使用角度而言，内部类的应用至少按照课程设计来讲，应该在2个月之后。

**3.5.1、内部类的基本概念**

内部类指的是在一个类的内部定义了其他类的情况，下面首先来看一下内部类的基本定义形式。

|  |
| --- |
| class Outer { // 外部类  private String msg = "Hello World " ; // 普通属性  **class Inner { // 内部类**  **public void print() {**  **System.out.println(msg) ;**  **}**  **}**  public void fun() {  Inner in = new Inner() ;  in.print() ;  }  }  public class TestDemo {  public static void main(String args[]) {  Outer out = new Outer() ;  out.fun() ;  }  } |

通过如上的代码相信首先最应该感受到的是：内部类属于在一个类内部定义的新的结构，不过按照类的基本组成来讲，一个类之中应该由属性和方法所组成，但是这个时候又多了一个类，这样的结构并不好，所以内部类本身最大的缺点破坏了程序的结构。

但是内部类本身也有自己的优点，而这个优点如果要想发现，最好的做法是将内部类拿到外面来，变为两个类。

|  |
| --- |
| class Outer { // 外部类  private String msg = "Hello World " ; // 普通属性  public void fun() {  Inner in = new Inner(**this**) ;  in.print() ;  }  **public String getMsg() {**  **return this.msg ;**  **}**  }  **class Inner { // 内部类**  **private Outer out = null ;**  **public Inner(Outer out) {**  **this.out = out ;**  **}**  **public void print() {**  **System.out.println(this.out.getMsg()) ;**  **}**  **}**  public class TestDemo {  public static void main(String args[]) {  Outer out = new Outer() ;  out.fun() ;  }  } |

这个时候感觉到代码所增加的复杂度是非常高的，那么很明显了，内部类的最大优点：**可以方便的访问外部类的私有操作，或者是由外部类方便的访问内部类的私有操作。**

|  |
| --- |
| class Outer { // 外部类  private String msg = "Hello World " ; // 普通属性  class Inner { // 内部类  private String info = "你好！" ; // 内部类私有  public void print() {  **System.out.println(msg) ; // 内部类访问外部类**  }  }  public void fun() {  Inner in = new Inner() ;  in.print() ;  **System.out.println(in.info) ; // 外部类直接调用**  }  }  public class TestDemo {  public static void main(String args[]) {  Outer out = new Outer() ;  out.fun() ;  }  } |

但是这个时候一个严重的问题来了，在之前一直强调：只要是访问类中的属性前面都要有“this.”，即：以上的程序之中，内部类访问外部类私有属性的时候，应该是“this.msg”才对，但是在内部类使用this表示的只是内部类的当前对象，如果要想表示出外部类的当前对象，使用“外部类.this”来表示。

|  |
| --- |
| class Outer { // 外部类  private String msg = "Hello World " ; // 普通属性  class Inner { // 内部类  public void print() {  System.out.println(**Outer.this.msg**) ;// 内部类访问外部类，Android开发之中，此代码常见  }  }  public void fun() {  Inner in = new Inner() ;  in.print() ;  }  }  public class TestDemo {  public static void main(String args[]) {  Outer out = new Outer() ;  out.fun() ;  }  } |

以上的程序都是通过了外部类的方法操作了内部类，但是如何可以在不同的类操作内部类呢？

为了解释本程序，下面首先来观察一下内部类的\*.class文件的名称：Outer**$**Inner.class，作为Java的标识符，$也是一个标识符的组成元素，但是对于这样的元素，一直从未使用过，而$是在文件中的表示，但是换到了程序之中，每一个$表示一个“.”，即：如果换到了程序里面，内部类的类名称就是：Outer.Inner，所以下面就可以给出在外部实例化内部类对象的操作格式：

|  |
| --- |
| 外部类.内部类 内部类对象 = **new** 外部类().**new** 内部类() ; |

之所以实例化外部类对象，主要是因为内部类需要访问外部类之中的普通属性，那么普通属性只有在对象实例化之后才会被实例化（可以使用）。

|  |
| --- |
| class Outer { // 外部类  private String msg = "Hello World " ; // 普通属性  class Inner { // 内部类  public void print() {  System.out.println(Outer.this.msg) ;// 内部类访问外部类  }  }  }  public class TestDemo {  public static void main(String args[]) {  **Outer.Inner in = new Outer().new Inner() ; // 这种格式几乎不会出现**  in.print() ;  }  } |

但是，如果说现在内部类不希望被外面看见呢？那么可以继续进行封装操作。

|  |
| --- |
| class Outer { // 外部类  private String msg = "Hello World " ; // 普通属性  **private class Inner** { // 内部类  public void print() {  System.out.println(Outer.this.msg) ;// 内部类访问外部类  }  }  } |

而在日后所进行的开发之中，特别是Android开发之中，这种使用private class定义的内部类是比较常见的。

**3.5.2、使用static定义内部类**

**回顾：**使用static定义的属性和方法，是独立于类之外的，可以在没有实例化对象的时候调用，而static也同样可以进行内部类的定义，而使用了static定义的内部类，则就表示为“外部类”，并且只能访问外部类之中static类型的操作。

|  |
| --- |
| class Outer { // 外部类  private static String msg = "Hello World " ; // 普通属性  static class Inner { // 内部类 = ”外部类“  public void print() {  System.out.println(Outer.msg) ;// 内部类访问外部类  }  }  }  public class TestDemo {  public static void main(String args[]) {  Outer.Inner in = new Outer.Inner() ;  in.print() ;  }  } |

一般而言，这样的代码操作是出现次数不多的，代码现在要求可以看懂即可。

**3.5.3、在方法中定义内部类**

内部类理论上可以在类的任意位置上进行定义，这就包括代码块之中，或者是普通方法之中，而在以后的开发过程之中，在普通方法里面定义内部类的情况是最多的。

**范例：**在方法之中定义内部类

|  |
| --- |
| class Outer { // 外部类  private String msg = "Hello World " ; // 普通属性  public void fun() {  class Inner { // 方法中定义的内部类  public void print() {  System.out.println(Outer.this.msg) ;// 内部类访问外部类  }  }  Inner in = new Inner() ; // 产生内部类对象  in.print() ;  }  }  public class TestDemo {  public static void main(String args[]) {  new Outer().fun() ;  }  } |

但是一个内部类如果要定义在方法之中，并且要访问方法的参数或者是方法中定义变量的时候，这些参数或变量前一定要增加一个“**final**”关键字。

|  |
| --- |
| class Outer { // 外部类  private String msg = "Hello World " ; // 普通属性  public void fun(**final** int x) { // 参数  **final** String info = "Hello MLDN" ; // 变量  class Inner { // 方法中定义的内部类  public void print() {  System.out.println(Outer.this.msg) ;  System.out.println(x) ;  System.out.println(info) ;  }  }  Inner in = new Inner() ; // 产生内部类对象  in.print() ;  }  }  public class TestDemo {  public static void main(String args[]) {  new Outer().fun(30) ;  }  } |

以上所讲解的操作语法，属于内部类的基本概念，暂时用不到。

**3.6、实际问题（核心，第3个代码模型）**

**总结：**现在为止的折腾过程：

· 类和对象：类的组成由属性和方法、对象的产生及内存分配、构造方法、private封装、匿名对象，最终这些概念通过一个简单Java类融合在了一起，代码；

· static（补充）：定义公共属性使用static，准备；

· String类：只是对操作的方法给了一些提示，但是并没有使用到实际的开发之中，准备；

· 对象数组：一组对象方便管理；

· 引用传递：与this一起完成对象比较。

下面给出这样的一个要求，问，应该由什么组成？

· 一个人有一辆汽车，问，如何设计这个要求？

下面换个方式，先不用程序理解，使用数据表理解，如果说现在让你定义数据表，表示出以上的数据存储？可以使用两张数据表来解决问题：一个表示人的数据表，另外一个表示车的数据表，则数据库创建脚本如下：

|  |
| --- |
| DROP TABLE car PURGE ;  DROP TABLE member PURGE ;  DROP SEQUENCE member\_seq ;  DROP SEQUENCE car\_seq ;  CREATE SEQUENCE member\_seq ;  CREATE SEQUENCE car\_seq ;  CREATE TABLE member(  mid NUMBER ,  name VARCHAR2(50) NOT NULL ,  CONSTRAINT pk\_mid PRIMARY KEY (mid)  ) ;  CREATE TABLE car(  mid NUMBER ,  title VARCHAR2(50) ,  color VARCHAR2(10) ,  CONSTRAINT pk\_mid PRIMARY KEY (mid) ,  CONSTRAINT fk\_mid FOREIGN KEY (mid) REFERENCES member(mid) ON DELETE SET NULL  ) ; |

如果按照以上的思路去设计类的话，现在应该由两个类完成关系，那么**这两个类和数据表的对应关系：**

**· 类中的属性 = 表的字段；**

**· 类的名称 = 表的名称；**

**· 类的一个实例化对象 = 数据表的一行记录；**

**· 类的对象数组 = 数据表的多行记录。**

以后实际工作之中，简单java类的设计原则参考数据表。

**范例：**实现代码 —— 一个人有一辆车，一辆车属于一个人

· 开发步骤一：先完成各自简单java类的定义；

· 开发步骤二：设置关联关系。

|  |
| --- |
| class Member {  private int mid ;  private String name ;  private Car car ; // car == null表示没有车  public Member(int mid,String name) {  this.mid = mid ;  this.name = name ;  }  public void setCar(Car car) {  this.car = car ;  }  public Car getCar() {  return this.car ;  }  public String getMemberInfo() {  return "人员编号：" + this.mid + "，姓名：" + this.name ;  }  }  class Car {  private String title ;  private String color ;  private Member member ;  public Car(String title,String color) {  this.title = title ;  this.color = color ;  }  public void setMember(Member member) {  this.member = member ;  }  public Member getMember() {  return this.member ;  }  public String getCarInfo() {  return "汽车名字：" + this.title + "，颜色：" + this.color ;  }  }  public class TestDemo {  public static void main(String args[]) {  Member mem = new Member(1,"付云松") ;  Car car = new Car("BMW，别摸我","红色") ;  mem.setCar(car) ; // 一个人有一辆车  car.setMember(mem) ; // 一辆车属于一个人  System.out.println(mem.getMemberInfo()) ;  System.out.println(mem.getCar().getCarInfo()) ;  System.out.println("=====================") ;  System.out.println(car.getMember().getMemberInfo()) ;  }  } |

每一个人都有一个自己的孩子，每个孩子都有自己的车，那么又该如何设计呢？

现在有两种做法：

· **做法一：**定义一个孩子的类，于是在Member表中设置这个类的关系，这样的设计过于重复；

· **做法二：**孩子本身依然属于一个人，那么依然具备了人所具备的所有属性，那么直接做一个内部的关联。

|  |
| --- |
| class Member {  private int mid ;  private String name ;  private Car car ; // car == null表示没有车  private Member child ; // 一个人有一个孩子  public Member(int mid,String name) {  this.mid = mid ;  this.name = name ;  }  public void setCar(Car car) {  this.car = car ;  }  public Car getCar() {  return this.car ;  }  public void setChild(Member child) {  this.child = child ;  }  public Member getChild() {  return this.child ;  }  public String getMemberInfo() {  return "人员编号：" + this.mid + "，姓名：" + this.name ;  }  }  class Car {  private String title ;  private String color ;  private Member member ;  public Car(String title,String color) {  this.title = title ;  this.color = color ;  }  public void setMember(Member member) {  this.member = member ;  }  public Member getMember() {  return this.member ;  }  public String getCarInfo() {  return "汽车名字：" + this.title + "，颜色：" + this.color ;  }  }  public class TestDemo {  public static void main(String args[]) {  Member mem = new Member(1,"付云松") ;  Member chd = new Member(100,"付三") ;  Car car = new Car("BMW，别摸我","红色") ;  Car c = new Car("AUTO","黑色") ;  mem.setCar(car) ; // 一个人有一辆车  car.setMember(mem) ; // 一辆车属于一个人  mem.setChild(chd) ; // 一个人有一个孩子  chd.setCar(c) ; // 孩子有一辆车  c.setMember(chd) ; // 车属于孩子  System.out.println("找到孩子的车：" + mem.getChild().getCar().getCarInfo()) ;  }  } |

**3.7、核心思考题（核心）**

利用此关系模型，表示出emp和dept的关系，使用字段：

· **emp表：**empno、ename、job、sal、comm、mgr、deptno；

· **dept表：**deptno、dname、loc。

**关系：**

· 一个部门有多个雇员；

· 一个雇员有一个领导；

· 一个雇员属于一个部门。

|  |
| --- |
| class Emp {  private int empno ;  private String ename ;  private String job ;  private double sal ;  private double comm ;  private Emp mgr ;  private Dept dept ;  public Emp() {}  public Emp(int empno,String ename,String job,double sal,double comm) {  this.empno = empno ;  this.ename = ename ;  this.job = job ;  this.sal = sal ;  this.comm = comm ;  }  public void setDept(Dept dept) {  this.dept = dept ;  }  public Dept getDept() {  return this.dept ;  }  public void setMgr(Emp mgr) {  this.mgr = mgr ;  }  public Emp getMgr() {  return this.mgr ;  }  public String getEmpInfo() {  return "雇员编号：" + this.empno + "，姓名：" + this.ename + "，职位：" + this.job + "，工资：" + this.sal + "，佣金：" + this.comm ;  }  }  class Dept {  private int deptno ;  private String dname ;  private String loc ;  private Emp emps [] ; // 多个雇员  public Dept() {}  public Dept(int deptno,String dname,String loc) {  this.deptno = deptno ;  this.dname = dname ;  this.loc = loc ;  }  public void setEmps(Emp emps[]) {  this.emps = emps ;  }  public Emp[] getEmps() {  return this.emps ;  }  public String getDeptInfo() {  return "部门编号：" + this.deptno + "，部门名称：" + this.dname + "，位置：" + this.loc ;  }  }  public class TestDemo {  public static void main(String args[]) {  // 1、第一层配置关系  Dept dept = new Dept(10,"ACCOUNTING","New Yrok") ;  Emp empa = new Emp(7369,"SMITH","CLERK",800.0,0.0) ;  Emp empb = new Emp(7566,"ALLEN","MANAGER",2450.0,0.0) ;  Emp empc = new Emp(7839,"KING","PRESIDENT",5000.0,0.0) ;  // 设置雇员和领导的关系  empa.setMgr(empb) ;  empb.setMgr(empc) ;  // 每个雇员属于一个部门  empa.setDept(dept) ;  empb.setDept(dept) ;  empc.setDept(dept) ;  // 每一个部门有多个雇员  dept.setEmps(new Emp[]{empa,empb,empc}) ;  // 2、第二层取得关系  System.out.println(dept.getDeptInfo()) ;  for (int x = 0 ; x < dept.getEmps().length ; x ++) {  System.out.println(dept.getEmps()[x].getEmpInfo()) ;  if (dept.getEmps()[x].getMgr() != null) { // 有领导  System.out.println("\t" + dept.getEmps()[x].getMgr().getEmpInfo()) ;  }  }  }  } |

以后要求掌握的能力：根据数据表，写出数据的关系模型。

## 3.8、集成设计模式（理解）

清楚了以上的引用的关系操作之后，可以通过以上的数据模型，得到进一步的抽象概念。世界上任何的组成都可以通过以上的数据模型完成。

**范例：**要求抽象出本教室

|  |
| --- |
| class 学生 {}  class 老师 {}  class 电脑 {}  class 桌子 {}  class 椅子 {}  class 窗户 {}  class 门 {}  class 教室 {  private 学生 [] 对象 ;  private 老师 对象 ;  private 电脑 [] 对象 ;  private 桌子 [] 对象 ;  private 椅子 [] 对象 ;  private 窗户 [] 对象 ;  private 门 对象 ;  } |

**范例：**现在要求抽象出电脑

电脑的零件：显示器、鼠标、键盘、硬盘、主板、机箱、CPU、内存。

|  |
| --- |
| class 显示器 {}  class 鼠标 {}  class 键盘 {}  class 硬盘 {}  class 主板 {}  class 机箱 {  private 主板 对象 ;  private CPU 对象 ;  private 硬盘 对象 ;  private 内存 对象 ;  }  class CPU {}  class 内存 {}  class 主机 {  private 机箱 对象 ;  }  class 电脑 {  private 显示器 对象 ;  private 主机 对象 ;  private 鼠标 对象 ;  private 键盘 对象 ;  } |

这种基本的抽象能力一定要有。

# 4、总结

1、 static定义的属性和方法可以由类名称直接调用、static定义的属性表示公共属性；

2、 内部类的基本操作；

# 5、预习任务

继承性、super关键字、方法的覆写、final关键字。

# 6、作业

1、 引用传递好好分析分析，关键是画图；

2、 完成以下数据模型的建立：字段挑几个有代表性的即可，不需要写setter、getter

（1）、商品类别和商品，一对多关系。

要求1：根据一个商品组的信息找到所有的商品；

要求2：根据一个商品找到其所属的商品组的信息；

（2）、一个用户可以下多个定单，一对多关系；

要求1：根据一个用户找到他的所有定单；

要求2：根据一个定单找到其下单的用户；

（3）、管理员-管理员组-权限，简单多对多；

要求1：根据一个管理员的信息可以找到这个管理员所在的所有管理员组，并且列出每个管理员组的权限；

要求2：根据一个管理员组可以列出这个管理员组的权限，以及所有的管理员；

要求3：根据一个权限可以找到具备此权限的所有管理员组，并且列出每个管理员组的所有管理员。