# 1.Java概述

## C/S与B/S区别：

**Java 中存在即是真理，就如一切事物*皆*对象；**

C/S 与 B/S 区别：

Client/Server是建立在局域网的基础上的.Browser/Server是建立在广域网的基础上的.

1. C/S：客户端/服务器端 ，服务器通常采用高性能的PC、工作站和小型机，并采用大型数据库Oracle或SQL server，客户端需要安装专用的软件
2. B/S：浏览器/服务器端 ，客户端只需要安装浏览器，服务器安装Oracle或sql server等数据库，在这种结构下用户界面完全通过WWW浏览器实现，一部分事务逻辑在前端实现，但是主要事务逻辑在服务器端实现，浏览器通过WEB server同数据库交互实现

区别：

1、硬件环境不同 2. 对安全要求不同 3，对程序架构不同 4、系统维护不同

5.处理问题不同 6、用户接口不同

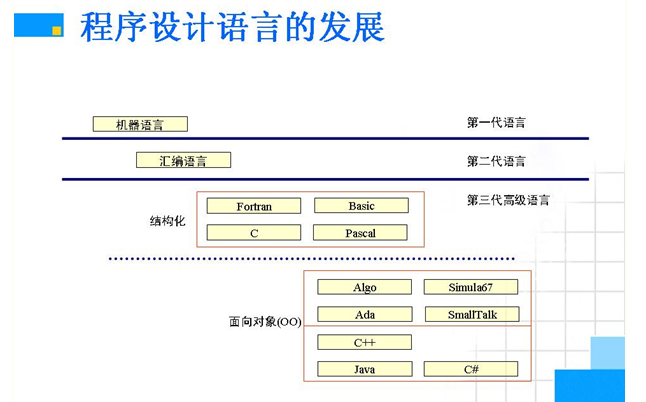
# 2.计算机语言的发展：

第一代语言：机器语言（速度最快，阅读性上最差）

第二代语言：汇编语言：出现的助记符 例如：0111 mod 1111 等

第三代语言：面向对象：C，pascal

面向过程： java（java jse j2ee jsp jsf）



Machine 计算机语言

C/C++ 编译型的语言

VB 解释型语言

Java 即是编译也是解释型语言

Shell 自身不能做事情，但是可以调用其他类型的语言的，可以用于用户与计算机沟通的桥梁

# 3.Java的为什么可以跨平台：

## 3.1Java的两个核心的技术：

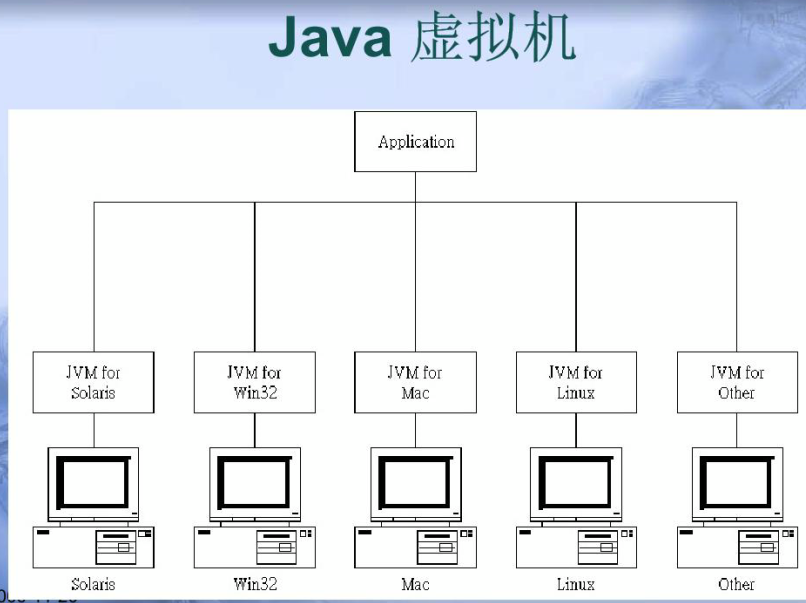
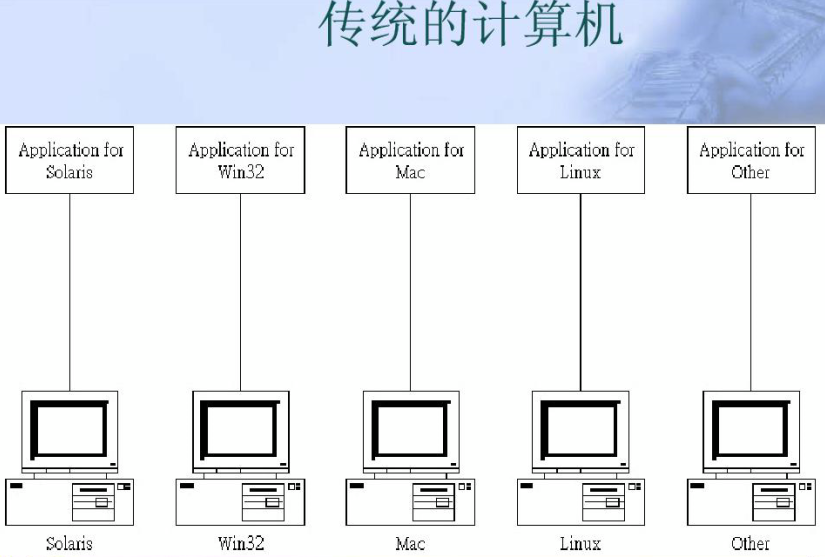
* Java 虚拟机
* Java垃圾回收机制（gc）

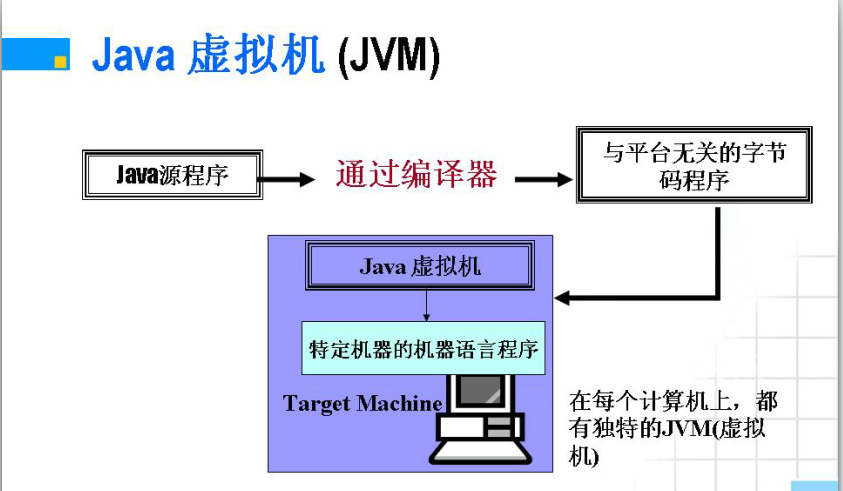
### 3.1.1 java虚拟机 jvm

Java是一个可以跨平台，跨平台就需要理解操作系统

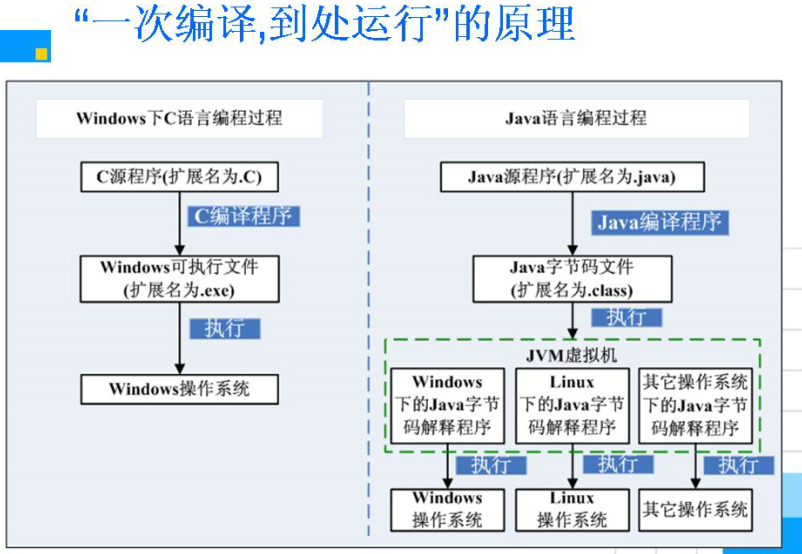
1. 操作系统：用来支持最基本的操作，譬如：创建文件及文件夹，在屏幕显示等
2. 所有的程序都与操作系统的通讯
3. 操作系统是用户与计算机键之间的主要接口

java跨平台就是利用java虚拟机（jvm）





C语言与java语言的编译的不同之处：

JRE:JAVA运行的环境，jvm

JDK: JDK是面向开发人员使用的SDK，它提供了Java的开发环境和运行环境

# JAVA 基础语法

基本实验：

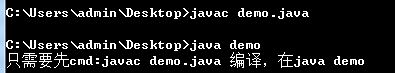
public class Demo{

public static void main(String[] arg){

System.out.println("只需要先cmd:javac demo.java 编译，在java demo");

}

}



## 4.1 创建网页文档API文档：

public class HelloWord{

//只有类名是大写的

//classc创建类（）（类名与文件名相同）

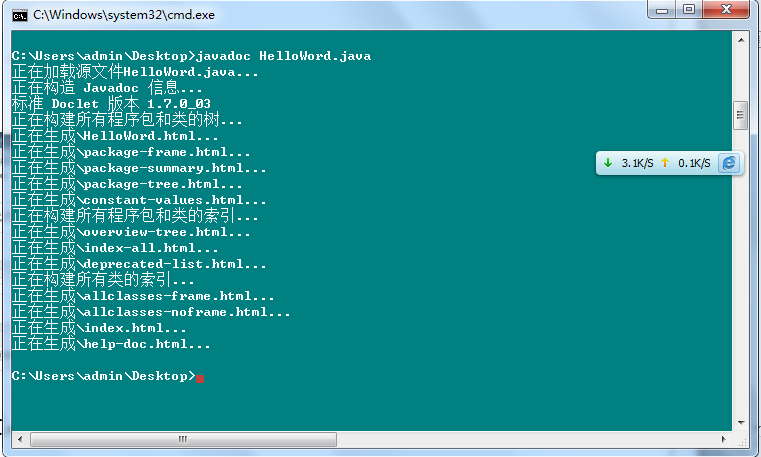
public static void main(Sring[] args){

System.out.println("HelloWorld");

}

}

**运行结果**：通过在命令行下输入命令“javadoc.HelloWord.java”，来生成网页文档



命令完毕后，会在程序的源代码同目录下生成文档相关的HTML文件



## 标示符：对变量，方法，类等要素的命名的字符序列；

Java标识符由数字，字母和下划线(\_)，美元符号（$）组成。在Java中是区分大小写的。而且还要求首位不能是数字。最重要的是，Java[关键字](http://baike.baidu.com/view/390935.htm)不能当作Java标识符。与c语言不同的是，c语言中不能用$开头 例如：Test $\_est I j num等

#### 命名约定

1.类和接口名。每个字的首字母大写，含有大小写。例如，MyClass，HelloWorld，Time等。

2.方法名。首字的首字母小写，其余的首字母大写，含大小写。尽量少用下划线。例如，myName，setTime等。

3.[常量](http://baike.baidu.com/view/346799.htm)名。基本数据类型的[常量](http://baike.baidu.com/view/346799.htm)名使用全部大写字母，字与字之间用下划线分隔。对象[常量](http://baike.baidu.com/view/346799.htm)可大小混写。例如，SIZE\_NAME。

4.[变量](http://baike.baidu.com/view/296689.htm)名。可大小写混写，首[字符](http://baike.baidu.com/view/263416.htm)小写，字间[分隔符](http://baike.baidu.com/view/1268377.htm)用字的首字母大写。不用下划线，少用美元符号。给[变量](http://baike.baidu.com/view/296689.htm)命名是尽量做到见名知义。

## 关键字：

什么是关键字：

关键字是[电脑语言](http://baike.baidu.com/view/390931.htm)里事先定义的,有特别意义的标识符,有时又叫[保留字](http://baike.baidu.com/view/421743.htm)，还有特别意义的[变量](http://baike.baidu.com/view/296689.htm)。

[**Java**](http://baike.baidu.com/view/29.htm)的关键字对[**java**](http://baike.baidu.com/view/29.htm)的[编译器](http://baike.baidu.com/view/487018.htm)有特殊的意义，他们用来表示一种[数据类型](http://baike.baidu.com/view/675645.htm)，或者表示**程序**的结构等，关键字不能用作[变量](http://baike.baidu.com/view/296689.htm)名、方法名、类名、包名。



### 关于基本数据类型：四类八种

**Boolean(true,false)，char， byte short int long， float double**

1. boolean类型只有两个值，true，false其他语言可以用数字表示，非零为真
2. char(2)在java上可以用中文字，是因为java编码是unicode（试图包含世界上所有的编码格式，包含c中ASC码，在uniconde编码最前），所以可知char是两个字节；

**byte(1),short(2),int(4),long(8)**

为啥需要四种整形类型：节省内存空间

Byte -128 – 127

Short -32768—32767

Int 。。。。。。

Long +L

整数类型的常量默认是int类型，所以数值过大，一定要加L

**float(4) double(8) 浮点型**

默认double，用float时，+f

**总结：**1. boolean不能传化为其他的数据类型

2. 容量小的自动传化为容量大的

Int num = 10；

Double n = num;

请写出将整数强制转换为浮点数的方法。

1. 用int的包装类 Integer

Intger I = new Intger(12);

Double j = i.doubleValue(); 就直接返回浮点

3.容量大的传化容量小的必须加强制类型传化（要注意溢出问题）

Int I = (int)n;

4.byte，char，short三个只要参加运算，java编译都会把其传化为int类型进行运算

5.混合运算的时候，把所有类型先传化为最大数据类型，在运算

### 关于创建类于方法：

Void（方法有无返回值） ，class （创建类），static（静态区有关）

**类的访问机制：**

不写访问控制符，那个该方法仅能再他所在的包中使用，不包含子包

public protected private

public 任何类都可以访问不管什么包

protected 与默认是增加了再在子包中可以使用，就是继承该类的包

private 私有的，只能在创造出该方法类中访问，只能是所在包

权限：public>protected> 默认>private

### [关于继承类型：](#_继承：)

Abstract 抽象类的使用 abstract class m{} 在定义某个类时是抽象类

Interface 接口的定义

Implements 用来表示子类对父类连接 public class Test implements People{}

Extends 用来表示子类对父类的继承 一个子类只有一个父类

This 子类的引用

Super 父类的引用

1. 相当于当前父类的引用
2. 如果在子类中需要使用父类的成员变量或被覆盖的方法的时候使用
3. 子类需要调用有参数的构造方法时用（当其中的父类没有参数时，则不需用）

### 条件语句：

If(布尔表达式) else 其中if必须是if(3>2) 而不是if(3)

Default Switch case break

Switch(整数表达式)语句中，格式： char是以先传化整形后存储，而在java7.0版Switch可以用字符串表达式

switch((short)1){//整数表达式的验证,

case 'b':

System.out.println("字符表达式");

case 97:

System.out.println("字符可以用整数存储");

//break;

default:

System.out.println("每次都执行");

}

switch("abc"){

case "abc":

System.out.println("可以用字符串表达式");

break;

case "abs":

System.out.println("用break跳出switch语句");

default:

System.out.println("defaul");

}

System.out.println("观察是否break跳出整个程序");

为啥一定要用整数表达式（String是新增）：

1.浮点型:可能损失精度

2.long：整数过大

3.switch((byte)1)char，short，int，String 可以的

解释：因为精度低的可以隐式的转换成精度高的，但反过来必须强制转换！只有当case语句一个都不执行的情况下，才会执行default

1. 当case执行时会一直向下执行，只到遇到break
2. 可知break是一个作用结束switch语句，还有一个是结束退出循环在if语句会提到
3. 总体来看switch就是条件表达式，在以后会提

### 循环语句：

For循环

增强循环：在容器一章有介绍，其实就是根据迭代来的

for(Object o : list){ }

例子： int nums[] = {1,2,3,4,5,}

for(int I : nums){}

While（continue，break）

Do while 首先执行，后判断条件是否符合

do{ j--- } while(j > 0) 类型

总结：只有当明确的循环次数的时候用for，不知道次数用while，while 与 do while最大区别在于至少执行一次

### **Continue与break区别**：

System.out.println("循环没有开始");

System.out.println("现在开始测试continue");

for (int i = 0; i < 3; i++) {

System.out.println("开始第" + i + "次for循环");

if (i == 1) {

continue;

}

System.out.println("看看continue后这里执行了吗？");

}

System.out.println("continue测试完毕\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("现在开始测试break");

for (int i = 0; i < 3; i++) {

System.out.println("开始第" + i + "次for循环");

if (i == 1){

break;

}

System.out.println("看看break后这里执行了吗？");

}

System.out.println("break测试完毕\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

可以看出，**continue是结束（i）本次循环，但会继续（i+1）开始循环**

**Break就是终止循环，不在继续**

### Final与finally,finalize区别：

Final表示一个修饰符，如果修饰一个类，说明该类是不能继承的；如果修饰一个变量，则变量不能改变，修饰一个方法，方法不能被重写

Finally 用于异常处理，它用来修饰一个代码块，即使前面的代码处理异常，该代码块也会执行，通常用于释放内存

Finalize 表示Objec类定义一个方法，它可以重写，用于回收资源

Final 是常量定义，final double PI = 3.1415926；

1. 要求常量在定义时，一定要求初始化，与局部变量和成员变量不同
2. Final不仅应用于基本的数据类型，还可以用于引用类型
3. 常量的命名需要大写，以区分变量
4. 外部类中局部变量在内部类调用时，必须要final声明才能调用

**final的功能是什么？什么情况下会用到它？**

1. final 修饰类 ：表示该类不能被继承，是顶级类。

修饰方法：表示不能重写。

修饰变量：表示不能修改，可在构造方法中赋值。

#### 修饰变量是注意：

**1.修饰基础类型数据为常量 final int i = 12;**

1. **修饰引用类型数据位常量 final WuLiang[] wl = new WuLiang[12];**
2. **Final与static共同修饰时（编译器常量 必须用大写表示，并使用下划线分隔各个单词，而没有要求，用final修饰变量就必须用大写表示，而在c语言中才是这样规定的）**
3. **Final int j = rand.nextInt(20);**

**我们不能因为某数据为final就认为在编译时就可以知道它的值，在运行时使用随机生成的数值类初始化**

**5． 空白final**

**Java允许“空白finnal”，指被声明为fianl但为给定初值的域（无任什么情况，编译器都确保空白final在使用前必须初始化。当然空白final在关键字fianl的使用提供更大的灵活性）**

public class Demo{

private final int i = 2;

**private final int j;//空白fianl**

**private final Boss boss;//空白final修饰引用数据**

**public Demo(){**

**j = 2;**

**boss = new Boss(1);**

**System.out.println("j ="+j);**

**}**

**public Demo(int x,Boss b){**

**j =x;**

**boss = b;//为空白fianl初始化，一定的，编译器必须确保空白fianl使用前初始化才能使用**

**System.out.println("j ="+j);**

**}**

public static void main(String[] arg){

**new Demo();**

**new Demo(3,new Boss(10));//通过这空白fianl可以提供final灵活性**

}

}

class Boss{

private int i ;

Boss(int i){

this.i = i;

}

}

2.当类不需要继承，当方法不能被重写，当使用常量时

Finally在后解释，与final没有联系，完全是两个不同的类型，就是在

Try { ……

}catch(Exception e ){ ……

e.printStackTrace();

}finally{ //无任是否异常都执行的语句

}

### Try catch Finally throw throws异常处理

1.在异常处理中，各个关键字作用：

**Try 块用于监控异常**

**Catch 块用于捕获异常**

**Finally 块用于释放资源、关闭文件、关闭数据库等**

**Throw 用于抛出一个异常类对象，通常用于处理自定义异常情况**

**Throw new MyException（）；**

**Throws 则是在方法声明时，告诉调用者该方法抛出什么类型的异常，而异常的捕获、处理交由调用该方法者去捕获**

**Int parseInt(String a) throws NumberFormatException**

1. 可以抛出异常用方法上写throws Exception在测试时可以这样写
2. 或者用try catch finally ，在真正的编程是这样写

**public** **class** Test{

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

System.out.println(Show.show());

}

}

**class** Show{

**public** **static** String show(){

System.*out*.println("开始试验try catch ");

**try**{

**int** a=2/0;

}**catch**(Exception e){

System.*out*.println("出现异常，执行catch语句");

**return** "执行该语句是前提是finally执行后在而且没有finally没有返回值";

}**finally**{

System.*out*.println("当有finally，catch有返回值时是不会执行，知道执行finally后才继续");

**return** "执行该语句会使catch语句中的返回值不起作用";

}

//return 3;

}

}结论1：当try里面的代码没有异常的时候，会执行该try块对应的finally块，并继续执行finally之后的代码。

结论2：当try里面的代码出现异常的时候，会执行该try块对应的catch块和finally块，且即使catch中有返回finally也是要执行。也就是先执行完finally中的语句，再返回catch中的return。

结论3：相对于结论2，finally中如果有return的情况，那么即使catch中有return，这个return也不起作用，该方法调用要到finally中的return才返回。

### Transient

序列化：Transient int time   
 简单点说，就是只保存一时的值，不会再叠加，每次在执行该数据，会重新开 始

注解：@Transient

**public** Set<Type> getChildren() {

**return** children;

}

**@transient** 当不保存数据时，用其就不保存包数据库中去，其中的意思就是数据不会再数据库中增加，但是肯定会在数据库中创建字段

Java的serialization提供了一种持久化对象实例的机制。当持久化对象时，可能有一个特殊的对象数据成员，我们不想用serialization机制来保存它。为了在一个特定对象的一个域上关闭serialization，可以在这个域前加上关键字transient。

Transient是Java语言的关键字，用来表示一个域不是该对象串行化的一部分。当一个对象被串行化的时候，transient型变量的值不包括在串行化的表示中，然而非transient型的变量是被包括进去的。我们把对象的这种能记录自己的状态以便将来再生的能力。叫作对象的持续性(persistence)。对象通过写出描述自己状态的数值来记录自己 ，这个过程叫对象的串行化(Serialization－连续)

### Import，package

Import 导入包

Package 打包。其实就是使全球唯一，在各个程序都可以使用，java为每个类都创建类字节码.class，类名与文件相同，因此同名的类可能发生冲突，为了解决这个，就使用了包来管理类名空间；

Java开源的，是每个人写项目时，不能类名有冲突，所以有包的概念，也就文件夹的概念

## [Static](#_Static)

静态的static，静态（唯一，不重复），存储在栈区中的静态区中，

静态属性：所有对象共享数据

静态方法：方法被声明为static后，则称为**类方法**。类方法相对于实例方法，前者区别于后者的地方：前者为属于该类的所有实例对象共享，无须实例化对象，仅通过类名即可访问(当然，是否能够直接访问，还取决于所声明的访问权限)。

1. 静态的方法只能访问静态的成员变量
2. 非静态的方法可以访问静态的方法（变量）和非静态的方法（变量）

[Static { }语句块：](../java资料/java网上资料/Static%7b%7d语句块详解.docx)（超链接）

**static{}语句块只会加载的时候才会执行，且只会执行一次**

**什么时候加载，就需要理解java的反射机制：**

**第一次加载：**

1、用Class.forName()显示加载的时候

2、实例化一个类的时候，如将main()函数的内容改为:Test t=new Test();//这种形式其实和1相比，原理是相同的，都是显示的加载这个类，读者可以验证Test t=new Test();和Test t=(Test)Class.forName().newInstance();这两条语句效果相同。

3、调用类的静态方法的时候，

4、调用类的静态变量的时候

注意，调用静态方法的时候不会加载

java反射机制：**通过拿到一个类字节码，然后可以做出与这个类的相关的所以操作**

session.load(Test.**class**, 1);

1. **一个java类只会加载一次，一个类只有一个字节码**
2. Class代表在内存中的字节码 Test.Class
3. 如何获取字节码：

Class c1 = T.**class**;

Class c2 = **new** T().getClass();

Class c3 = Class.forName("com.wuliang.T");

总结：由上特性，static变量都会被类加载时初始化，而方法同时随类加载而进驻内存。

### New

New + 构造方法（）；唯一的创建对象方法

### Null：

空指针，只有当对象（为null）调用相应的方法或成员是才会出现空指针异常

### Strictfp（不常用）

一旦使用了关键字strictfp来声明某个类、接口或者方法时，那么在这个关键字所声明的范围内所有浮点运算都是精确的，符合IEEE-754规范的。例如一个类被声明为strictfp，那么该类中所有的方法都是strictfp的。

### Volatile（不常用） Synchronized

线程中的锁的概念，能够保证在同一时刻最多只有一个线程执行该段代码

volatile修饰变量。在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

看看Java Language Specification中的例子。

条件：一个线程不停的调用方法one()，一个线程不停的调用方法two()。我测试过多次，这种情况好像一直没有出现。

Java代码

class Test {

static int i = 0, j = 0;

static void one() { i++; j++; }

static void two() {

System.out.println("i=" + i + " j=" + j);

}

}

结果偶尔会出现j大于i的情况，因为方法没有同步，所以会出现i和j可能不是一次更新。一种防止这种情况发生的办法就是声明两个方法为synchronized 的。

Java代码

class Test {

tatic int i = 0, j = 0;

static synchronized void one() { i++; j++; }

static synchronized void two() {

System.out.println("i=" + i + " j=" + j);

}

}

这样可以防止两个方法同时被执行，还可以保证j和i被同时更新，这样一来i和j的值一直是一样的。

另外一种途径就是把i和j声明为volatile。

Java代码

class Test {

static volatile int i = 0, j = 0;

static void one() { i++; j++; }

static void two() {

System.out.println("i=" + i + " j=" + j);

}

}

### Native（不常用）

可以将native方法比作Java程序同Ｃ程序的接口，其实现步骤：

　　 １、在Java中声明native()方法，然后编译；

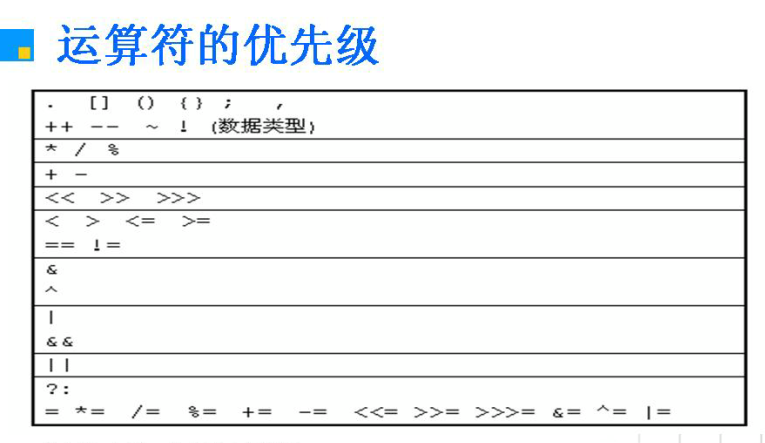
　　 ２、用javah产生一个.h文件；

　　 ３、写一个.cpp文件实现native导出方法，其中需要包含第二步产生的.h文件（注意其中又包含了JDK带的jni.h文件）；

　　 ４、将第三步的.cpp文件编译成动态链接库文件；

　　 ５、在Java中用System.loadLibrary()方法加载第四步产生的动态链接库文件，这个native()方法就可以在Java中被访问了。

## 3.Java运算符



位运算符：&，|，~，^ ,>>,<<,>>>,<<<

算数运算符：+，-，\*，/,%

逻辑运算符：！，&&，||

关系运算符：>,<,>=,<=,!=,==

三目运算符：？：

位运算符：&（按位与） 和&&类似，1&1=1，1&0=0 就是有假为假

|（按位或） 和 ||类似，1|0=1,1|1=1,0|0=0 有真为真

~（按位取反）~1=0

^（按位异或） 1^1=0,1^0=1 与&相反

>> 右移 对于有符号数字,在右移时，符位同时移动。当正数，最高数为0，最高补0，而为复数时，最高位为1，最高位补1

(7 >> 2;0000 0111>> 2; 0000 0001 = 1)

<< 左移符号运算符 （7 <<2;0000 0111 << 2; 0001 1100 =28）

>>> 右移无符号运算符 (当出现符号时，则运算无穷)

**技巧：位运算的效率最高，是二进制运算**

**其中左运算 << 相当于把操作数乘以2的N次幂**

**而 右运算 >> 相当于把操作数除以2的N次幂，与左运算相反**

注意：java中“+”有两个作用

1. 普通的算术运算符和自增
2. 字符串连接符 （在遇到字符串连接符时会自动调用toString（）方法）
3. 1. &&和 ||、！是逻辑运算符，根据逻辑关系生成一个布尔值（true,false）,其中&& 是一个与运算，必须判断都为真则为真，而“||”判断一个为真就为真

if(true && false){

System.out.println("&& 是一个与运算，必须保证判断都为真才为真");

}

2.与按位与“&”，按位或“|”不同的是，逻辑与或是有“短路”，即一旦能够明确无误确定整个表达式的值，就不在计算表达式余下部分，而按位运算符不会，所以用逻辑运算符判断提高性能

## 4.条件语句和循环例子

前面已提到条件和循环语句，不过，有些重要例子

九九乘法：for(int i=1;i<10;i++){

**for**(**int** j=1;j<i+1;j++){

**int** t=i\*j;

System.out.print(i+"\*"+j+"= ");

System.out.printf("%-2d",t);

System.out.print(" ");

}

System.out.println();}

求质数：

1.质数是i=101-200;

2.用除数从k=2开始，直到200-1,k=2;k\*k<=i

3.相除i/k==0时，break，退出循环

4.不等于零时，判断k\*k>=i+1，打印出i

### 图形界面：

\*

\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

int k,i;

for(i=4;i>=0;i--){

for(int j=0;j<i;j++){

System.out.print(" ");

}

for(k=9-2\*i;k>0;k--){

System.out.print("\*");

}

System.out.println();

}

# 4. 面向对象

## 1.面向对象与面向过程区别：

**面向过程**就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了。

**面向对象**是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。

例如五子棋，面向过程的设计思路就是首先分析问题的步骤：1、开始游戏，2、黑子先走，3、绘制画面，4、判断输赢，5、轮到白子，6、绘制画面，7、判断输赢，8、返回步骤2，9、输出最后结果。把上面每个步骤用分别的函数来实现，问题就解决了。

而面向对象的设计则是从另外的思路来解决问题。整个五子棋可以分为 1、黑白双方，这两方的行为是一模一样的，2、棋盘系统，负责绘制画面，3、规则系统，负责判定诸如犯规、输赢等。第一类对象（玩家对象）负责接受用户输入，并告知第二类对象（棋盘对象）棋子布局的变化，棋盘对象接收到了棋子的i变化就要负责在屏幕上面显示出这种变化，同时利用第三类对象（规则系统）来对棋局进行判定。

可以明显地看出，面向对象是以功能来划分问题，而不是步骤。同样是绘制棋局，这样的行为在面向过程的设计中分散在了总多步骤中，很可能出现不同的绘制版本，因为通常设计人员会考虑到实际情况进行各种各样的简化。而面向对象的设计中，绘图只可能在棋盘对象中出现，从而保证了绘图的统一。

功能上的统一保证了面向对象设计的可扩展性。比如我要加入悔棋的功能，如果要改动面向过程的设计，那么从输入到判断到显示这一连串的步骤都要改动，甚至步骤之间的循序都要进行大规模调整。如果是面向对象的话，只用改动棋盘对象就行了，棋盘系统保存了黑白双方的棋谱，简单回溯就可以了，而显示和规则判断则不用顾及，同时整个对对象功能的调用顺序都没有变化，改动只是局部的。

再比如我要把这个五子棋游戏改为围棋游戏，如果你是面向过程设计，那么五子棋的规则就分布在了你的程序的每一个角落，要改动还不如重写。但是如果你当初就是面向对象的设计，那么你只用改动规则对象就可以了，五子棋和围棋的区别不就是规则吗？（当然棋盘大小好像也不一样，但是你会觉得这是一个难题吗？直接在棋盘对象中进行一番小改动就可以了。）而下棋的大致步骤从面向对象的角度来看没有任何变化。

当然，要达到改动只是局部的需要设计的人有足够的经验，使用对象不能保证你的程序就是面向对象，初学者或者很蹩脚的程序员很可能以面向对象之虚而行面向过程之实，这样设计出来的所谓面向对象的程序很难有良好的可移植性和可扩展性。

什么是面向对象：对于发生一件事物时，

**面向对象是按照要完成的功能来实现，面向过程按照解决这个问题的步骤来实现**

考虑方向是对象，状态，结果。忽略过程

考虑：什么的对象，干嘛（方法）

什么是面向过程：对于发生一件事物时，考虑方向是按过程来按部就班一步步写好；

主要考虑的是步骤

例如：银行取钱，面向对象：就是吴亮，取钱

面向过程：坐车去银行，到柜台，取款机，插卡，输入密码

面向对象结构：类，对象，方法

什么是对象：java的眼中一切皆对象；

对象的属性：名词（变量）

对象的方法：动词（方法）

### 什么是类：类是具有相同性质和功能的事物所构成的集合体，是对现实事物的抽象描述

**类与对象的区别**：类是一个抽象的概念，是创建对象。对象是类的实例，是客观世界真实存在的实体

类是抽象，对象是具体存在的；

人类 我

Java编程的思想：自己创建类，说明把自己当成上帝；

### 修饰符：

有四种（public，private，protected，default）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 权限修饰赋 | 当前类 | 同一包 | 子孙类 | 其他包 |
| Public | √ | √ | √ | √ |
| Protect | √ | √ | √ |  |
| 默认 | √ | √ |  |  |
| Private | √ |  |  |  |

注意：修饰符既可以用在方法上，也可以用在变量上

**局部变量前不能放置任何修饰符**

### 构造方法：

1. 是一个特殊的方法
2. 没有返回值，连void都没有
3. 用来创建对象，显示不了其调用的过程
4. 每个类都有默认一个空间的构造方法，称之为隐式构造方法
5. 如果自己写了一个构造方法时，则隐式构造方法消失d

Public 类名（）{}

//构造方法最重要的应用是创建对象

New + 构造方法

### 方法的重载（overload）：

**什么是方法的重载：**指多个方法享有相同方法名，但是参数列表必须不同，类修饰符和返回值可以不同可以同

**区别是否是重载：**就根据参数列表是否相同，其中参数列表（个数与类型）循序不同都可以

### **方法的重写：**（override方法的覆盖）

1. 方法的重写发生在继承关系中(也就是子类对父类的方法的引申)
2. 重写的方法与父类方法要一致，返回值，方法名，参数列表
3. **唯一可以不同的重写方法修饰符期限<=父类中被重写的方法的修饰符**

@Override//该注解表示方法的重写

**public** **boolean** equals(Object obj) {

**return** **super**.equals(obj);

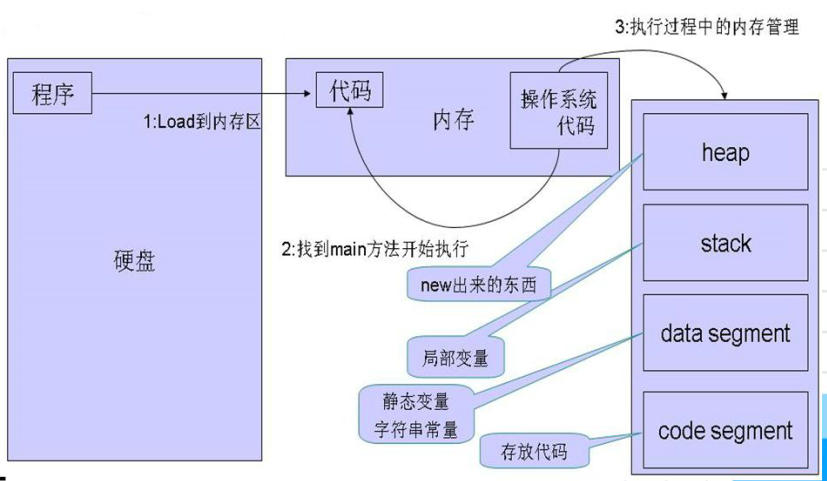
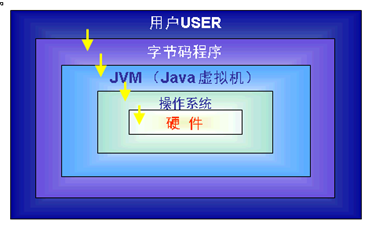
}

**注意:**

**重写和重载的产生是基于继承，没以后继承就不会重写与重载**

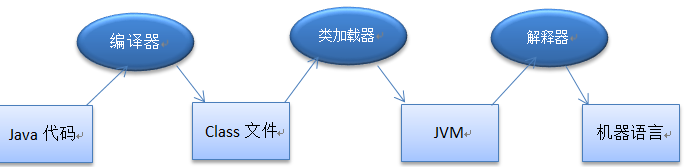
**Static 是静态，唯一性，所有的静态的方法不能被重写**

## 2.Java内存解析：



Java程序执行过程：

编写好程序 Test.java ---.class字节码—jvm内存—



----🡪 1.代码区 （存储字节码） code segment

2.栈区 （存储成员变量）stack

3.堆区 （存储对象new+构造（）） heap

4.静态区 （静态变量和字符串常量） date segment

Jvm内存解析：

堆区：

1. 只存入对象，每个对象都有相对应的class文件，从而得到操作指令
2. Jvm只有一个堆区被所有的线程所共有，没有对象引用和变量
3. 对象不存在时，由系统在某一时间释放gc垃圾回收机制

栈区：

1. 每个线程都有一个栈区，栈区存放于基本数据类型变量和对象的引用
2. 每个栈区中的数据都是私有，其他栈区不能访问
3. 自动释放，存储函数的参数值和局部变量的值

静态区：  
 1.由所有线程所共有

2.存放在静态区的成语变量；和字符串和static变量

Static int i=12；

Java中六种数据存储方式

寄存器：

1. 存在与CPU中，存取速度最快，但数量有限

2、Stack(栈、堆栈)

存在与内存中，存取的速度和效率仅次于Register，在程序加载的时候编译器必须准确知道所需的所有空间大小，通过指针来操作

3、Heap(堆中)

存在与内存中，存取方式灵活，无需编译器关心，速度比不上Stack

4、Static (静态类型)

整个程序都可以访问，对象的属性可以是静态的，但JAVA对象不会是静态的.

5、Constant（常量）

存在于程序代码中，不能改变

6、非易失性数据

a、流类型数据 Transient

b 、持久性数据

## String类型

String是特殊的包装类型

可以用String str =”abc” 和 String str = new String(“abc”);来创建对象，第一个隐藏了对象的创建，首先在栈中创建“abc”的值得地址也就引用。在堆中创建一个对象，此对象不是str，而是str指向对象，是堆中对象指向栈中的数据；没有用new的是可以共享的

用new出现不可以共享、、、

“abc”字符串是放在静态区中，用new创建的对象指向”abc”,当再次创建对象，还是必须在静态区中创建“abc”，所以说new出来的不可以共享，而第一个是使用String str1=”abc”;不必创建对象，str1直接指向”abc”,所以可以共享，效率高

String str1 = "abc";

String str2 = "abc";

String str3 = new show().toString();

String str4 = new String("abc");

String str5 = new String();

str5="abc";

System.out.println(str1.hashCode());

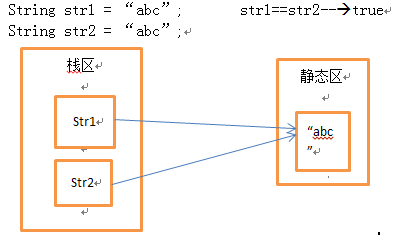
System.out.println(str2.hashCode());

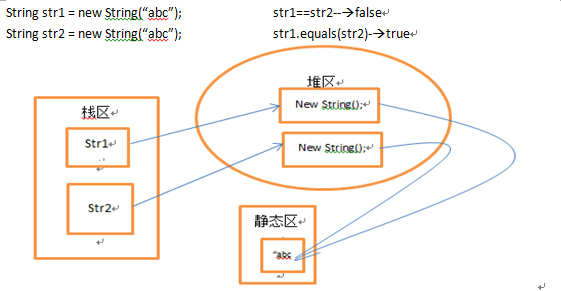
System.out.println(str4.hashCode());

System.out.println(str3.hashCode());//以查看hashCode,得到虚拟内存地址

System.out.println(str5.hashCode());

### String str = “abc” 与 String str = “abc”区别：





### Static

public class Test {

public static void main(String[] args){

Show s = new Show();

s.name = "b=not is static";

s.age=19;

s.name1="static";

s.age1=23;

Show s1 = new Show();

s1.name = "b=is static";

s1.age=191;

s1.name1="static1";

s1.age1=22;

Show1 s2 = new Show1();

s2.name1="static2";

s2.age1=3333;

System.out.println(s.name+" "+s.age+" "+s.name1+" "+s.age1);

System.out.println(s1.name+" "+s1.age+" "+s1.name1+" "+s1.age1);

System.out.println(s2.name1+" "+s2.age1);

}结果：

}

class Show{

String name;

int age;

static String name1;

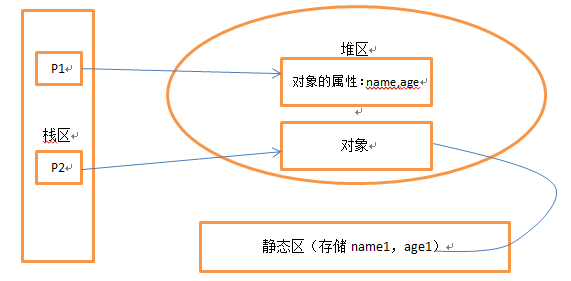
static int age1;

}

class Show1{

static String name1;

static int age1;

}

1. Static变量：从中可以得到：static是不能保存每个对象的数据，在同一类中所有对象中static是共享的，只有个在不同的类中static是不同的每次使用同一类中static都会覆盖原来的数据，所以不能用static保存每个对象的数据。
2. Static 方法：1.静态方法只能访问静态的成员变量和静态的方法

2.非静态方法既可以访问静态的也可以访问非静态的

### This 与 Super区别：

都是引用，都是指向本身，This是栈区中的引用，Super堆区中的引用

This 子类继承父类所有成员（包含成员变量与方法）

Super 1.相当于当前父类的引用

2.如果在子类使用调用父类的重名成员变量和子类覆盖父类的方法的时候，就需要Super来标明是父类，用this标明是子类

3.子类调用构造方法是，注意一点：先有父亲，再有儿子

所以在使用构造方法，创建子类对象的时候，先把父类对象创建出来，java默认调用的是父类没有参数的构造方法，所以当父类中有没有参数的构造方法，那么子类可以不写构造方法或者写构造时，不必考虑父类，因为系统会自动创建对象

必须写好super(参数)

例子：

**public** **class** Test{

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

// Super s = new Super();

// s.show();

Son son = **new** Son();

son.show();

}

}

**class** Super{

String name = "wuliang";

**public** Super(String name){

**this**.name = "吴亮";

System.*out*.println("子类默认调用空的父类，当父类不为空时，就必须先创建父类的对象");

}

**public** **void** show(){

**int** age = 19;

System.*out*.println("name= "+name+"----->age="+age);

}}

**class** Son **extends** Super{

String name ="aaa";

**public** Son(){

**super**("name");

System.*out*.println("子类调用构造方法必须写创建父类的对象");

}

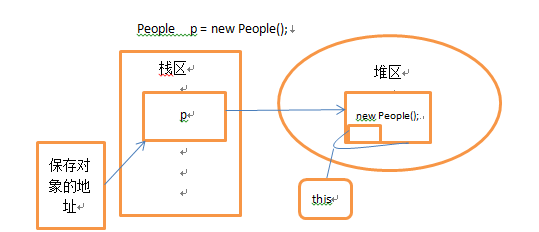
**public** **void** show(){

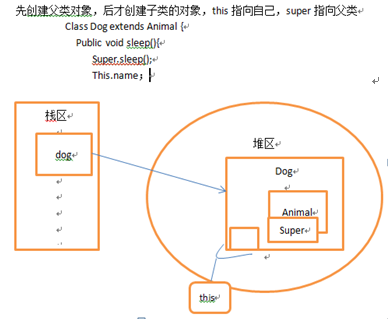
**super**.show();//当子类调用父类的重名变量或者被覆盖的方法的时候，就需要使用super以区别子类的成员变量与方法

**int** age = 23;

System.*out*.println("name= "+**super**.name+"----->age="+**super**.age);//父类变量

System.*out*.println("name= "+ name+"----->age="+ age);

}}



# 5.面向对象：类与类之间的关系

面向对象的三大方法：

1. 继承
2. **多态** ：指把类中类似的功能的不同方法使用同一个方法名来实现，从而使用相同的方法类调用不同功能的同名方法。面向对象程序设计中的多态可以通过重写（子类对父类的覆盖）和重载（在同一类中定义同一方法名不同功能的方法）的方式来实现
3. 封装

## 继承：

**什么是继承：什么是一种什么（老师是人）**

作用：1. 代码的复用 2. 功能的扩展（子类继承父类的，子类还可以增加功能）

可以提下：聚合关系：什么是什么的一部分

被继承：父类，超类，基类（super class）

需继承：子类 （子类名 extends 父类名）

在java中仅仅只能是单继承的，而不是c++中的多继承，不过，可以用接口来表示

子类可以继承父类的成员变量,方法；

**class** Animal{

**public** Animal(**int** d){

System.*out*.println("当第一排就是，说明在继承中，必须先创建父类对象才可以"+d);

}

**public** **void** show(){

System.*out*.println("1111111111111111111111");

}

}

**class** Dog **extends** Animal{

**public** Dog(**int** i){

**super**(1); //super已经介绍了，当子类中需要调用父类的成员变量和被覆盖的方法的时候，才需要用

2.当子类需要重写构造方法的有参数的时候，必须先创建父类的对象

3.当父类的构造方法没有参数，系统默认先创建父类对象，不需要手动创建

System.out.println("使用构造方法是需要super创建父类");

}

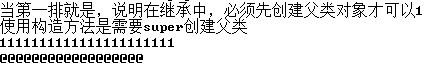
**public** **void** show(){

**super**.show();

System.*out*.println("@@@@@@@@@@@@@@@@@@");

}}

结果：



注意：

请写出继承的语法。通过super关键字可以明确访问父类的变量或方法。

1. class Father{

//父类

}

class Son extends Father{

//继承父类的方法与变量，可重写

}

2. 通过super关键字不但可以调用父类的构造方法，也可以访问父类的变量和方法，其格式如下：

下面通过一段程序来讲解访问父类的变量和方法，其代码如下：

class NeiOne

{

int a;

}

class NeiOne1 extends NeiOne

{

int a;

NeiOne1(int x,int z)

{

super.a=x;//调用父类被子类隐藏的变量

a=z;

}

void print()

{

System.out.println(""+super.a);

System.out.println(""+a);

}

}

class text

{

public static void main(String args[])

{

NeiOne1 a1=new NeiOne1(1,2);

a1.print();

}

}

包：也就是文件夹的概念；

由于java是开源的，在相互整合项目的同时，必须要求类名是唯一的，当其中类名就要用包来区别

※全球唯一：IP地址-域名 package com.wuliang.util;(记得说唯一的时候，就要用Ip地址)

Java 中 java.lang 包中是默认的import java.lang.\*;而其中有String类包，所以不需要

## OBJect 类：（要记住屏幕上打印出来的都是字符串）

Java 中所有的类都继承object，自己写的类都是默认继承Object类，class Test extends Object ,除非他已继承一个父类，而父类肯定也继承Object类

1. toString方法

**Object 类的 toString 方法返回一个字符串，该字符串由类名（对象是该类的一个实例）、at 标记符“@”和此对象哈希码的无符号十六进制表示组成。换句话说，该方法返回一个字符串，它的值等于：**

**getClass().getName() + '@' + Integer.toHexString(hashCode())**

String str = new Test().toString();

System.out.println(str);//

**new Test().toString();得到的：Test@5bf825cc类名+@+哈希码序列**

两种情况下自动调用toString()

1. 在println中

System.out.println(“new Test()”);

1. 用字符串连接符（”+”）出现在表达式中有字符串的时候

Int I = 1;

System.out.println(“wulinag”+i); wulinag1

## Equals：

重写equals方法为什么重写hashCode，是由于equals是判断对象的内容是否相等，而不代表地址就相等，就必须找到该对象的中哈希码，也是类似于地址，从而找到对象的值

**public** **class** Test{

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

T t1 = **new** T("lll",11);

T t2 = **new** T("lll",11);

System.*out*.println(t1.equals(t2));

}

}

**class** T{

String name ;

**int** age ;

**public** T(String name,**int** age){

**this**.name=name;

**this**.age=age;

}

String str1 = **new** String("abc");

String str2 = **new** String("abc");

**public** **void** showEquals(){

**if**(str1 == "abc"){

System.*out*.println("------------==------");

}

**if**(str1.equals("abc")){

System.*out*.println("-------equals-------");

}

**if**(str1 == str2 ){

System.*out*.println("等号==是比较对象的地址的");

}

**if**(str1.equals(str2)){

System.*out*.println("equals是比较对象值");

}

}

@Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + age;

result = prime \* result + ((name == **null**) ? 0 : name.hashCode());

**return** result;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

T other = (T) obj;

**if** (age != other.age)

**return** **false**;

**if** (name == **null**) {

**if** (other.name != **null**)

**return** **false**;

} **else** **if** (!name.equals(other.name))

**return** **false**;

**return** **true**;

}

}



自己写的错误例子：

**public** **boolean** equals(Object obj){

**if**(obj **instanceof** Test){

T t = (T)obj;

**if**(**this**.name.equals(t.name) && **this**.age == t.age){

**return** **true**;

}**else**{

**return** **false**;

}

}**else**{

**return** **false**;

}

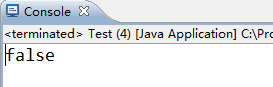
}

**public** **int** hashCode(){

**return** **this**.name.hashCode()+((Integer)**this**.age).hashCode();

}

没写及写错equals和hashCode了：



1. **什么时候需要重写equals和hashCode**

**在自定义类中，就需要重写，由于自定义类中没有给他相等的规则，就无法判断对象的属性是否相等（自定义类就是自己创建的类）**

**hasCode就是为了使对象相等的时候，哈希吗相等**

## 向上转型：

一个对象既可以作为它自己本身的类型使用，也可以作为它的基类使用，**而把这中对某个对象引用视为对基类的引用的做法称为向上转型**

**Cat cat = new Cat();**

**Animal.sleep(cat);**

**用List<String> list = new ArrayList<String>();.使用的是向上转型，而使用接口的目的就为了当你认为程序查询次数没有更新重要，就可以改为List<String> list = new LinkedList<String>();**

**不过，用这个List接口，可以更好实现，但是ArrayList与LinkedList有独自的成员，所以需要时，就不能用向上转型了**

Animal a = new Cat();向上转型

Java中允许基类的引用指向子类对象

## 向下转型

Cat c = （Cat）a; 只能强制转换，a 不能使用子类中多出来的成员与方法

禁止使用子类多出来类型

## 代理类型

第三中关系称为代理，java并没有提供对它的直接支持，只是继承与组合之间的中庸之道

public class Demo{

public static void main(String[] arg){

Secretary s = new Secretary("小庄");

s.wages();

}

}

**class Boss{**

**void wages(String name){**

**System.out.println(name+"------10000$----");**

**}**

**}**

**class Secretary{**

**private String name;**

**private Boss boss = new Boss();**

**public Secretary(String name){**

**this.name = name;**

**}**

**void wages(){**

**boss.wages(this.name);**

**}**

**}**

**代理就是我们不能直接访问的对象（Boss），系统为我们配置一个代理对象（Sercretary），我们可以通过该对象类访问我们需要的对象，不需要继承关系**

## 封装：

就是将重复出现的代码写成一个方法，使用的时候直接调用；

封装的含义：屏蔽细节

## 多态：

**什么是多态**：指把类中具有相似的功能不同的方法使用同意方法名来实现，从而可以使用相同的方法名来调用具有不同功能的同名方法

在程序执行的过程中根据实际传入的参数来调用相应的方法

条件：1. 继承 2. 方法的重写 3. 父类的引用指向子类的对象

## abstract抽象类：

**原因：有时候，基类并不与具体的事物相联系，而是只表代一种抽象的概念，用以他的派生类提供一个公共的界面。因此，java中引入了抽象类的概念**

抽象类定义： 当我们的一个类，只是为了多态或其他的一些形式存在，他们的方法一定要被重写并且它的对象永远不能被创建，那么他就是抽象类

1. 抽象类不能直接实例化，对抽象类不能new

创建时：class abstract Test

1. 抽象类可以包括抽象成员，但非抽象类中不可以

Abstract void m();

1. 一个类中有抽象类方法，这个类必须是抽象类
2. 这个类是抽象类是可以有非抽象方法，也可以有抽象类方法

如果一个非抽象类从抽象成员中派生，则必须通过覆盖来实现所有继承而来的抽象成员

### 抽象方法：

抽象方法就是在声明方法，加上abstract关键字，声明抽象方法需要注意四点：

1. 抽象方法必须在抽象类中
2. 声明抽象方法时，不能用static和private修饰符
3. 抽象方法只是引入了新方法，但不提供该方法实现，只包含一个分号
4. 当抽象类派生一个非抽象类，必须重写非抽象类的成员

解释：为啥不能用static和private，由于static存储在静态区中，是唯一，不能被重写，但是抽象类目的就是提供一个公共界面，需重写，所以，而private是私有的，那么外界将无法访问，接口也就失去意义

## Interface接口：

由于Java不像C++一样可以实现多继承，而在Java中实现类似于多继承，需要借助于Java中的接口。**接口是一种用来定义程序的协议，**它描述可以属于任何类或结构的一组相关行为。Java声明接口，使用interface关键字

其多继承格式：

修饰符 interface 接口名称 extends 继承接口列表

**interface** D **extends** A,B,C{ }

特殊的抽象类：抽像类可以有非抽象的方法和成员变量，接口中只能有抽象的方法与常量

用implements来连接接口

关于接口具有的特性，总结如下：

* 接口只能定义方法，不能实现
* 接口只可由常量或方法这两种成员的任何组合构成
* 接口成员最好用public来修饰
* 接口可以使用extends继承一个或多个接口，多个接口用逗号隔开
* 接口类似于抽象类，实现接口的类必须要实现接口中所有方法

注意：**接口是规范，类是实现；接口的目的是定义一个规范，所以类都需要遵守这个规范**

### 接口和抽象类的区别：

抽象类可以提供成员的实现细节，而接口不能。设计抽象时如果要求提供成员的实现细节选择抽象类

如果设计抽象是选择抽象类，在以后的版本中可以随意为抽象类添加成员，而接口方面没有这么灵活，只能修改代码

一个类可以同时实现多个接口，只能继承一个抽象类，如果要实现类似于多重继承的效果请选择接口

# 6. 各种类的作用：

## 内部类：

#### 创建非静态内部类：

**内部变量其实就是外部类一个成员而已**

public class Demo{

private class D{

int i = 1;

}

public static void main(String[] arg){

System.out.println("创建一个非静态内部类");

}

}

**2.在外部类中访问内部类**

**public class Demo{**

**public static void main(String[] arg){**

**D d = new D();**

**d.getD1();**

**}**

**}**

**class D{**

**private class D1{**

**int i = 1;**

**}**

**public void getD1(){**

**D1 d1 = new D1();**

**System.out.println("外部类访问内部类变量 i="+d1.i);**

**}**

**}**

**3.在外部类外中访问内部类**

**D.D1 d1 = new D().new D1();外部类外可以直接通过这，来访问内部类，但不好理解，故可以分解成下面的**

**D d = new D();**

**D.D1 d1 = d.new D1();**

**两个方法都可以用**

public class Demo{

public static void main(String[] arg){

//D.D1 d1 = new D().new D1();

D d = new D();

D.D1 d1 = d.new D1();

System.out.println("访问内部变量 i = "+d1.i);

}

}

class D{

class D1{

int i = 1;

}

}

**注意：外部类外访问时，如果内部是private，则使用不了 D.D1，只能在D中访问D1，在面试与编写是小心**

#### 局部内部类

4.内部类访问外部类

其实可以看做内部类为成员方法，可以直接调用其中成员变量等，当然如果重名，比如外部类有int i=9；内部类有 int i= 8；可以通过this区别，在内部类中this代表内部类，外部类.this.i代表外部类i

注意：**在内部类访问外部类时，访问的必须是成员，而不局部，当访问局部变量的时候，就必须要声明局部变量 final int I =9;**

**class D{**

**public void getVoid(){**

**final int i =9;**

**class D1{**

**public void getD1(){**

**System.out.println("内部类访问外部类i ="+i);**

**}**

**}**

**D1 d1 = new D1();**

**d1.getD1();**

**}**

#### 静态内部类 （与非静态类似）

main 是程序的入口，是一个特殊的方法，他区别于其他的静态方法，不能用对待其他静态函数的思想去看待main。

静态方法及变量属于整个类，数据将会被存储在公共区域，**非静态方法及变量属于对象。**所以在类static方法调用非static方法或变量时必须先创建对象，才能调用

#### 匿名内部类：

public class Demo{

public static void main(String[] arg){

**D d = new D(){**

**{ i = 9;**

**}//重写父类的方法**

**public void getD(){**

**System.out.println("匿名内部类中成员变量的值 i="+i);**

**}**

**};**

d.getD();

}

}

class D{

int i;

public void getD(){

System.out.println("内部类的父类的方法");

}

}

## 1.String类：

### charAt：

char charAt(int index)得到是指定索引的char值（从0开始） str.charAt(2)

endsWith: 测试此字符串是否以指定的后缀结束。

### StartsWith

### equalsIgnoreCase:

### equals:

spit: String[] split(String regex) 分割字符串

### substring：

### replace：

### trim：在聊天室中忽略空格（忽略前导和后到空格，不忽略其余空格）

### 例子：

String str = "wu,love,you";

System.out.println(str.charAt(2));

System.out.println(str.endsWith("u"));//

System.out.println(str.equalsIgnoreCase("WulianG"));//忽略大小写的比较，验证码上的需要使用

System.out.print(str.equals("wuliang"));//正常的字符串的比较大小

System.out.println(str.length());

String[] str1 = str.split(",");//分割字符串

for(int i = 0;i < str1.length;i++){

System.out.println(str1[i]);

String str3 = str1[i];

String str2 = str3.replace(str3, "\*\*\*");//替换字符串

System.out.println(str2);

}

System.out.println(str.substring(2));//截取字符串

System.out.println(str.substring(2,4));

String str4 = " jlkjk lkjlkjkl ";

System.out.println(str4.trim());

结果：,

* true
* false
* false11
* wu
* \*\*\*
* love
* \*\*\*
* you
* \*\*\*
* ,love,you
* ,l
* jlkjk lkjlkjkl

## 2.StringBuffer类：可变的，效率比起String高

### Append:追加

### Delete:删除

### Insert:插入

1. StringBuffer sb1 = new StringBuffer();//默认初始容量为16个字符
2. StringBuffer sb2 = new StringBuffer("sssssssssssssss");//不能直接赋值sb = "z"赋值是String类型
3. sb1.append("wu,no,yes");
4. System.out.println(sb1);
5. sb1.append(1);//追加字符串中的值，无任什么值都可以
6. System.out.println(sb1);
7. System.out.println(sb2);
8. System.out.println(sb1.delete(5,6));//删除操作
9. System.out.println(sb1.insert(5, ","));//插入操作

结果：wu,no,yes

wu,no,yes1

sssssssssssssss

wu,noyes1

wu,no,yes1

## 3.本数据类型的包装类：

由于基本数据类型是没有方法的，所以就不能传化成其他的；所以出现了包装类

四类八种：其中就是都有包装类：下面须记住的包装类

Charactrer 🡪 char 与 Integer 🡪 int 其余的只要首字母大写

Java5.0出现了自动拆箱和自动装箱

### 自动拆箱与自动装箱：

Int num = 10;

Integer I = num;//自动装箱

Int n = I;//自动拆箱

## 4.math类:

System.*out*.println(Math.*floor*(109.877));//返回最大double数

System.*out*.println(Math.*random*()\*100+1);//random()中0.0<=x<1.0

Math.round(Math.random()\*100)+1为整数

其余的自己查

## 5.Random类 util包中

## 6.File类(io包中)

### List:返回字符串数组，得到是子目录

### ListFiles:返回字符串数组，得到是本目录，子目录

例子：File files = **new** File("./a");

//String[] str = files.list();//只出现子目录

File[] str = files.listFiles();//得到本目录与子目录

**for**(**int** i = 0;i < str.length;i++){

System.*out*.println(str[i]);

}//结果1:aa

//bb

//结果2：.\a\aa

// .\a\bb

### getName：返回序列中最后一个名称

### isDirectory：测试抽象路径表示的文件是否在同一个目录

### ※例子：文件目录形式

### 使用File类操作目录

1. 示例1：**判断文件是否存在，不存在则创建，存在则删除**

**import** java.io.File; // 需要引入相应的包

**public** **class** FileTest01 {

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

File f = **new** File("c:/1.txt");

**if** (f.exists()) // 判断文件是否存在

f.delete(); // 删除文件（其实没有真正的删除）

**else**

**try** {

f.createNewFile(); // 创建新文件 用f.mkdir();可以真正创建

} **catch** (Exception e) {

e.printStackTrace();

}

System.*out*.println("文件名称：" + f.getName());

System.*out*.println("文件路径：" + f.getPath());

System.*out*.println(f.canWrite() ? "文件可写" : "文件不可写");

System.*out*.println(f.isDirectory() ? "是目录" : "不是目录");

System.*out*.println("文件长度:" + f.length() + "Bytes");

}

}

1. 示例2：**创建文件夹（在D盘目录下创建目录： source）**

**import** java.io.File;

**public** **class** FileTest02 {

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

File f = **new** File("d:" + File.*separator* + "source");

f.mkdir();

}

}

1. 示例3：**列出指定目录下的文件 （显示出C盘下全部内容）**

**import** java.io.File;

**public** **class** FileTest03 {

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

File f = **new** File("c:" + File.*separator*);

String[] str = f.list();

**for** (**int** i = 0; i < str.length; i++) {

System.*out*.println(str[i]);

}

文件目录显示问题

}

}

1. **示例4：列出文件夹的递归**

**import** java.io.File

**public** **class** StringTest {

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

File files = **new** File("./a");

StrTest.*show*(files,0);

}

}

**class** StrTest{

**public** **static** **void** show(File files,**int** lever){

String str ="";

**for**(**int** i = 0;i<lever;i++){

str += " ";

}//String[] str = files.list();//只出现子目录

File[] file = files.listFiles();

**for**(**int** i = 0;i < file.length;i++){

System.*out*.println(str+file[i].getName());//getName得到是该路径序列中最后一个名称

**if**(file[i].isDirectory()){

**if**(file[i].getName().equals("bb")){

lever = 0;

}

*show*(file[i],++lever);

}结果：aa

} aaa

} bbb

} bb

aaa

ddd

**File类只是针对文件进行操作，对文件内容操作须使用文件输入输出流**

### mkdirs与mkdir：创建文件夹

File files = new File("f./c/a");

Files.mkdirs() files.mkdir()

其中区别：mkdirs既可以在不存在父目录中创建，也可以与mkdir必须在存在父目录中一样创建

# 7.异常处理：

**异常与错误：** 异常：  在Java中程序的错误主要是语法错误和语义错误，一个程序在编译和运行时出现的错误我们统一称之为异常，它是VM（虚拟机）通知你的一种方式，通过这种 方式，VM让你知道，你（开发人员）已经犯了个错误，现在有一个机会来修改它。Java中使用异常类来表示异常，不同的异常类代表了不同的异常。但是在 Java中所有的异常都有一个基类，叫做Exception。

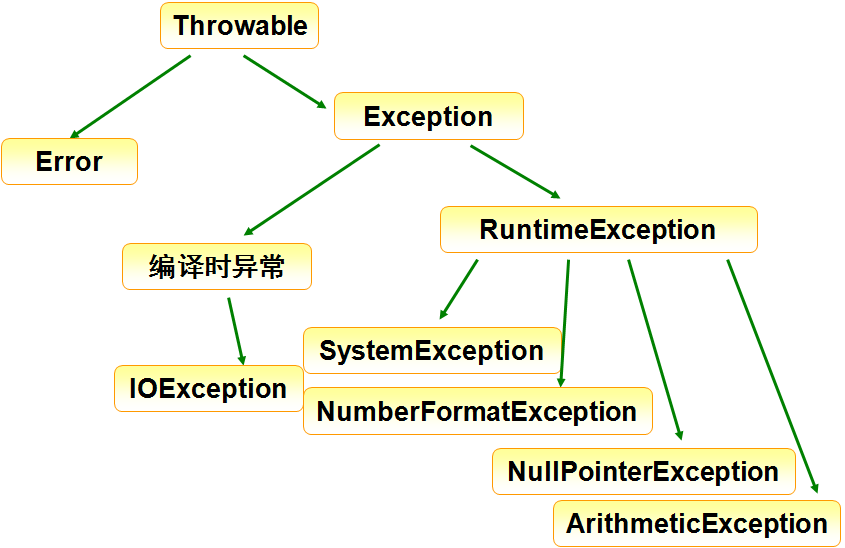
错误：它指的是一个合理的应用程序不能截获的严重的问题。大多数都是反常的情况。错误是VM的一个故障（虽然它可以是任何系统级的服务）。所以，错误是很 难处理的，一般的开发人员（当然不是你）是无法处理这些错误的，比如内存溢出。 和异常一样，在Java中用错误类来表示错误，不同的错误类代表了不同的 错误。 但是在Java中所有的错误都有一个基类，叫做Error。  
综上，我们可以知道异常和错误最本质的区别就是异常能被开发人员处理而错误时系统本来自带的，一般无法处理也不需要我们程序员来处理。

异常处理原因（try catch）：1.为了普通用户看的懂

2.使程序继续向下执行

程序会出现：Error 致命错误

Exception 代码有问题，可以解决的



一般情况下，自定义异常都继承自Exception类，但所有异常类的父类则是Throwable类

## Java中异常处理之程序错误分类

 Java程序中的错误有不同的性质,按照错误的性质可以将程序错误分成3类:

**1: 语法错** 违反语法规范的错误称为语法错(syntax error), 这类错误通常是在所以又可以称为是编译错误.引起语法错误的主要有:标识符未声明,表达式中的运算符与操作数类型不兼容,变量赋值时的类型与声明的类型不 匹配,括号不匹配,语句末尾缺少分号等等.

**2:语义错**  语义错误是指程序在语法上正确,但是在语义上存在错误,这样的错误就是语义错误(semantic error),语义错误是不被系统发现的,只,语义错误的程序称为运行错误(run-time error)Java 解释器在运行时能够发现语义错,一旦发现语义错Java将停止程序运行,并给出错误的位置和性质. 引起语义错误的主要有:输入数据的格式错,除数为0错,给变量赋予超出其范围的值等..

  3:逻辑错 如果程序通过编译,可以运行,但是运行的结果不是我们所期望的,那么这类错误就被称为逻辑错误(logic error). 引起这类错误的主要是: 循环条件不正确 循环次数不对 . 由于系统无法找到逻辑错,所以逻辑错误是很难以确定和排除的.

语法错也就是编译异常

语义错误就是运行期异常

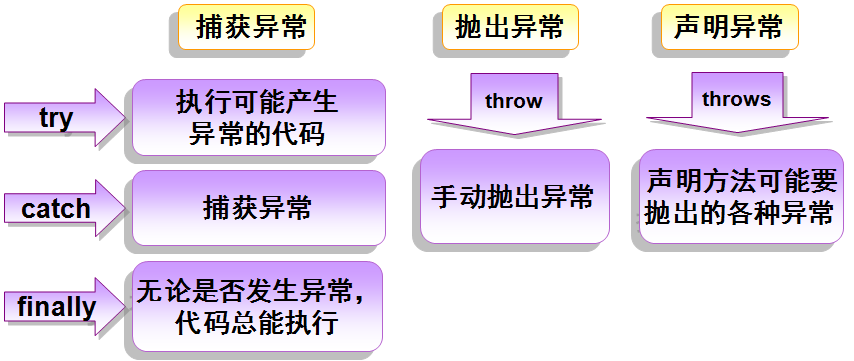
## Java是如何处理异常的：

1. Java有五个关键字：try ,catch,finally,throw,throws

注意：当try{} 中有return语句，还会执行finally{}吗，执行流程

会，由于return返回并跳出方法，但是，程序会先执行try中遇到return语句就跳转到finally执行，再执行return语句，跳出方法体

1. 在什么时候使用：



Exception 的子类

最特殊的：RuntimeException(运行期异常)

异常的分类：

编译期异常：就是在编译的时候就会检查，必须 try catch

运行期异常：就是运行期的时候检查 可不需要写 try catch

### Tyr catch finally:例子多重异常

### 多重异常

1. **程序中的一段代码可能会引发多种类型的异常，此时可以在一个try语句块后面跟多个catch语句块，分别处理不同的异常**
2. **程序运行时，系统会从上至下分别对每个catch语句块处理的异常进行检测，并执行第一个与异常类型匹配的catch语句。执行其中的一条catch语句后，后面的catch语句都将被忽略**
3. **示例：**

**public** **class** Demo004 {

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

**int** firstNum = 80, secondNum = 0;

**try** {

**int** result = firstNum / secondNum;

System.*out*.println("两个数字相除的结果是:" + result);

} **catch** (ArithmeticException e) {

System.*out*.println("算术异常:");

} **catch** (NumberFormatException e) {

System.*out*.println("数字转换异常");

} **catch** (ArrayIndexOutOfBoundsException e) {

System.*out*.println("数组下标越界异常");

} **catch** (Exception e) {

System.*out*.println("其他异常");

}

}

}

在上面的代码示例中，当发生异常时，会从第一个catch开始进行对比，如果发生的异常类型是第一个，就忽略后面的几个catch，如果不是第一个，就对比第二个异常类型，如此类推。

1. **多重异常在处理时，捕获范围小的异常必须放在捕获范围大的异常之前，否则程序在编译时会出现错误，并提示“执行不到的 NumberFormatException 的 catch 块。它已由 Exception 的 catch 块处理。”。如：**

**public** **class** Demo005 {

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

**int** firstNum = 80, secondNum = 0;

**try** {

firstNum = Integer.*parseInt*("A");

secondNum = Integer.*parseInt*("B");

} **catch** (ArithmeticException e) {

System.*out*.println("算术异常。");

**异常将被Exception捕获，后面的catch不会捕获任何异常**

} **catch** (Exception e) {

System.*out*.println("其他异常。");

} **catch** (NumberFormatException e) {

System.*out*.println("数字转换异常。");

}

}

}

### 没有匹配catch块的处理

1. **如果程序中产生的异常类型和以上catch块中的异常类都不匹配，程序就不能捕获并处理此异常**

**将产生**

**NumberFormatException**

**异常**

**public** **class** Demo006 {

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

**try** {

**int** firstNum = Integer.*parseInt*("A");

} **catch** (ArithmeticException e) {

System.*out*.println("算术异常。");

} **catch** (ArrayIndexOutOfBoundsException e) {

System.*out*.println("数组下标越界异常。");

}

}

}

1. 上面的代码中，因为没有匹配的catch块，所以将不会捕获异常，程序会因为发生异常而停止
2. 如何解决呢：那就要**处理所有异常**
   * + 1. **在程序中对异常进行处理时，建议使用Exception进行异常捕获，这样程序中出现的全部异常均可能被处理。**

**public** **class** Demo007 {

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

**int** firstNum = 80, secondNum = 0;

**try** {

**int** result = firstNum / secondNum;

System.*out*.println("商的结果是：" + result);

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

## 2.常见的异常类型

|  |  |
| --- | --- |
| **异常类** | **说明** |
| **NullPointerException** | **访问Null对象的方法** |
| **IllegalArgumentException** | **接收非法参数** |
| **ClassNotFoundException** | **不能加载所需要的类** |
| **ArithmeticException** | **算术运算异常，如除数为零** |
| **ArrayIndexOutOfBoundsException** | **数组下标越界** |
| **NumberFormatException** | **格式化数据异常** |
| **IOException** | **文件读写异常** |

1. **NullPointerException**

**对象为空，调用方法则出现空指针异常**

1. **NumberFormatException :**

**public** **class** StringTest {

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

String stuName = " tom";

**int** result = Integer.*parseInt*(stuName);

}

}

1. IoException

**try** {

InputStream is = **new** FileInputStream("./ddd/dd");

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

## 3. 抛出异常：

Throw 自定义异常是需要使用抛出异常给调用它环境

Throws 在方法中抛出，不需要使用try catch

### Throws方法抛出：

Public void show throws Excetpion(){异常}

注意：这只是一种简单教学是才能使用的，尽量不用

### 自定义异常：(一定要继承RuntimeException)

**当程序员检查异常时，要正确找到异常的位置，使用自定义异常，可以最快的找到异常处**

1.先创建异常需继承RuntimeException

**package** com.wuliang.excepiton;

**public** **class** PasswordsException **extends** RuntimeException {}

2.抛出异常，在需要用该异常是

**if**(!password.equals(rs.getString(1))){

**throw** **new** PasswordsException();}

3.调用抛出异常的方法是，需try catch接收

**try**{

UserDao4MySqlImpl.*setUser*(username, password);

}**catch** (UsernameException e){

request.setAttribute("msg","用户名错误 ！ "); }**catch** (PasswordsException e){

request.setAttribute("msg","密码错误 ！");}

在一个程序中使用自定义异常：

import java.util.Scanner;

public class Tset{

public static void main(String args[]) throws Exception {

Scanner scanner = new Scanner(System.in);

int age = scanner.nextInt();

if ((age < 0) || (age > 100))

throw new MyException("输入的年龄非法");

}

**⑶ 使用自定义异常类**

}

class MyException extends RuntimeException {

public MyException(String msg) {

super(msg);

**⑵ 构造方法中调用父类构造方法**

## 4.使用细节

1. **一个方法被重写时，重写它的方法必须抛出相同的异常类或异常子类**

示例代码：

**class** Father {

**public** **void** speak() **throws** IOException {

System.*out*.println("Father");

}

}

**class** Son **extends** Father {

**public** **void** speak() **throws** Exception {

System.*out*.println("son");

}

}

因为：**Exception不是父类方法抛出的异常IOException的子类，程序编译异常**

所以上面的代码在编译时就会提出有错误：

异常 Exception 与 Father.speak() 中的 throws 子句不兼容

1. **如果父类方法抛出多个异常类，则重写方法必须抛出父类方法异常类的一个子集，即不能抛出新的异常**

示例代码：

**class** Father2 {

**public** **void** speak() **throws** IOException, NullPointerException {

System.*out*.println("Father");

}

}

**class** Son **extends** Father2 {

**public** **void** speak() **throws** SQLException {

System.*out*.println("son");

}

}

**因为：SQLException不是父类方法抛出的异常的子集，程序编译异常**

所以上面的代码在编译时就会提出有错误：

异常 SQLException 与 Father.speak() 中的 throws 子句不兼容

# 8.容器(数组与泛型和集合)

## 数组：

**数组是一个变量，用于存储多个相同数据类型的一组数据**

声明一个变量就是在内存空间中划出一块合适的空间，然后将数据存储在其中，而**声明数组就是在内存空间中划出一串连续的空间**，存储多个相同数据类型数据

#### 数组的特点：

* + - 1. 不管有多少元素，都只有一个名称，就是数组名
      2. 数组元素按编号循序存储，编号从0开始的整数，编号也就是元素**下标（索引）**
      3. 每个元素都是通过唯一对应的索引来查找出
      4. 数组的大小就是定义的长度，就是length，new int[4];4就是数组的长度
      5. 数组一旦定义就会改变

#### 声明数组：告诉系统什么数据类型？

**Int**[] nums ;

**int** nums[];

#### 分配空间：告诉系统分配多大的空间

数组为引用数据类型

Int[] nums = new int[6];

#### 赋值：

* 1. 边声明边赋值

**int** nums[] = {1,2,3,4};

**int** nums1[] = **new** **int**[]{1,2,3,4};

* 1. 动态键盘输入（在c与c++中很多基本测试输入，而java在我七个月中，基本上没用到过）

**int** nums2[] = **new** **int**[5];

Scanner sc = **new** Scanner(System.*in*);

**for**(**int** i=0;i<5;i++){

nums2[i] = sc.nextInt();

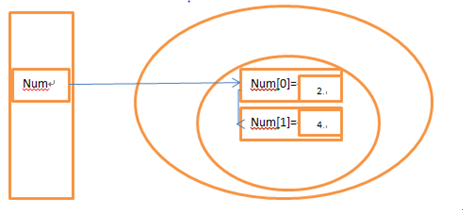
}

其中系统默认数组：引用数据为null，bool false ，char 空字符串 ，其余默认为0

#### 数组保存数据的内存位置

**下面错误内存，还不知道非常清楚的知道内存位置**

int[] nums = {new int(),new int(),……}



Length是数组的属性，如图就可以知道，数组的中的元素是存放在堆区中，所以，只要是在数组，就可以知道用equals判断

**修改后：**

**int**[] ch = {45,84,45,75};

String[] str = {"23","234","wrw"};

System.*out*.println(ch[1]==84);

**if**("23".equals(str[0])){

System.*out*.println("数组中字符串equals判断是否相等");

}

**if**("23"==str[0]){

System.*out*.println("数组中字符串能用等于号判断是否相等");

}

小结：从上面可知，其中**基本类型数据来说除String既可以用equals比较两字符串的Unicode序列是否相当，也可以“==”判断两字符串地址;其他”==”都是用等于号；所以上面结论错误**

**可以判断出，数组是开辟一串内存空间，每一个空间都还是根据原本数据类型来存储，根据我判断数组只是给其一连续地址，可以根据索引查找到**

**二维数组：**  
1、二维数组：是一种平面的二维结构，本质上是数组的数组。

1. 格式：type[][] 数组名；例如：int[][] a;//声明，没有初始化

2. 固定行列长度声明，初始为默认值（由数据类型决定）。

格式：type[][] 数组名 = new type[一维长度][二维长度];

例：int[][] a = new int a[2][3]; //注意：a[行号]，是一个一维数组。比如上例中a[0]和a[1]都是一维数组

c)动态声明  
动态声明首次声明时只声明一维的长度，二维的长度动态指定（一旦指定就不可改变）。

格式：  
type[][] 数组名 = new type[一维长度][];  
数组名[一维索引号] = new type[该行列长度];//可行原因，见b)里面的注意事项

例如：int[][] a = new int[3][];  
a[0] = new int[2];//初始一维索引号为1的数组，长度为2  
a[1] = new int[3];  
a[2] = new int[1];

注意：数组完全初始化后才能赋值，下例是错误的。  
int[][] a = new int[2][];  
a[0][1] = 1;//错误，第二维没有初始化

d)常量初始化声明

格式：type[][] 变量名 = new type[][]{{枚举行1所有元素的值},..,{枚举行n所有元素值}};//每行的元素个数不必相同。  
type[][] 变量名 =  {{枚举行1所有元素的值},..,{枚举行n所有元素值}};

int[][] a = new int[][]{{1,2},{3,4},{5,6}};//各行长度相同  
int[][] b = new int[][]{{1,2,3},{4},{5,6,7,8}};//各行长度不同

注意：如果指定了初始化值，就不能指定长度。  
int[] a = new int[2][1]{{1,2},{1}};//错误

e)常量动态始化声明

实际上就是上两种方法的综合使用，格式如下：

type[][] 变量名 = new type[一维长度值][];  
变量名[一维索引] = new type[]{枚举该索引下所有数组元素的常量值};

例如：  
int[][] a =new int[2][];  
a[0] = new int[] {1,2,3};

注意：数组常量只能在初始化时使用。  
int[][] a =new int[2][];  
a[0] ={1,2,3,4};//错误，只能在初始化时使用

#### Arrays

**此类作用用来操作数组（比如排序和搜索）的各种方法**

**copyOf:**

**int**[] nums = Arrays.*copyOf*(ch, 5);

**for**(**int** u : nums){

System.*out*.println(u);

}

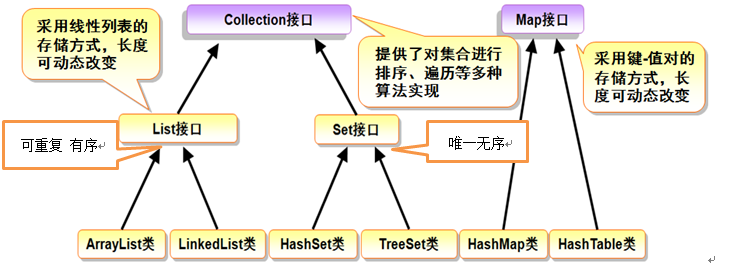
数组与容器的不同：

数组定义了后是不可改变的 String[] nums = new String[5];

容器定义是可以改变的 Collection

## 容器：

### 基本概念：



#### Collection，一个独立元素序列

**List 必须按照插入的循序保存元素，可以根据源代码**

ArrayList 查询快LinkedList 更新快只有底层原理不同，其中由于ArrayList类似于数据结构中顺序表，所以存取时容易，插入和删除时消耗资源快，LinkedList类似于双向链表，可以快速插入与删除，而存取消耗资源多

**public** **boolean** add(E e) {

ensureCapacityInternal(size + 1); //插入时索引size+1

elementData[size++] = e;//对象保存

**return** **true**; }

再查看elementDate可以知道，是用

**public** ArrayList(Collection<? **extends** E> c) {

elementData = c.toArray();

**Set 不保存重复的元素，而且**

如果此 set 没有包含满足 (e==null ? e2==null : e.equals(e2)) 的元素 e2，则向该 set 中添加指定的元素 e。如果此 set 已经包含该元素，则该调用不改变此 set 并返回 false。结合构造方法上的限制，这就可以确保 set 永远不包含重复的元素

Set接口有很多实现类。**HashSet查询快，TreeSet自动排序，LinkedSet按照添加顺序存储。**使用HashSet保存元素可以提高获取元素的效率；使用LinkedHashSet保存元素可以保留元素存入顺序；使用TreeSet保存元素可以实现元素的自动排序功能，当然可以按照其指定的方式排序。

HashSet中而看源代码：

**public** HashSet() {

HashMap map = **new** HashMap<>(); }

**public** **boolean** add(E e) {

**return** map.put(e, *PRESENT*)==**null**; }// *PRESENT是Object*

*所以了解到，Set就是Map实现，所以性能差不多*

**Queue 按照排队规则来确定对象产生的顺序，不常用**

Map。一组成对 键值对 对象，可以通过键来查值。ArrayList可使用数字来查找值，因此某种意义上，将数字与对象关联在一起。映射表使用另一个对象来查找某个对象，被称为“关联数组”，因为将某些对象与另外一些对象关联在一起，或者被称为“字典”

**HashMap，TreeMap，LinkedMap，**与HashSety一样，HashMap提供最快的查找技术，也没有按照任何明显的顺序来保存，而TreeMap按照比较结果升序保存键，LinkedMap则按照插入顺序保存键，同时还保留HashMap查询速度

### list：

1.方法

List list = **new** ArrayList();

list.add("吴亮");//add增加数据

list.add(1,'o');//list有两种类型增加

list.add(111);

System.*out*.println(list.get(0));//get得到数据

System.*out*.println(list.contains("吴亮"));//contains包含 判断列表是否存在指定数据，boolean

System.*out*.println(list.indexOf('o'));//indexOf 指定元素在该列表中索引，如果不存在返回-1

System.*out*.println(list.remove(2));//remove移除一个指索引元素

list.set(1, 999);//替换

System.*out*.println(list.get(1));

System.*out*.println(list.size());//列表大小

list.clear();

2.

3.用Collections类：

**package** com.wuliang;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** CollectionsTest {

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

T.*show*();

List<T> lists = **new** ArrayList<T>();

lists.add(**new** T("w",12));

lists.add(**new** T("wu",13));

lists.add(**new** T("wul",14));

Collections.*sort*(lists);//会报错，要接口Comparable 排序的时候，必须要知道其中大小比例

System.*out*.println(lists+" ");

**for**(Iterator<T> i = lists.iterator(); i.hasNext();){

T ch = i.next();

System.*out*.println("name="+ch.name+" "+"age="+ch.age);

}

}

}

**class** T **implements** Comparable{

String name;

**int** age;

**public** T(String name,**int** age){

**this**.name = name;

**this**.age = age;

}

**public** **static** **void** show(){

List<Character> list = **new** ArrayList<Character>();

list.add('1');

list.add('3');

list.add('6');

list.add('0');

System.*out*.println(Collections.*binarySearch*(list, '1'));;

Collections.*sort*(list);

**for**(Iterator<Character> i = list.iterator(); i.hasNext();){

**char** ch = i.next();

System.*out*.println(ch);

}

}

@Override

**public** **int** compareTo(Object o) {

**if**(o **instanceof** T){

T t = (T)o;

**if**(**this**.age > t.age){

**return** 1;

}**else if(this.age < t.age)**{

**return** -1;

}else{return 0}

}**else**{

**return** 0;

}

}//如果不重写toString方法时，则会导致得到 com.wuliang.T@470285d8 需要的

**public** String toString(){

**return** "name="+**this**.name+" "+"age="+**this**.age;

}

}

2.Set：

其实set是唯一无序的，所以就没有get,indexOf,set替换等

### 迭代器---容器的遍历：

所有实现Collection接口的类都有一个iterator方法，这个方法会返回一个实现Iterator迭代器接口的类的对象

List用get（）可以获取值，但是当需要修改代码成Set，那么就用for循环就需要重新修改代码，而用迭代器只是一种容器，不需要关系容器类型

**迭代器（也是一种设计模式），轻量级对象：创建它的代价小**

例如：Java的Iterator只能单向移动

1).使用方法iterator()要求容器返回一个Iterator。Iterator将准备返回序列的第一个元素

2）使用next()获得序列中的下一个元素

3）使用hasNext() 检查序列中是否还有元素

4）使用remove容器中删除元素，使用在next方法下

例子：

**for**(Iterator i = list.iterator();i.hasNext();){

Object o = i.next();//如果是用泛型则必须String str = i.next()

**if**(list.equals('o')){

i.remove();//用容器中迭代是删除元素，用remove}

System.*out*.println(o);}

ListIterator

listIterator是一个更加强大Iterator的子类型，只能用于各种List类的访问，尽管Iterator只能单向向前移动，而ListIterator可以双向移动

List<String> list = **new** ArrayList<String>(Arrays.*asList*("one","two"));

ListIterator<String> li = list.listIterator();

**while**(li.hasNext()){

System.*out*.print(li.next()+" "+li.nextIndex()+" "+li.previousIndex()+" ;" ); }

System.*out*.println();

**while**(li.hasPrevious()){

System.*out*.print(li.previous()+" ;"); }

nextIndex得到元素下一个索引，previousIndex得到元素上一个索引，hasPrevious判断元素后一个元素是否存在，previous得到元素上一个元素

### 增强循环：

增强的for循环在迭代容器的时候，内部也是使用Iterator方法；缺点：不存在索引，不能删除，例子：

**for**(Object o1 : list){

System.*out*.println(o1);}

### 泛型：（只要是容器就一定使用泛型）

为啥需要泛型：1.什么都放入容器，就没有意义（将容器分类）

2.放进的是Object，出来的是Object

List<String> list = new ArraryList<String>

### Map：其实就是可以自定义索引的容器

Map 就是键值对，键是唯一无序的可以用Set (map.KeySet)，值是就直接用Collection(map.values);其中显示出键值是，可以用map.entrySet()方法，以下

例子：

**package** com.wuliang;

**import** java.util.Collection;

**import** java.util.HashMap;

**import** java.util.Iterator;

**import** java.util.Map;

**import** java.util.Set;

**public** **class** StringTest {

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

ListDemo.*show*();

}

}

**class** ListDemo{

**public** **static** **void** show(){

Map<String,Integer> map = **new** HashMap<String,Integer>();//键值对 key - value == entry Map保存的是entry

map.put("1", 1);

map.put("6", 7);

map.put("2", 5);

map.put("2", 4);//Map可以重复，但是是覆盖，无序

System.*out*.println(map.get("2"));

Collection<Integer> coll = map.values();

System.*out*.println(coll);//[345, 4, 1, 7, 34]

// for(Integer i : coll){

// System.out.println(i);

// }

**for**(Iterator<Integer> i = coll.iterator(); i.hasNext();){

**int** c = i.next();

System.*out*.println(c);

}

Set<String> set = map.keySet();

System.*out*.println(set);//set =[3, 2, 1, 6, 5]

//也上一致

Set<Map.Entry<String,Integer>> se = map.entrySet();

**for**(Iterator<Map.Entry<String,Integer>> it = se.iterator();it.hasNext();){

Map.Entry<String,Integer> e = it.next();

System.*out*.println("key="+e.getKey()+" value= "+e.getValue());}}}//结果：

// key=2 value= 4

// key=1 value= 1

// key=6 value= 7

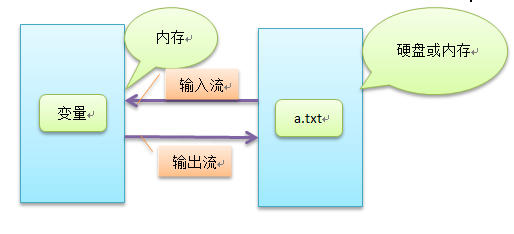
// key=5 value= 34

# 9.IO流：

## 1．IO流介绍：

输入input 输出output **以java内存为基**

1.内存与文件中存在交流，通过管道（流）流过字节



2.IO流的分类

内存与内存之间的交流：

1）：字节流（一个字节输入\出）

2）：字符流（一个字符输入\出）

功能性：

1）：节点流 （节点流的名称+流）

2）：处理流 在节点流之间使用一个处理功能（不能再两节点之间）

四个抽象类：

InputStream（字节流） Reader（字符流）

OutputStream Writer

使用字节流和字符流读取文件与写入文件

### 使用流进行文件操作：

#### 使用File类进行文件与目录的操作

### 例子1：读取文件与写入文件

**import** java.io.\*;

**public** **class** FilesTest01 {

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

FileInputStream fis = **null**;

FileOutputStream fos = **null**;

**try** {

fis = **new** FileInputStream("f:/a.txt");

fos = **new** FileOutputStream("f:/b.txt");

**int** i = -1;

**while**((i = fis.read()) != -1){//read读取字节

fos.write((char)i);//写入文件

}} **catch** (IOException e) {

e.printStackTrace();

}**finally**{

**try** {fis.close(); fos.close();

} **catch** (IOException e) {

e.printStackTrace();} }}}

### 例子2：在控制台中输入文件中的内容

FileInputStream fis = **null**;

fis = **new** FileInputStream("f:/a.txt");// 第1步：创建输入流对象

**byte**[] data = **new** **byte**[1024]; // 创建用于保存数据的字节数组

// 第2步：将所有数据读取到字节数组中

**int** i = fis.read(data);//从输入流中读取一定数量的字节，并将其存储在缓冲区数组 b 中

// 将读取出的数据解析成字符串

String s = **new** String(data, 0, i);//(byte[] bytes, int offset, int length)

System.*out*.println(s); // 在控制台输出

fis.close();// 第3步：关闭输入流对象

### 例子3：在程序中写入到文件中

FileOutputStream fos = **new** FileOutputStream("F:/a.txt");

String str = "HelloWorld";

fos.write(str.getBytes());

## **缓冲区：**

为啥使用缓冲区：使用字节流等都是微量流入，容易伤害硬盘，而且效率不高，就出现了缓冲区

缓冲区：8kb内存（当文件输出时，最后没有达到8kb也会输出）

**BufferedInputStream 输入 BufferedOutputStream 输出 例子与上类似**

## 转换流：实现字节向字符流转换

* + - 1. **InputStreamReader** 
         1. **实现将字节输入流转换为字符输入流**
         2. **为了达到最高效率，可要考虑在 BufferedReader 内包装 InputStreamReader。例如：**

**BufferedReader in = new BufferedReader(new InputStreamReader(System.in));**

* + - 1. **OutputStreamWriter** 
         1. **实现将字符输出流转换为字节输出流**

**Writer wt = new BufferedReader(new OutputStreamReader(System.out));**

### 例子：有关控制台输入：

**import** java.io.\*;

**public** **class** CollectionsTest {

**public** **static** **void** main(String[] args) **throws** Exception{

String s = **null**;// 用于保存用户输入的字符串

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(System.*in*));//将控制台输入的字节转换为字符

**do** { System.*out*.println("请输入：");// 提示用户输入

s = br.readLine();// 按行输入

System.*out*.println(s);// 在控制台输出用户的输入

} **while** (!s.equals("quit"));

System.*out*.println("结束输入！");

br.close();}}

## 数据流：

DataOutStream DataInputStream

DataOutputStream dos = **new** DataOutputStream(**new** FileOutputStream("F:/b.txt"));

dos.writeUTF("Hello");

DataInputStream dis = **new** DataInputStream(**new** FileInputStream("F:/b.txt"));

System.*out*.println(dis.readUTF());

dis.close();

dos.close();//由于用数据流处理了，而不能在其他的文件中找到

## 序列化：Object流：（常用在于游戏保存）

[**新手问：java中的Serializable接口到底是做什么用的？**](http://www.chinaunix.net/old_jh/26/395684.htmlADSESSION=1352179672&ADTAG=CLIENT.QQ.4735_.0)

序列化：**把一个对象通过字节流写出来**

反序列化：通过IO把序列化的文件读取出来（直接把其封装成一个对象）

序列化必须实现接口Serializable;

**解释1：举个例子，你编写了一款游戏，保存记录时把所有状态一一保存非常麻烦，这时就可以使用Serializable（序列化接口），它的作用是可以将一个对象实例序列化，序列化后你可以选择将它保存在你需要的位置。   
相对的，读取后生成的对象所有属性（除了设置为瞬时值的属性）将和原对象的属性相同（只是内存地址不同）。   
这样可以方便的将一个java对象写入到磁盘中，保存该对象的所有状态！   
值得注意的是序列化的对象中包含的属性和其他对象都需要实现序列化接口，不然无法正常序列化！   
在hibernate里，并非所有的实体类必须实现序列化接口，因为在hibernate中我们通常是将基本类型的数值映射为数据库中的字段。而基础类型都实现了序列化接口（String也实现了）。   
所以，只有在想将一个对象完整存进数据库（存储为二进制码），而不是将对象的属性分别存进数据库，读取时再重新构建的话，就可以不用实现序列化接口。  
实现了Serializable，可以方便保存数据。**

**解释2：**

序列化是把一个对象的状态写入一个字节流的过程，它执行RMI，RMI允许一台机器上的JAVA对象调用不同机器上的JAVA对象方法，对象可以作为参数提供给那个远程方法，发送机序列化该对象并传送它，接收机执行反序列化。   
序列化和反序列化的关系图表可形成包含循环引用的顺序图表。这是整个序列化的总体思想。   
而Serializable接口属于支持序列化的一个接口，只有一个实现它的对象可以被序列化工具存储和回复，Serializable接口没有定义任何成员，只用来表示一个累可以被序列化，若该类可以序列化，那么它的所有子类都可以。

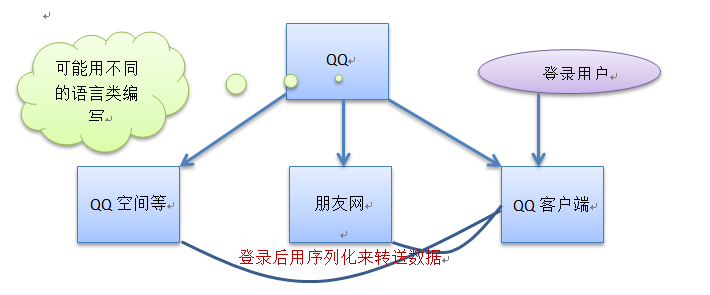
Object serialization的定义：   
Object serialization 允许你将实现了Serializable接口的对象转换为字节序列，这些字节序列可以被完全存储以备以后重新生成原来的对象。    
  
serialization不但可以在本机做，而且可以经由网络操作（就是猫小说的RMI）。这个好处是很大的----因为它自动屏蔽了操作系统的差异，字节顺序（用Unix下的c开发过网络编程的人应该知道这个概念，我就容易在这上面犯错）等。比如，在Window平台生成一个对象并序列化之，然后通过网络传到一台Unix机器上，然后可以在这台Unix机器上正确地重构这个对象。   
  
Object serialization主要用来支持2种主要的特性：   
1。Java的RMI(remote method invocation).RMI允许象在本机上一样操作远程机器上的对象。当发送消息给远程对象时，就需要用到serializaiton机制来发送参数和接收返回直。   
  
2。Java的JavaBeans.   Bean的状态信息通常是在设计时配置的。Bean的状态信息必须被存起来，以便当程序运行时能恢复这些状态信息。这也需要serializaiton机制。   
  
二。sakulagi和rollingpig说的持久化我也说一下。   
我觉得你们说的应该是英文里的persistence.但是Java语言里现在只支持lightweight persistence,就是轻量级持久化，这是通过serialization机制来实现的。   
  
persistence是指一个对象的生命周期不由程序是否执行来决定，即使是在程序终止时这个对象也存在。它把一个serializable的对象写到磁盘（本机或其他机器上的非RAM存储器），并在程序重新调用时再读取对象到通常的RAM存储器。

解释3：

Hibernate 上：

**hibernate有二级缓存，缓存会将对象写进硬盘，就必须序列化，以及兼容对象在网络中的传输等等...  
目的：将二级缓存中的内容持久化保存下来，便于恢复缓存的信息，hibernate的缓存机制通过使用序列化，断定应该是基于序列化的缓存，如没有 serializable接口，在序列化时，使用objectOutputStream的write（object）方法将对象保存到文件时将会出现异常。**

**分布式开发：**



**hibernate中实体类的序列化就是将类序列化后存入磁盘并释放内存空间。**

### 例子：序列化

**import** java.io.\*;

**import** java.util.\*;

**public** **class** CollectionsTest {

**public** **static** **void** main(String[] args) **throws** Exception{

List<Count> list = **new** ArrayList<Count>();

list.add(**new** Count("wu",19,78));

list.add(**new** Count("wl",49,53));

**ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("F:/a.txt"));**

oos.writeObject(list);

oos.close();

**ObjectInputStream ois = new ObjectInputStream(new FileInputStream("F:/a.txt"));**

List<Count> list1 = (List<Count>)ois.readObject();//由于object流不知道那个类是被序列化的，故必须要强制转换成该类

**for**(Count l : list1){

System.*out*.println("name= "+l.name+'\t'+"age="+l.age+'\t'+"time= "+l.time);}

ois.close();

}

}

**class** Count **implements** **Serializable**{//Object流必须加接口Serializable

String name;

**int** age;

**transient** **int** time;//transient是暂时的，重新写入

**public** Count(String name, **int** age, **int** time) {

**this**.name = name;**this**.age = age; **this**.time = time;}}

思路：1.先写入Count类，序列化前提必须实现接口Serializable，其中time要用transient修饰

2.再写List容器来保存Count类的数据的对象

3.再序列化用 oos.writerObject(对象)；

4.反序列化时必ois.readObject()强制转换为List<Count>，后遍历

# 10.Java多线程机制：

多线程：多个程序同时进行

线程：一个进程里多条执行路径

1.）操作系统真的同时运行多个程序（进程）？？？

不是，由于cpu在同一个时间点上只运行一个进程，因为cpu运算的速度非常快，每一次都分成多个程序，cpu不同进程间不停的切换的，速度过快，人感觉不到

双核操作系统可以同时运行两个进程

2.）线程之间可以切换

1. **线程是比进程更小的执行单位，是在进程的基础上进行的进一步划分；多线程是指一个进程在执行过程中可以产生多个线程**
2. **Java中实现多线程有三种方式：继承Thread类、实现Runnable接口**
3. **继承Thread类不适合于多个线程共享资源**
4. **实现Runnable接口可以方便地实现资源共享**
5. **解决线程中资源共享的安全问题可以使用同步代码块和同步方法实现**
6. **线程在整个生命周期中具有创建、运行、阻塞、死亡4种状态**

### 在程序中实现多线程：

### 实现接口 Runnable和继承类 Thread(也已经实现了Runnable)

T1 t1 = new T1();

new Thread(t1).start();//用这样创建就可以直接的开启线程

t1.run();//此时线程并没有开始，所以不会进行多线程

必须要实现创建类Thread的对象，再开启start

例子：**public** **class** CollectionsTest {

**public** **static** **void** main(String[] args) **throws** Exception{

Thread th1 = **new** Thread(**new** TestRunnable());

Thread th2 = **new** Thread(**new** TestRunnable2());

th1.start();

th2.start();}}

**class** TestRunnable **implements** Runnable{

**public** **void** run() {

**for**(**int** i = 0; i< 100;i++){

System.*out*.println("----------run-----------"+i);}}}

**class** TestRunnable2 **extends** Thread{

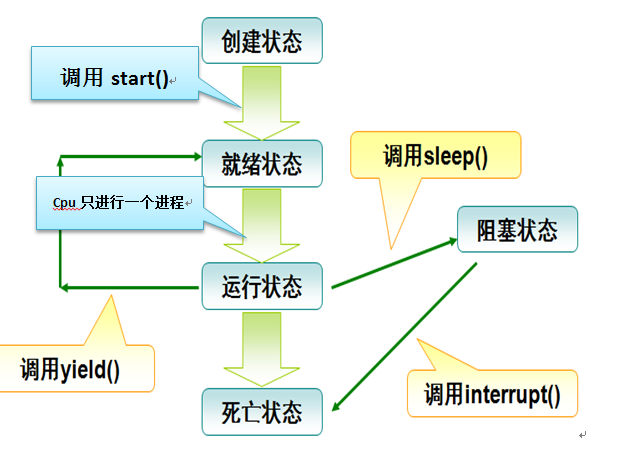
**public** **void** run(){

**for**(**int** i = 0; i< 100;i++){

System.*out*.println("----------thread-----------"+i);}}}

## 线程生命周期

1. **线程生命周期包含如下状态：**



* + - 1. **调用yield()：暂停当前正在执行的线程对象，和其他程序相同执行概率。**
      2. **调用interrupt()：判断是否中断线程。如果中断了，返回true**
      3. **调用sleep()：在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）**

### 实现Thread 类方法：

**public** **class** CollectionsTest {

**public** **static** **void** main(String[] args) **throws** Exception{

T12 t1 = **new** T12();

System.*out*.println(t1.isAlive());

t1.start();//public final boolean isAlive()测试此是否处于活动状态；如果线程未终止，就处于活动状态

System.*out*.println(t1.isAlive());

System.*out*.println(t1.*interrupted*());//public static boolean interrupted()判断线程是否中断

//t1.setPriority();public final void setPriority更改线程优先级

T23 t2 = **new** T23();

t2.start();}}

**class** T12 **extends** Thread{

**public** **void** run(){

**for**(**int** i = 1; i < 100; i++){

// if(i == 50){

// try{

// Thread.sleep(3000);//休眠多少时候，定时醒

// }catch(InterruptedException e){}

**if**(i%2==0){

Thread.*yield*();//使下一个线程执行 Thread中yield()是让当前线程暂停,转入就绪状态,让系统的线程调度器重新调度一次

}

System.*out*.println("-------------T1-----------"+i);}}}

**class** T23 **extends** Thread {

**public** **void** run() {

**for**(**int** i = 1;i < 100; i++){

System.*out*.println("-------T2----------"+i);}}}

## 线程同步：（有关锁的问题）：

什么是同步：

在编写多线程程序时，经常需要使用同步技术，其根本原因在与线程之间存在与共享资源，在多个线程同时修改一个整数的过程中，容易发生数据错误，这是典型的非同步问题。在独立测试中不容易发现，调试难度加大

**同步是一种控制多线程依次使用共享资源的能力，从而避免了错误的发生**

线程同步：同一个对象被多个进程所用

1.死锁问题：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。

由于线程被无限期地阻塞，因此程序不可能正常终止。

死锁产生根本原因是资源竞争。如果若干个线程正常运行，遇到彼此需要对方的占用资源，就可能发生线程死锁。如果若干个线程异常运行，资源请求与释放的循序不当，也可能发生死锁

死锁的条件：

1. 资源互斥 资源仅能让一个线程使用不支持多个线程
2. 请求保持 已经拥有的线程在请求其他的资源是没有释放当前资源
3. 不能剥夺 已经获得资源在自主释放前不能使用其他的手段剥夺
4. 循环等待 各个线程对资源的需求构成一了循环

### 2.哲学家吃饭的问题

3.生产者与消费者的问题

public class ThreadTest {

public static void main(String[] args) {

Test test = **new** Test(1);

**new** Thread(test).start();

Test test1 = **new** Test(2);

**new** Thread(test1).start();}}

**class** Test **implements** Runnable{

**int** i ;

**static** Integer *o1* = 12;

**static** Integer *o2* = 13;//不能用int类型，有static int i = 12；i是变量

**public** Test(**int** i){

**this**.i = i;}

**public** **void** run(){

**if**(i == 1){

**synchronized**(*o1*){

System.*out*.println("AAAAAAAAAAAAAAAA");

**synchronized**(*o2*){

System.*out*.println("BBBBBBBBBBBBBBBBBB");}}}

**else** **if**(i == 2){

**synchronized**(*o2*){

System.*out*.println("CCCCCCCCCCCCCCCCC");

**synchronized**(*o1*){

System.*out*.println("DDDDDDDDDDDDDDDDDDDD"); }}}}}

**一、当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。**

**二、当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它**

**synchronized(this)同步代码块的访问将被阻塞。**

**三、然而，当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问**

**该object中的除synchronized(this)同步代码块以外的部分。**

**四、第三个例子同样适用其它同步代码块。也就是说，当一个线程访问object的一个synchronized(this)**

**同步代码块时，它就获得了这个object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。**

**五、以上规则对其它对象锁同样适用**

**面试题解：**

**简述Object类中提供的线程的方法及作用**

**Notify（） 唤醒在此对象监视器上等待的单个线程**

**notifyAll（） 所有线程**

**wait（） 线程处于等待，直到其他线程调用notify或notifyAll**

**wait（time）超过指定的时间被唤醒或者notify等（毫秒单位）**

**那个接口可以创建有返回值的线程**

**Callable（）**

**那个类可以用来创建线程池？**

**Exceutors创建一个新的线程是有一定代价，如果系统中需要创建大量短周期线程，就应该使用线程池，用Exceutors （例如使用 newCachedThreadPool（）创建的线程池可以在没有可用的线程下创建一个新的线程，而newFixedThreadPool（）方法可以创建一个固定的大小的线程池）**

# 11.网络编程：

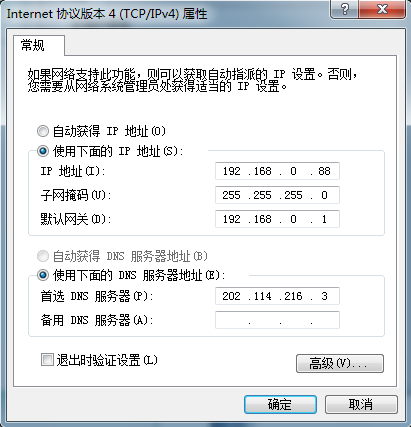
## 网络编程基础：

### 1.为什么需要网络编程：

1. **网络编程实质：两个或多个设配（计算机）之间的数据传输**
2. **对于网络编程主要的计算机之间的通信，首先要解决的问题是网络上的计算机**
3. **找到网络上的电脑，就必须找到其中IP地址**

### 2.IP地址与端口号

1. **IP地址是区分不同计算机的唯一标识**
2. **端口号是网络应用程序的区分标识**



**中国IP地址都是动态的，由于IP地址不够，每次上网会注册一个，关上则换**

### 3.网络通讯过程：

1. **在网络通讯中，第一次主动发起通讯的程序就是客户端（Client）程序，而在第一次通讯中等待连接的程序就是服务器端(Server)程序。**
2. **一旦通讯建立，则客户端和服务器端完全一样，没有区别**

### 4.协议 TCP与UDP

1. 数据在传输会按照一定的格式进行编码，在接收时也必须按照同样的方式进行解码，在这个过程中，发送方和接收方所使用的编码格式，就可以称作协议。
2. 而这种格式必须能够生成唯一的编码，按照该编码可以唯一的解析出发送数据的内容

协议：-🡪语言（网页与服务器中的一门语言）

为什么需要协议：由于网页与服务器间沟通需要桥梁，就需要语言

网络通讯协议TCP：传字节，不会流失数据（有服务器的）qq聊天

用户数据包协议UDP：将字节打包，允许流失（没有服务器的）视频

## Socket简介

1. **Socket（套接字）是网络驱动层提供给应用程序的接口和机制**
2. **Socket是TCP网络编程的核心，发送接收消息都需要使用Socket**

### 例子1：建立服务器

**import** java.io.DataInputStream;

**import** java.net.\*;

**public** **class** StringTest {

**public** **static** **void** main(String[] args) **throws** Exception{

//服务器Server

ServerSocket ss = **new** ServerSocket(6666);

Socket s = ss.accept();//接收文件

DataInputStream dis = **new** DataInputStream(s.getInputStream());

System.*out*.println(dis.readUTF());

dis.close();

s.close();

ss.close(); }}

### 例子2：建立客户端

**import** java.io.\*;

**import** java.net.Socket;

**public** **class** CollectionsTest {

**public** **static** **void** main(String[] args) **throws** Exception{

//客户端

Socket st = **new** Socket("127.0.0.1",6666);//网络编程两个设配之间的数据传输，其中要找到电脑的IP+端口号（什么程序使用他），

//使用服务器，是TCP必须得前提 1.建立客户端，找到IP

DataOutputStream dos = **new** DataOutputStream(st.getOutputStream());//2.建立输出流，以写入

dos.writeUTF("HelloWorld");//用args[0]可以由自己输入得到相应的 }}

### 问题1：怎么要求服务器不断的接收东西：

由于服务器中在接受一次就会自动关闭

只需写个死循环，就可以不断的接收数据

while(true){

Socket s = ss.accept();

DataInputStream dis = new DataInputStream(s.getInputStream());

System.out.println(dis.readUTF());

dis.close();

s.close(); }