JAVA 的面向对象编程……课堂笔记

面向对象主要针对面向过程。
面向过程的基本单元是函数。

什么是对象：**EVERYTHING IS OBJECT**（万物皆对象）

所有的事物都有两个方面：
有什么（属性）：用来描述对象。
能够做什么（方法）：告诉外界对象有那些功能。
后者以前者为基础。
大的对象的属性也可以是一个对象。

为什么要使用面向对象：
首先，面向对象符合人类看待事物的一般规律。
对象的方法的实现细节是屏蔽的，只有对象方法的实现者了解细节。
方法的定义非常重要。方法有参数，也可能有返回值。

注意区分：对象（本身）、对象的实现者、对象的调用者。
分析对象主要从方法开始。
我们通过类来看待对象，类是对象的抽象。

其次，采用面向对象方法可以使系统各部分各司其职、各尽所能。
对象之间的耦合性一定要低（比如不同硬盘和不同主板之间的关系）。这样才能使每个对象本身做成最好的。

对于对象的要求：高内聚、低耦合，这样容易拼装成为一个系统。
实现高内聚就是要最大限度低提高复用性（复用性好是因为高内聚）。
可复用性是 **OOP** 的基础。

比较面向过程的思想和面向对象的思想：
面向过程的思想：由过程、步骤、函数组成，以过程为核心；
面向对象的思想：以对象为中心，先开发类，得到对象，通过对象之间相互通信实现功能。
面向过程是先有算法，后有数据结构。
面向对象是先有数据结构，然后再有算法。

在用面向对象思想开发的过程中，可以复用对象就进行复用，如无法进行复用则开发新的对象。
开发过程是用对个简单的对象的多个简单的方法，来实现复杂的功能 。
从语法上来看，一个类是一个新的数据类型。
在面向对象编程中，除了简单数据类型，就是对象类型。
定义类的格式：

```
class   Student{
   代码
}
```

注意类名中单词的首字母大写。
实例变量：定义在类中但在任何方法之外。（New 出来的均有初值）
局部变量：定义在方法之中的变量。
局部变量要先赋值，再进行运算，而实例变量均已经赋初值。这是局部变量和实例变量的一大区别。
实例变量的对象赋值为 null。
局部变量不允许范围内定义两个同名变量。实例变量的作用域在本类中完全有效，当被其他的类调用的时候也可能有效。
实例变量和局部变量允许命名冲突。

书写方法的格式：

修饰符	返回值	方法名	调用过程中可能出现的例外	方法体
public	**int/void**	**addNumber(参数)**	**throw Excepion**	**{}**

例：

```
public    int    addNumber(int a,int b){
}
```

注：方法名中的参数 int a,int b 为局部变量

类方法中的一类特殊方法：构造方法。
构造方法是当用类生成对象时，系统在生成对象的过程中利用的方法。
注意：构造方法在生成对象的时候会被调用，但并不是构造方法生成了对象。
构造方法没有返回值。格式为：public 方法名。
构造方法的方法名与类名相同。
构造方法是在对象生成的过程中自动调用，不可能利用指令去调用。
在一个对象的生成周期中构造方法只用一次，一旦这个对象生成，那么这个构造方法失效。

用类来生成对象的语句：
Student s=new Student()。
第一个 Student 表示这是用 Student 类进行定义。“Student()”表示调用一个无参数的构造方法。
如果()中有参数，则系统构造对象的过程中调用有参的方法。
此时 S 称为一个对象变量。
Student s 的存储区域存放的是地址：一个对象在硬盘上占有一个连续地址，首地址赋予 s 空间。
S 称为对象 Student 的引用。
注意：在对象变量中存放的是引用（地址）；在简单变量中存放的是数值。

可以构造多个构造方法，但多个构造方法的参数表一定不同，参数顺序不同即属于不同的构造方法：

```
public student(string name,int a){
}
public student(int a,string name){
}
```

为两个不同的构造方法。
如果我们未给系统提供一个构造方法，那么系统会自动提供一个为空的构造方法。
练习：写一个类，定义一个对象，定义两个构造方法：一个有参，一个无参。
（编写一个程序验证对象的传递的值为地址）
注意下面这种形式：

```
static void changename(student stu){stu.setName “LUCY”}
```

注意生成新的对象与旧对象指向无关，生成新对象生命消亡与旧对象无关。

面向对象方法的重载（overloading）和覆盖（overriding）。
在有些 JAVA 书籍中将 overriding 称为重载，overloading 称为过载。
Overloading 在一个类中可以定义多个同名方法，各个方法的参数表一定不同。但修饰词可能相同，返回值也可能相同。
在程序的编译过程中根据变量类型来找相应的方法。因此也有人认为 overloading 是编译时的多态，以后我们还会学到运行时多态。

为什么会存在 overloading 技术呢？作为应对方法的细节。
利用类型的差异来影响对方法的调用。
吃（）可以分为吃肉，吃菜，吃药，在一个类中可以定义多个吃方法。

构造方法也可以实现 overloading。例：

```
public   void   teach(){};
public   void   teach(int a){};
public   void   teach(String a){}
```

为三种不同的方法。

Overloading 方法是从低向高转。
Byte—short—float—int—long—double。

在构造方法中，this 表示本类的其他构造方法：

```
student(){};
student(string n){
   this();//表示调用 student()
}
```

如果调用 student(int a)则为 this(int a)。
特别注意：用 this 调用其他构造方法时，this 必须为第一条语句，然后才是其他语句。

This 表示当前对象。

```
Public   void   printNum(){
   Int number=40;
   System.out.println(this.number);
}
```

此时打印的是实例变量，而非局部变量，即定义在类中而非方法中的变量。

This.number 表示实例变量。
谁调用 this.number 那么谁即为当前(this)对象的 number 方法。

封装：使对象的属性尽可能私有，对象的方法尽可能的公开。用 private 表示此成员属性为该类的私有属性。

Public 表示该属性（方法）公开；
Private 表示该属性（方法）为只有本类内部可以访问（类内部可见）。
（想用 private 还要用 set 和 get 方法供其他方法调用，这样可以保证对属性的访问方式统一，并且便于维护访问权限以及属性数据合法性）

如果没有特殊情况，属性一定私有，方法该公开的公开。
如果不指明谁调用方法，则默认为 **this**。
区分实例变量和局部变量时一定要写 **this**。

11.29

继承：
父类（**SuperClass**）和 子类（**SonClass**）。
父类的非私有化属性和方法可以默认继承到子类。
Class Son extends Father{
}
而如果父类中的私有方法被子类调用的话，则编译报错。

父类的构造方法子类不可以继承，更不存在覆盖的问题。（非构造方法可以）
如果子类访问父类的构造方法，则在编译的时候提示访问不到该方法。
JAVA 中不允许多继承，一个类有且只有一个父类（单继承）。
JAVA 的数据结构为树型结构，而非网状。（**JAVA** 通过接口和内部类实现多继承）

方法的覆盖（**overriding**）
方法的重载并不一定是在一个类中：子类可以从父类继承一个方法，也可以定义一个同名异参的方法，也称为 **overloading**。
当子类从父类继承一个无参方法，而又定义了一个同样的无参方法，则子类新写的方法覆盖父类的方法，称为覆盖。（注意返回值类型也必须相同，否则编译出错。）
如果方法不同，则成重载。

对于方法的修饰词，子类方法要比父类的方法范围更加的宽泛。
父类为 **public**，那么子类为 **private** 则出现错误。
之所以构造方法先运行父类再运行子类是因为构造方法是无法覆盖的。
以下范围依次由严到宽：
private ： 本类访问；
default ： 表示默认，不仅本类访问，而且是同包可见。
Protected：同包可见+不同包的子类可见
Public ：表示所有的地方均可见。

当构造一个对象的时候，系统先构造父类对象，再构造子类对象。
构造一个对象的顺序：（注意：构造父类对象的时候也是这几步）
① 递归地构造父类对象；
② 顺序地调用本类成员属性赋初值语句；
③ 本类的构造方法。

Super()表示调用父类的构造方法。
Super()也和 **this** 一样必须放在第一行。
This()用于调用本类的构造方法。
如果没有定义构造方法，那么就会调用父类的无参构造方法，即 **super()**。

要养成良好的编程习惯：就是要加上默认的父类无参的构造方法。
思考：可是如果我们没有定义无参的构造方法，而在程序中构造了有参的构造方法，那么如果方法中没有参数，那么系统还会调用有参的构造方法么？应该不会。

多态：多态指的是编译时类型变化，而运行时类型不变。
多态分两种：
① 编译时多态：编译时动态重载；
② 运行时多态：指一个对象可以具有多个类型。

对象是客观的，人对对象的认识是主观的。
例：
Animal a=new Dog()；查看格式名称；
Dog d=(Dog)a。声明父类来引用子类。
（思考上面的格式）

运行时多态的三原则：（应用时为覆盖）
1、 对象不变；（改变的是主观认识）
2、 对于对象的调用只能限于编译时类型的方法，如调用运行时类型方法报错。
在上面的例子中：Animal a=new Dog()；对象 a 的编译时类型为 Animal，运行时类型为 dog。
注意：编译时类型一定要为运行时类型的父类（或者同类型）。
对于语句：Dog d=(Dog)a。将 d 强制声明为 a 类型，此时 d 为 Dog()，此时 d 就可以调用运行时类型。
注意：a 和 d 指向同一对象。
3、 在程序的运行时，动态类型判定。运行时调用运行时类型，即它调用覆盖后的方法。

关系运算符：**instanceof**
a instanceof Animal;(这个式子的结果是一个布尔表达式**)**
a 为对象变量，**Animal** 是类名。
上面语句是判定 **a** 是否可以贴 **Animal** 标签。如果可以贴则返回 **true**，否则返回 **false**。
在上面的题目中： **a instanceof Animal** 返回 **True**，
a instanceof Dog 也返回 **True**，
instanceof 用于判定是否将前面的对象变量赋值后边的类名。
Instanceof 一般用于在强制类型转换之前判定变量是否可以强制转换。

如果 **Animal a=new Animal();**
Dog d=Dog()a;
此时编译无误，但运行则会报错。

Animal a=new Dog()相当于下面语句的功能：
Animal a=getAnimal();
Public static Animal.getAnimal;
Return new Dog();

封装、继承、多态为面向对象的三大基石（特性）。

运行时的动态类型判定针对的是方法。运行程序访问的属性仍为编译时属性。

Overloading 针对的是编译时类型，不存在运行时的多态。

习题：建立一个 **shape** 类，有 **circle** 和 **rect** 子类。
Shape 类有 **zhouchang()**和 **area()**两种方法。

（正方形）**squ** 为 **rect** 子类，**rect** 有 **cha()**用于比较长宽的差。

覆盖时考虑子类的 **private** 及父类的 **public**（考虑多态），之所以这样是避免调用 **A** 时出现实际调用 **B** 的情况。而出现错误。
11.29 下午讲的是教程上的 **Module6**
Module6-7 包括：面向对象高级、内部类、集合、反射（暂时不讲）、例外。
面向对象高级、集合和例外都是面向对象的核心内容。

面向对象高级：　　修饰符：
static:①可修饰变量（属性）；②可修饰方法；③可修饰代码块。
Static int data 语句说明 **data** 为类变量，为一个类的共享变量，属于整个类。
Int data 为实例变量。
例：
static int data;
m1.data=0;
m1.data++的结果为 **1,**此时 **m2.data** 的结果也为 **1**。
Static 定义的是一块为整个类共有的一块存储区域，其发生变化时访问到的数据都时经过变化的。
其变量可以通过类名去访问：类名.变量名。与通过访问对象的编译时类型访问类变量为等价的。

Public static void printData(){}
表明此类方法为类方法（静态方法）
静态方法不需要有对象，可以使用类名调用。
静态方法中不允许访问类的非静态成员，包括成员的变量和方法，因为此时是通过类调用的，没有对象的概念。**This.data** 是不可用的。

一般情况下，主方法是静态方法，所以可调用静态方法，主方法为静态方法是因为它是整个软件系统的入口，而进入入口时系统中没有任何对象，只能使用类调用。

覆盖不适用于静态方法。
静态方法不可被覆盖。（允许在子类中定义同名静态方法，但是没有多态，严格的讲，方法间没有多态就不能称为覆盖）
当 **static** 修饰代码块时（注：此代码块要在此类的任何一个方法之外），那么这个代码块在代码被装载进虚拟机生成对象的时候可被装载一次，以后再也不执行了。
一般静态代码块被用来初始化静态成员。

Static 通常用于 **Singleton** 模式开发：
Singleton 是一种设计模式，高于语法，可以保证一个类在整个系统中仅有一个对象。

11.30

final 可以修饰类、属性、方法。

当用 **final** 修饰类的时候，此类不可被继承，即 **final** 类没有子类。这样可以用 **final** 保证用户调用时动作的一致性，可以防止子类覆盖情况的发生。

当利用 **final** 修饰一个属性（变量）的时候，此时的属性成为常量。
JAVA 利用 **final** 定义常量（注意在 **JAVA** 命名规范中常量需要全部字母都大写）：
Final int AGE=10;
常量的地址不可改变，但在地址中保存的值（即对象的属性）是可以改变的。

Final 可以配合 static 使用。 ？
Static final int age=10;

在 **JAVA** 中利用 **public static final** 的组合方式对常量进行标识（固定格式）。

对于在构造方法中利用 **final** 进行赋值的时候，此时在构造之前系统设置的默认值相对于构造方法失效。

常量（这里的常量指的是实例常量：即成员变量）赋值：
①在初始化的时候通过显式声明赋值。Final int x=3;
②在构造的时候赋值。
局部变量可以随时赋值。

利用 final 定义方法：这样的方法为一个不可覆盖的方法。
Public final void print(){};
为了保证方法的一致性（即不被改变），可将方法用 final 定义。

如果在父类中有 final 定义的方法，那么在子类中继承同一个方法。

如果一个方法前有修饰词 private 或 static，则系统会自动在前面加上 final。即 private 和 static 方法默认均为 final 方法。

注：final 并不涉及继承，继承取决于类的修饰符是否为 private、default、protected 还是 public。
也就是说，是否继承取决于这个方法对于子类是否可见。
Abstract(抽象)可以修饰类、方法
如果将一个类设置为 abstract，则此类必须被继承使用。此类不可生成对象，必须被继承使用。
Abstract 可以将子类的共性最大限度的抽取出来，放在父类中，以提高程序的简洁性。
Abstract 虽然不能生成对象，但是可以声明，作为编译时类型，但不能作为运行时类型。
Final 和 abstract 永远不会同时出现。

当 abstract 用于修饰方法时，此时该方法为抽象方法，此时方法不需要实现，实现留给子类覆盖，子类覆盖该方法之后方法才能够生效。

注意比较：
private void print(){}；此语句表示方法的空实现。
Abstract void print(); 此语句表示方法的抽象，无实现。

如果一个类中有一个抽象方法，那么这个类一定为一个抽象类。
反之，如果一个类为抽象类，那么其中可能有非抽象的方法。

如果让一个非抽象类继承一个含抽象方法的抽象类，则编译时会发生错误。因为当一个非抽象类继承一个抽象方法的时候，本着只有一个类中有一个抽象方法，那么这个类必须为抽象类的原则。这个类必须

为抽象类，这与此类为非抽象冲突，所以报错。

所以子类的方法必须覆盖父类的抽象方法。方法才能够起作用。

只有将理论被熟练运用在实际的程序设计的过程中之后，才能说理论被完全掌握！

为了实现多态，那么父类必须有定义。而父类并不实现，留给子类去实现。此时可将父类定义成 **abstract** 类。如果没有定义抽象的父类，那么编译会出现错误。

Abstract 和 **static** 不能放在一起，否则便会出现错误。（这是因为 **static** 不可被覆盖，而 **abstract** 为了生效必须被覆盖。）

例：（本例已存在\CODING\abstract\TestClass.java 文件中）

```
public class TestClass{
   public static void main(String[] args){
      SuperClass sc=new SubClass();
      Sc.print();
      }
      Abstract class SuperClass{
         Abstract void print();}
   }
   class SubClass extends SuperClass(){
      void print(){
      System.out.println(“print”);}
   }
```

JAVA 的核心概念：接口（**interface**）
接口与类属于同一层次，实际上，接口是一种特殊的抽象类。
如:

```
   interface IA{
}
```

public interface：公开接口
与类相似，一个文件只能有一个 **public** 接口，且与文件名相同。
在一个文件中不可同时定义一个 **public** 接口和一个 **public** 类。

一个接口中，所有方法为公开、抽象方法；所有的属性都是公开、静态、常量。

一个类实现一个接口的格式：

```
class IAImple implements IA{
};
```

一个类实现接口，相当于它继承一个抽象类。

类必须实现接口中的方法，否则其为一抽象类。
实现中接口和类相同。

接口中可不写 **public**，但在子类中实现接口的过程中 **public** 不可省。
（如果剩去 **public** 则在编译的时候提示出错：对象无法从接口中实现方法。）

注：
① 一个类除继承另外一个类，还可以实现接口；
class IAImpl extends java.util.Arrylist implement IA{}
继承类 实现接口
这样可以实现变相的多继承。
② 一个类只能继承另外一个类，但是它可以继承多个接口，中间用“,”隔开。
Implements IA,IB
所谓实现一个接口，就是指实现接口中的方法。
③ 接口和接口之间可以定义继承关系，并且接口之间允许实现多继承。
例：interface IC extends IA,IB{};
接口也可以用于定义对象
IA I=new IAImpl();
实现的类从父类和接口继承的都可做运行时类型。
IAImple extends A implement IA,IB
IB I=new IAImple();
I instance of IAImple;
I instance of A;
I instance of IA;
I instance of IB;
返回的结果均为 true.

接口和多态都为 JAVA 技术的核心。

接口往往被我们定义成一类 XX 的东西。
接口实际上是定义一个规范、标准。

① 通过接口可以实现不同层次、不同体系对象的共同属性；
通过接口实现 write once as anywhere.
以 JAVA 数据库连接为例子：JDBC 制定标准；数据厂商实现标准；用户使用标准。
接口通常用来屏蔽底层的差异。
②接口也因为上述原因被用来保持架构的稳定性。

JAVA 中有一个特殊的类： **Object**。它是 **JAVA** 体系中所有类的父类（直接父类或者间接父类）。

此类中的方法可以使所的类均继承。

以下介绍的三种方法属于 **Object:**
(1) finalize 方法：当一个对象被垃圾回收的时候调用的方法。
(2) toString():是利用字符串来表示对象。
当我们直接打印定义的对象的时候，隐含的是打印 **toString()**的返回值。
可以通过子类作为一个 **toString()**来覆盖父类的 **toString()**。
以取得我们想得到的表现形式，即当我们想利用一个自定义的方式描述对象的时候，我们应该覆盖 **toString()**。
(3)equal
首先试比较下例：
String A=new String(“hello”);

String A=new String(“hello”);
A==B(此时程序返回为 FALSE)
因为此时 **AB** 中存的是地址，因为创建了新的对象，所以存放的是不同的地址。

附加知识：
字符串类为 **JAVA** 中的特殊类，**String** 中为 **final** 类，一个字符串的值不可重复。因此在 **JAVA VM**（虚拟机）中有一个字符串池，专门用来存储字符串。如果遇到 **String a=”hello”**时（注意没有 **NEW**，不是创建新串），系统在字符串池中寻找是否有**”hello”**，此时字符串池中没有**”hello”**，那么系统将此字符串存到字符串池中，然后将**”hello”**在字符串池中的地址返回 **a**。如果系统再遇到 **String b=”hello”**，此时系统可以在字符串池中找到 **“hello”**。则会把地址返回 **b**，此时 **a** 与 **b** 为相同。

String a=”hello”;
System.out.println(a==”hello”);
系统的返回值为 **true**。

故如果要比较两个字符串是否相同（而不是他们的地址是否相同）。可以对 **a** 调用 **equal:**
System.out.println(a.equal(b));
equal 用来比较两个对象中字符串的顺序。
a.equal(b)是 **a** 与 **b** 的值的比较。

注意下面程序：
```
student a=new student(“LUCY”,20);
student b=new student(“LUCY”,20);
System.out.println(a==b);
System.out.println(a.equal(b));
```
此时返回的结果均为 **false**。

以下为定义 **equal**（加上这个定义，返回 **ture** 或 **false**）
```
public boolean equals(Object o){
   student s=(student)o;
   if (s.name.equals(this.name)&&s.age==this.age)
else return false;
}
```
如果 **equals()**返回的值为

以下为实现标准 **equals** 的流程：
```
public boolean equals(Object o){
   if (this==o) return trun;   //此时两者相同
   if (o==null) return false;
   if (! o instanceof strudent) return false;   //不同类
   studeng s=(student)o; //强制转换
   if (s.name.equals(this.name)&&s.age==this.age) return true;
else return false;
}
```

以上过程为实现 **equals** 的标准过程。

练习：建立一个 employee 类，有 String name,int id,double salary.运用 get 和 set 方法，使用 toString，使用 equals。

封装类：
JAVA 为每一个简单数据类型提供了一个封装类，使每个简单数据类型可以被 Object 来装载。
除了 int 和 char，其余类型首字母大写即成封装类。
转换字符的方式：
int I=10;
String s=I+" ";
String s1=String.valueOf(i);

Int I=10;
Interger I_class=new integer(I);

看 javadoc 的帮助文档。
附加内容：
"=="在任何时候都是比较地址，这种比较永远不会被覆盖。

程序员自己编写的类和 JDK 类是一种合作关系。（因为多态的存在，可能存在我们调用 JDK 类的情况，也可能存在 JDK 自动调用我们的类的情况。）
注意：类型转换中 double\interger\string 之间的转换最多。

12.01

内部类：
（注：所有使用内部类的地方都可以不用内部类，使用内部类可以使程序更加的简洁，便于命名规范和划分层次结构）。
内部类是指在一个外部类的内部再定义一个类。
内部类作为外部类的一个成员，并且依附于外部类而存在的。
内部类可为静态，可用 PROTECTED 和 PRIVATE 修饰。（而外部类不可以：外部类只能使用 PUBLIC 和 DEFAULT）。

内部类的分类：
成员内部类、
局部内部类、
静态内部类、
匿名内部类（图形是要用到，必须掌握）。

① 成员内部类：作为外部类的一个成员存在，与外部类的属性、方法并列。
内部类和外部类的实例变量可以共存。
在内部类中访问实例变量：this.属性
在内部类访问外部类的实例变量：外部类名.this.属性。

成员内部类的优点：
⑴内部类作为外部类的成员，可以访问外部类的私有成员或属性。（即使将外部类声明为 PRIVATE，但是对于处于其内部的内部类还是可见的。）

⑵用内部类定义在外部类中不可访问的属性。这样就在外部类中实现了比外部类的 **private** 还要小的访问权限。
注意：内部类是一个编译时的概念，一旦编译成功，就会成为完全不同的两类。
对于一个名为 **outer** 的外部类和其内部定义的名为 **inner** 的内部类。编译完成后出现 **outer.class** 和 **outer$inner.class** 两类。

（编写一个程序检验：在一个 **TestOuter.java** 程序中验证内部类在编译完成之后，会出现几个 **class.**）

成员内部类不可以有静态属性。（为什么？）

如果在外部类的外部访问内部类，使用 **out.inner.**

建立内部类对象时应注意：
在外部类的内部可以直接使用 **inner s=new inner();**（因为外部类知道 **inner** 是哪个类，所以可以生成对象。）
而在外部类的外部，要生成（**new**）一个内部类对象，需要首先建立一个外部类对象（外部类可用），然后在生成一个内部类对象。
Outer.Inner in=Outer.new.Inner()。
错误的定义方式：
Outer.Inner in=new Outer.Inner()。
注意：当 **Outer** 是一个 **private** 类时，外部类对于其外部访问是私有的，所以就无法建立外部类对象，进而也无法建立内部类对象。

② 局部内部类：在方法中定义的内部类称为局部内部类。
与局部变量类似，在局部内部类前不加修饰符 **public** 和 **private**，其范围为定义它的代码块。

注意：局部内部类不仅可以访问外部类实例变量，还可以访问外部类的局部变量（但此时要求外部类的局部变量必须为 **final**）？？
在类外不可直接生成局部内部类（保证局部内部类对外是不可见的）。
要想使用局部内部类时需要生成对象，对象调用方法，在方法中才能调用其局部内部类。

③ 静态内部类：（注意：前三种内部类与变量类似，所以可以对照参考变量）
静态内部类定义在类中，任何方法外，用 **static** 定义。
静态内部类只能访问外部类的静态成员。
生成（**new**）一个静态内部类不需要外部类成员：这是静态内部类和成员内部类的区别。静态内部类的对象可以直接生成：
Outer.Inner in=new Outer.Inner();
而不需要通过生成外部类对象来生成。这样实际上使静态内部类成为了一个顶级类。
静态内部类不可用 **private** 来进行定义。例子：
对于两个类，拥有相同的方法：

```
People
{
   run();
}
Machine{
    run();
}
```

此时有一个 **robot** 类：
class Robot extends People implement Machine.
此时 **run()**不可直接实现。
注意：当类与接口（或者是接口与接口）发生方法命名冲突的时候，此时必须使用内部类来实现。
用接口不能完全地实现多继承，用接口配合内部类才能实现真正的多继承。

④ 匿名内部类（必须掌握）：
匿名内部类是一种特殊的局部内部类，它是通过匿名类实现接口。
IA 被定义为接口。
IA I=new IA(){};
注：一个匿名内部类一定是在 **new** 的后面，用其隐含实现一个接口或实现一个类，没有类名，根据多态，我们使用其父类名。
因其为局部内部类，那么局部内部类的所有限制都对其生效。
匿名内部类是唯一一种无构造方法类。
匿名内部类在编译的时候由系统自动起名 **Out$1.class**。

如果一个对象编译时的类型是接口，那么其运行的类型为实现这个接口的类。
因匿名内部类无构造方法，所以其使用范围非常的有限。
（下午：）**Exception**（例外/异常）（教程上的 **MODEL7**）
对于程序可能出现的错误应该做出预案。
例外是程序中所有出乎意料的结果。（关系到系统的健壮性）

JAVA 会将所有的错误封装成为一个对象，其根本父类为 **Throwable**。
Throwable 有两个子类：**Error** 和 **Exception**。
一个 **Error** 对象表示一个程序错误，指的是底层的、低级的、不可恢复的严重错误。此时程序一定会退出，因为已经失去了运行所必须的物理环境。
对于 **Error** 错误我们无法进行处理，因为我们是通过程序来应对错误，可是程序已经退出了。
我们可以处理的 **Throwable** 对象中只有 **Exception** 对象（例外/异常）。
Exception 有两个子类：**Runtime exception**（未检查异常）
非 **Runtime exception**（已检查异常）
（注意：无论是未检查异常还是已检查异常在编译的时候都不会被发现，在编译的过程中检查的是程序的语法错误，而异常是一个运行时程序出错的概念。）
在 **Exception** 中，所有的非未检查异常都是已检查异常，没有另外的异常!!

未检查异常是因为程序员没有进行必要的检查，因为他的疏忽和错误而引起的异常。一定是属于虚拟机内部的异常（比如空指针）。

应对未检查异常就是养成良好的检查习惯。
已检查异常是不可避免的，对于已检查异常必须实现定义好应对的方法。
已检查异常肯定跨越出了虚拟机的范围。（比如“未找到文件”）

如何处理已检查异常（对于所有的已检查异常都要进行处理）：
首先了解异常形成的机制：
当一个方法中有一条语句出现了异常，它就会 **throw**（抛出）一个例外对象，然后后面的语句不会执行返回上一级方法，其上一级方法接受到了例外对象之后，有可能对这个异常进行处理，也可能将这个异常转到它的上一级。
对于接收到的已检查异常有两种处理方式：**throws** 和 **try** 方法。

注意：出错的方法有可能是 **JDK**，也可能是程序员写的程序，无论谁写的，抛出一定用 **throw**。

例：**public void print() throws Exception.**

对于方法 **a**，如果它定义了 **throws Exception**。那么当它调用的方法 **b** 返回异常对象时，方法 **a** 并不处理，而将这个异常对象向上一级返回，如果所有的方法均不进行处理，返回到主方法，程序中止。（要避免所有的方法都返回的使用方法，因为这样出现一个很小的异常就会令程序中止）。

如果在方法的程序中有一行 **throw new Exception()**，返回错误，那么其后的程序不执行。因为错误返回后，后面的程序肯定没有机会执行，那么 **JAVA** 认为以后的程序没有存在的必要。

对于 **try……catch** 格式：
try {可能出现错误的代码块**}** **catch(exception e){**进行处理的代码**}** ；
对象变量的声明

用这种方法，如果代码正确，那么程序不经过 **catch** 语句直接向下运行；
如果代码不正确，则将返回的异常对象和 **e** 进行匹配，如果匹配成功，则处理其后面的异常处理代码。（如果用 **exception** 来声明 **e** 的话，因为 **exception** 为所有 **exception** 对象的父类，所有肯定匹配成功）。处理完代码后这个例外就完全处理完毕，程序会接着从出现异常的地方向下执行（是从出现异常的地方还是在 **catch** 后面呢？利用程序进行验证）。最后程序正常退出。

Try 中如果发现错误，即跳出 **try** 去匹配 **catch**，那么 **try** 后面的语句就不会被执行。
一个 **try** 可以跟进多个 **catch** 语句，用于处理不同情况。当一个 **try** 只能匹配一个 **catch**。
我们可以写多个 **catch** 语句，但是不能将父类型的 **exception** 的位置写在子类型的 **excepiton** 之前，因为这样父类型肯定先于子类型被匹配，所有子类型就成为废话。**JAVA** 编译出错。

在 **try**，**catch** 后还可以再跟一子句 **finally**。其中的代码语句无论如何都会被执行（因为 **finally** 子句的这个特性，所以一般将释放资源，关闭连接的语句写在里面）。

如果在程序中书写了检查（抛出）**exception** 但是没有对这个可能出现的检查结果进行处理，那么程序就会报错。
而如果只有处理情况（**try**）而没有相应的 **catch** 子句，则编译还是通不过。
如何知道在编写的程序中会出现例外呢
1. 调用方法，查看 **API** 中查看方法中是否有已检查错误。
2. 在编译的过程中看提示信息，然后加上相应的处理。

Exception 有一个 **message** 属性。在使用 **catch** 的时候可以调用：
Catch(IOException e){System.out.println(e.message())};
Catch(IOException e){e.printStackTrace()};
上面这条语句回告诉我们出错类型所历经的过程，在调试的中非常有用。

开发中的两个道理：
①如何控制 try 的范围：根据操作的连动性和相关性，如果前面的程序代码块抛出的错误影响了后面程序代码的运行，那么这个我们就说这两个程序代码存在关联，应该放在同一个 try 中。
② 对已经查出来的例外，有 throw(积极)和 try catch（消极）两种处理方法。
对于 try catch 放在能够很好地处理例外的位置（即放在具备对例外进行处理的能力的位置）。如果没

有处理能力就继续上抛。

当我们自己定义一个例外类的时候必须使其继承 excepiton 或者 RuntimeException。
Throw 是一个语句，用来做抛出例外的功能。
而 throws 是表示如果下级方法中如果有例外抛出，那么本方法不做处理，继续向上抛出。
Throws 后跟的是例外类型。
断言是一种调试工具（assert）
其后跟的是布尔类型的表达式，如果表达式结果为真不影响程序运行。如果为假系统出现低级错误，在屏幕上出现 assert 信息。
Assert 只是用于调试。在产品编译完成后上线 assert 代码就被删除了。

方法的覆盖中，如果子类的方法抛出的例外是父类方法抛出的例外的父类型，那么编译就会出错：子类无法覆盖父类。
结论：子类方法不可比父类方法抛出更多的例外。子类抛出的例外或者与父类抛出的例外一致，或者是父类抛出例外的子类型。或者子类型不抛出例外。
如果父类型无 **throws** 时，子类型也不允许出现 **throws**。此时只能使用 **try catch**。

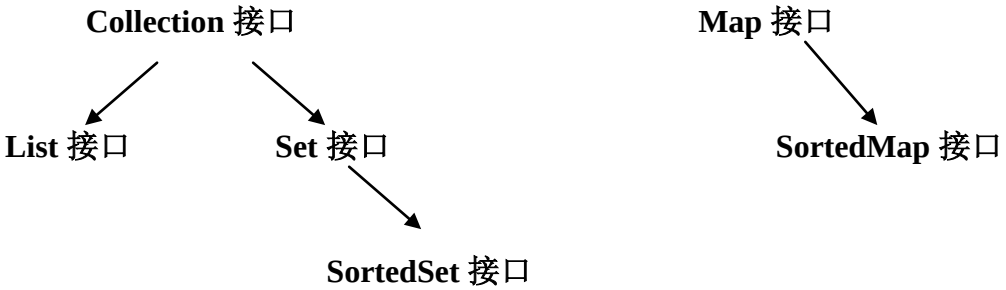
练习：写一个方法：**int add(int a,int b)**

```
{
   return a+b;
}
```

当 **a+b=100;**抛出 **100** 为异常处理。

12.02

集合（从本部分开始涉及 **API**）
集合是指一个对象容纳了多个对象，这个集合对象主要用来管理维护一系列相似的对象。
数组就是一种对象。（练习：如何编写一个数组程序，并进行遍历。）
java.util.*定义了一系列的接口和类，告诉我们用什么类 **NEW** 出一个对象，可以进行超越数组的操作。
（注：**JAVA1.5** 对 **JAVA1.4** 的最大改进就是增加了对范型的支持）
集合框架接口的分类：（分 **collection** 接口 和 **map** 接口）



JAVA 中所有与集合有关的实现类都是这六个接口的实现类。

Collection 接口：集合中每一个元素为一个对象，这个接口将这些对象组织在一起，形成一维结构。

List 接口代表按照元素一定的相关顺序来组织（在这个序列中顺序是主要的），**List** 接口中数据可重复。

Set 接口是数学中集合的概念：其元素无序，且不可重复。（正好与 **List** 对应）

SortedSet 会按照数字将元素排列，为“可排序集合”。

Map 接口中每一个元素不是一个对象，而是一个键对象和值对象组成的键值对（**Key-Value**）。
Key-Value 是用一个不可重复的 **key** 集合对应可重复的 **value** 集合。（典型的例子是字典：通过页码的 **key** 值找字的 **value** 值）。
例子：
key1—value1;
key2—value2;
key3—value3.
SortedMap：如果一个 **Map** 可以根据 **key** 值排序，则称其为 **SortedMap**。（如字典）
!!注意数组和集合的区别：数组中只能存简单数据类型。**Collection** 接口和 **Map** 接口只能存对象。

以下介绍接口：
List 接口：（介绍其下的两个实现类：**ArrayList** 和 **LinkedList**）
ArrayList 和数组非常类似，其底层①也用数组组织数据，**ArrayList** 是动态可变数组。
① 底层：指存储格式。说明 **ArrayList** 对象都是存在于数组中。
注：数组和集合都是从下标 **0** 开始。
ArrayList 有一个 **add(Object o)**方法用于插入数组。
ArrayList 的使用：（完成这个程序）
先 **import java.util.*;**
用 **ArrayList** 在一个数组中添加数据，并遍历。
ArrayList 中数组的顺序与添加顺序一致。
只有 **List** 可用 **get** 和 **size**。而 **Set** 则不可用（因其无序）。
Collection 接口都是通过 **Iterator()**（即迭代器）来对 **Set** 和 **List** 遍历。
通过语句：**Iterator it=c.iterator();** 得到一个迭代器，将集合中所有元素顺序排列。然后可以通过 **interator** 方法进行遍历，迭代器有一个游标（指针）指向首位置。
Interator 有 **hasNext()**，用于判断元素右边是否还有数据，返回 **True** 说明有。然后就可以调用 **next** 动作。**Next()**会将游标移到下一个元素，并把它所跨过的元素返回。（这样就可以对元素进行遍历）
练习：写一个程序，输入对象信息，比较基本信息。
集合中每一个元素都有对象，如有字符串要经过强制类型转换。
Collections 是工具类，所有方法均为有用方法，且方法为 **static**。
有 **Sort** 方法用于给 **List** 排序。
Collections.Sort()分为两部分，一部分为排序规则；一部分为排序算法。
规则用来判断对象；算法是考虑如何排序。
对于自定义对象，**Sort** 不知道规则，所以无法比较。这种情况下一定要定义排序规则。方式有两种：
① java.lang 下面有一个接口：Comparable（可比较的）
可以让自定义对象实现一个接口，这个接口只有一个方法 comparableTo(Object o)
其规则是当前对象与 o 对象进行比较，其返回一个 int 值，系统根据此值来进行排序。
如 当前对象>o 对象，则返回值>0；（可将返回值定义为 1）
如 当前对象=o 对象，则返回值=0；
如 当前对象<o 对象，则返回值〈0。（可将返回值定义为-1）
看 TestArraylist 的 java 代码。
我们通过返回值 1 和-1 位置的调换来实现升序和降序排列的转换。

② java.util 下有一个 Comparator(比较器)
它拥有 compare()，用来比较两个方法。
要生成比较器，则用 Sort 中 Sort（List,List(Compate)）
第二种方法更灵活，且在运行的时候不用编译。

注意：要想实现 comparTo()就必须在主方法中写上 implement comparable.

练习：生成一个 EMPLOYEE 类，然后将一系列对象放入到 ArrayList。用 Iterator 遍历，排序之后，再进行遍历。

集合的最大缺点是无法进行类型判定（这个缺点在 **JAVA1.5** 中已经解决），这样就可能出现因为类型不同而出现类型错误。
解决的方法是添加类型的判断。

LinkedList 接口（在代码的使用过程中和 **ArrayList** 没有什么区别）
ArrayList 底层是 **object** 数组，所以 **ArrayList** 具有数组的查询速度快的优点以及增删速度慢的缺点。
而在 **LinkedList** 的底层是一种双向循环链表。在此链表上每一个数据节点都由三部分组成：前指针（指向前面的节点的位置），数据，后指针（指向后面的节点的位置）。最后一个节点的后指针指向第一个节点的前指针，形成一个循环。
双向循环链表的查询效率低但是增删效率高。所以 **LinkedList** 具有查询效率低但增删效率高的特点。
ArrayList 和 **LinkedList** 在用法上没有区别，但是在功能上还是有区别的。
LinkedList 经常用在增删操作较多而查询操作很少的情况下：队列和堆栈。
队列：先进先出的数据结构。
堆栈：后进先出的数据结构。
注意：使用堆栈的时候一定不能提供方法让不是最后一个元素的元素获得出栈的机会。
LinkedList 提供以下方法：（**ArrayList** 无此类方法）

addFirst();
removeFirst();
addLast();
removeLast();

在堆栈中，**push** 为入栈操作，**pop** 为出栈操作。

Push 用 **addFirst()**；**pop** 用 **removeFirst()**，实现后进先出。
用 **isEmpty()**--其父类的方法，来判断栈是否为空。

在队列中，**put** 为入队列操作，**get** 为出队列操作。
Put 用 **addFirst()**，**get** 用 **removeLast()**实现队列。

List 接口的实现类（**Vector**）（与 **ArrayList** 相似，区别是 **Vector** 是重量级的组件，使用使消耗的资源比较多。）
结论：在考虑并发的情况下用 **Vector**（保证线程的安全）。
　　在不考虑并发的情况下用 **ArrayList**（不能保证线程的安全）。

面试经验（知识点）：
java.util.stack（**stack** 即为堆栈）的父类为 **Vector**。可是 **stack** 的父类是最不应该为 **Vector** 的。因为 **Vector** 的底层是数组，且 **Vector** 有 **get** 方法（意味着它可能访问到并不属于最后一个位置元素的其他元素，很不安全）。
对于堆栈和队列只能用 **push** 类和 **get** 类。
Stack 类以后不要轻易使用。
!!! 实现堆栈一定要用 **LinkedList**。
（在 **JAVA1.5** 中，**collection** 有 **queue** 来实现队列。）

Set-HashSet 实现类：
遍历一个 **Set** 的方法只有一个：迭代器（**interator**）。
HashSet 中元素是无序的（这个无序指的是数据的添加顺序和后来的排列顺序不同），而且元素不可重复。
在 **Object** 中除了有 **final()，toString()，equals()，**还有 **hashCode()**。
HashSet 底层用的也是数组。
当向数组中利用 **add(Object o)**添加对象的时候，系统先找对象的 **hashCode：**
int hc=o.hashCode(); 返回的 **hashCode** 为整数值。
Int I=hc%n;（**n** 为数组的长度），取得余数后，利用余数向数组中相应的位置添加数据，以 **n** 为 **6** 为例，如果 **I=0** 则放在数组 **a[0]**位置，如果 **I=1,**则放在数组 **a[1]**位置。如果 **equals()**返回的值为 **true**，则说明数据重复。如果 **equals()**返回的值为 **false**，则再找其他的位置进行比较。这样的机制就导致两个相同的对象有可能重复地添加到数组中，因为他们的 **hashCode** 不同。
如果我们能够使两个相同的对象具有相同 **hashcode**，才能在 **equals()**返回为真。
在实例中，定义 **student** 对象时覆盖它的 **hashcode**。
因为 **String** 类是自动覆盖的，所以当比较 **String** 类的对象的时候，就不会出现有两个相同的 **string** 对象的情况。
现在，在大部分的 **JDK** 中，都已经要求覆盖了 **hashCode**。
结论：如将自定义类用 hashSet 来添加对象，一定要覆盖 hashcode()和 equals()，覆盖的原则是保证当两个对象 hashcode 返回相同的整数，而且 equals()返回值为 True。
如果偷懒，没有设定 **equals()**，就会造成返回 **hashCode** 虽然结果相同，但在程序执行的过程中会多次地调用 **equals()**，从而影响程序执行的效率。

我们要保证相同对象的返回的 **hashCode** 一定相同，也要保证不相同的对象的 **hashCode** 尽可能不同（因为数组的边界性，**hashCode** 还是可能相同的）。例子：

```
public int hashCode(){
   return name.hashcode()+age;
}
```

这个例子保证了相同姓名和年龄的记录返回的 **hashCode** 是相同的。

使用 **hashSet** 的优点：
hashSet 的底层是数组，其查询效率非常高。而且在增加和删除的时候由于运用的 **hashCode** 的比较开确定添加元素的位置，所以不存在元素的偏移，所以效率也非常高。因为 **hashSet** 查询和删除和增加元素的效率都非常高。
但是 **hashSet** 增删的高效率是通过花费大量的空间换来的：因为空间越大，取余数相同的情况就越小。**HashSet** 这种算法会建立许多无用的空间。
使用 **hashSet** 接口时要注意，如果发生冲突，就会出现遍历整个数组的情况，这样就使得效率非常的低。

练习：new 一个 hashset，插入 employee 对象，不允许重复，并且遍历出来。

添加知识点：
集合对象存放的是一系列对象的引用。
例：

```
Student S
Al.add(s);
s.setName("lucy");
Student s2=(Student)(al.get(o1));
```

可知 **s2** 也是 **s**。

12.05

SortedSet 可自动为元素排序。
SortedSet 的实现类是 TreeSet:它的作用是字为添加到 TreeSet 中的元素排序。

练习：自定义类用 TreeSet 排序。
与 HashSet 不同，TreeSet 并不需要实现 HashCode()和 equals()。
只要实现 compareable 和 compareTo()接可以实现过滤功能。
（注：HashSet 不调用 CompareTo()）。

如果要查询集合中的数据，使用 Set 必须全部遍历，所以查询的效率低。使用 Map，可通过查找 key 得到 value，查询效率高。
集合中常用的是：ArrayList，HashSet，HashMap。其中 ArrayList 和 HashMap 使用最为广泛。

使用 HashMap，put()表示放置元素，get()表示取元素。

遍历 Map，使用 keySet()可以返回 set 值，用 keySet()得到 key 值，使用迭代器遍历，然后使用 put()得到 value 值。
上面这个算法的关键语句：
Set s=m.keySet();
Interator it=new interator();
Object key=it.next();
Object value=m.get(key);

注意：HashMap 与 HashCode 有关，用 Sort 对象排序。
如果在 HashMap 中有 key 值重复，那么后面一条记录的 value 覆盖前面一条记录。

Key 值既然可以作为对象，那么也可以用一个自定义的类。比如：
m.put(new sutdent(“Liucy”,30),”boss”)
如果没有语句来判定 Student 类对象是否相同，则会全部打印出来。

当我们用自定义的类对象作为 key 时，我们必须在程序中覆盖 HashCode()和 equals()。

注：HashMap 底层也是用数组，HashSet 底层实际上也是 HashMap，HashSet 类中有 HashMap 属性（我们如何在 API 中查属性）。HashSet 实际上为(key.null)类型的 HashMap。有 key 值而没有 value 值。

正因为以上的原因，TreeSet 和 TreeMap 的实现也有些类似的关系。
注意：TreeSet 和 TreeMap 非常的消耗时间，因此很少使用。
我们应该熟悉各种实现类的选择——非常体现你的功底。

HashSet VS TreeSet：HashSet 非常的消耗空间，TreeSet 因为有排序功能，因此资源消耗非常的高，我们应该尽量少使用，而且最好不要重复使用。
基于以上原因，我们尽可能的运用 HashSet 而不用 TreeSet，除非必须排序。
同理：HashMap VS TreeMap:一般使用 HashMap，排序的时候使用 TreeMap。
HashMap VS Hashtable（注意在这里 table 的第一个字母小写）之间的区别有些类似于 ArrayList 和 Vector，Hashtable 是重量级的组件，在考虑并发的情况，对安全性要求比较高的时候使用。

Map 的运用非常的多。

使用 **HashMap()**，如果使用自定义类，一定要覆盖 **HashCode()**和 **equals()**。

重点掌握集合的四种操作：增加、删除、遍历、排序。

Module8—12 利用两天的时间完成。
Module8：图形界面
Module9：事件模型（在本部分最重要）
Module10：**AWT**
Module11：**Swing**
Module12：**Applet**（这个技术基本已经被淘汰）

软件应用的三个发展阶段：
单机应用
网络应用（**C/S** 结构）
BS 结构：**B** 表示浏览器，**S** 表示 **server** 端。即利用浏览器作为客户端，因此对于图形界面的要求已经不高，现在的发展趋势是不使用安装，即不用任何的本地应用，图形很快就会被服务器构件开发所取代。

经验之谈：**Swing** 的开发工作会非常的累，而且这项技术正在走向没落。避免从事有这种特征的工作。
AWT 也即将被取代。
Module8—Module11 所使用的技术都将被 **JSF** 技术所取代。
JSF 是服务器端的 **Swing**：目前技术已经成熟，但是开发环境（工具）还不成熟。
Module12 的 **Applet** 技术也将被 **WebStart** 所取代。
Module9 为重点，所谓事件模型是指观察者设计模式的 **JAVA** 应用。事件模型是重点。

Module8：图形界面（**java.awt.***）
Awt：抽象窗口工具箱，它由三部分组成：
①组件：界面元素；
②容器：装载组件的容器（例如窗体）；
③布局管理器：负责决定容器中组件的摆放位置。
图形界面的应用分四步：
① 选择一个容器：
⑴window:带标题的容器（如 Frame）；
⑵Panel:面板
通过 add()想容器中添加组件。
Java 的图形界面依然是跨平台的。但是在调用了一个窗体之后只生成一个窗体，没有事件的处理，关闭按钮并不工作。此时只能使用 CTRL+C 终止程序。
②设置一个布局管理器：用 setLayout()；
③向容器中添加组件；
③ 添加组件的事务处理。P198

P204：Panel 也是一种容器：但是不可见的。在设置容易的时候不要忘记设置它们的可见性。
Panel pan=new Panel;
Fp.setLayout(null);表示不要布局管理器。

五种布局管理器：
P206：Flow Layout(流式布局)：按照组件添加到容器中的顺序，顺序排放组件位置。默认为水平排列，如果越界那么会向下排列。排列的位置随着容器大小的改变而改变。
Panel 默认的布局管理器为 Flow Layout。

Border Layout：会将容器非常五个区域：东西南北中。
语句：
Button b1=new Botton(“north”);//botton 上的文字
f.add(b1,”North”);//表示 b1 这个 botton 放在 north 位置
注：一个区域只能放置一个组件，如果想在一个区域放置多个组件就需要使用 Panel 来装载。
Frame 和 Dialog 的默认布局管理器是 Border Layout。

Grid Layout：将容器生成等长等大的条列格，每个块中放置一个组件。
f.setLayout GridLayout(5,2,10,10)//表示条列格为 5 行 2 类，后面为格间距。

CardLayout:一个容器可以放置多个组件，但每次只有一个组件可见（组件重叠）。
使用 first()，last()，next()可以决定哪个组件可见。可以用于将一系列的面板有顺序地呈现给用户。

重点：GridBag Layout:在 Grid 中可指定一个组件占据多行多列，GridBag 的设置非常的烦琐。

Module9:AWT:事件模型
事件模型指的是对象之间进行通信的设计模式。
对象 1 给对象 2 发送一个信息相当于对象 1 引用对象 2 的方法。
模型即是一种设计模式（约定俗成）
对象对为三种：
①事件源：发出事件者；
②事件对象：发出的事件本身；
④ 事件监听器：提供处理事件指定的方法。

Java AWT 事件模型也称为授权事件模型，指事件可以和监听器之间事先建立一种关系：约定那些事件如何处理，由谁去进行处理。这种约定称为授权。
一个事件源可以授权多个监听者（授权也称为监听者的注册）；
多个事件源也可以注册多个事件监听器。
监听者对于事件源的发出的事件作出响应。

在 java.util 中有 **EventListener** 接口：所有事件监听者都要实现这个接口。
java.util 中有 **EventObject** 类：所有的事件都为其子类。

事件范例在**\CoreJava\Girl.java** 文件中。**(**文件已加注释**)**

注意：接口因对不同的事件监听器对其处理可能不同，所以只能建立监听的功能，而无法实现处理。

下面程序建立监听功能：

```
//监听器接口要定义监听器所具备的功能，定义方法
{
    void WhatIdoWhenGirlHappy(EmotionEvent e);
```

```
    void WhatIdoWhenGirlSad(EmotionEvent e);
}
```
注意查看参考书：事件的设置模式，如何实现授权模型。

事件模式的实现步骤：
开发事件对象（事件发送者）——接口——接口实现类——设置监听对象
一定要理解透彻 **Gril.java** 程序。

重点：学会处理对一个事件源有多个事件的监听器（在发送消息时监听器收到消息的排名不分先后）。
事件监听的响应顺序是不分先后的，不是谁先注册谁就先响应。
事件监听由两个部分组成（接口和接口的实现类）。

事件源	事件对象	事件监听
gril	**EmotinEvent**	**EmotionListener(接口)、Boy(接口的实现类)**

鼠标事件：**MouseEvent**，接口：**MouseListener**。
P235 ActionEvent。
注意在写程序的时候：**import java.awt.*;**以及 **import java.awt.event.***注意两者的不同。

在生成一个窗体的时候，点击窗体的右上角关闭按钮激发窗体事件的方法：窗体 **Frame** 为事件源，**WindowsListener** 接口调用 **Windowsclosing()**。
为了配合后面的实现，我们必须将 **WindowsListener** 所有的方法都实现，除了 **Windowsclosing** 方法，其余的方法均为空实现。
（练习：写一个带 **button** 窗体，点关闭按钮退出。）
上面程序中实现了许多不必要的实现类，虽然是空实现。
为了避免上面那些无用的实现，可以利用 **WindowEvent** 的一个 **WindowEvent** 类，还是利用 **windowsListener**。还有 **WindowAdapter** 类，它已经实现了 **WindowsListener**。它给出的全部都是空实现，那就可以只写想要实现的类，去覆盖其中的类，就不用写空实现。
注意：监听过多，会抛 **tooManyListener** 例外。

12.06

Module 10
Canvas 组件：画布，可以实现动画操作。
TextArea：文本域。
在单行文本域中回车会激发 **ActionEvent**。
用 **CheckBoxGroup** 实现单选框功能。
Java 中，单选框和复选框都是使用 **CheckBox** 实现。
菜单：**new MenuBar()**，**MenuBar** 表示菜单条。
菜单中的每一项为 **MenuItem**，一般级联菜单不应该超过三级。
练习：
设计一个计算器：注意设置一个 **boolean** 值（**append**）来判断输入数字是位于第一个数的后面还是属于输入的第二个数。
设置一个变量来存放“+”，点完运算符后，将 **append** 设置为 **false**。
String number1
Char operator 存放运算符。

Module 11 Swing
AWT 是 **Java** 最早出现的图形界面，但很快就被 **Swing** 所取代。
Swing 才是一种真正的图形开发。

AWT 在不同平台所出现的界面可能有所不同：因为每个 OS 都有自己的 UI 组件库，java 调用不同系统的 UI。
注意 AWT 为重量级组件，相当消耗资源，且不同系统的组件可能不同。因为这个问题使得 AWT 开发的软件难以作到跨平台。
更为要命的是：不同 OS 的组件库都存在 BUG。必须多种平台进行测试，并且 AWT 的组件库并不丰富。
为解决以上问题，SUN 和 IBM 以及 NETSCAPE 联合开发出 JAVA 基础类包 Swing：注意 JAVA 的基础类以 Swing 为核心。
注意引用：javax.swing.*;javax 表示 JAVA 的扩展。

我们在学习 JDBC 的时候会过度到 J2EE。

在 Swing 的组件中，基本上都是在 AWT 组件的名称前面加“J”。
一般情况下，除了 Choise 等组件:
import javax.swing.*;好要加上：import java.awt.*以及 import java.awt.event.*。
Swing 与 AWT 的最大区别是 Swing 为 JAVA 自身的组件。已经不是对等实体，与底层的 OS 无关。（JBUILDER 就是使用 Swing 写的）

Swing 与 AWT 在事件模型处理上是一致的。
Jframe 实际上是一堆窗体的叠加。
Swing 比 AWT 更加复杂且灵活。
在 JDK1.4 中，给 JFRAME 添加 Button 不可用 jf.add(b)。而是使用 jf.getContentPane().add(b)。
content 是先申请面板。不过在 JDK1.5 中可以使用 add.。
Jpanel 支持双缓冲技术。
在 Jbutton 中可以添加图标。
JscrollPane 可以管理比屏幕还要大的组件。
TextArea 只有装入 JscrollPane 中才能实现滚动条。
JeditorPane 用于显示浏览器。
注意：Tabbed Panel 与 Border 的比较。
进度条：ProgressBar。
JcomboBox：下拉菜单：在 AWT 中同类组件是 choice。
JlistPanel：选择列表
BorderPanel：设置边框
JsplitPanel：可将容器分为两个部分，其中一个部分有 Jtree。
TextBox：也是一种新的容器，可以设置组件的间距。
TextFileChoose：文件选择器。
ColorChoose：颜色选择器
Module 12　Applet
Applet 为 Panel 的子类
Applet 是 java 的自动执行方式（这是它的优势，主要用于 HTML）。
工作四种语法：init()，start()，stop()，destory()。
Swing 中有一个 Japplet，如使用 Swing 组件。
Applet 消亡的原因：
①java 为安全起见对 Applet 有所限制：Applet 不允许访问本地文件信息、敏感信息，不能执行本地指令（比如 FORMAT），不能访问初原服务器之外的其他服务器。
② IE 不支持新版本的 Applet。
Applet 的优势：
网络传输，自动下载。

Application 的优势：没有执行限制。
WebStart：可在网络传输，并且在本地无限制。因此前景光明。

练习：
使用 Swing 实现一个界面，分为上下两个部分，南边为 JtextField 组件，可编辑，上面为 JtextArea 组件，不可编辑，在 JtextField 组件输入字符，按回车，就可以将内容输入到 JtextArea 组件。(AREA 区域可以滚动)

12.07

多线程
进程：任务
任务并发执行是一个宏观概念，微观上是串行的。
进程的调度是有 OS 负责的（有的系统为独占式，有的系统为共享式，根据重要性，进程有优先级）。

由 OS 将时间分为若干个时间片。
JAVA 在语言级支持多线程。
分配时间的仍然是 OS。
参看 P377

线程由两种实现方式：
第一种方式：

```
class MyThread extends Thread{
    public void run(){
    需要进行执行的代码，如循环。
     }
}

public class TestThread{
   main(){
    Thread t1=new Mythread();
    T1.start();
     }
}
```

只有等到所有的线程全部结束之后，进程才退出。

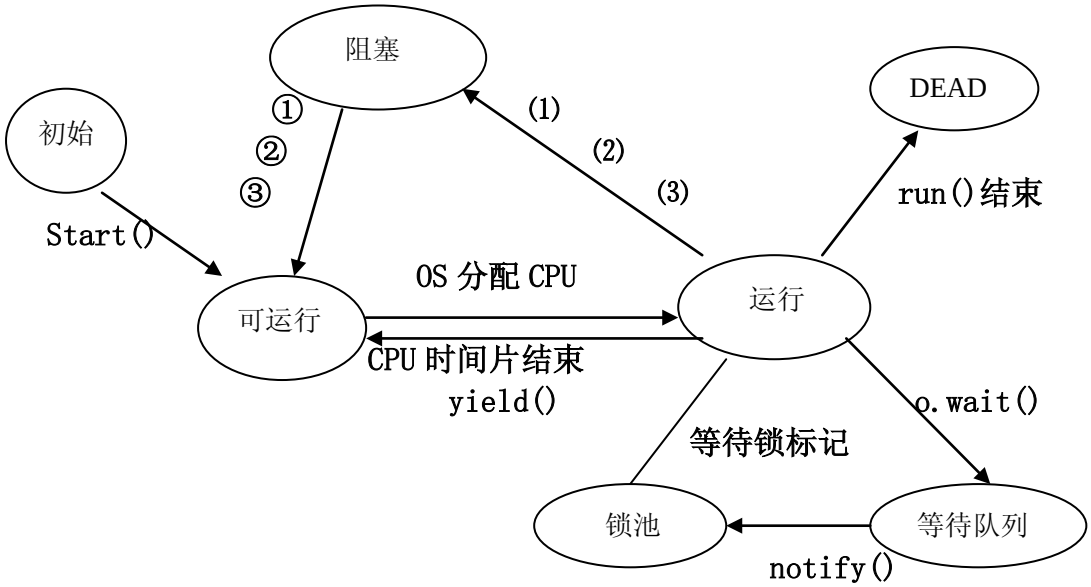
第二种方式：

```
Class MyThread    implements Runnable{
        Public    void run(){
        Runnable target=new MyThread();
        Thread t3=new Thread(target);
        Thread.start();//启动线程
        }
}
```

P384:通过接口实现继承

练习：写两个线程：
① 输入 200 个“###”②输入 200 个“***”

下面为线程中的 7 中非常重要的状态：（有的书上也只有认为前五种状态：而将“锁池”和“等待队列”都看成是“阻塞”状态的特殊情况：这种认识也是正确的，但是将“锁池”和“等待队列”单独分离出来有利于对程序的理解）



注意：图中标记依次为
①输入完毕；②wake up③t1 退出
⑴如等待输入（输入设备进行处理，而 CUP 不处理），则放入阻塞，直到输入完毕。
⑵线程休眠 sleep（）
⑶t1.join()指停止 main()，然后在某段时间内将 t1 加入运行队列，直到 t1 退出，main()才结束。
特别注意：①②③与⑴⑵⑶是一一对应的。

进程的休眠：Thread sleep(1000);//括号中以毫秒为单位
当 **main()**运行完毕，即使在结束时时间片还没有用完，**CPU** 也放弃此时间片，继续运行其他程序。
Try{Thread.sleep(1000);}
Catch(Exception e){e.printStackTrace(e);}
T1.join()表示运行线程放弃执行权，进入阻塞状态。
当 **t1** 结束时，**main()**可以重新进入运行状态。
T1.join 实际上是把并发的线程编程并行运行。
线程的优先级：**1-10**，越大优先级越高，优先级越高被 **OS** 选中的可能性就越大。（不建议使用，因为不同操作系统的优先级并不相同，使得程序不具备跨平台性，这种优先级只是粗略地划分）。
注：程序的跨平台性：除了能够运行，还必须保证运行的结果。

一个使用 **yield()**就马上交出执行权，回到可运行状态，等待 **OS** 的再次调用。

下午：
程序员需要关注的线程同步和互斥的问题。
多线程的并发一般不是程序员决定，而是由容器决定。
多线程出现故障的原因：
两个线程同时访问一个数据资源（临界资源），形成数据发生不一致和不完整。
数据的不一致往往是因为一个线程中的两个关联的操作只完成了一步。

避免以上的问题可采用对数据进行加锁的方法
每个对象除了属性和方法，都有一个 monitor（互斥锁标记），用来将这个对象交给一个线程，只有拿到 monitor 的线程才能够访问这个对象。
Synchronized:这个修饰词可以用来修饰方法和代码块

Object obj;
Obj.setValue(123);
Synchronized 用来修饰方法，表示当某个线程调用这个方法之后，其他的事件不能再调用这个方法。只有拿到 obj 标记的线程才能够执行代码块。
注意：Synchronized 一定使用在一个方法中。
锁标记是对象的概念，加锁是对对象加锁，目的是在线程之间进行协调。

当用 Synchronized 修饰某个方法的时候，表示该方法都对当前对象加锁。
给方法加 Synchronized 和用 Synchronized 修饰对象的效果是一致的。

一个线程可以拿到多个锁标记，一个对象最多只能将 monitor 给一个线程。
Synchronized 是以牺牲程序运行的效率为代价的，因此应该尽量控制互斥代码块的范围。

方法的 Synchronized 特性本身不会被继承，只能覆盖。

线程因为未拿到锁标记而发生的阻塞不同于前面五个基本状态中的阻塞，称为锁池。
每个对象都有自己的一个锁池的空间，用于放置等待运行的线程。
这些线程中哪个线程拿到锁标记由系统决定。

锁标记如果过多，就会出现线程等待其他线程释放锁标记，而又都不释放自己的锁标记供其他线程运行的状况。就是死锁。
死锁的问题通过线程间的通信的方式进行解决。
线程间通信机制实际上也就是协调机制。
线程间通信使用的空间称之为对象的等待队列，则个队列也是属于对象的空间的。
Object 类中又一个 wait()，在运行状态中，线程调用 wait()，此时表示着线程将释放自己所有的锁标记，同时进入这个对象的等待队列。
等待队列的状态也是阻塞状态，只不过线程释放自己的锁标记。
Notify()
如果一个线程调用对象的 notify()，就是通知对象等待队列的一个线程出列。进入锁池。如果使用 notifyall()则通知等待队列中所有的线程出列。

注意：只能对加锁的资源进行 wait()和 notify()。

释放锁标记只有在 Synchronized 代码结束或者调用 wait()。
注意锁标记是自己不会自动释放，必须有通知。
注意在程序中判定一个条件是否成立时要注意使用 WHILE 要比使用 IF 要严密。
WHILE 会放置程序饶过判断条件而造成越界。
补充知识：
suspend()是将一个运行时状态进入阻塞状态(注意不释放锁标记)。恢复状态的时候用 resume()。Stop()指释放全部。
这几个方法上都有 Deprecated 标志，说明这个方法不推荐使用。

一般来说，主方法 main()结束的时候线程结束，可是也可能出现需要中断线程的情况。对于多线程一般每个线程都是一个循环，如果中断线程我们必须想办法使其退出。

如果主方法 main()想结束阻塞中的线程（比如 sleep 或 wait）
那么我们可以从其他进程对线程对象调用 interrupt()。用于对阻塞（或锁池）会抛出例外 InterruptedException。
这个例外会使线程中断并执行 catch 中代码。

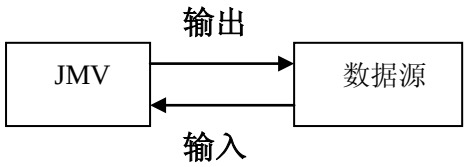
多线程中的重点：实现多线程的两种方式，Synchronized, 以及生产者和消费者问题（ProducerConsumer.java 文件）。

练习：
① 存车位的停开车的次序输出问题；
② 写两个线程，一个线程打印 1-52，另一个线程答应字母 A-Z。打印顺序为 12A34B56C……5152Z。通过使用线程之间的通信协调关系。
注：分别给两个对象构造一个对象 **o**，数字每打印两个或字母每打印一个就执行 **o.wait()**。在 **o.wait()**之前不要忘了写 **o.notify()**。

补充说明：通过 Synchronized，可知 Vector 较 ArrayList 方法的区别就是 Vector 所有的方法都有 Synchronized。所以 Vector 更为安全。
同样：Hashtable 较 HashMap 也是如此。

12.08

Module 10：I/O 流（java 如何实现与外界数据的交流）
Input/Output：指跨越出了 JVM 的边界，与外界数据的源头或者目标数据源进行数据交换。



注意：输入/输出是针对 **JVM** 而言。
File 类（**java.io.***）可表示一个文件，也有可能是一个目录（在 **JAVA** 中文件和目录都属于这个类中，而且区分不是非常的明显）。
Java.io 下的方法是对磁盘上的文件进行磁盘操作，但是无法读取文件的内容。
注意：创建一个文件对象和创建一个文件在 **JAVA** 中是两个不同的概念。前者是在虚拟机中创建了一个文件，但却并没有将它真正地创建到 **OS** 的文件系统中，随着虚拟机的关闭，这个创建的对象也就消失了。而创建一个文件才是在系统中真正地建立一个文件。
例如：**File f=new File(“11.txt”);//创建一个名为 11.txt 的文件对象**
f.CreateNewFile(); //真正地创建文件

f.CreateMkdir()：创建目录
f.delete()；删除文件
f.deleteOnExit();在进程退出的时候删除文件，这样的操作通常用在临时文件的删除。

对于命令：**File f2=new file(“d:\\abc\\789\\1.txt”)**
这个命令不具备跨平台性，因为不同的 **OS** 的文件系统很不相同。
如果想要跨平台，在 **file** 类下有 **separtor()**，返回锁出平台的文件分隔符。
File.fdir=new File(File.separator);

String str=”abc”+File.separator+”789”;
使用文件下的方法的时候一定注意是否具备跨平台性。

List()：显示文件的名（相对路径）
ListFiles()：返回 **Files** 类型数组，可以用 **getName()**来访问到文件名。
使用 **isDirectory()**和 **isFile()**来判断究竟是文件还是目录。

练习：
写一个 **javaTest** 程序，列出所有目录下的***.java** 文件，把子目录下的 **JAVA** 文件也打印出来。

使用 **I/O** 流访问 **file** 中的内容。
JVM 与外界通过数据通道进行数据交换。
分类：
按流分为输入流和输出流；
按传输单位分为字节流和字符流；
还可以分为节点流和过滤流。
节点流：负责数据源和程序之间建立连接；
过滤流：用于给节点增加功能。
过滤流的构造方式是以其他流位参数构造（这样的设计模式称为装饰模式）。

字节输入流：**io** 包中的 **InputStream** 为所有字节输入流的父类。
Int read();读入一个字节（每次一个）；
可先使用 **new byte[]**=数组，调用 **read(byte[] b)**
read (byte[])返回值可以表示有效数；**read (byte[])**返回值为**-1** 表示结束。

字节输出流：**io** 包中的 **OutputStream** 位所有字节输入流的父类。
Write 和输入流中的 **read** 相对应。

在流中 **close()**方法由程序员控制。因为输入输出流已经超越了 **VM** 的边界，所以有时可能无法回收资源。
原则：凡是跨出虚拟机边界的资源都要求程序员自己关闭，不要指望垃圾回收。
以 **Stream** 结尾的类都是字节流。
如果构造 **FileOutputStream** 的同时磁盘会建立一个文件。如果创建的文件与磁盘上已有的文件名重名，就会发生覆盖。
用 **FileOutputStream** 中的 **boolean**，则视，添加情况，将数据覆盖重名文件还是将输入内容放在文件的后面。（编写程序验证）

DataOutputStream:输入数据的类型。
因为每中数据类型的不同，所以可能会输出错误。
所有对于：**DataOutputStream**
DataInputStream
两者的输入顺序必须一致。
过滤流：
bufferedOutputStream
bufferedInputStream
用于给节点流增加一个缓冲的功能。
在 **VM** 的内部建立一个缓冲区，数据先写入缓冲区，等到缓冲区的数据满了之后再一次性写出，效率很高。
使用带缓冲区的输入输出流的速度会大幅提高，缓冲区越大，效率越高。（这是典型的牺牲空间换时间）

切记：使用带缓冲区的流，如果数据数据输入完毕，使用 **flush** 方法将缓冲区中的内容一次性写入到外部数据源。用 **close()**也可以达到相同的效果，因为每次 **close** 都会使用 **flush**。一定要注意关闭外部的过滤流。

（非重点）管道流：也是一种节点流，用于给两个线程交换数据。
PipedOutputStream
PipedInputStream
输出流：**connect(**输入流**)**

RondomAccessFile 类允许随机访问文件
GetFilepoint()可以知道文件中的指针位置，使用 **seek()**定位。
Mode(“r”:随机读；**”w”**：随机写；**”rw”**：随机读写**)**
练习：写一个类 **A**，**JAVA A file1 file2**
file1 要求是系统中已经存在的文件。**File2** 是还没有存在的文件。
执行完这个命令，那么 **file2** 就是 **file1** 中的内容。

字符流：**reader\write** 只能输纯文本文件。
FileReader 类：字符文件的输出

字节流与字符流的区别：
字节流的字符编码：
字符编码把字符转换成数字存储到计算机中，按 **ASCii** 将字母映射为整数。
把数字从计算机转换成相应的字符的过程称为解码。
编码方式的分类：
ASCII（数字、英文）**:1** 个字符占一个字节（所有的编码集都兼容 **ASCII**）
ISO8859-1（欧洲）：**1** 个字符占一个字节
GB-2312/GBK：**1** 个字符占两个字节
Unicode: 1 个字符占两个字节（网络传输速度慢）
UTF-8：变长字节，对于英文一个字节，对于汉字两个或三个字节。

原则：保证编解码方式的统一，才能不至于出现错误。
Io 包的 **InputStreamread** 称为从字节流到字符流的桥转换类。这个类可以设定字符转换方式。
OutputStreamred:字符到字节
Bufferread 有 **readline()**使得字符输入更加方便。
在 **I/O** 流中，所有输入方法都是阻塞方法。
Bufferwrite 给输出字符加缓冲，因为它的方法很少，所以使用父类 **printwrite**，它可以使用字节流对象，而且方法很多。

练习：做一个记事本
swing/JfileChoose: getSelect file()
InputStreeamReader：把字节变为字符
JAVA 中对字符串长无限制　**bufferedReader**（**ir**）

12.09

class ObjectOutputStream 也是过滤流，使节点流直接获得输出对象。
最有用的方法：**WriteObject(Object b)**
用流传输对象称为对象的序列化，但并不使所有的对象都可以进行序列化的。只有在实现类时必须实现一个接口：**IO** 包下的 **Serializable(**可序列化的**)**。此接口没有任何的方法，这样的接口称为标记接口。
Class Student implements Serializable
把对象通过流序列化到某一个持久性介质称为对象的可持久化。
Hibernate 就是研究对象的可持久化。
ObuectInputStream in =new ObjectInputStream;
Object o1=in.readObuect();
Student s1=(Student)o1;
注意：因为 **o1** 是一个对象，因为需要对其进行保存。

Transient 用来修饰属性。
Transient int num;
表示当我们对属性序列化时忽略这个属性（即忽略不使之持久化）。
所有属性必须都是可序列化的，特别是当有些属性本身也是对象的时候，要尤其注意这一点。
判断是否一个属性或对象可序列化：**Serialver**。
Serialver TestObject（**TestObject** 必须为已经编译）
执行结果：如果不可序列化；则出现不可序列化的提示。如果可以序列化，那么就会出现序列化的 **ID**：**UID**。
java.until.*有
StringTokenizer（参数 **1**，参数 **2**）按某种符号隔开文件
StringTokenizer(s,":") 用“**:**”隔开字符，**s** 为对象。

练习：将一个类序列化到文件，然后读出。下午：
1、 网络基础知识
2、 **JAVA** 网络编程

网络与分布式集群系统的区别：每个节点都是一台计算机，而不是各种计算机内部的功能设备。
Ip:具有全球唯一性，相对于 **internet**，**IP** 为逻辑地址。
端口**(port)**：一台 **PC** 中可以有 **65536** 个端口，进程通过端口交换数据。连线的时候需要输入 **IP** 也需要输入端口信息。
计算机通信实际上的主机之间的进程通信，进程的通信就需要在端口进行联系。
192.168.0.23:21
协议：为了进行网络中的数据交换（通信）而建立的规则、标准或约定。
不同层的协议是不同的。

网络层：寻址、路由（指如何到达地址的过程）
传输层：端口连接
TCP 模型：应用层/传输层/网络层/网络接口
端口是一种抽象的软件结构，与协议相关：**TCP23** 端口和 **UDT23** 端口为两个不同的概念。
端口应该用 **1024** 以上的端口，以下的端口都已经设定功能。
套接字**(socket)**的引入：

Ip+Port=Socket（这是个对象的概念。）
Socket 为传输层概念，而 JSP 是对应用层编程。例：
java.net.*;
(Server 端定义顺序)
ServerSocket(intport)
Socket.accept()；//阻塞方法，当客户端发出请求是就恢复
如果客户端收到请求：
则 Socket SI=ss.accept();
注意客户端和服务器的 Socket 为两个不同的 socket。
Socket 的两个方法：
getInputStream()：客户端用
getOutputStream() 服务器端用
使用完毕后切记 Socket.close()，两个 Socket 都关，而且不用关内部的流。
在 client 端，Socket s=new Socket(“127.0.0.1”,8000);
127.0.0.1 为一个默认本机的地址。
练习：
1、 客户端向服务器发出一个字符串，服务器转换成大写传回客户端。
大写的函数：String.toUpperCase()
2、 服务器告诉客户端：“自开机以来你是第 n 个用户”。

12.12

UDP 编程：
DatagramSocket（邮递员）：对应数据报的 Socket 概念，不需要创建两个 socket，不可使用输入输出流。
DatagramPacket（信件）：数据包，是 UDP 下进行传输数据的单位，数据存放在字节数组中。
UDP 也需要现有 Server 端，然后再有 Client 端。
两端都是 DatagramPacket（相当于电话的概念），需要 NEW 两个 DatagramPacket。
InetAddress:网址
这种信息传输方式相当于传真，信息打包，在接受端准备纸。
模式：
发送端：Server:
DatagramPacket inDataPacket=new DatagramPacket ((msg,msg.length); InetAdress.getByName(ip),port);
接收端：
clientAddress=inDataPack.getAddress();//取得地址
clientPort=inDataPack.getPort();//取得端口号
datagramSocket.send; //Server
datagramSocket.accept; //Client
URL:在应用层的编程
注意比较：
http://Localhost:8080/directory //查找网络服务器的目录
file://directory //查找本地的文件系统
java 的开发主要以 http 为基础。
反射：主要用于工具和框架的开发。
反射是对于类的再抽象；通过字符串来抽象类。
JAVA 类的运行：classLoader:加载到虚拟机（vm）
Vm 中只能存储对象（动态运行时的概念），.class 文件加载到 VM 上就成为一个对象，同时初始静态成员及静态代码（只执行一次）。

Lang 包下有一个类为 **Class**：在反射中使用。此类中的每个对象为 **VM** 中的类对象，每个类都对应类类的一个对象（**class.class**）。

例：对于一个 **Object** 类，用 **getClass()**得到其类的对象，获得类的对象就相当于获得类的信息，可以调用其下的所有方法，包括类的私有方法。

注意：在反射中没有简单数据类型，所有的编译时类型都是对象。

反射把编译时应该解决的问题留到了运行时。