# 《面向对象与多线程综合实验》指导书

本课程旨在使学生了解软件开发的一般过程，深入理解面向对象语言的基本概念和基本原理，理解和掌握继承与多态、异常处理、输入输出流、GUI设计、JDBC数据库操作、网络编程、多线程等技术；熟练掌握在Java语言环境下，上述技术的具体实现方法，并深入领会Java程序设计实用开发技术。

为达到上述目标，本课程计划让学生以迭代方式逐步编程开发一个小型档案管理系统，实现上述7个部分的训练，从而培养学生分析和解决问题的能力。该小型档案系统要求如下：

1. 系统基于C/S模式，包括客户端子系统，服务器端子系统
2. 用户需登录，验证口令通过后才能使用系统。用户分为系统管理人员、档案录入人员，档案浏览人员
3. 系统中相关用户信息、档案属性存放于关系数据库中，档案本身以文件形式存放于服务器相关目录中
4. 客户端和服务器端基于Socket实现通信，服务器端需实现多线程功能，可同时处理、响应多个客户端的数据请求
5. 用户信息管理功能，普通用户登录后可对自己基本信息(姓名、性别、民族、出生日期、职务、密码等信息)的查询、修改。系统管理人员除上述功能外壳增加、删除用户、修改用户的相关属性和权限
6. 档案数据录入功能，档案录入人员可输入新的档案文件信息，并将档案文件上传至服务器
7. 档案数据查询功能，普通用户可按条件查询相应的档案文件信息，对相关档案文件可以下载保存

## 封装、继承与多态

### 1.1 知识要点

（1）封装

封装(Encapsulation)是类的三大特征之一，即将类的状态信息隐藏在类内部，不允许外部程序直接访问，必须通过该类提供的方法来实现对隐藏信息的操作和访问。简而言之就是隐藏对象的属性和实现细节，仅向外提供访问接口。

封装的好处主要有：隐藏类的实现细节；让使用者只能通过程序员规定的方法来访问数据；可以方便的加入存取控制语句，限制不合理操作。

封装的具体步骤：修改属性的可见性来限制对属性的访问；为每个属性创建一对赋值(setter)方法和取值(getter)方法，用于对这些属性的存取；在赋值方法中，加入对属性的存取控制、有效性检查等。

（2）继承

继承(Inheritance)是Java中实现代码重用的重要手段，它是一种软件复用形式。继承是使用已存在的类的定义作为基础建立新类的技术，新类可以吸收现有类的成员，并可以增加新的功能或修改原有的功能。这种技术使得复用以前的代码非常容易，能够大大缩短开发周期，降低开发费用。

继承表达的是is-a的关系，或者说是一种特殊和一般的关系。Java中只支持单继承，即每个类只能有一个父类。在Java中所有的类都直接或间接的继承了java.lang.Object类，即Object类是所有java类的祖先。

在Java中，子类可以从父类中继承的有：继承public和protected修饰的属性和方法，不管子类和父类是否在同一个包里；继承默认权限修饰符修饰的属性和方法,但子类和父类必须在同一个包里。子类无法继承父类的有：无法继承private修饰的属性和方法；无法继承父类的构造方法。

（3）多态

多态(Polymorphism) 即多种状态，是指计算机程序运行时，系统可依据对象所属类，引发对应类的方法，从而有不同的行为。简单来说，所谓多态意指相同的消息给予不同的对象会引发不同的动作。

多态又可分为设计时多态和运行时多态。重载又被称为设计时多态，而对于覆盖或继承的方法，Java运行时系统根据调用该方法的实例的类型来决定选择调用哪个方法则被称为运行时多态。多态的优点可归纳为五个方面：可替换性、可扩充性、接口性、灵活性和简化性。

在Java中实现多态的三个条件：继承的存在(继承是多态的基础,没有继承就没有多态)；子类重写父类的方法(多态下调用子类重写的方法)；父类引用变量指向子类对象(子类到父类的类型转换)。

特别注意：子类转换成父类时的规则：将一个父类的引用指向一个子类的对象，称为向上转型，自动进行类型转换。此时通过父类引用调用的方法是子类覆盖或继承父类的方法，不是父类的方法。此时通过父类引用变量无法调用子类特有的方法。如果父类要调用子类的特有方法就得将一个指向子类对象的父类引用赋给一个子类的引用，称为向下转型,此时必须进行强制类型转换。

### 实验目的

掌握类的定义、对象的创建、对象的属性的引用和方法的调用；熟悉Java中的继承机制，方法的重载与覆盖；掌握多态、抽象类、接口的使用。

### 1.3 实验内容

编写一个程序，实现档案管理系统中的用户管理模块。要求模块中实现用户的模拟登录过程。通过用户输入，获取用户名和口令；与事先记录在程序中的用户信息进行对比，通过口令验证后才能使用系统。用户分为系统管理人员、档案录入人员，档案浏览人员三类，相关类图如下所示。

（1）要求在用户类中实现封装，并构造setter、getter方法实现属性的访问。

（2）通过继承方式加以实现上述类。

（3）通过多态，实现用户菜单项的展示，根据用户角色不同，系统自动调用对应showMenu()方法。

（4）在未讲数据库之前，系统中已存在用户的信息放置在Hashtable中。提供DataProcessing类实现数据的查找、插入、更新和删除。（此类可提供给学生直接使用）

（5）在未讲I/O之前，系统中的文件操作的方法可虚化，只用打印语句即可。

## 2. 异常处理

### 2.1 知识要点

（1）异常

异常指不期而至的各种状况，如：文件找不到、网络连接失败、数据库错误等。异常是一个事件，它发生在程序运行期间，干扰了正常的指令流程。Java通过API中Throwable类的众多子类描述各种不同的异常。因而，Java异常都是对象，是Throwable子类的实例，描述了出现在一段编码中的错误条件。当条件生成时，错误将引发异常。



Throwable：有两个重要的子类：Exception（异常）和Error（错误），二者都是Java异常处理的重要子类，各自都包含大量子类。

Error：是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM出现的问题。

Exception：是程序本身可以处理的异常。异常和错误的区别：异常能被程序本身可以处理，错误是无法处理。

通常，Java的异常(包括Exception和Error)分为可查的异常（checked exceptions）和不可查的异常（unchecked exceptions）。不可查异常是编译器不要求强制处置的异常：包括运行时异常（RuntimeException与其子类）和错误（Error）。运行时异常是不检查异常，程序中可以选择捕获处理，也可以不处理。除了RuntimeException及其子类以外，其他的Exception类及其子类都属于可查异常。这种异常的特点是Java编译器会检查它，也就是说，当程序中可能出现这类异常，要么用try-catch语句捕获它，要么用throws子句声明抛出它。

（2）处理异常机制

在 Java 应用程序中，异常处理机制为：抛出异常，捕捉异常。

A．抛出异常

当一个方法出现错误引发异常时，方法创建异常对象并交付运行时系统，异常对象中包含了异常类型和异常出现时的程序状态等异常信息。运行时系统负责寻找处置异常的代码并执行。

* throws抛出异常

如果一个方法可能会出现异常，但没有能力处理这种异常，可以在方法声明处用throws子句来声明抛出异常。throws语句的语法格式为：

methodname throws Exception1,Exception2,..,ExceptionN {…… }

* throw抛出异常

throw总是出现在方法体中，用来抛出一个Throwable类型的异常。程序会在throw语句后立即终止，它后面的语句执行不到。throw语句的语法格式为：

throw new exceptionname;

B．捕获异常

在方法抛出异常之后，运行时系统将转为寻找合适的异常处理器（exception handler）。当异常处理器所能处理的异常类型与方法抛出的异常类型相符时，即为合适的异常处理器。运行时系统从发生异常的方法开始，依次回查调用栈中的方法，直至找到含有合适异常处理器的方法并执行。当运行时系统遍历调用栈而未找到合适的异常处理器，则运行时系统终止。同时，意味着Java程序的终止。

* 捕获异常：try、catch 和 finally

try {

// 可能会发生异常的程序代码

} catch (Type1 id1) {

// 捕获并处理try抛出的异常类型Type1

} catch (Type2 id2) {

// 捕获并处理try抛出的异常类型Type2

} finally {

// 无论是否发生异常，都将执行的语句块

}

try 块：用于捕获异常。其后可接零个或多个catch块，如果没有catch块，则必须跟一个finally块。

catch 块：用于处理try捕获到的异常。

finally 块：无论是否捕获或处理异常，finally块里的语句都会被执行。当在try块或catch块中遇到return语句时，finally语句块将在方法返回之前被执行。

### 2.2 实验目的

理解异常的基本概念；了解Java异常的层次结构；熟悉并掌握Java异常的捕获处理方法。

### 2.3 实验内容

（1）阅读Java™ Platform, Standard Edition 8 API Specification文档，了解后续编程中将要处理的IOException及其子类FileNotFoundException、EOFException，SocketException，SQLException以及运行时异常RuntimeException与其子类IllegalStateException。

（2）根据新提供的DataProcessing类（因还未讲SQL，此类模拟异常出现情况，以一定概率随机产生异常），在所编写的Administrator、Operator和Browser类，增加异常处理功能。

## 3. 输入输出流

主要内容：I/O流的概念和划分；字节流处理，包括文件类，文件的顺序处理，随机访问处理；字符流处理，包括Reader，Writer，字节流和字符流的转化；对象串行化的概念和方法。

重点内容：字节流处理，字符流处理

难点内容：对象串行化的概念和方法

培养能力：抽象思维、分析问题、解决问题与创新能力

## 4. GUI设计

主要内容： Java图形界面程序的基本结构；Java布局管理和常用组件的使用；Java事件处理机制。

重点内容：图形界面元素

难点内容：事件处理机制

培养能力：抽象思维、分析问题、解决问题与创新能力

## 5. JDBC数据库操作

主要内容： JDBC的特点，结构，应用模型；JDBC驱动程序的类型；通过JDBC访问数据库执行SQL 语句的方法。

重点内容：通过JDBC访问数据库

难点内容：通过JDBC访问数据库

培养能力：抽象思维、分析问题、解决问题与创新能力

## 6. 网络编程

主要内容： Java网络编程基础；使用URL类访问网络资源；基于Socket的客户和服务器编程；基于Datagram客户和服务器编程。

重点内容：使用URL类访问网络资源

难点内容：基于Socket的客户和服务器编程

培养能力：抽象思维、分析问题、解决问题与创新能力

## 7. 多线程

主要内容：线程的基本概念；线程的构造，调度，控制方法；多线程互斥和同步处理。

重点内容：线程的构造，调度，控制方法

难点内容：多线程互斥和同步处理

培养能力：抽象思维、分析问题、解决问题与创新能力

1. 继承和组合(Inheritance & Composition)

本文主要说明Java中继承与组合的概念，以及它们之间的联系与区别。首先文章会给出一小段代码示例，用于展示到底什么是继承。然后演示如何通过“组合”来改进这种继承的设计机制。最后总结这两者的应用场景，即到底应该选择继承还是组合。

## 1、继承

假设我们有一个名为Insect（昆虫）的类，这个类包含两个方法：1）移动move()； 2）攻击attack()。  
代码如下：

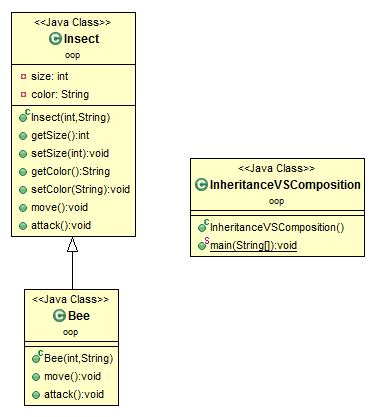
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | class Insect {      private int size;      private String color;        public Insect(int size, String color) {          this.size = size;          this.color = color;      }        public int getSize() {          return size;      }        public void setSize(int size) {          this.size = size;      }        public String getColor() {          return color;      }        public void setColor(String color) {          this.color = color;      }        public void move() {          System.out.println("Move");      }        public void attack() {          move();  //假设昆虫在攻击前必须要先移动一次          System.out.println("Attack");      }  } |

现在，你想要定义一个名为Bee（蜜蜂）的类。Bee（蜜蜂）是Insect（昆虫）的一种，但实现了不同于Insect（昆虫）的attack()和move方法。这时候我们可以用继承的设计机制来实现Bee类，就像下面的代码一样：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Bee extends Insect {      public Bee(int size, String color) {          super(size, color);      }        public void move() {          System.out.println("Fly");      }        public void attack() {          move();          super.attack();      }  } |
| 1  2  3  4  5  6 | public class InheritanceVSComposition {      public static void main(String[] args) {          Insect i = new Bee(1, "red");          i.attack();      }  } |

InheritanceVSComposition作为一个测试类，在其main方法中生成了一个Bee类的实例，并赋值给Insect类型的引用变量 i。所以调用i的attack方法时，对应的是Bee类实例的attack方法，也就是调用了Bee类的attack方法。

类的继承结构图如下，非常简单：

[](http://www.importnew.com/12907.html/inheritance-vs-composition-1)**输出：**

|  |  |
| --- | --- |
| 1  2  3 | Fly  Fly  Attack |

Fly被打印了两次，也就是说move方法被调用了两次。但按理来讲，move方法只应当被调用一次，因为无论是昆虫还是蜜蜂，一次攻击前只移动一次。

问题出在子类（即Bee类）的attack方法的重载代码中，也就是super.attack()这一句。因为在父类（即Insect类）中，调用 attack方法时会先调用move方法，所以当子类（Bee）调用super.attack()时，相当于也同时调用了被重载的move方法（注意是子 类被重载的move方法，而不是父类的move方法）。

为了解决这个问题，我们可以采取以下办法：

1. 删除子类的attack方法。这么做会使得子类的attack方法的实现完全依赖于父类对于该方法的实现（因为子类继承了父类的attack方法）。如果 父类的attack方法不受控制而产生了变更。比如说，父类的attack方法中调用了另外的move方法，那么子类的attack方法也会产生相应的变 化，这是一种很糟糕的封装。
2. 也可以重写子类的attack方法，像下面这样：

|  |  |
| --- | --- |
| 1  2  3  4 | public void attack() {      move();      System.out.println("Attack");  } |

这样保证了结果的正确性，因为子类的attack方法不再依赖于父类。但是，子类attack方法的代码与父类产生了重复（重复的attack方法会使得很多事情变得复杂，不仅仅是多打印了一条输出语句）。所以第二种办法也不行，它不符合软件工程中关于重用的思想。

如此看来，继承机制是有缺点的：子类依赖于父类的实现细节，如果父类产生了变更，子类的后果将不堪设想。

## 2、组合

在上面的例子中，可以用组合的机制来替代继承。我们先看一下运用组合如何实现。

attack这一功能不再是一个方法，而是被抽象为一个接口。

|  |  |
| --- | --- |
| 1  2  3  4 | interface Attack {      public void move();      public void attack();  } |

通过对Attack接口的实现，就可以在实现类当中定义不同类型的attack。

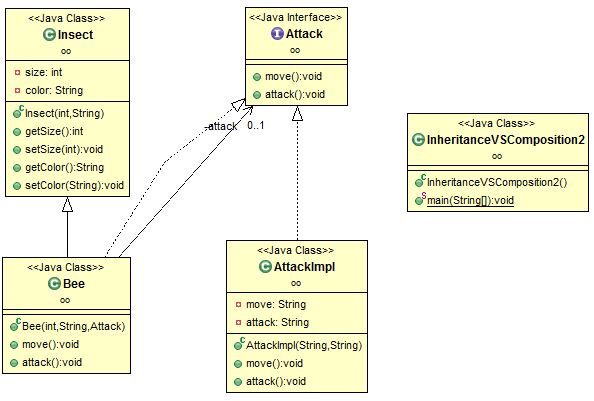
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class AttackImpl implements Attack {      private String move;      private String attack;        public AttackImpl(String move, String attack) {          this.move = move;          this.attack = attack;      }        @Override      public void move() {          System.out.println(move);      }        @Override      public void attack() {          move();          System.out.println(attack);      }  } |

因为attack功能已经被抽象为一个接口，所以Insect类不再需要有attack方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class Insect {      private int size;      private String color;        public Insect(int size, String color) {          this.size = size;          this.color = color;      }        public int getSize() {          return size;      }        public void setSize(int size) {          this.size = size;      }        public String getColor() {          return color;      }        public void setColor(String color) {          this.color = color;      }  } |

Bee类一种Insect类，它具有attack的功能，所以它实现了attack接口：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | // 这个封装类封装了一个Attack类型的对象  class Bee extends Insect implements Attack {      private Attack attack;        public Bee(int size, String color, Attack attack) {          super(size, color);          this.attack = attack;      }        public void move() {          attack.move();      }        public void attack() {          attack.attack();      }  } |

**类图：**  
[](http://www.importnew.com/12907.html/inheritance-vs-composition-2)

测试类代码，将AttackImpl的实例作为Attack类型的参数传给Bee类的构造函数：

|  |  |  |  |
| --- | --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | | public class InheritanceVSComposition2 {      public static void main(String[] args) {          Bee a = new Bee(1, "black", new AttackImpl("fly", "move"));          a.attack();            // if you need another implementation of move()          // there is no need to change Insect, we can quickly use new method to attack            Bee b = new Bee(1, "black", new AttackImpl("fly", "sting"));          b.attack();      }  } | |
| 1  2  3  4 | fly  move  fly  sting | |

## 3、什么时候该用继承，什么时候该用组合？

以下两条原则说明了应该如何选择继承与组合：

* 如果存在一种IS-A的关系（比如Bee“是一个”Insect），并且一个类需要向另一个类暴露所有的方法接口，那么更应该用继承的机制。
* 如果存在一种HAS-A的关系（比如Bee“有一个”attack功能），那么更应该运用组合。

总结来说，继承和组合都有他们的用处。只有充分理解各对象和功能之间的关系，才能充分发挥这两种机制各自的优点。

1 继承和组合(Inheritance & Composition)

封装、继承与多态2 Polymorphism

2 异常处理 Exception

3 输入输出流 I/O

4 GUI设计

5 JDBC数据库操作

6 网络编程

7 多线程

8

1. 基本语法：   
   public class Sub extends Base{   
   }   
   当Sub和Base在同一个package时，Sub继承了Base中的public、protected和默认（即未指定访问级别）访问级别的成员变量和方法。   
   当Sub和Base在不同package时，Sub继承了Base中的public、protected访问级别的成员变量和方法。   
   2.Java不支持多重继承，但可以通过串行的间接继承进行多重继承   
   3.所有Java类都直接或间接的集成了java.lang.object类。Object类定义了Java对象具有的相同行为。   
   4.重载overload   
   同一个类中，相同名字的方法，但是参数类型、个数或者顺序至少有一项不同。   
   5.覆盖override   
   子类覆盖了父类的方法，子类方法的名字、参数签名、返回类型必须与父类相同。若想怪边返回类型等，但是还想用同一个名字，则可以使用先覆盖再重载的策略：   
   public class Base{   
       public void method(){   
       }   
   }   
   public class Sub extends Base{   
       public void method(){//覆盖Base类的method方法   
       }   
       public int method(int a){//重载Sub类的method方法   
       }   
   }   
   注意：   
   a.子类方法不能缩小父类方法的访问权限（可以扩大），而且子类不能抛出比父类方法更多的异常，其必须抛出和父类方法抛出的相同的异常，或者子类方法抛出的异常类是父类方法抛出的异常类的子类。   
   b.父类的静态方法不能被子类覆盖为非静态方法。同样，父类的非静态方法不能被子类覆盖为静态方法。   
   c.子类可以定义与父类的静态方法同名的静态方法，以便在父类中隐藏父类的静态方法。子类覆盖静态方法和覆盖一般方法的区别是，运行时，JVM把静态方法和所属类绑定，而将一般方法和所属实例绑定。   
   d.父类的私有方法不能被子类覆盖。虽然不会出现编译错误，但是实际上子类并没有从父类继承这个方法，所以，其并没有覆盖关系。   
   e.父类的抽象方法通过子类实现父类的抽象方法或者子类重新声明父类的抽象方法来实现覆盖。   
   f.父类的非抽象方法可以被覆盖为抽象方法。   
   6.super关键字   
   super和this的功能都是使被屏蔽的方法或变量变为可见。不能在静态方法或者静态代码块内不能使用super关键字。   
   7.多态   
   实质：指当系统A访问系统B的服务时，系统B可以通过多种实现方式来提供服务，而这一切对系统A是透明的。例如：   
   public class Feeder{   
       public void feed(Animal animal,Food food){   
           animal.eat(food);   
       }   
   }   
   Feeder feeder=new Feeder();   
   Animal animal=new Dog();   
   Food food = new Bone();   
   feeder.feed(animal,food);//喂狗骨头吃   
   animal=new Cat();   
   food = new Fish();   
   feeder.feed(animal,food);//喂猫鱼吃   
   注意animal被定义为Animal类型，但是有可能引用Dog或Cat类型，在调用animal.eat的时候，JVM会执行animal变量所引用的实例的eat方法。   
   需要注意的是：   
   1）静态绑定：对于一个引用类型的变量，Java编译器按照它声明的类型来处理。例如，Base who = new Sub();   
   这个who是Base类型的引用变量，Java编译器就会按照Base类型来处理。要访问Sub类的成员，则使用强制类型转换：((sub)who).subVar="123";   
   这适用于成员变量（包括静态变量和实例变量），静态方法   
   2）动态绑定：对于一个引用类型的变量，运行时JVM按照其实际引用的对象的方法进行绑定。实例方法就属于这一类。   
   3）Java允许在具有直接或间接的继承关系的类之间进行类型转换，对于向上转型，不必使用强制类型转换。   
   继承的利弊：   
   提高代码可重用性，是提高系统扩展性的有效手段。但是继承树若是异常复杂，或被随意扩展，反而会削弱系统的可扩展性和可维护性。继承的最大弱点是打破了封装，削弱了子类的独立性。   
   继承的原则：   
   1）不考虑Object，继承树应该尽量保持在两到三层。   
   2）当一个系统使用一棵继承树上的类时，应该尽可能地把引用变量声明为继承树的上层类型。   
   3）若继承树上有接口类型，那么应该尽可能地把引用变量声明为继承树上层的接口类型。   
   4）上层完成那些适用于所有子类或大多数子类的方法，否则则设定为抽象方法。   
   5）在设计继承类的时候，尽可能的封装父类的实现细节，也就是把代表实现细节的属性和方法定义为private类型，若某些实现细节必须被子类访问则设为protected类型。   
   6）将不允许子类覆盖的方法定义为final类型。   
   7）父类的构造方法中不要调用可以被子类覆盖的方法。   
   8）某些类不是为了继承而设计的，那么采用将类声明为final类型，或者将这个类的所有构造方法声明为private类型，然后通过一些静态方法来负责构造自身的实例的两种方法禁止继承。   
   9）对于一棵设计合理的继承树，子类之间会具有不同的属性和行为。   
   10）组合关系指的是：类A中包含有类C的属性，则类A称为包装类（整体类），而类C称作被包装类（局部类）。在开发中，组合关系虽然不比继承关系减少代码量，但是维护起来，有更好的松耦合性。不使用继承，而是在类中加入要继承的类的一个实例成员就是组合。继承关系是静态的，在运行时，子类无法改变它的弗雷，但组合关系在运行时可以根据需要改变实现方式。在继承关系中，子类只能继承父类的接口，不能取消父类的方法，但是除非覆盖该方法，并直接抛出java.lang.UnsupportedOperationExcepetion。而组合则不会自动去继承父类的方法。

1，多态的概念

多态是面向对象编程的一大特征。体现了程序的可扩展性，也体现了程序代码的重复使用特性。简单的说就是指一个对象的行为方式可以有很多种操作形态，根据不同对象，会有不同的操作。所以多态寄托于对象。

2，多态的应用

在面向对象编程中，多态主要是通过[方法的重载和覆盖](http://www.cnblogs.com/liwustore/archive/2013/01/10/2854416.html)体现的。方法是通过给不同对象发送相同的信息，根据不同的对象来完成不同的工作。

3，构造方法和多态

一个类可以有多个构造方法，当使用同一个类的不同构造方法创建多个类对象时，会呈现多种对象，所以这也就体现了多态性。

4，多态应用例题：

设计程序，说明世界上主要几个国家的语言：