1. 前言

1.1. JAVA 特点

1) 简单(Java 语法是 C++语法的一个“纯净”版本);
2) 可移植性 (一次编译到处运行)
3) 面向对象
4) 分布式(Java 把打开套接字连接等繁琐的网络任务变得非常容易)
5) 健壮性(Java 编译器会检查出很多其他语言在运行时刻才显示出来的错误; Java 采用的指针模型可以消除重写内存和数据崩溃的可能)
6) 多线程(多线程编程的简单性是 Java 成为流行的服务器端开发语言的主要原因之一)
7)安全(用 Java 可以构建防病毒和防篡改的系统)
9) 动态(Java 可随意增加新的方法以及实例变量,而客户端却不需做任何的更改)
10)体系结构中立(字节码与计算机体系结构无关，只要存在运行时系统，可在多种处理器上执行)

1.2. 运行原理

先编译 *.java 文件－－－－>*.class 文件
运行 *.class －－加载－－> JVM（JAVA 虚拟机）

1.3. JAVA 目录

JRE－－－－－－－－运行环境
SRC－－－－－－－－类库
BIN－－－－－－－－应用程序

2. 一·基础知识

2.1. 配置环境

LINUX 系统（修改环境配置文件）
1 打开 shell
2 vi .bash_profile
3 JAVA_HOME=JAVA 目录路径

4 PATH=$JAVA_HOME/bin:其他路径
5 CLASSPATH=.
6 export JAVA_HOME CLASSPATH
Windows 系统
我的电脑属性－－－>环境变量
设置环境变量：
JAVA_HOME=路径
PATH = %PATH%;c:\j2sdk1.4.2_05\bin;
CLASSPATH = .;

2.2. Java 中基本概念

1) 源文件
在最顶层只包括一个 public 类型的类/接口，文件名与类/接口名同并以.java 作为文件后缀。
2) 包(package ,在源文件中 this identify 只能放在第一行，且最多只能是一行)
一个将类和接口组织在一块的实体，在文件系统中以目录/文件夹型式呈现。

3. 二·定义，关键字和类型

3.1. 注释的三种形式

// 单行注释
/* 一或多行注释 */
/** 文档注释 */

3.2. Java 代码中的“;”、“{}”、“ ”

Java 语句以分号分隔;
Java 代码块包含在大括号内;
忽略空格。

3.3. 标识符

1) 用以命名类、方法和变量、以及包;
遵守 JAVA 的命名规范
类以每个单词都以大写字母开头。
方法和变量第一个字母不大写，其他依旧
2) 以字符、“_”或“$”开头;
3) 无长度限制。

3.4. 数据类型

1) 整型

byte	1B	8 位	-128 到 127
short	2B	16 位	-2^15 到 2^15-1
int	4B	32 位	-2^31 到 2^31-1
long	8B	64 位	-2^63 到 2^63-1

2) 浮点类型

float	4B	32 位
double	8B	64 位

3) 字符类型

char	2B	16 位

4) 布尔型

boolean	false/true

注：1) char 是无符号的 16 位整数,字面值必须用单引号括起来； ‘a’

2) String 是类，非原始数据类型；

3) 长整型数字有一个后缀为“L”或“l”，八进制前缀为“0”，十六进制前缀为“0x”；

4) 默认浮点类型为 double；

5) float 数据类型有一个后缀为“f”或“F”,Double 数据类型后可跟后缀“D”或“d“

3.5. 命名规则

1) 类/接口名首字母大写；
2) 方法、变量名第一个字母小写，其余首字母大写；
3) 常量名称全部大写；
4) 包名全部小写。

4. 三・表达式和控制流

4.1. 变量和作用域

1) 局部变量

定义在方法内部，其作用域为所在代码块，也称为临时变量、栈变量。

存在于栈中。

2) 实例变量

定义在类内部方法之外，其作用域为整个类。如未定义初值，系统会自动

为其赋默认值。存在于堆中
默认数值

类型	默认值
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000' 空格
boolean	false
*All reference types	null

4.2. 操作符

System.out.println(3/2) 按整型计算 得 1

1) >> 前面是零补零，前面是一补一;

2) >>> 无符号右移;

>>和>>>对于负数不一样
正数：>>等于除以 X 的 2 次方
负数：正数的结果＋1

3) && 短路与，前面为假，表达式为假，后面不须计算;

4) || 短路或，前面为真，表达式为真，后面不计算;

例：

if(a<3&(b=a)==0) b 赋值

if(a<3&&(b=a)==0) b 不赋值

4.3. 数字类型之间的转换

1) byte ——→ short ——→ int ——→ long

2) char ——→ int - - - → float

3) float ——→ double

4) long - - - → float

5) long - - - → double

6) int ——→ double

注：1)实箭头表示无信息损失的转换，虚箭头表示转换可能引起损失;

2)int 和 float 同为 32 位，但 float 要有几位表示幂的位数，在精度位数上要比 int 要小，所以有可能会有损失。long 转为 double 同理;

3)char 和 short 同为 16 位，但 char 属无符号数，其范围为 0~2^16, short

的范围为-2^15~2^15-1，所以 char 和 short 不能相互转换；

4)byte、short、char 属 child 型，在计算时会自动转为 int 型，然后转为更大范围类型(long、short、double)。

4.4. 强制类型转换

1) 语法：圆括号中给出要转换的目标类型，随后是待转换的变量名。例：doublc x=9.997;int nx = (int)x;

2) 如果试图强制转换的类型超出了目标类型的范围，结果是一个被截取的不同的数值;

3) 不能在布尔值和任何数字类型间强制类型转换，如确要转换，可使用条件表达式，例：b?1:0。

4.5. 转换的二种类型

1) 赋值

double a = 1.0f

int = ‘j’;

2) 方法调用

```
double converValue(int value){
    return value;
}
```

3) 数值计算转换 -9.232e20+1;

4.6. 控制流

if()
if()….else
if()…..else if()….else

```
switch(){
case variable:……..
case variable:……..
default:
……
}
```

注解：switch()内数据类型为 child 类型 byte short char 自动转换为 int
case 块中不加 break 时顺序执行下面的语句。

4.7. 循环语句

for(int i=0;i<n;i++){}

while(){}
do{} while();----------→加分号
例子：

```
loop:for(int i=0;i<n;i++){
    for(int j=0;j<m;j++){
        if(3==j){
            break  loop;//----------------→loop 为标签  只能用在循环语句中，循环
//嵌套中用于跳到外层循环
}
    }
}
```

辨析：
int x,a=6,b=7;
x=a++ + b++; //----------a=7,b=8,x=13
int x=6;x=~x;//---------- 6 的二进制 0110 取反得 11001 再转成补码（取反加一）10111 ＝ －7

5. 四·数组

5.1. 声明数组

1) 一组相同类型(可以是类)数据的集合;
2) 一个数组是一个对象;
3) 声明一个数组没有创建一个对象;
4) 数组能以下列形式声明：
int[] i 或 int i[]
Car[] c 或 Car c[]
*C++中只能 Car c[]
*JAVA 中推荐用 Car[] c;

5.2. 创建数组

1) 创建基本数据类型数组 int[] i = new int[2];
2) 创建引用数据类型数组 Car[] c = new Car[100];
3) 数组创建后有初始值。
数字类型为 0 布尔类型为 false 引用类型为 null

5.3. 初始化数组

1) 初始化、创建、和声明分开

```
int[] i;
i = new int[2];
i[0] = 0;
i[1] = 1;
```

2) 初始化、创建、和声明在同一时间

```
int[] i = {0,1};
Car[] c = {new Car(),new Car()};
```

5.4. 多维数组

1) 有效

```
int[][] i1 = new int[2][3];
int[][] i2 = new int[2][];
i2[0] = new int[2],i2[1] = new int[3];
 *C++中  int[][] =new int[][3];有效
```

2) 无效

```
int[][] i1 = new int[][3];
```

3) 数组长度 ----------→数组的属性 length

```
int[] i = new int[5];
int len = i.length;//len = 5;
Student[][] st = new Student[4][6];
len = st.length;//len = 4;
len = st[0].length;//len = 6;
```

请问以下哪段代码哪个可正确执行？(a,c)

1. a `char[] i = {'a','b'}; i = new char[]{'b','c'};`
   b `char[] i = {'a','b'}; i = {'b','c'};`
   c `char[] i = new char[2]; i = new char[]{'b','c'};`
   d `char[] i = new char[2]; i = {'b','c'};`

5.5. 数组拷贝

System.arrayCopy(Object src, int srcPos, Object dest, int destPos, int length);
拷贝一个数组到另一个数组。

6. 五·对象和类

6.1. 面向对象与面向过程

面向对象：
Everything is an Object
为什么面向对象？
符合我们看待客观世界的思维方式
各司其职 各尽所能

可重用性
可移植性
可插入性
可扩展性
弱耦合性

面向过程：先有算法，后有数据结构
面向对象：先有数据结构，后有算法

定义类
定义属性：实例变量
定义方法：声明 实现
Overload：方法名相同，参数表不同
　　编译器根据参数，选择一个方法，允许自动类型提升，就近匹配原则
定义构造方法：1.分配空间 2.初始化属性 3.调用某一个构造方法
ClassName cn=new ClassName();

对象变量：引用、句柄 存储的是对象的地址 （栈空间）

参数传递：简单类型参数：值传递 对象类型参数：地址传递

this:当前对象
　　调用本类其他的构造方法，必须放在构造方法的第一行

封装：该隐藏的隐藏，该公开的公开
属性：隐藏 private
方法：该公开的公开 public
方法声明：可以公开
方法实现：不可以公开 实现细节的改变对架构的影响最小化

6.2. 对象的概念

什么是对象：EVERYTHING IS OBJECT（万物皆对象）
所有的事物都有两个方面：
1．有什么（属性）：用来描述对象。
2．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。
*一个对象的属性也可以是一个的对象。这是一种对象的关联（associate）

```
public class Student{
private String name;---------对象
private int age;---------------基本类型
private gender;
public void study(){}---------方法
}
```

6.3. 成员变量和局部变量

1．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）
实例变量中对象默认值为 null。
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。

2．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。
局部变量不允许范围内定义两个同名变量。实例变量和局部变量允许命名冲突，但在局部变量的作用域内局部变量优先，访问实例变量是使用 this.variableName。

对于引用型类型变量
Car cart = new Car();
变量 cart 存在在栈中，而 cart 对象实际存在在堆中。

如果没有引用
New Cart();
垃圾回收会很快回收这个匿名对象。

简单的垃圾收集器原理
每个对象都包含了一个引用计数器，Garbage Collection 运行时检查对象的这个计数器，垃圾收集器会在整个对象列表中移动巡视，一旦它发现其中一个引用计数成为 0，就释放它占据的存储空间。

6.4. 成员方法

方法定义
1) 格 式 <modifiers><return_type><name>([argument_list>])[throws

<exception>]{<block>}

例如：public String getName(){ return name; }

2) 当没有返回值时，返回类型必须被定义为 void。

3) 构造方法没有返回类型。

4) 返回类型必须与方法相邻，其他修饰符号可以调换位置。

参数传递

Java 语言总是使用传值调用。这意味着方法得到的只是所有参数值的拷贝。因此，方法不能修改传递给它的任何参数变量的内容。对于对象类型的参数传递的也是该对象的引用值，方法中并不能改变对象变量，但能通过该变量调用对象的方法或修改对象的成员。

***方法与函数的区别，方法的内容是和对象的属性紧密相连的，函数只为了解决一个功能。

6.5. This 关键字

1) this 是个隐式参数，代表被构造的对象;

```
publie class Person{
    private String name;
    public void setName(String name){
this.name=name;----------→this.name 为成员变量
}
}
```

2) 如果构造器的第一个语句具有形式 this(...)，那么这个构造器将调用同一类中的其他构造器。

3)在构造器中 this()必须放在方法的第一行。

*Super 关键字也是个隐形参数，代表被构造对象的父类。

同样也必须在构造方法的第一行

6.6. 访问控制符

权限高

public	全部可见
protected	本类可见，同包可见，子类可见
default	本类可见，同包可见
private	本类可见

权限低

6.7. 构造方法

构造方法是在生成对象的过程中调用的方法，但构造方法并不能创建对象。

其特点为：

1．构造方法没有返回值。
2．构造方法的方法名与类名相同。

格式为：public ClassName(){}
构造方法也可以是其他的限制符－－private protected default private 一般用在 singleton 模式中。
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。
*接口不能创建实例，因为没有构造方法

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：----------→构造方法的重载

```
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。
如果我们提供了有参的构造方法，那么系统不会再提供无参的构造方法了。这样当被子类继承时，如果子类构造方法不人为调用父类的有参构造方法就会出现异常。

6.8. 数据和方法的隐藏－－－封装

```
    public class Person{
    private String name;----------→数据的隐藏
    private int age;
    public String getName(){----------→方法尽量公开
        return name;
}
public int getAge(){
    return age;
}
}
```

有些方法也需要隐藏起来，比如：

```
Class Singleton{
    Private Singleton(){
        ……
    }
    Public static Singleton getInstance(){
        …..
        …..
        If(…..){
```

```
            Singleton=new Singleton();
        }
        //对生产对象的条件进行限制
        Return singleton;
    }
}
```

6.9. 方法的重载

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。Overloading 被认为是编译时的多态。Overloading 只是为方便程序员编程的解决方法，编译之后的方法名实际加上了各参数类型成为唯一的名字。

普通方法
public void aa(int a,double b) throws IOException{}
private int aa(double a,int b){}
protected double aa(String a,String b){}
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 方法对于不匹配的参数是从低向高转换的。
Byte—short—float—int—long—double。

System.out.println(“%5.2f,%10.3f”, num,num2); 按格式输入

7. 六·高级语言特性

7.1. 封装 (encapsulation)

1．事物的内部实现细节隐藏起来
2．对外提供一致的公共的接口－－间接访问隐藏数据
3．可维护性

7.2. 继承 (inherit)

7.2.1. JAVA 继承特点

继承：父类的成员能否继承到子类？
子类能否访问到父类的成员

private：本类内部可以访问 不能继承到子类
（default）：本类内部可以访问，同包其他类也可 以访问
能否继承到子类？ 不一定

protected：本类内部可以访问，不同包的子类也可以访问， 同包其他类也可以访问
能继承到子类
public：任何地方都可以访问 能继承到子类

从严 到宽

覆盖：
方法名：相同
参数表：相同
访问限制符：相同或者更宽
返回值类型：相同 或者 子类返回的类型是父类返回的类型的子类

对象的构造过程：
1.递归的构造父类对象
2.分配空间
3.初始化属性
4.调用本类的某一个构造方法

super：调用父类的某一个构造方法
父类对象

多态：
1. 对象不变
2. 只能对对象调用编译时类型中定义的方法
3. 运行时，根据对象的运行时类型，找覆盖过的方法来调用（运行时动态类型判定）

强制类型转换 instanceof

屏蔽子类差异，利用父类共性做出通用编程

属性的遮盖（shadow）没有多态
方法的重载看参数的编译时类型

7.2.2. 父类（SuperClass）和 子类（SubClass）的关系

父类的非私有化属性(不同包的子类无法访问 default 修饰符)和方法可以默认继承到子类。

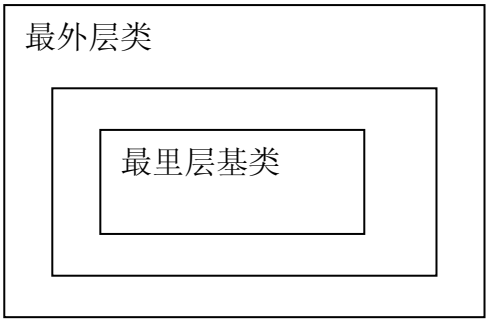
```
Class Son extends Father{
}
```

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。
所以子类构造方法默认调用父类的无参构造方法。(所以养成写无参构造的习惯)
如果子类访问父类的有参构造方法,必须在子类构造方法第一行使用 super(参数)
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。

```
Public class BMWcar extends Car{
    Public BMWcar(){
        Super(int alength); //显式的调用父类的构造，默认调用无参构造
        //所以父类没有无参构造的话，子类如果不加显示调用其他构造就会报错。这里的 super 是一个对父类的引用
    }
}
```

7.2.3. 系统构造一个对象的顺序

1 先为最里层类成员属性赋初值;
2 再构造该类对象;
3 返回外层，重复 1（上一层类）、2 步骤直到完成最外层类的构造。



super() this() 不会同时出现

```
A(){
super();
}
```

```
A(int a){
this();
}
```

7.3. 多态(polymorphism)

多态：一个对象变量可以指向多种实际类型的现象。

7.3.1. 方法的覆盖（overridding）

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。）
如果方法参数表不同，则成重载。
特点：
1．对于方法的访问限制修饰词，子类方法要比父类的访问权限更高。
父类为 public，那么子类为 private 则出现错误。
2．子类抛出的异常应该是父类抛出的异常或其子类。

7.3.2. 多态的分类

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，方法的覆盖
这样对于对象而言分为：
理解运行时多态：
Car c = new Bus();
Car 编译时类型　　编译时检查变量类型是否存在，是否有调用的方法
Bus 运行时类型　　实际运行是访问 heep 中的对象，调用实际的方法。
运行时多态是由运行时类型决定的
编译时多态是由编译时类型决定的
猫，小鸟，狗 都是动物，都可以安上动物的标签。

```
Interface Animal{}
Class Car implements Animal{}
Class Bird implements Animal{}
Class Dog implements Animal{}
```

方法中

```
Animal a = new Car();
Animal b = new Bird();
Animal c = new Dog();
```

*方法重载看的是参数的编译时类型

```
public class Animal{
    public static void main(String[] args){

    }
}
```

(1) 是覆盖吗？不能多态了

```
abstract class MyClass{
    priavate void m();
}
class Sub extends MyClass(){
    public void m();
}
```

(2) 错误的修饰符组合

```
abstract class MyClass{
    priavate abstract void m();
}
class Sub extends MyClass(){
    public void m();
}
```

(3) 5.0 新 非覆盖

```
abstract class MyClass{
    private final void m();
}
class Sub extends MyClass(){
    public void m();
}
```

7.3.3. 运行时多态的三原则

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.动态类型判定实际调用的方法。即它调用覆盖后的方法。

7.3.4. 关系运算符：instanceof

instanceof Animal;(这个式子的结果是一个布尔表达式)
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，
用于判定前面的对象是否是后边的类或者子类。

```
Animal a = new Car();
If(a instanceof Dog){
Dog b =(Dog)a;
}
else if(a instanceof Car){
    Car c =(Car)a
}
```

不会错。

7.4. 静态变量，方法和类

静态变量
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。
例：

```
Class M{
static int data;
}
M m1=new M(); M m2=new M();
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。
其变量可以通过类名去访问：类名.变量名。与通过对象引用访问变量是等价的。

2) 静态方法

```
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。方法中 this.data 和 super.data 是不可用的。
原因：从根本来说，是静态变量不管类是否实例化都会存在，而实例变量只有类实例化了才存在。直接调用静态方法时并不确定实例变量是否存在。

一般情况下，主方法是静态方法，所以 JVM 可以直接调用它，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。
猜想：JVM 在代码中有这样的语句：
ClassName.main(arg); ClassName 通过命令行的”java 类名”取得，所以类名不用加.class 扩展名

*覆盖不适用于静态方法。
静态方法不可被覆盖。
如果子类中有和父类重名的静态方法，虽然编译通过，但它并不能实现多态，所以不能称作覆盖。

```
public class    Test {
    public static void main(String[] arg) {
            Super s = new Sub();
            s.show();

    }
}
class Super
{
     static public void show(){System.out.println("in Super");}
}
class Sub extends Super
{
     static public void show(){System.out.println("in Sub");}
}
```

执行结果是： in Super

3) 静态内部类----→只能是成员内部类

```
class Out{
    public static class Inner{}
}
```

4) 初始化块
1. 只被执行一次;
2. 初始化块在类被加载后首先被运行，不管类是否实例化
3. 一般用来初始化静态变量

```
    Public static void main(String[] args){
System.out.println(Car.name);//这时加载 Car Class 进入 JVM 并执行静态代//码块
    }
```

7.5. Singleton 模式

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。
特点：
1．有一个静态属性
2．私有的构造－－－singleton 不能 new
3．公共的静态方法来得到静态属性
实现单例模式的原理：
　　利用类属性（静态变量）在系统中唯一的特性，建立这样一个唯一的引用并控制这个引用所指的空间是不变的。

```
public class ConnectionFactory{
    private static Connection conn;
private Connection(){
        if(conn==null)
        conn = new Connction();
    }
    public Connection getInstance(){
        return conn;
}
}
```

实现 2

```
public class ConnectionFactory{
    private static Connection conn;
static{
        conn = new Connection();
    }
    public static Connection getInstance(){
        return conn;
    }
}
```

7.6. final 关键字

7.6.1. final 变量不能被改变;

当利用 final 修饰一个属性（变量）的时候，此时的属性成为常量。
注意 JAVA 命名规范中常量全部字母大写：
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。
在 JAVA 中利用 public static final 的组合方式对常量进行标识（固定格式）。

Final 变量是在整个类被创建时候被赋值，之后就不能改变了。
对于 final 变量，如果在声明的时候和构造的时候均不进行赋值，编译出错。

对于利用构造方法对 final 变量进行赋值的时候，此时在构造之前系统设置的默认值被覆盖。

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。

```
Class A{
    Final int x=3;
    Public A(){
        x=4;
    }
}
```

7.6.2. final 方法不能被改写;

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类对于子类是否可见。
Final 和 abstract 永远不会同时出现。

7.6.3. final 类不能被继承;

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。
String 类数据 final 类 ，目的是提供效率保证安全。

7.6.4. String 类

String 的声明：public final class String 无法继承，强不变模式
字符串池－－－－－－－－－－－－－－－－>池化思想 数据库连接池，EJB 池

```
public class TestString {
```

```
    public static void main(String[] args){
        String s1=new String("abc");
        String s2=s1;
        s1+="d";
        System.out.println( s1 );
        System.out.println( s1==s2 );
    }
}
```

便于实例重用
不要轻易在 heap 里创建空间
intern()　返回池地址

对于字符串连接

str=”123”+”456”+”789”＋“123”;
产生：
123456
123456789
123456789123
产生多余对象

应该使用 StringBuffer(线程安全的) 或者 StringBuilder（线程不安全的）

```
String str="hello:nihao:happy";
StringTokenizer st=new StringTokenizer(s,":");
while(st.hasMoreTokens()){
    String str=st.nextToken();
    System.out.println(str);
}
```

7.7. 抽象类

1) Abstract(抽象)可以修饰类、方法
如果将一个类声明为 abstract，此类不能生成对象，只能被继承使用。
Abstract 类的设计是将子类的共性最大限度的抽取出来，以提高程序的统一性。

2) 一个类中包含有抽象方法必须声明为抽象类;
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。

3) 抽象类不能实例化，但仍可以声明;
Abstract 类可以作为编译时类型，但不能作为运行时类型。

4) 子类继承抽象类必须实现其中抽象方法
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

7.8. 接口 (模板方法模式)

1) 接口是抽象类的另外一种形式(没有实例变量的抽象类);
2) 在一个接口中所有方法都是抽象方法;
3) 接口中所有变量都必须被定义为 final static；
4) 接口可以继承多个接口。
5) 可插入性的保障－－工程模式
6) 是规范的制订者和规范的实现者分开。－－JDBC
注：1) 接口中的方法自动被置为 public, 因经，在接口中声明方法并不需要提供 public 关键字。但在实现接口时，必须把方法声明为 public。

接口与抽象类的区别：

接口	抽象类
无实现方法	可以有非抽象方法－－实现代码
可以把子类的共有代码提取出来放在抽象类中	

在 JAVA 中的接口分类：
普通 -----有方法和属性
常量 -----存常量
标记 -----没有方法，属性，只为了做编译类型的标记
实例：
简单工厂模式（接口和多态的实际使用）

```
public interface Car {
    public void brake();
    public void grade();
    public void engineer();

}
public class BMW implements Car{
    private String name;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
```

```
    }
    public void brake() {
        // TODO Auto-generated method stub
        System.out.println("BMWcar's brake()");
    }
    public void engineer() {
        // TODO Auto-generated method stub
        System.out.println("BMWcar's engieer()");
    }
    public void grade() {
        // TODO Auto-generated method stub
        System.out.println("BMWcar's greade()");
    }
    /**
     * @param args
     */
}
Public class Banz implements Car{
    ……
}
Public class Factory{
    Public static Car factory(String name){
        If(name.equals(“BMW”){
            Return new BMWCar(); //返回一个引用
        }
        If(name.equals(“Banz”){
            Return new BanzCar();
        }
    }
}
```

注意：

1 这个问题不是对象有没有 toString 方法的问题,而是用接口作为编译时类型能否调用的问题.

2 接口就是接口,尽管本质上是抽象类,但是总归不是类,接口没有父类,Object 类绝不是接口的父类,这种理解大错而特错!!!

3.我们来看看虚拟机规范是怎么说的

JVM 会解析类型为 CONSTANT_InterfaceMethodref_info 的常量池的入口,会按照如下步骤执行接口方法解析:

1) 检查接口是否具有用户调用的方法
2) 检查接口的父接口是否具有用户调用的方法
3) 检查 java.lang.Object 是否具有用户调用的方法
4) 抛出 NoSuchMethodException

因此,不是接口类型有 toString 方法,用反射是看不到这个方法的,而是虚拟机在作方法调用连接时,会自动去找 Object 类中的方法.虚拟机就是这么工作的.

所以,对以接口作为编译时类型的方法调用,根据以上说的第三点,Object 类中的方法都可以得到调用

7.9. Object 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）。
此类中的方法可以使所的类均继承。

以下介绍的三种属于 Object 的方法:
(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 B=new String(“hello”);
A==B(此时程序返回为 FALSE)
因为此时 AB 中存的是不同的对象引用。
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。
因为 Student 继承的是 Object 的 equals()方法，此时 toString()等于＝＝
为了实现对象的比较需要覆盖 equals（加上这个定义，返回 ture 或 false）
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;
}
```

7.10. 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。

注：
“==”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。
封装类・字符串・基本类型

```
Interger--------------------(Double(a.toString))------------>Double
String -----------------(Integer.valueOf() )------------------>Integer
Integer-----------------(String.valueOf() )-------------------> String
Int----------------------(100+””)------------------------------>String
String------------------(Integer.parseInt() )-----------------><int
Integer-----------------(Integer.intValue() )-----------------><int
```

7.11. 内部类

（注：所有使用内部类的地方都可以不用内部类，但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。
*内部类可为静态，可用 PROTECTED 和 PRIVATE 修饰。（而外部类不可以：顶级类只能使用 PUBLIC 和 DEFAULT）。
*JAVA 文件中没有 publie class 可以类名和文件不同名。

7.11.1. 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）。

7.11.2. 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。
在外部类的外部访问内部类，使用 out.inner.

成员内部类的特点：
1．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。）
2．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。
3．成员内部类不能有静态属性

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 o=new Outer();
Outer.Inner in=o.new.Inner()。

7.11.3. 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，但可以访问外部类的局部常量
在类外不可直接访问局部内部类（保证局部内部类对外是不可见的）。
在方法中才能调用其局部内部类。

7.11.4. 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。

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。
例子：
对于两个类，拥有相同的方法：

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

此时有一个 robot 类：

```
class Robot extends People implement Machine.
```

此时 run()不可直接实现。

```
interface Machine
{
      void run();
}
class Person
```

```
{
    void run(){System.out.println("run");}
}
class Robot extends Person
{
    private class MachineHeart implements Machine
    {
        public void run(){System.out.println("heart run");}
    }
    public void run(){System.out.println("Robot run");}
    Machine getMachine(){return new MachineHeart();}
}
class Test
{
    public static void main(String[] args)
    {
        Robot robot=new Robot();
        Machine m=robot.getMachine();
        m.run();
        robot.run();
    }
}
```

7.11.5. 匿名内部类

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。

7.12. 集合

集合对象：用于管理其他若干对象的对象
数组：长度不可变

List: 有顺序的，元素可以重复
  遍历：for 迭代

排序：Comparable Comparator Collections.sort()
ArrayList：底层用数组实现的 List
特点：查询效率高，增删效率低 轻量级 线程不安全
LinkedList：底层用双向循环链表 实现的 List
特点：查询效率低，增删效率高
Vector: 底层用数组实现 List 接口的另一个类
特点：重量级，占据更多的系统开销 线程安全

Set：无顺序的，元素不可重复（值不相同）
遍历：迭代
排序：SortedSet
HashSet：采用哈希算法来实现 Set 接口
唯一性保证：重复对象 equals 方法返回为 true
重复对象 hashCode 方法返回相同的整数
不同对象 哈希码 尽量保证不同（提高效率）

SortedSet：对一个 Set 排序
TreeSet：在元素添加的同时，进行排序。也要给出排序规则
唯一性保证：根据排序规则，compareTo 方法返回为 0，就可以认定两个对象中有一个是重复对象。

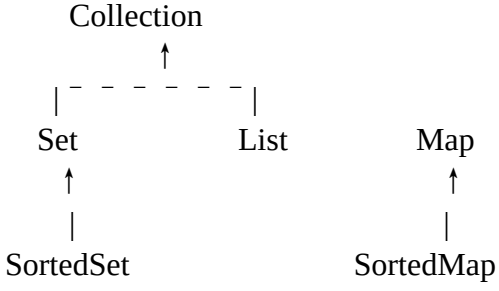
Map：元素是键值对 key：唯一，不可重复 value：可重复
遍历：先迭代遍历 key 的集合，再根据 key 得到 value
HashMap:轻量级 线程不安全 允许 key 或者 value 是 null
Hashtable：重量级 线程安全 不允许 key 或者 value 是 null
Properties：Hashtable 的子类，key 和 value 都是 String

SortedMap：元素自动对 key 排序
TreeMap：

集合是指一个对象可以容纳了多个对象（不是引用），这个集合对象主要用来管理维护一系列相似的对象。

7.12.1. 集合接口类层次

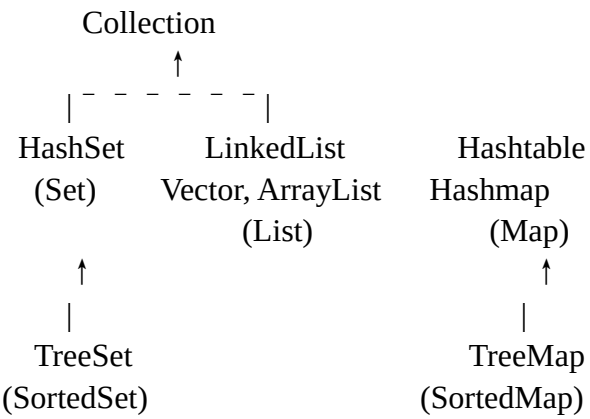
位于 package java.util.*;



1) Set: 集合类中不允许有重复对象;
2) SortedSet: 和 Set 接口同，但元素按升序排列;
3) List: 元素加载和移出时按照顺序，可以保存重复对象。
4) Map: (key-value 对)存储了唯一关键字辨识和对应的值。
5) SortedMap: 和 Map 类同，但对象按他们关键字的升序排列。

7.12.2. 集合类层次

（注：JAVA1.5 对 JAVA1.4 的最大改进就是增加了对范型的支持）



Collection 接口的方法：

add(Object o)
addAll(Collection c)
contains(Object o)
containsAll(Collection c)
remove(Object o)
removeAll(Collection c)
clear()
equals(Object o)
isEmpty()
iterator()
size()
toArray()
toArray(Object[] o)

7.12.3. 五个最常用的集合类之间的区别和联系

1．ArrayList: 元素单个，效率高，多用于查询
2．Vector: 元素单个，线程安全，多用于查询

3．LinkedList:元素单个，多用于插入和删除
4．HashMap: 元素成对，元素可为空
5．HashTable: 元素成对，线程安全，元素不可为空

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。
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。
栈：后进先出的数据结构。
注意：使用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。

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 finalize()，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 类时要注意，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。

7.12.4. 比较

Collections 类（工具类－－－全是 static 方法）

```
Public static int binarySearch(List list,Object key)
Public static void Sort(List list,Comparator com)
Public static void sort(List list)
```

方法一：
Comparator 接口

```
Int compare(Object a,Object b)
Boolean equals(Object o)
```

例子：

```
import java.util.*;
public class    Test {
    public static void main(String[] arg) {
            ArrayList al = new ArrayList();
            Person p1 = new Person("dudi");
            Person p2 = new Person("cony");
            Person p3 = new Person("aihao");
            al.add(p1);
            al.add(p2);
            al.add(p3);
            Collections.sort(al,p1);
            for(Iterator it = al.iterator();it.hasNext();){
                Person p = (Person)it.next();
                System.out.println(p.name);
            }
    }
}
class Person implements java.util.Comparator
{
     public String name;
     public Person(String name){
          this.name = name;
     }
     public int compare(Object a,Object b){
          if(a instanceof Person&&b instanceof Person){
               Person pa = (Person)a;
               Person pb = (Person)b;
               return pa.name.compareTo(pb.name);
          }
          return 0;
     }
     public boolean equals(Object a){return true;}
}
```

方法二

```
Java.lang.Comparable
Public int compareTo(Object o)
Class Person implements java.lang.Comparable{
    Public int compareTo(Object o){
        Comparable c1=(Comparable)this;
        Comparable c2=(Comparable)o;
        Return    c1.name.compareTo(c2.name );
}
    }
    …….
```

}
Main 方法中
Collections.sort(list);
*注意：程序员和类库之间是平等的关系，而不是上下级关系，以前的类可以是通过接口调用后来的类 。
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。

7.13. 反射

用于工具，架构，动态开发等开发工程
三种得到类对象的途径：
Class.forName(“name”) //输入全类名
object.getClass() //得到该对象的类对象
object.class

无参构造一个对象
newInstance()
有参的构造一个对象
Constructor con=c.getConstructor(String.class)
Object o=con.newInstance(“liucy”);

调用对象方法
Class newClass=Student.class; //得到一个类对象
Object o = newClass.newInstance(); //产生对象
Method newMethod = newClass.getDeclaredMethod("study");//得到这个类的方法类
newMethod.invoke(o);//调用方法，需要提供是哪个对象调用和参数

1) 确定一个对象的类;
2) 获得一个类的修改符、变量、方法、构器函数、和父类的相类信息;
3) 找出哪些常量和方法是从一个接口声明的;
4) 创建一个在运行时才知道名称的类;
5) 调用对象的方法;

8. 七・异常

8.1. 异常的基本概念

1) 异常事件改变程序流程;

2) 当一个异常事件发生时，一个异常被抛出;
3) 响应处理异常的代码被称为 exception handler;
4) exception handler 捕获异常;
5) 异常处理能让你集中精力在一个地方解决问题，然后将处理错误的代码分开来放在另一个地方。

8.2. 捕获异常

1) 设置一个 try/catch 的代码块;
2) 如果 try 块内的任何代码抛出了由 catch 子句指定的异常，则
   a. 程序跳过 try 块中的其他代码;
   b. 程序执行 catch 从句中的处理器代码。
3) 如 try 块内没有抛出异常，直接跳过 catch 从句内容。
4) 如 try 块内抛出的异常没有在 catch 从句中指定, 则该方法会立即退出。

8.3. 处理异常

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。
对已经查出来的例外，有 throw(消极)和 try catch（积极）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。
3) 对子类方法抛出的异常不能超出父类方法 throws 指令的范围。如父类方法不抛出任何异常，在子类方法中必须捕捉每一个“已检查异常”。

8.4. 捕捉多个异常

1) 每个异常类型使用一个 catch 从句;
2) 如前面 catch 从句捕获异常，将直接跳过后面 catch 从句内容;
3) 建议按异常类型的子类->超类的顺序排列 catch 从句的先后顺序。

8.5. finally 声明

无论是否捕获异常，都会执行 finally 从句中的代码;
例子：

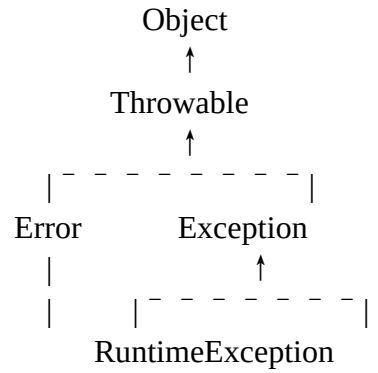
finally{ con.close();}

8.6. 异常调用栈

1. 哪个方法调用的代码发生异常，返回到调用方法的地方;
2. main 方法调用的代码发生异常，返回到虚拟机。

8.7. 异常层次

答：1) 起源于 Error 的类代表不常用的环境(通常是硬件层面);
2) 应用程序不能够从 Error 中恢复正常;
3) 所有的 Java 异常都起源于 Exception;
4) RuntimeExcepiton 也称为未检查异常;
5) 未检查异常无须捕获;
6) 其它异常，也称为检查异常，必须处理



8.8. 一些未检查的异常

答：1) java.lang.ArithmeticException 如：除 0;
2) java.lang.NullPointerException 如：没初始化一个 References 便使用;
3) java.lang.ArrayIndexoutofBoundsException 如：调用一个有十个元素的 Array 的第十一个元素的内容;
4) java.lang.NumberFORMatException 如：Integer.parseInt("a");

8.9. 写你自己的异常

答：1) 要做的仅仅是从 Exception 继承或者是从 Exception 的一个子类衍生出自己需要的类就可;

2) 习惯为每一个异常类提供一个默认的构造器以及一个包含详细信息的构造器。

8.10. 抛出你自己的异常

答：1) 在方法的定义中增加一个 throws 修饰符，以通知调用者可能会抛出一个异常;
2) 在方法中构造一个该类的实例，然后抛出该实例。

9. 八・图形用户接口

构造 GUI 的步骤
1.选择容器
2.设置布局管理器
3.添加组件
4.设置事件监听

布局管理器
1. FlowLayout 流式布局 Panel 的默认
2. BorderLayout 东西南北中 Frame 的默认
3. GridLayout 网格布局
4. CardLayout 卡片布局
5. GridBagLayout 复杂的网格布局

JButton ：按钮
JTextField：单行文本域
JTextArea：多行文本区
JScrollPane：滚动窗体
JComboBox：下拉选择框

JRadioButton:单选按钮
JCheckBox：多选按钮
JList:多行列表
JLabel：标签
JPasswordField：密码输入框
JEditorPane：显示结构化文档
Border:边框

JMenuBar：菜单条
JMenu：菜单
JMenuItem：菜单项

JPopupMenu：弹出式菜单

JSlider：滑动条
JProgressBar：进度条
JTabbedPane：分层面板
JSplitPane：分隔面板
JToolBar：工具条

JFileChooser：文件选择器
JColorChooser：颜色选择器

显示对话框
JoptionPane 里面有很多静态方法可以弹出对话框

10. 九・AWT（Abstract Window Toolkit） 事件模型

Jbutton ---->
ActionListener ----------> getAcitonCommand()
Window -------->

11. 十・The AWT Component Library

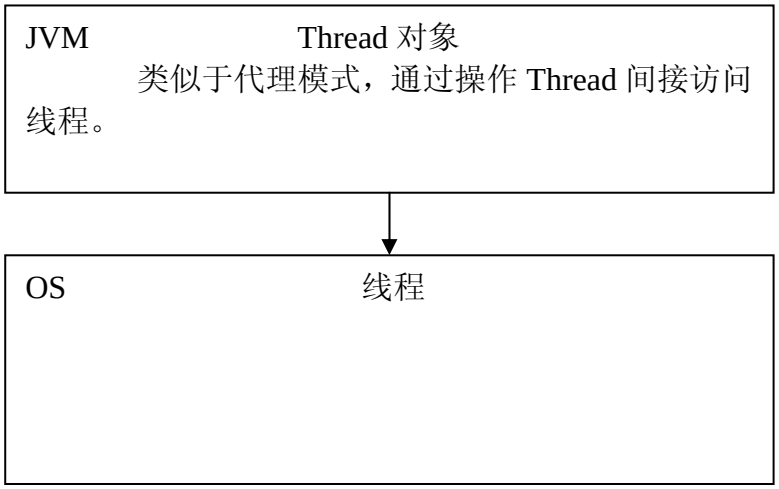
12. 十一・JFC（Java Foundation Classes）

13. 十二・Applets

14. 十三・线程 Thread

14.1. 线程原理

进程是数据独占的
线程是数据共享的（所以需要处理数据并发）
并发原理：宏观并行，微观串行
OS 将一段时间分为多个时间片，每个时间片 CPU 只能运行一个任务。



注意： 凡是访问 JVM 外部的资源时，都不能直接访问的，都需要一个代理的对象。

14.2. 线程实现的两种形式

继承 java.lang.Thread：

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

启动线程

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

实现 java.lang.Runnable 接口：

```
Class MyThread    implements Runnable{
   Public    void run(){

   }
}
```

这种实现可以再继承其他类。

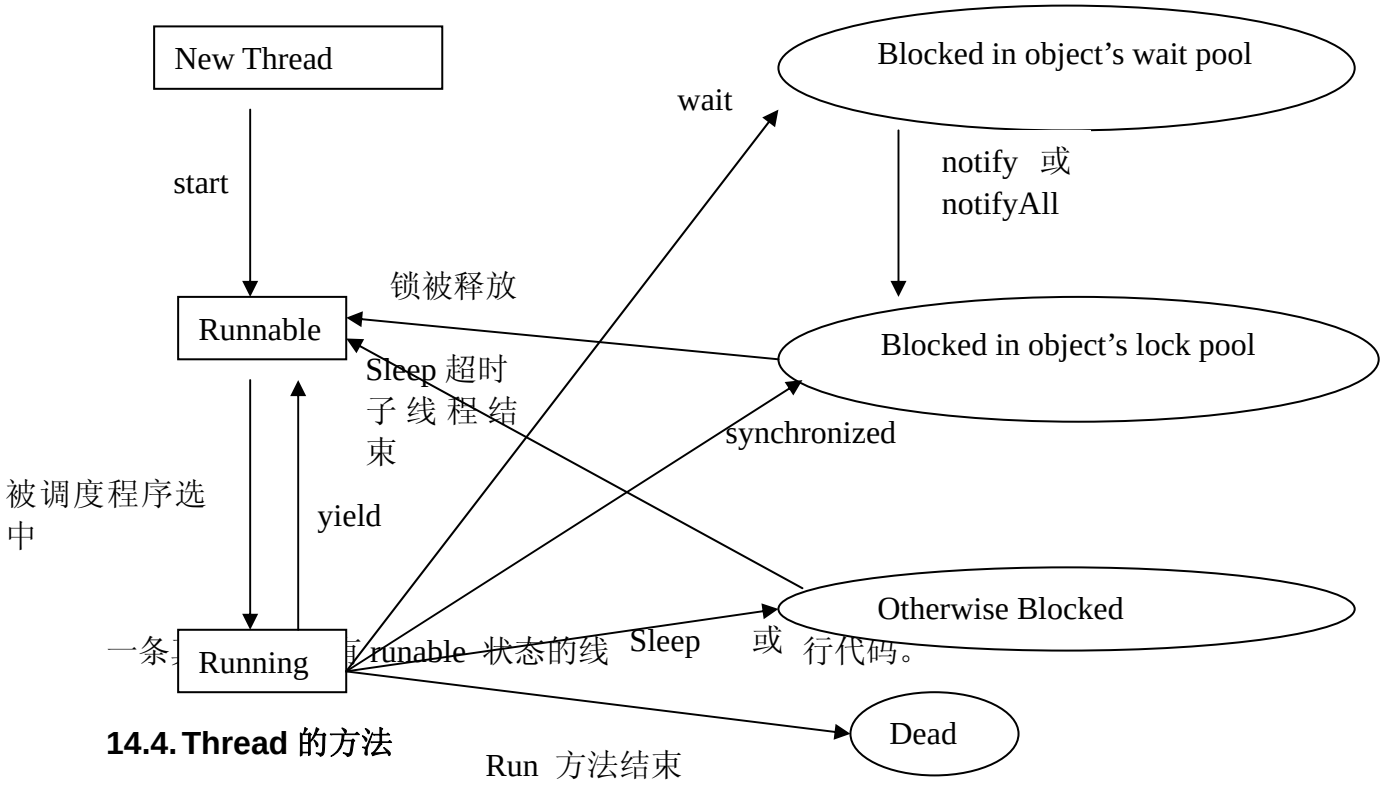
启动线程时不同前者

```
public static void main(){
     Runnable myThread = new MyThread();
     Thread t = new Thread(myThread);
     t.start();
}
```

当调用 start 方法时，JVM 会到 OS 中产生一个线程。

14.3. 线程的生命周期

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“等待池”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“等待池”单独分离出来有利于对程序的理解）



14.4. Thread 的方法

public static void sleep(long millis)
throws InterruptedException
括号中以毫秒为单位，使线程停止一段时间，间隔期满后，线程不一定立即恢复执行。
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(InterruptedException e){e.printStackTrace(e);}

Public final void join() throws InterruptedException
表示其他运行线程放弃执行权，进入阻塞状态，直到调用线程结束。
实际上是把并发的线程变为串行运行。
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。

Public static void field()
使当前线程马上交出执行权，回到可运行状态，等待 OS 的再次调用。

Public final Boolean isActive()
验证当前线程是否是活动的，不管它是否正在运行。

14.5. 共享数据的并发处理

两个线程修改共享资源时会出现数据的不一致，为避免这种现象采用对访问的线程做限制的方法。利用每个对象都有一个 monitor(锁标记)，当线程拥有这个锁标记时才能访问这个资源，没有锁标记便进入锁池。

1．Synchronized 修饰代码块

```
public void push(char c){
        synchronized(this){
        ...
        }
}
```

对括号内的对象加锁，只有拿到锁标记的对象才能执行该代码块

2．Synchronized 修饰方法

```
public synchronized void push(char c) {
    ...
    }
```

对当前对象的加锁，只有拿到锁标记的对象才能执行该方法
注：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阻塞进入锁池（lock pool）。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。由系统决定哪个线程拿到锁标记并运行。

14.6. 使用互斥锁的注意事项

举例：男孩和女孩例子，每个女孩是一个对象，每个男孩是个线程。每个女孩都有自己的锁池。每个男孩可能在锁池里等待。

```
Class Girl{
    Public void hand(){

    }
    Public syncronized void kiss(){

    }
}
Class Boy extends Thread{
```

```
    Public void run(){

    }
}
```

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。

死锁的两种处理方法

统一排列锁顺序(解决不同方法中对多个共享资源的访问)

对象 1 的方法

synchronized(a)

synchronized(b)

对象 2 的方法

synchronized(a)

synchronized(b)

2．线程间通信(也就是线程间的相互协调)

线程间通信使用的空间称之为对象的等待池（wait pool），该队列也是属于对象的空间的。

进入等待池

使用 Object 类中 wait()的方法，在运行状态中，线程调用 wait()，此时表示线程将释放自己所有的锁标记和 CPU 的占用，同时进入这个对象的等待池。等待池的状态也是阻塞状态，只不过线程释放自己的锁标记。

退出等待池进入锁池

notify()：将从对象的等待池中移走一个任意的线程，并放到锁池中，那里的对象一直在等待，直到可以获得对象的锁标记。

notifyAll(): 将从等待池中移走所有等待那个对象的线程并放到锁池中，只有锁池中的线程能获取对象的锁标记，锁标记允许线程从上次因调用 wait()而中断的地方开始继续运行

注意：只能对加锁的资源进行 wait()和 notify()。

1) wait()：交出锁和 CPU 的占用;

2) notify()：将从对象的等待池中移走一个任意的线程，并放到锁池中，那里的对象一直在等待，直到可以获得对象的锁标记。

3) notifyAll(): 将从等待池中移走所有等待那个对象的线程并放到锁池中，只有锁池中的线程能获取对象的锁标记，锁标记允许线程从上次因调用 wait()而中断的地方开始继续运行

注：在 java.io 包中 Vector 和 HashTable 之所以是线程安全的，是因为每个方法都有 synchronized 修饰。Static 方法可以加 synchronized，锁的是类对象。

但是 Vector 是 jdk 1.0 的 ArrayList 是 jdk1.2 所以实际应用还是使用 ArrayList

例子：

生产者和消费者

一个柜台一定数量的产品,柜台满时不能生产，空时不能够买。

15. 十四・标准 I/O 流与文件

15.1. 对文件的操作

1．File 类（java.io.File）可表示文件或者目录（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）。
File 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

2．File 的方法
Boolean createNewFile() //创建文件
Boolean mkdir() //创建目录
Boolean mkdirs() //创建多个目录
Boolean delete() //删除文件
Boolean 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。
String[] List()：返回当前 File 对象下所以显文件和目录名（相对路径）
File[] ListFiles()：返回当前 File 对象所有 Files 对象，可以用 getName()来访问到文件名。
isDirectory()和 isFile()来判断究竟是目录还是文件。
String getParent() 得到父类文件名
File getParentFile() 。。。
String getPath() 。。。路径
exists() 判断文件是否存在

15.2. 处理跨平台性

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 的文件系统的分隔符是不相同。
使用 file 类的 separtor 属性，返回当前平台文件分隔符。

```
File newD = new File("aa"+File.separator+"bb"+File.separator+"cc");
        File newF = new File(newD,"mudi.txt");
        try{
        newD.mkdirs();
```

```
newF.createNewFile();
}catch(Exception e){}
```

15.3. 对象的序列化接口

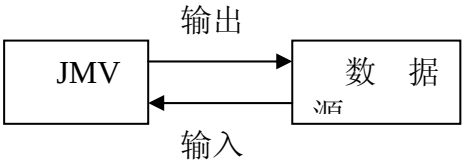
Serializable 接口没有方法，是标识接口。
序列化的步骤：

1）实现 Serializable 接口
2）实例化对象文件输出对象
3）将对象输出到文件里
4）有些临时变量生命周期不需要超过虚拟机的生命周期，需要加上：transient 关键字，这个属性不被序列化。

一个序列化对象内部属性的对象也需要序列化接口。

15.4. I/O 流基础

Input/Output：指跨越出了 JVM 的边界，与外界进行数据交换。



注意：输入/输出是针对 JVM 而言。

15.5. 流的分类

1) 从数据类型分：字节流和字符流

字节流类：
抽象父类： InputStream,OutputStream
实现类：

BufferedInputStream 缓冲流－过虑流
BufferedOutputStream
ByteArrayInputStream 字节数组流－节点流
ByteArrayOutputStream
DataInputStream 处理 JAVA 标准数据流－过虑流
DataOutputStream
FileInputStream 处理文件 IO 流－节点流
FileOutputStream
FilterInputStream 实现过虑流－字节过虑流父类
FilterOutputStream
PipedInputStream 管道流

PipedOutputStream
PrintStream 包含 print() 和 println()
RandomAccessFile 支持随机文件

抽象父类：Reader, Writer
实现类：

BufferedReader
BufferedWriter
PrintWriter
CharArrayReader
CharArrayWriter
FileReader
FileWriter
FilterReader
FilterWriter
InputStreamReader
OutputStreamWriter
PipedReader
PipedWriter
StringReader
StringWriter

2) 从数据方向分：输入流和输出流
InputXXXXX ， OutputXXXXX
3) 从流的功能分：节点流和过滤流（使用到油漆工模式）
节点流用来传输数据。
过滤流用来封装节点流或者其他过滤流，从而给节点流或其他的过滤流增加一个功能。

15.6. I/O 输入输出

流的标准写法：

```
OutputStream os=null;
    OutputStream os=null;
    try{
        String a="hello";
        byte[] b=a.getBytes();
        os=new FileOutputStream("D:\\aa.txt");
        os.write(b);
    }catch(IOException ioe){
        ioe.printStackTrace();
    }finally{
        if(os!=null){
```

```
                try {
                    os.close();
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }
```

1. InputStream 类

所有字节输入流的父类，如：FileInputStream,ObjectInputStream,PipedInputStrean

1) 三个基本的 read()方法

a. int read(): 从流里读出的一个字节或者-1; (实际读了多长）

b. int read(byte[])：将数据读入到字节数组中，并返回所读的字节数; （期望读了多长）

c. int read(byte[], int , int)：两个 int 参数指定了所要填入的数组的子范围。

2) 其它方法

a. void close(): 关闭流，如使用过滤器流，关闭栈顶部的流，会关闭其余的流。

b. int available(): 返回可从流中读取的字节数。

c. skip(long): 丢弃了流中指定数目的字符。

d. boolean markSupported()

e. void mark(int)

f. void rese()

2. OutputStream 方法

答：1) 三个基本的 read()方法

a. void write():

b. void write(byte[])：

c. void write(byte[], int , int)：

写输出流。

2) 其它方法

a. void close(): 关闭流，如使用过滤器流，关闭栈顶部的流，会关闭其余的流。

b. void flush(): 允许你强制执行写操作。

注：在流中 close()方法由程序员控制。因为输入输出流已经超越了 JVM 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。

3. FileInputStream 和 FileOutputStream

答：1) 结点流，使用磁盘文件。

2) 要构造一个 FileInputStream, 所关联的文件必须存在而且是可读的。

3) 要构造一个 FileOutputStream 而输出文件已经存在，则它将被覆盖。

```
FileInputStream infile = new FileInputStream("myfile.dat");
FIleOutputStream outfile = new FileOutputStream("results.dat");
```

FileOutputStream outfile = new FileOutputStream(“results.dat”,true);

参数为 true 时输出为添加，为 false 时为覆盖。

FileOutputStream 类代码：(为什么能建文件)

```
Public FileOutputStream(String name){
    This(name!=null？ new File(String):null,false);
}
```

键盘流

PrintWriter : System.in

4．DataInputStream 和 DataOutputStream

为过滤流。通过流来读写 Java 基本类,注意 DataInputStream 和 DataOutputStream 的方法 是成对的。

过滤流。输出输入各种数据类型。

writeBoolean(boolean b) ------以 1bit 数据传送

writeByte(int) ------以 1 byte 数据传送

writeBytes(String s) --------以 byte 序列数据传送

writeChar(int v) －－－－－－以 2 byte

writeChars(String s)----------以 2 byte 序列

writeDouble(double d) -------以 8 byte

writeInt(int v)

writeLong(long l)

writeShort(short s)

writeUTF(String)----------能输出中文！

6．ObjectInputStream 和 ObjectOutputStream

过滤流。处理对象的持久化

```
Object o = new Object();
FileOutputStream fos=new FileOutputStream("Object.txt");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(o);
oos.close();
FileInputStream fis =new FileInputStream(“Object.txt”);
ObjectInputStream ois =new ObjectInputStream(fis);
Object o = (Object)Ois.readObject();
ois.close();
```

7. BufferInputStream 和 BufferOutputStream

过滤流，可以提高 I/O 操作的效率

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。

8. PipedInputStream 和 PipedOutputStream

用来在线程间通信.

```
PipedOutputStream pos=new PipedOutputStream();
    PipedInputStream pis=new PipedInputStream();
    try
    {
        pos.connect(pis);
        new Producer(pos).start();
        new Consumer(pis).start();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
```

9.RandomAccessFile 随机访问

可以得到文件指针。

long getFilePointer() 得到从文件开始处到文件指针的位置。

seek(long point) 将文件指针移动到此处。

10 Reader 和 Writer

1) Java 技术使用 Unicode 来表示字符串和字符，而且提供 16 位版本的流，以便用类似的方法处理字符。

2) InputStreamReader 和 OutputStreamWriter 作为字节流与字符流中的接口。

3) 如果构造了一个连接到流的 Reader 和 Writer，转换规则会在使用缺省平台所定义的字节编码和 Unicode 之间切换。

4) 字节流与字符流的区别：

编码是把字符转换成数字存储到计算机中。把数字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。

10

都是过滤流。

BufferedReader 的方法：readLine():String

PrintWriter 的方法：println(….String,Object 等等)和 write()

11. 随机存取文件

1) 实现了二个接口：DataInput 和 DataOutput;

2) 只要文件能打开就能读写;

3) 通过文件指针能读写文件指定位置;

4) 可以访问在 DataInputStream 和 DataOutputStream 中所有的 read()和 write()

操作;

5) 在文件中移动方法:

a. long getFilePointer(): 返回文件指针的当前位置。

b. void seek(long pos): 设置文件指针到给定的绝对位置。

c. long length(): 返回文件的长度。

12．编码问题：

编码的方式：

每个字符对应一个整数。

不同的国家有不同的编码，当编码方式和解码方式不统一时，产生乱码。

因为美国最早发展软件，所以每种的编码都向上兼容 ASCII 所以英文没有乱码。

ISO-8859-1 西方字符

GB2312 。。。。。。镕

GBK

Big5

Unicode

UTF-8

补充：

字节流结束返回-1

字符流结束返回 null

对象流结束返回 EOFException

引申---------〉异常经常被用在流程控制， 异常也是方法的一种返回形式。

16. 十五・网络编程

16.1. 网络基础知识

网络编程的目的就是指直接或间接地通过网络协议与其他计算机进行通讯。
计算机网络形式多样，内容繁杂。网络上的计算机要互相通信，必须遵循一定的协议。目前使用最广泛的网络协议是 Internet 上所使用的 TCP/IP 协议。
IP 地址：具有全球唯一性，相对于 internet，IP 为逻辑地址。

IP 地址分类：
1. A 类地址
A 类地址第 1 字节为网络地址，其它 3 个字节为主机地址。另外第 1 个字节的最高位固定为 0。

A 类地址范围：1.0.0.1 到 126.155.255.254。
A 类地址中的私有地址和保留地址：
10.0.0.0 到 10.255.255.255 是私有地址（所谓的私有地址就是在互联网上不使用，而被用在局域网络中的地址）。
127.0.0.0 到 127.255.255.255 是保留地址，用做循环测试用的。

2. B 类地址

B 类地址第 1 字节和第 2 字节为网络地址，其它 2 个字节为主机地址。另外第 1 个字节的前两位固定为 10。
B 类地址范围：128.0.0.1 到 191.255.255.254。
B 类地址的私有地址和保留地址
172.16.0.0 到 172.31.255.255 是私有地址
169.254.0.0 到 169.254.255.255 是保留地址。如果你的 IP 地址是自动获取 IP 地址，而你在网络上又没有找到可用的 DHCP 服务器，这时你将会从 169.254.0.0 到 169.254.255.255 中临得获得一个 IP 地址。

3. C 类地址
C 类地址第 1 字节、第 2 字节和第 3 个字节为网络地址，第 4 个个字节为主机地址。另外第 1 个字节的前三位固定为 110。
C 类地址范围：192.0.0.1 到 223.255.255.254。
C 类地址中的私有地址：
192.168.0.0 到 192.168.255.255 是私有地址。

4. D 类地址
D 类地址不分网络地址和主机地址，它的第 1 个字节的前四位固定为 1110。
D 类地址范围：224.0.0.1 到 239.255.255.254

Mac 地址：每个网卡专用地址，也是唯一的。
端口(port)：OS 中可以有 65536（2^16）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。
不同层的协议是完全不同的。

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。

TCP/IP 模型

Application

(FTP,HTTP,TELNET,POP3,SMPT)
Transport
(TCP,UDP)
Network
(IP,ICMP,ARP,RARP)
Link
(Device driver,….)

注：
IP：寻址和路由
ARP（Address Resolution Protocol）地址解析协议：将 IP 地址转换成 Mac 地址
RARP（Reflect Address Resolution Protocol）反相地址解析协议：与上相反
ICMP（Internet Control Message Protocol）检测链路连接状况。利用此协议的工具：ping , traceroute

16.2. TCP Socket

TCP 是 Tranfer Control Protocol 的简称，是一种面向连接的保证可靠传输的协议。通过 TCP 协议传输，得到的是一个顺序的无差错的数据流。发送方和接收方的成对的两个 socket 之间必须建立连接，以便在 TCP 协议的基础上进行通信，当一个 socket（通常都是 server socket）等待建立连接时，另一个 socket 可以要求进行连接，一旦这两个 socket 连接起来，它们就可以进行双向数据传输，双方都可以进行发送或接收操作。

1) 服务器分配一个端口号，服务器使用 accept()方法等待客户端的信号，信号一到打开 socket 连接，从 socket 中取得 OutputStream 和 InputStream。

2) 客户端提供主机地址和端口号使用 socket 端口建立连接，得到 OutputStream 和 InputStream。

TCP/IP 的传输层协议

16.2.1. 建立 TCP 服务器端

创建一个 TCP 服务器端程序的步骤：

1). 创建一个 ServerSocket
2). 从 ServerSocket 接受客户连接请求
3). 创建一个服务线程处理新的连接
4). 在服务线程中，从 socket 中获得 I/O 流
5). 对 I/O 流进行读写操作，完成与客户的交互
6). 关闭 I/O 流

7). 关闭 Socket

```
ServerSocket server = new ServerSocket(post)
Socket connection = server.accept();
ObjectInputStream put=new ObjectInputStream(connection.getInputStream());
ObjectOutputStreamo put=newObjectOutputStream(connection.getOutputStream());
```

处理输入和输出流;
关闭流和 socket。

16.2.2. 建立 TCP 客户端

创建一个 TCP 客户端程序的步骤：

1).创建 Socket
2). 获得 I/O 流
3). 对 I/O 流进行读写操作
4). 关闭 I/O 流
5). 关闭 Socket

```
Socket connection = new Socket(127.0.0.1, 7777);
ObjectInputStream input=new ObjectInputStream(connection.getInputStream());
ObjectOutputStream utput=new ObjectOutputStream(connection.getOutputStream());
```

处理输入和输出流;
关闭流和 socket。

16.3. 建立 URL 连接

UDP 是 User Datagram Protocol 的简称，是一种无连接的协议，每个数据报都是一个独立的信息，包括完整的源地址或目的地址，它在网络上以任何可能的路径传往目的地，因此能否到达目的地，到达目的地的时间以及内容的正确性都是不能被保证的。

比较：TCP 在网络通信上有极强的生命力，例如远程连接（Telnet）和文件传输（FTP）都需要不定长度的数据被可靠地传输；既然有了保证可靠传输的 TCP 协议，为什么还要非可靠传输的 UDP 协议呢？主要的原因有两个。一是可靠的传输是要付出代价的，对数据内容正确性的检验必然占用计算机的处理时间和网络的带宽，因此 TCP 传输的效率不如 UDP 高。二是在许多应用中并不需要保证严格的传输可靠性，比如视频会议系统，并不要求音频视频数据绝对的正确，只要保证连贯性就可以了，这种情况下显然使用 UDP 会更合理一些。

如：
http://www.tarena.com.cn:80/teacher/zhuzh.html

协议名://机器名＋端口号＋文件名

2 . URL 类的常见方法

一个 URL 对象生成后，其属性是不能被改变的，但是我们可以通过类 URL 所提供的方法来获取这些属性：

public String getProtocol() 获取该 URL 的协议名。
public String getHost() 获取该 URL 的主机名。
public int getPort() 获取该 URL 的端口号，如果没有设置端口，返回-1。
public String getFile() 获取该 URL 的文件名。
public String getRef() 获取该 URL 在文件中的相对位置。
public String getQuery() 获取该 URL 的查询信息。
public String getPath() 获取该 URL 的路径
public String getAuthority() 获取该 URL 的权限信息
public String getUserInfo() 获得使用者的信息
public String getRef() 获得该 URL 的锚

3. 例子，将 tarena 网站首页拷贝到本机上。

```
import java.net.*;
import java.io.*;
import java.util.*;

public class TestURL{

    public static void main(String[] arg){

        System.out.println("http://www.tarena.com.cn:80/index.htm===>");
        //System.out.println(getWebContent());
        writeWebFile(getWebContent());
        }

        public static String getWebContent(){

            URL url = null;
            HttpURLConnection uc = null;
            BufferedReader br = null;
            final int buffLen = 2048;
            byte[] buff = new byte[buffLen];
            String message = "";
            String tmp = "";
            int len = -1;

            String urlStr = "http://www.tarena.com.cn:80/index.htm";
```

```
            try{
                url = new URL(urlStr);
                //连接到 web 资源
                System.out.println("before openConnection ====>"+new Date());
                uc = (HttpURLConnection)url.openConnection();
                System.out.println("end openConnection ====>"+new Date());
                br = new BufferedReader( new
InputStreamReader(uc.getInputStream()));
                System.out.println("end getINputStream() ====>"+new Date());

                while( ( tmp = br.readLine())!=null){
                    message += tmp;
                }
                System.out.println("end set message ====>"+new Date());

            }catch(Exception e){e.printStackTrace();System.exit(1);}
            finally{

                if(br!=null){
                    try{
                        br.close();
                    }catch(Exception ioe){ioe.printStackTrace();}
                }
            }

            return  message;
    }

    public static void writeWebFile(String content){

        FileWriter fw = null;
        try{
            fw = new FileWriter("index.htm");
            fw.write(content,0,content.length());
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(fw!=null){
                try{
                    fw.close();
                }catch(Exception e){}
            }
        }
```

```
    }
}
```

16.4. UDP socket

这种信息传输方式相当于传真，信息打包，在接受端准备纸。
特点：
1) 基于 UDP 无连接协议
2) 不保证消息的可靠传输
3) 它们由 Java 技术中的 DatagramSocket 和 DatagramPacket 类支持
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，其中包括了目标地址和端口以及传送的信息（所以不用建立点对点的连接）。
DatagramPacket 的分类：
用于接收：DatagramPacket(byte[] buf,int length)
DatagramPacket(byte[] buf,int offset,int length)
用于发送：DatagramPacket(byte[] buf,int length, InetAddress address,int port)
DatagramPacket(byte[] buf,int offset,int length,InetAddress address,int port)
注：InetAddress 类网址用于封装 IP 地址
没有构造方法，通过
InetAddress.getByAddress(byte[] addr):InetAddress
InetAddress.getByName(String host):InetAddress
等。

16.4.1. 建立 UDP 发送端

创建一个 UDP 的发送方的程序的步骤：

1). 创建一个 DatagramPacket，其中包含发送的数据和接收方的 IP 地址和端口
号。
2). 创建一个 DatagramSocket，其中包含了发送方的 IP 地址和端口号。
3). 发送数据
4). 关闭 DatagramSocket

```
byte[] buf = new byte[1024];
DatagramSocket datagramSocket = new DatagramSocket(13);// set port
DatagramPackage intputPackage = new DatagramPackage(buf,buf.length);
datagramSocket.receive(inputPackage);
```

DatagramPackage outputPackage = new DatagramPackage(buf,buf.length,
inetAddress,port);
datagramSocket.send(outputPackage);
没建立流所以不用断开。

16.4.2. 建立 UDP 接受端

创建一个 UDP 的接收方的程序的步骤：

1). 创建一个 DatagramPacket，用于存储发送方发送的数据及发送方的 IP 地址和端口号。
2). 创建一个 DatagramSocket，其中指定了接收方的 IP 地址和端口号。
3). 接收数据
4). 关闭 DatagramSocket

byte[] buf = new byte[1024];
DatagramSocket datagramSocket = new DatagramSocket();//不用设端口，因为发送的包中端口
DatagramPackage outputPackage=new DatagramPackage(
Buf,buf.length,serverAddress,serverPort);
DatagramPackage inputPackage=new DatagramPackage(buf,buf.length);
datagramSocket.receive(inputPackage);

17. java5.0 的新特性

JAVA 的又一次革命。
它的特点：
JVM 不变
JAVA 和 C++不断的融合， 5.0 里有许多和 C++ 类似。留下的只是思想。
程序员的开发工作越来越简单。

17.1. 泛型

17.1.1. 说明

增强了 java 的类型安全，可以在编译期间对容器内的对象进行类型检查，在运行期不必进行类型的转换。而在 j2se5 之前必须在运行期动态进行容器内对象的检查及转换，泛型是编译时概念，运行时没有泛型

减少含糊的容器，可以定义什么类型的数据放入容器

```
ArrayList<Integer> aList = new ArrayList<Integer>();
   aList.add(new Integer(1));
   // ...
   Integer myInteger = aList.get(0);
```

我们可以看到，在这个简单的例子中，我们在定义 aList 的时候指明了它是一个直接受 Integer 类型的 ArrayList，当我们调用 aList.get(0)时，我们已经不再需要先显式的将结果转换成 Integer，然后再赋值给 myInteger 了。而这一步在早先的 Java 版本中是必须的。也许你在想，在使用 Collection 时节约一些类型转换就是 Java 泛型的全部吗？远不止。单就这个例子而言，泛型至少还有一个更大的好处，那就是使用了泛型的容器类变得更加健壮：早先，Collection 接口的 get()和 Iterator 接口的 next()方法都只能返回 Object 类型的结果，我们可以把这个结果强制转换成任何 Object 的子类，而不会有任何编译期的错误，但这显然很可能带来严重的运行期错误，因为在代码中确定从某个 Collection 中取出的是什么类型的对象完全是调用者自己说了算，而调用者也许并不清楚放进 Collection 的对象具体是什么类的；就算知道放进去的对象“应该”是什么类，也不能保证放到 Collection 的对象就一定是那个类的实例。现在有了泛型，只要我们定义的时候指明该 Collection 接受哪种类型的对象，编译器可以帮我们避免类似的问题溜到产品中。我们在实际工作中其实已经看到了太多的 ClassCastException，不是吗？

17.1.2. 用法

声明及实例化泛型类：

```
HashMap<String,Float> hm = new HashMap<String,Float>();
```

编译类型的泛型和运行时类型的泛型一定要一致。没有多态。
不能使用原始类型

```
GenList<int> nList = new GenList<int>(); //编译错误
```

J2SE 5.0 目前不支持原始类型作为类型参数(type parameter)

定义泛型接口：

```
public interface GenInterface<T> {
  void func(T t);
```

```
}
```

定义泛型类：

```
public class ArrayList<ItemType> { ... }

public class GenMap<T, V> { ... }
```

例 1：

```
public class MyList<Element> extends LinkedList<Element>
{
    public void swap(int i, int j)
    {
        Element temp = this.get(i);
        this.set(i, this.get(j));
        this.set(j, temp);
    }

    public static void main(String[] args)
    {
        MyList<String> list = new MyList<String>();
        list.add("hi");
        list.add("andy");
        System.out.println(list.get(0) + " " + list.get(1));
        list.swap(0,1);
        System.out.println(list.get(0) + " " + list.get(1));
```

```
    }

}
```

17.1.3. 泛型的通配符"?"

```
package day16;
import java.util.*;
import static java.lang.System.*;
public class TestTemplate {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        List<String> l1=new ArrayList<String>();
        l1.add("abc");
        l1.add("def");
        List<Number> l2=new ArrayList<Number>();
        l2.add(1.3);
        l2.add(11);
        List<Integer> l3=new ArrayList<Integer>();
        l3.add(123);
        l3.add(456);

    //  print(l1);
        print(l2);
        print(l3);
    }
    static void print(List<? extends Number> l){    //所有 Number 的子类
        for(Object o:l){
            out.println(o);
        }
    }

    static void print(List<? super Number> l){      //所有 Number 的父类
        for(Object o:l){
            out.println(o);
        }
    }
```

}
"?"可以用来代替任何类型，例如使用通配符来实现 print 方法。

public static void print(GenList<?> list) {})

17.1.4. 泛型方法的定义

```
<E> void copyArrayToList(E[] os,List<E> lst){
        for(E o:os){
            lst.add(o);
        }
    }

static <E extends Number> void copyArrayToList(E[] os,List<E> lst){
        for(E o:os){
            lst.add(o);
        }
    }

static<E extends Number & Comparable> void copyArrayToList(E[] os,List<E> lst){
        for(E o:os){
            lst.add(o);
        }
    }
```

受限泛型是指类型参数的取值范围是受到限制的. extends 关键字不仅仅可以用来声明类的继承关系，也可以用来声明类型参数(type parameter)的受限关系.例如，我们只需要一个存放数字的列表，包括整数(Long, Integer, Short)，实数(Double, Float)，不能用来存放其他类型，例如字符串(String)，也就是说，要把类型参数 T 的取值泛型限制在 Number 极其子类中.在这种情况下，我们就可以使用 extends 关键字把类型参数(type parameter)限制为数字
只能使用 extends 不能使用 super ，只能向下，不能向上。
调用时用 <? > 定义时用 <E>

17.1.5. 泛型类的定义

类的静态方法不能使用泛型，因为泛型类是在创建对象的时候产生的。

```
class MyClass<E>{
    public void show(E a){
        System.out.println(a);
    }
```

```
    public E get(){
        return null;
    }

}
```

受限泛型

```
class MyClass <E extends Number>{
    public void show(E a){

    }
}
```

17.1.6. 泛型与异常

类型参数在 catch 块中不允许出现，但是能用在方法的 throws 之后。例：

```
import java.io.*;

interface Executor<E extends Exception> {

    void execute() throws E;

}


public class GenericExceptionTest {

    public static void main(String args[]) {

        try {

                Executor<IOException> e = new Executor<IOException>() {

                    public void execute() throws IOException{

                        // code here that may throw an

                        // IOException or a subtype of

                        // IOException
```

```
                    }
                    };
                e.execute();
            } catch(IOException ioe) {
                System.out.println("IOException: " + ioe);
                ioe.printStackTrace();
            }
        }
}
```

17.1.7. 泛型的一些局限型

不能实例化泛型

```
T t = new T(); //error
```

不能实例化泛型类型的数组

```
T[] ts= new T[10];    //编译错误
```

不能实例化泛型参数数

```
Pair<String>[] table = new Pair<String>(10); // ERROR
```

类的静态变量不能声明为类型参数类型

```
public class GenClass<T> {
  private static T t;    //编译错误
}
```

泛型类不能继承自 Throwable 以及其子类

public GenExpection<T> extends Exception{} //编译错误

不能用于基础类型 int 等

Pair<double> //error

Pair<Double> //right

17.2. 增强的 for 循环

for in loop
解决遍历数组和遍历集合的不统一。

```
package com.kuaff.jdk5;
import java.util.*;
import java.util.Collection;

public class Foreach
{
   private Collection<String> c = null;
   private String[] belle = new String[4];
   public Foreach()
   {
          belle[0] = "西施";
          belle[1] = "王昭君";
          belle[2] = "貂禅";
          belle[3] = "杨贵妃";
          c = Arrays.asList(belle);
   }
   public void testCollection()

   {

      for (String b : c)

      {

          System.out.println("曾经的风化绝代:" + b);

      }
```

```
    }
    public void testArray()
    {
        for (String b : belle)
        {
            System.out.println("曾经的青史留名:" + b);
        }
    }
    public static void main(String[] args)
    {
        Foreach each = new Foreach();
        each.testCollection();
        each.testArray();
    }
}
```

对于集合类型和数组类型的，我们都可以通过 foreach 语法来访问它。上面的例子中，以前我们要依次访问数组，挺麻烦：

```
for (int i = 0; i < belle.length; i++)
{
        String b = belle[i];
        System.out.println("曾经的风化绝代:" + b);
}
```

现在只需下面简单的语句即可：

```
for (String b : belle)
{
    System.out.println("曾经的青史留名:" + b);
 }
```

对集合的访问效果更明显。以前我们访问集合的代码：

```
for (Iterator it = c.iterator(); it.hasNext();)
{
    String name = (String) it.next();
    System.out.println("曾经的风化绝代:" + name);
}
```

现在我们只需下面的语句：

```
for (String b : c)
{
    System.out.println("曾经的风化绝代:" + b);
}
```

Foreach 也不是万能的，它也有以下的缺点：

在以前的代码中,我们可以通过 Iterator 执行 remove 操作。

```
for (Iterator it = c.iterator(); it.hasNext();)
```

```
{
    itremove()
}
```

但是，在现在的 foreach 版中，我们无法删除集合包含的对象。你也不能替换对象。

同时，你也不能并行的 foreach 多个集合。所以，在我们编写代码时，还得看情况而使用它。

17.3. 自动装箱和自动拆箱

自动封箱解箱只在必要的时候才进行。还有其它选择就用其它的
byte b　　-128~127
Byte b　　多一个null
简单类型和封装类型之间的差别
封装类可以等于null　，避免数字得0时的二义性。
Integer i=null;
int ii=i;　　会抛出NullException 异常。
相当于 int ii=i.intValue();

17.3.1. 在基本数据类型和封装类之间的自动转换

5.0之前
Integer i=new Integer（4）;
int　ii= i.intValue();

5.0之后
Integer i=4;
Long l=4.3;
静态导入(Static Imports)

17.4. 类型安全的枚举

在 5.0 之前使用模式做出枚举

```
final class Season{
    public static final Season SPRING=new Season();
    public static final Season WINTER=new Season();
    public static final Season SUMMER=new Season();
    public static final Season AUTUMN=new Season();
    private Season(){}
}
```

完全等价于

```
enum Season2{
    SPRING,
    SUMMER,
    AUTUMN,
    WINTER
}
```

枚举类(Enumeration Classes)和类一样，具有类所有特性。Season2 的父类是 java.lang.Enum;

隐含方法： Season2[] ss=Season2.values(); 每个枚举类型都有的方法。enum 可以 switch 中使用（不加类名）。

```
switch( s ){
    case SPRING:
            …….
    case SUMMER:
            …….
    ……..
}
```

枚举的有参构造

```
enum Season2{
    SPRING(“春”),      ----------逗号
    SUMMER(“夏),   ----------逗号
    AUTUMN(“秋”),   ----------逗号
    WINTER(“冬”);   ----------分号
    private String name;
    Season2(String name){
        this.name=name;
    }
    String getName(){
        return name;
    }

}
```

Season2.SPRING.getName() ----------→春

枚举的高级用法：

```
enum Operation{
    ADD{
        public double calculate(double s1,double s2){
            return s1+s2;
        }
    },
    SUBSTRACT{
        public double calculate(double s1,double s2){
            return s1-s2;
        }
},
    MULTIPLY{
        public double calculate(double s1,double s2){
            return s1*s2;
        }
},
    DIVIDE{
        public double calculate(double s1,double s2){
            return s1/s2;
        }
};
    public abstract double calculate(double s1 ,double s2);
}
```

有抽象方法枚举元素必须实现该方法。

17.5. 静态引入

静态成员的使用，使用 import static 引入静态成员。

很简单的东西，看一个例子：

没有静态导入

```
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
```

有了静态导入

```
import static java.lang.Math.*;

sqrt(pow(x, 2) + pow(y, 2));
```

其中 import static java.lang.Math.*;就是静态导入的语法，它的意思是导入 Math 类中的所有 static 方法和属性。这样我们在使用这些方法和属性时就不必写类名。

需要注意的是默认包无法用静态导入，另外如果导入的类中有重复的方法和属性则需要写出类名，否则编译时无法通过。

17.6. C 风格的格式化输出

格式化 I/O(Formatted I/O)

增加了类似 C 的格式化输入输出，简单的例子：

```
public class TestFormat{
  public static void main(String[] args){
    int a = 150000, b = 10;
    float c = 5.0101f, d = 3.14f;

    System.out.printf("%4d %4d%n", a, b);
    System.out.printf("%x %x%n", a, b);
    System.out.printf("%3.2f %1.1f%n", c, d);
    System.out.printf("%1.3e %1.3e%n", c, d*100);
  }
}
```

输出结果为：

```
150000   10
249f0 a
5.01 3.1
```

5.010e+00 3.140e+02

17.7. Building Strings(StringBuilder 类)

在 JDK5.0 中引入了 StringBuilder 类，该类的方法不是同步(synchronized)的，这使得它比 StringBuffer 更加轻量级和有效。

17.8. 可变长的参数

使用条件：只在必要的时候进行。同时有数组，没变参，有变参，没输组，不能共存。一个方法最多只能有一个变长参数，而且是最后一个参数。
5.0 之前

```
public static void main(String[] args){

}
```

JVM 收到数据封装在数组里，然后传入方法
5.0 之后

```
public static void m(String... s){
        System.out.println("m(String)" +s);
}
```

调用 m(String… s)

```
for(String s2:s){
    System.out.println(s2);
}
```

17.9. JAVA5.0 的注释 (Annotation)

描述代码的代码。给编译器看的代码，作用是规范编译器的语法。

```
class Student{
    @Override
    public String toString(){
        return “student”;
    }

}
```

类型（接口）

1．标记注释
@Override
2．单值注释

@注释名（prameter=10）
int parameter
特例:
@注释名 (value “134”)

@SuperessWarning({“ddd”,”aaa”,”ccc”}) //JVM 还没有实现这个注释
3．普通注释
(key1=value,……)

4．自定义注释
public @interface Test{

}
注释的属性类型可以是
8 种基本类型
String
Enum
Annotation
以及它们的数组

三个新加的多线程包
Java 5.0 里新加入了三个多线程包：java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks.
java.util.concurrent 包含了常用的多线程工具，是新的多线程工具的主体。
java.util.concurrent.atomic 包含了不用加锁情况下就能改变值的原子变量，比如说 AtomicInteger 提供了 addAndGet()方法。Add 和 Get 是两个不同的操作，为了保证别的线程不干扰，以往的做法是先锁定共享的变量，然后在锁定的范围内进行两步操作。但用 AtomicInteger.addAndGet()就不用担心锁定的事了，其内部实现保证了这两步操作是在原子量级发生的，不会被别的线程干扰。
java.util.concurrent.locks 包包含锁定的工具。

17.10. Callable 和 Future 接口

Callable 是类似于 Runnable 的接口，实现 Callable 接口的类和实现 Runnable 的类都是可被其它线程执行的任务。Callable 和 Runnable 有几点不同：
Callable 规定的方法是 call()，而 Runnable 规定的方法是 run().
Callable 的任务执行后可返回值，而 Runnable 的任务是不能返回值的。
call（）方法可抛出异常，而 run（）方法是不能抛出异常的。
运行 Callable 任务可拿到一个 Future 对象，通过 Future 对象可了解任务执行情况，可取消任务的执行，还可获取任务执行的结果。

以下是 Callable 的一个例子：

```
public class DoCallStuff implements Callable{ // *1
        private int aInt;
        public DoCallStuff(int aInt) {
                this.aInt = aInt;
        }
        public String call() throws Exception { //*2
                boolean resultOk = false;
                if(aInt == 0){
                        resultOk = true;
                }   else if(aInt == 1){
                        while(true){ //infinite loop
                                System.out.println("looping....");
                                Thread.sleep(3000);
                        }
                } else {
                        throw new Exception("Callable terminated with
Exception!"); //*3
                }
                if(resultOk){
                        return "Task done.";
                } else {
                        return "Task failed";
                }
        }
}
```

*1: 名为 DoCallStuff 类实现了 Callable，String 将是 call 方法的返回值类型。例子中用了 String，但可以是任何 Java 类。

*2: call 方法的返回值类型为 String，这是和类的定义相对应的。并且可以抛出异常。

*3: call 方法可以抛出异常，如加重的斜体字所示。

以下是调用 DoCallStuff 的主程序。

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
public class Executor {
        public static void main(String[] args){
                //*1
                DoCallStuff call1 = new DoCallStuff(0);
                DoCallStuff call2 = new DoCallStuff(1);
                DoCallStuff call3 = new DoCallStuff(2);
                //*2
                ExecutorService es = Executors.newFixedThreadPool(3);
```

```
                    //*3
                    Future future1 = es.submit(call1);
                    Future future2 = es.submit(call2);
                    Future future3 = es.submit(call3);
                    try {
                            //*4
                            System.out.println(future1.get());
                             //*5
                            Thread.sleep(3000);
                            System.out.println("Thread 2 terminated? :" +
future2.cancel(true));
                            //*6
                            System.out.println(future3.get());
                    } catch (ExecutionException ex) {
                            ex.printStackTrace();
                    } catch (InterruptedException ex) {
                            ex.printStackTrace();
                    }
            }
}
```

*1: 定义了几个任务
*2: 初始了任务执行工具。任务的执行框架将会在后面解释。
*3: 执行任务，任务启动时返回了一个 Future 对象，如果想得到任务执行的结果或者是异常可对这个 Future 对象进行操作。Future 所含的值必须跟 Callable 所含的值对映，比如说例子中 Future 对印 Callable
*4: 任务 1 正常执行完毕，future1.get()会返回线程的值
*5: 任务 2 在进行一个死循环，调用 future2.cancel(true)来中止此线程。传入的参数标明是否可打断线程，true 表明可以打断。
*6: 任务 3 抛出异常，调用 future3.get()时会引起异常的抛出。
运行 Executor 会有以下运行结果：

```
looping....
Task done. //*1
looping....
looping....//*2
looping....
looping....
looping....
looping....
Thread 2 terminated? :true //*3
//*4
java.util.concurrent.ExecutionException: java.lang.Exception: Callable terminated
with Exception!
        at java.util.concurrent.FutureTask$Sync.innerGet(FutureTask.java:205)
        at java.util.concurrent.FutureTask.get(FutureTask.java:80)
```

```
        at concurrent.Executor.main(Executor.java:43)
        …….
```

*1: 任务 1 正常结束
*2: 任务 2 是个死循环，这是它的打印结果
*3: 指示任务 2 被取消
*4: 在执行 future3.get()时得到任务 3 抛出的异常