JAVA 重点难点

--前人学习笔记+自己学习笔记整理加工.

一、基础

1、1995 年正式发布 JAVA

2、OAK-JAVA 雏形，JAVA 发布之前用(语法借鉴 C++，思想借鉴 SMOLLTALK)；

3、1998 年,JAVA2 产生(JAVA 的最大特点：开源 OPEN);bea 公司 J2EE 正式应用火红

4、JDK1.5(简称 Tigger)最新

5、j2EE 表示层，业务层，数据层三大组成

6、环境变量：JAVAHOME：JDK 的安装路径，

PATH：%JAVAHOME%\bin —虚拟机执行命令的路径

CLASSPATH:%JAVAHOME%\lib 字节码文件的路径，可用；号间隔多个选择路径

JBUILDER ORICAL 数据库，自带有 JDK1.4，有了 JBUILDER 可以不用安装 JDK；

ORICAL 数据库，自带有 JDK1.3,应把想要应用的 JDK 版本的路径写在环境变量的最前面.

7、编译器：全部解释再执行 解释器：解释一条执行一条

JAVA 程序是先编译后解释执行.

8、系统垃圾收集是自动的，即使显示调用，也是由系统控制的；

(显示调用也不一定马上收集，还是由系统控制)(java.lang)System.gc();Runtime.getRuntime().gc();

9、JVM 工作：类加载，字节码校验，解释器→成为机器能执行的机器语言

二、简单程序

1、源文件名一定要和 public 定义的类的类名一样；

没有 public 类的话，源文件名可以随意；

2、main()方法可以处于同意源文件中的不同类中;是 java 执行的入口；

Java 后面跟的是含有 main()方法的类的类名；java 根据类名进入不同入口；

一个源文件中有多少类就有多少个.class 文件

三、打包

1、把源文件(字节码文件)打包到某个包中，使用 package 关键字，且一定是放在文件头，一定只有一句!；

2、包名小写；

3、使用打包后，java Hello.class 执行程序(注意顺序一定是：package-import-class,不能将其他东西插到中间)

4. 强制打包:javac -d . Hello.java. 将在当前目录下生成 Hello.java 内声明的 package.

四、帮助文档

1、生成帮助文档：javadoc 包名.源文件名

五、JAVA 的 JDK 分成 2 部分：JRE 和类库

1、系统只要装了 JRE 就可以运行 JAVA 字节码程序（必须要是通过编译后）

2、JAR 一般打包的是.class 文件;一般 jar 包双击后可以运行（包名.jar);

3、要运行的话要在清单文件中加：Main-Class: Hello

4、JAR 打包：先用 jar cvf ~.jar Hollo.class(或包名)(是 Class 文件，不是 JAVA 文件)

改.mf 文件：加 Main-Class: 包名.Hollo(运行的 Class 文件)

标识符,数据类型,运算符,条件控制,数组

标识符：可以以 __ 和 $ 符开头.

命名规范：

1. 类：首字母大写，第二个单词开始首字母大写，一般为名词.

2. 方法：首字母小写，第二个单词开始首字母大写，动词.

3. 变量：首字母小写，第二个单词开始首字母大写.

4. 常量：大写，单词之间用下划线连接.

变量：

实例变量声明时可以不初始化，系统会自动初始化为 0 或 null.

局部变量在使用前必先附值.

局部变量与实例变量重名时以局部变量为准.

局部变量在它的作用域内不允许同名.

基本数据类型：

各原始数据类型的默认值

Byte	Short	Int	Long	Float	Double	Char	boolean	reference
0	0	0	0L	0.0F	0.0D	'\u0000'	flase	Null
8 位	16 位	32 位	64 位	32 位	64 位	32 位		

基本数据类型从小到大转化系统可自动转换，从大往小转换需显式强制转换.

基本数据类型在进行动算时可能会发生精度损失.

num1 + num2　　以大的数据类型为准，否则为 int.

byte ba=1;　ba=ba+1;错误

　　ba+=1;　正确

条件控制：

switch(temp)语句中，default：可以放在任意位置，效果一样。括号中能放 32 位及以下的有范围的类型表达式。

System.exit(0)-----退出整个程序。

1－temp 的类型只能为 byte,short,char,int；
2－case 之后的表达式只能是一个常量整型表达式，即任何字符常量的组合，及能计算成一个常量整数值的整型常量(非 boolean 型)；
3－每个 case 子句后面，应该跟一个 break;
4－default 子句，能放在 switch 的任何地方：top,end,middle

数组：

数组初始化时一定要指定其维数。

动态初始化数组：int[] iArray=new int[5];

静态初始化数组：int[] iArray={1,2,3,4,5}; iArray.length=5;

附值： int[] iArray=new int[] {1,2,3,4,5} iArray.length=5;

iArray 为数组的引用，存放着该数组的起始地址。

二维数组：

int[] [] a=new int[2][3]; 即数组的数组。

列数先不建立的二维数组：

```
int[] [] b=new int[2][];
a[0]      =new int[4];
a[1]      =new int[3];
```

break,continue,return,exit();的区别

break－跳出本层循环
continue－跳出本次循环
return－跳出方法
exit()－跳出执行的程序

面向对象

面向对象主要针对面向过程。面向过程的基本单元是函数。
在 JAVA 中，一切都是对象，但我们用来操纵对象的却是引用。

对象包括变量(属性)和方法，变量表示对象的属性，方法用来描述对象的功能，处理过程。

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。

面向对象是先有数据结构，然后再有算法。

从语法上来看，一个类是一个新的数据类型。

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。构造方法的方法名与类名相同。如果用户没有为类编写构数方法，则系统会为该类生成一个无参的默认构造方法，否则，系统将不再为该类自动生成无参构造方法。

this 表示当前对象，this()表示本类构造方法。super 表示直接父类对象，super()表示直接父类构造方法。this(),super()均可带参数。

继承

父类（SuperClass）和 子类（SonClass）。
父类的非私有化属性和方法（非构造方法）可以默认继承到子类。

```
class Son extends Father{
}
```

JAVA 中只支持单继承。JAVA 通过接口和内部类实现多继承。
父类的构造方法子类不能继承，子类只能在自己的构造方法中访问父类的构造方法。
方法覆盖(OVERRIDE)：
当父类中的非私有方法跟子类的方法名一样，参数一样，返回类型也一样时，称为子类方法对父类方法的覆盖。这时，子类方法的访问控制权限不能比父类方法的访问控制权限更严格，也不能抛出更多的异常，否则编译出错。这也是 JAVA 之所以有动态多态的原因。

修饰属性和方法的修饰符，以下范围依次由严到宽：
private ：本类访问；
default ：同包可见。
protected：同包可见+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，系统先递归构造父类对象，再构造子类对象。
super()表示调用父类的构造方法。this()用于调用本类的构造方法,可以有参或无参,以此调用相应的构造函数。
Super()和 this()一样，如果出现，则必须放在构造函数的第一行,且两者不能同时出现在同一构造函数中。
如果没有调用 super()和 this()构造方法，那么系统会自动调用父类的无参构造方法，相当于 super()。
因此在编写类时，如果定义了有参构造方法，则应同时定义一个无参构造方法，以便别人断承。

构造对象的顺序：

1. 加载类
   - 静态变量
   - 静态初始化块
   - static {}
2. 判断构造函数中有没有 this()和 super(),如果有 this(),则调用本类相应的构造函数;如果有 super(),则调用父类相应的构造函数;如果两个都没有，则调用父类无参构造函数；以此递归类推。
3. 实例变量初始化。

4. 执行动态初始化块。{…….}
5. 执行本类的构造方法。

提示：

1. 为什么初始化一个子对象时必须先调用基类的构造方法？
   因为子对象内部包含着一个父对象，所以，初始化子对象前必须先初始化这个在子对象内部的父对象。父类在子类构造器可以访问它之前就已经完成了初始化。
2. 执行类的所有特定的清理动作，其顺序跟生成对象的顺序相反，通常这就要求父类元素仍旧存活。
3. 组合和继承都允许在新的类中放置子对象，组合是显式地这样做，而继承是隐式地做。组合技术通常用于想在新类中使用现有类的功能而非它的接口的这种情形。当想使用现有类，并开发一个它的特殊版本时，则使用继承。

多态

多态（迟后联编）：多态指的是运行时类型识别。当对象调用覆盖方法时，系统在编译过程中不对对象类型进行识别和方法绑定，而是在运行时才对该对象进行类型识别，并根据该对象的具体类型绑定执行相应类型中定义的覆盖方法。如果在子类里找不到相应的方法，才到父类里去打。多态之所以能体现出来，正是由于子类跟父类间发生的方法覆盖。

多态典型使用：

```
class Animal{
void eat(){};
}

class Dog extends Animal{
void eat(){};
}
class Pig extends Animal{
void eat(){};
}
Void go(Animal animal){
animal . eat();
}
Animal a=new Dog();     Pig p=new Pig();

go(a);     go(p);
```

关系运算符：instanceof

instance　instanceof　class

Animal　dog=new Dog;

if (dog1　instanceof　Dog)(这个式子的结果是一个布尔表达式)

Dog　dog2=(Dog)dog1;

instanceof 用于判定前面的对象是否属于后面的类型。是则返回 **true** ，否则返回 **false**。

如果 **dog is a Dog**，则可以对 **dog** 强制转换为 **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。

面向对象高级特性：

static 修饰符

static:①可修饰属性；②可修饰方法；③可修饰代码块。

一般情况下，只能在类中定义静态方法和属性，而不能在一个方法中定义。静态属性用来作为一个计数器。静态属性和方法都可以（最好）用类名直接调用，因为静态的东西属于类而不是属于对象，静态方法只能调用静态方法和属性。

提示：

1. main 方法为什么是 static 的？

因为 main 方法是程序的入口，在起动程序时尚未建立对象，所以 main 方法必须是 static 的才能被调用，程序才能起动.

当执行 java calssname 时，classname 自动调用静态 main 方法,相当于 classname.main()。

2. 静态方法不体现多态性,编译器只看编译时类型。

```
class A{
static void eat(){System.out.println(“this is static A.eat()”);}
}
class B extends A{
static void eat(){System.out.println(“this is static B.eat()”);}
}
class Test{
A a=new B();   a.eat();//输出为：this is static A.eat()。
}
```

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）：

static { }静态初始化块

静态代码块在代码被装载进虚拟机时被执行。一般静态代码块被用来初始化静态成员。

{ }动态初始化块。

创建对象时才执行。

3. 如何使一个类只能生成一个对象？单例模式

```
public class Test{
      public static void main(String[] args){
   A a=A.get();   A b=A.get();     System.out.print(a= =b);//true
  }
}

class A{
      static A a=new A();
      private A(){  }
public static A get(){   return a;}
```

```
class B{
private   static B b;
private B{}
public static B get(){
  if(b= =null)   b=new B(); return b;
```

}　　}

final 修饰符

final 可以修饰类、属性、方法。

当用 **final** 修饰的类不能被继承，即 **final** 类没有子类。用来确保声明的类不被修改。

当利用 **final** 修饰一个属性（变量）的时候，此属性成为常量。跟据命名规范，变量名全部大写。
当属性声明为 **final** 时，最好跟 **static** 联合在一起使用，以节省资源。
final 的属性在声明时一定要初始化，否则，就要在该类的构造函数中初始化。局部 final 变量可以随时赋值
当一个引用声明为 **final** 时，引用的值不能变，但引用所指向的对象的属性可变。

final 的方法不能被覆盖。用于在保持方法的一致性时。但可被重载。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 修饰符

abstract(抽象)可以修饰类、方法
如果将一个类设置为 abstract，此类不可生成对象，必须被继承使用。抽象类除了不能生成对象外，其它与变通类无异。抽像类继承一个类时,该类必须有无参构造方法,否则出错.
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 abstract 类的引用。
final 和 abstract 永远不会同时出现。
1－－抽象类，一般无构造方法;
2－－若抽象类有构造方法，一般声明为“protected”，供它的子类通过”super”调用。
当 abstract 用于修饰方法时，此时该方法为抽象方法，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()； 此语句表示抽象方法，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。

如果一个类继承一个抽象类，则这个类必须实现(覆盖)该抽象类的所有抽象方法。

如果有抽象方法未被实现，则子类将继承父类的抽象方法，所以这个子类也一定是抽象类。

abstract 和 **static,private,final** 同时修饰一个类或方法。

interface　　接口

接口与类属于同一层次，接口是一种特殊的抽象类。接口不能生成对象，但可以声明引用；接口没有构造方法。接口里定义的所有属性都一定是 **public static final** 的。所有方法都是
public abstract 的。

```
interface IA{
}
```

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。

1.

一个类实现一个接口的格式：

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

一个接口可以 **extends** 另一个接口。一个类实现接口时，必须实现接口中及其父类的所有的方法。
接口中声明方法时可不写 **public**，方法默认为 **public abstract** 的。但在子类中实现接口的过程中 **public** 不可省。

提示：

1. 为什么要有接口？为什么不直接用抽象类呢？
   用接口可以实现多继承；

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，但是它可以实现多个接口，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，并且接口之间允许多继承。
例：`interface IC extends IA,IB{};`

接口和多态都为 JAVA 技术的核心。

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。

可以修饰类的修饰符有：

public , abstract , final , default

private , protected 不能修饰类。

String 类

```
String str1=”nihao”;
String str2=”nihao”;
str1 = = str2   //这种创建方法两个引用指向常量域的同一个”nihao”。
String str3=new String(“nihao”);
String str4=new String(“nihao”);
```
str3!=str4

char	**charAt**(int index) Returns the char value at the specified index.
int	indexOf(int ch) Returns the index within this string of the first occurrence of the specified character.
String	substring(int beginIndex) Returns a new string that is a substring of this string.
String	substring(int beginIndex, int endIndex) Returns a new string that is a substring of this string.//beginIndex<= substring <endIndex

字符串的解析:

```
public static void main(String [] args){
    String tom=”12,152,54,1,22,96,696”;
        int i=0;
        String []s=null;
    StringTokenizer st=new StringTokenizer(tom,”,”);
    int n=st.countTokens();//获取分隔后的元素个数.
    while(st.hasMoreTokens()){//还有更多元素?
        s[i++]=st.nextToken();//获取下一个元素.
    }}
```

bject 类

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）。

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。如果本类没有覆盖 toString()，则系统自动调用其父类的 toString（）方法。可以通过子类定义 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) equals();

==判断的是变量的值是否相等.equals()方法判断的是对象的值是否相等。

用户如果想比较自定义的类的对象是否相等，就必须要在该类中覆盖 equals()方法。

```
class Student{
String name;
int age;

public String toString(){
        return “name”+name+ “age:”+age;
}
public boolean equals(Object o){
if(o==null) return false;
if(o==this) return true;
if(!(o instanceof Student)) return false;
        Student s=(Student)o;
        return name.equals(o.name)&&(age==age);
}
protected void finalize(){//全靠系统决定，我们无法确定它什么时候执行。但一定是回收对象前做的事。
System.out.print(“完了完了”);
}
}
```

自反性　　a.equals(a)=true

对称性　　a.equals(b)=b.equals(a)

传递性　　a.equals(b)=true

　　&　　b.equals(c)=true　　➔a.equals(c)=true

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。

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。不能和外部类同名.
内部类和外部类的实例变量可以共存。成员内部类不能定义静态成员。

成员内部类可以访问外部类的所有属性和方法：外部类名.this.普通属性（方法）。

外部类名.静态属性（方法）

外部类方法也可以访问内部类属性和方法：可以在外部类方法中生成一个内部类对象。

Inner in=new Inner();　　in.属性（方法）；

在别的类访问一个类中的成员内部类：Outer out=new Outer();//先取得外部类的对象

Outer.Inner in=out.new Inner();

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。）

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。

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。静态内部类可以定义静态成员。

静态内部类只能访问外部类的静态成员。外部类名.静态属性（方法）

在别的类中生成（new）一个静态内部类不需要外部类对象：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();

不能和外部类同名.但可以定义静态和非静态成员。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()};
上面这条语句会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。
③ 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。

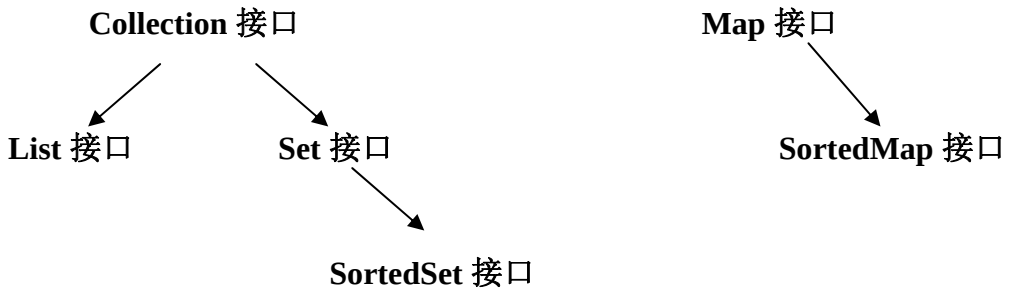
练习：写一个方法：int add(int a,int b)

```
{
   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：数组中能存基本数据类型。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。

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。

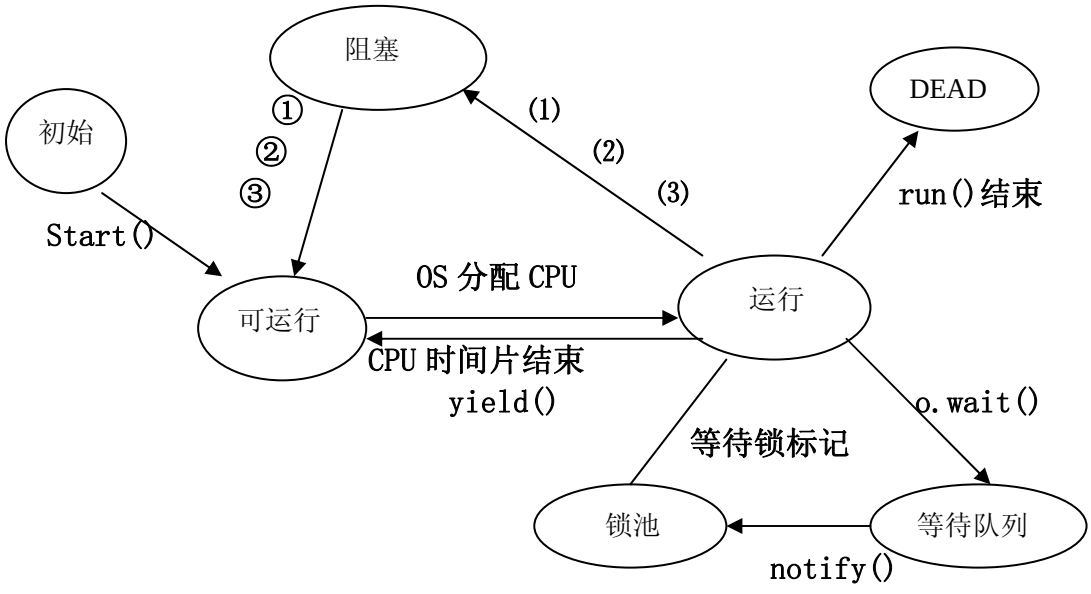
第二种方式：

```
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。
加锁时应考虑锁哪个对像?锁哪段代码?锁包含同步变量的类的对像.
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 Interrupted Exception。
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）。

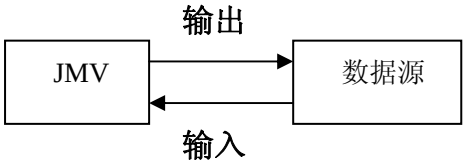
练习：
① 存车位的停开车的次序输出问题；
② 写两个线程，一个线程打印 1-52，另一个线程答应字母 A-Z。打印顺序为 12A34B56C……5152Z。通过使用线程之间的通信协调关系。
注：分别给两个对象构造一个对象 o，数字每打印两个或字母每打印一个就执行 **o.wait()**。在 **o.wait()**之前不要忘了写 **o.notify()**。

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。