#关于java一些的日常练习

java环境变量的配置

Java对象与类

Java数据类型

在工具类中还会涉及到String类

对象与类

[方法](#方法)

[类](#类)

[对象](#对象)

[类与对象关系](#类与对象关系)

[类与对象的基本定义](#类与对象的基本定义)

[封装](#封装)

[继承](#继承)

[多态](#多态)

[垃圾回收机制](#垃圾回收机制)

[构造方法](#构造方法)

[静态static关键字](#静态static关键字)

[静态属性](#静态属性)

[静态方法](#静态方法)

[静态语句块](#静态语句块)

[final和static和abstract区别和使用](#final和static和abstract区别和使用)

[单例模式](#单例模式)

接口与内部类

[抽象类和接口](#抽象类和接口)

[异常及异常处理(Exception Handing)](#异常及异常处理)

[工具类](#工具类)

[集合类](#集合类)

[文件管理](#文件管理)

[多线程](#多线程)

---
main方法

main方法是java的程序的入口方法，JVM在运行程序时首先找的就是main方法，如果找到了main方法再执行，找不到的话就会报错。
* public static void main(String[] args)这句代码进行分析。
* public： 声明方法为公有的，表明了任何对象和类都可以访问这个方法，这样JVM就可以调用这个方法。
* static ： 表明了这个方法为静态的，我们知道静态方法在类进行加载后就可以使用类名+方法名进行调用，这对于main这个入口方法是很有用的。因为类刚进行加载不存在任何实例，只能使用类名.main进行调用。
* void： 表明方法没有返回值。为什么要使用void类型呢？因为main返回任何值对程序都没任何意义（main是最先执行的，不需要返回值），所以设计成void，意味着main不会有任何值返回。
* main： main是JVM识别的特殊方法，即程序的入口方法，必须为main。
* String[] args ： 字符串数组给开发人员在命令行状态下与程序交互提供了一种手段。为什么使用数组呢？因为传入的参数可能多个，需要使用数组来存。args

可以换，而String不可以换。
* 注：也可以这么写： String args[]　　上面提到了由于这里面装的是数组，所以想想怎么声明数组的就OK了！dataType[] arrayRefVar;
* // 首选的方法或dataType arrayRefVar[];
* // 效果相同，但不是首选方法同样的，在main 方法中这样写也是没有问题的！
* 注：main方法必须有public static void修饰，也可以使用final或者synchronized来修饰，但是不能使用abstract来修饰，因为main方法是入口方法。修饰符顺序可以变化。

JDK与JRE区别：

```
    JRE(Java Runtime Enviroment)是Java的运行环境。面向Java程序的使用者，而不是开发者。JRE是运行Java程序所必须环境的集合，包含JVM标

准实现及 Java核心类库。它包括Java虚拟机、Java平台核心类和支持文件。它不包含开发工具(编译器、调试器等)。

    JDK(Java Development Kit)又称J2SDK(Java2 Software Development Kit)，是Java开发工具包，它提供了Java的开发环境(提供了编译器

javac等工具，用于将java文件编译为class文件)和运行环境(提 供了JVM和Runtime辅助包，用于解析class文件使其得到运行)。如果你下载并安装

了JDK，那么你不仅可以开发Java程序，也同时拥有了运 行Java程序的平台。JDK是整个Java的核心，包括了Java运行环境(JRE)，一堆Java工具

tools.jar和Java标准类库 (rt.jar)
```

---

java环境变量的配置

下面是我对应的JDK安装位置： JDK 位置：C:\Program Files\Java\jdk1.8.0_73

环境变量配置：
* (1) JAVA_HOME 编辑：
C:\Program Files\Java\jdk1.8.0_73

* (2)CLASSPATH 编辑：
%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar

* (3)PATH 点击编辑文本：
%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin

* (4)检查是否环境配置成功
使用CMD查看环境是否配置成功：java javac java -version

---

java环境变量的配置的理解

设置环境变量
在java 中需要设置三个环境变量（1.5之后不用再设置classpath了，但个人强烈建议继续设置以保证向下兼用问题）
JDK安装完成之后我们来设置环境变量：右击“我的电脑”，选择“属性”，选择“高级”标签，进入环境变量设置，分别设置如下三个环境变量：
（1）配置path变量：
为什么要配置path变量？因为电脑系统将根据该变量的值找到java编程中需要的一些程序，比如javac.exe、java.exe、javah.exe等等,其中javac.exe程序是用于编译java源代码，java.exe程序是用于执行后缀为class的代码。
如何配置path变量？
path变量一般电脑系统都已经创建好了，所以不用新建该变量，只需在系统变量里选中到path变量，点击“选择” ->“编辑"，需要注意的是，不要随便删除方框里原有的其他path值，而是先在那一大串值之后，添加一个分号，再把C:\Program Files\Java\jdk1.5.0_17\bin填上去。下面列出其键值对：
变量名: PATH(不区分大小写但建议使用大写)
变量值: C:\Program Files\Java\jdk1.5.0_17\bin

path指定一个路径列表，用于搜索可执行文件。执行一个可执行文件时，如果该文件不能再当前路径下找到，则依次寻找PATH中的每一个路径，直至找到。java编译命令（javac）,执行命令（java）和一些工具命令（javadoc，jdb等）都在其安装路径下的bin目录中，因此应该将该路径添加到Path环境变量中，以方便今后在任何地方执行与调用。
不把JDK的bin目录加到PATH的话，需要path\to\jdk\bin\javac Xxx.java 这样编译程序，加到PATH后，可以直接 javac Xxxx.java， 输入省事多了。
（2）配置classpath变量：
为什么要配置classpath变量？配置classpath变量，才能使得java解释器知道到哪里去找标准类库，这些标准类库是别人已经写好了的，我们只管使用。比如我们常用到java.lang包中的类，在配置classpath变量后被设为默认导入，所以在写程序时就不用import这个包了。那么这些标准类库在哪呢？在以JDK的lib目录下以jar为后缀的文件中：一个是dt.jar，一个是tools.jar，这两个jar包都位于C:/jdk1.6.0/lib目录下，所以通常我们都会把这两个jar包加到我们的classpath环境变量的值为：.; C:\Program Files\Java\jdk1.5.0_17\\lib\tools.jar; C:\Program Files\Java\jdk1.5.0_17\\lib\dt.jar;
如何配置classpath变量？
在系统环境变量那一栏中点击新建classpath ，具体过程省略，下面只列出其键值对：
变量名: CLASSPATH
变量值: .;%JAVA_HOME%\lib\tools.jar;%JAVA_HOME%\lib\dt.jar;
（注意，CLASSPATH最前面是有个 “.”的，表示当前目录。用两个%包围JAVA_HOME这个变量的意思是引用变量的值，当然如果你不想这样做，也可以这样配置classpath的值为：.;%JAVA_HOME%\lib\tools.jar;%JAVA_HOME%\lib\dt.jar; 也就是用JAVA_HOME变量的值替换%JAVA_HOME%）;
CLASSPATH也指定一个路径，用于搜索java编译或者运行时需要用到的类。在CLASSPATH列表中出了可以包含路径外，还可以包含Jar文件。Java查找类时会把这个Jar文件当做一个目录来进行查找。通常，需要将JDK安装路径下的jre\lib\rt.jar包含在CLASSPATH中。
JDK1.5后，就不必再配置classpath了。但建议继续设置以保证向下兼用问题最初的jdk自己找不到jdk\lib目录下的tools.jar，所以需要配置这个环境变量。
CLASSPATH=.;%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar
其中"."表示在当前目录下寻找所要编译的类配置classpath变量，才能使得java解释器知道到哪里去找标准类库，这些标准类库是别人已经写好了的，我们只管使用。比如我们常用到java.lang包中的类，在配置classpath变量后被设为默认导入，所以在写程序时就不用import这个包了。这些标准类库在哪？在以JDK的lib

目录下以jar为后缀的文件中：一个是dt.jar，一个是tools.jar，这两个jar包都位于jdk/lib目录下。
（3）配置JAVA_HOME变量：
为什么要配置classpath环境变量？一是为了方便引用，比如，JDK安装在C:\Program Files\Java\jdk1.5.0_17目录里，则设置JAVA_HOME为该目录路径，那么以后要使用这个路径的时候，只需输入%JAVA_HOME%即可，避免每次引用都输入很长的路径串；二则是归一原则，当JDK路径改变的时候，仅需更改JAVA_HOME的变量值即可，否则,就要更改任何用绝对路径引用JDK目录的文档，要是万一没有改全，某个程序找不到JDK，后果是可想而知的----系统崩溃！三则是第三方软件会引用约定好的JAVA_HOME变量，不然，你不能正常使用该软件。
如何配置JAVA_HOME变量？
在系统环境变量那一栏中点击新建JAVA_HOME （JAVA_HOME指向的是JDK的安装路径），变量的值实际上是JDK安装路径的根目录。具体过程省略，下面只列出其键值对：
变量名： JAVA_HOME
变量值：C:\Program Files\Java\jdk1.5.0_17

对象与类区别

* `对象:` 对象是类的实例，有状态和行为两种属性。状态影响行为，行为影响状态。

* 对象的三个主要特性：
 > 1. 对象的行为（behavior）：可以对对象施加哪些操作、方法；
 > 2. 对象的状态（state）：当施加那些方法时，对象是如何响应的？
 > 3. 对象标识（identity）：如何辨别具有相同行为、状态的不同对象。
 ps:同一个类的所有对象实例，由于支持相同的行为而具有家族式的相似性。对象的行为是用可调用的方法定义的。
* `类:` 类是一个模板，它用来描述对象的行为和状态。类是构造对象的模板或蓝图。
 由类构造（Contruct）对象过程为创建类的实例（instance）。
* 类之间的关系：依赖（use-a）、聚合（has-a）、继承（is-a）。
* 类包含的类型变量：

>* 局部变量：在方法、构造方法或者语句块中定义的变量被称为局部变量。变量声明和初始化都是在方法中，方法结束后，变量就会自动销毁。
>* 方法会运用形参。调用的一方会传入实参。实参是传给方法的值，当它传入方法后就成为了形参。参数跟局部变量是一样的。它有类型与名称，可以在方法内使用。
>* 成员变量：成员变量是定义在类中，方法体之外的变量。这种变量在创建对象的时候实例化。成员变量可以被类中方法、构造方法和特定类的语句块访问。
>* 类变量：类变量也声明在类中，方法体之外，但必须声明为static类型。

Java的基本数据类型
--
在java中，变量分为两种：`primitive主数据类型(基本数据类型)`、`引用数据类型`
primitive主数据类型
primitive主数据类型(基本数据类型)包括八种：`boolean`,`char`, `byte`,`short`,`int,lon`g,`float,double`对于变量的赋值，都是将某个变量的值赋给另一个变量；这两个变量之间并没有其他什么联系，只是在初次赋值时，其值刚好相等而已；之后则各走各路，并无任何瓜葛；
>* char类型 2字节 short 2字节,char类型与short类型之间,需要转换,比较特殊，表示正数。
>* char类型与char类型运算，晋级为Int类型,然后再进行计算。
* 隐式数据类型的顺序：

    byte < short < int < long < float < double
* 需要强制转换的顺序：

    double > float > long > int > short > byte > char(特殊，表示正数)
* 大类型转换为小类型值，需要进行强制转换：

```
    float c = 9.0f/2;
    double d = 9.0/2;
    d = c;  //将c赋值给d
    c = (float)d;   //强制类型转换
```

* 本节要点
>* 变量有两种：primitive主数据类型和引用
>* 变量的声明必须有类型和名称
>* primitive主数据类型变量值是该值的字节所表示的
>* 引用变量的值代表位于堆之对象的存取方法
>* 引用变量如同遥控器，对引用变量使用圆点云算法可以如同按下遥控器按钮般的存取它的方法或实例变量
>* 没有引用到任何对象的音乐变量的值为null值
>* 数组一定是一个对象，不管所声明的元素是否为primitive主数据类型，并且没有primitive主数据类型的数组，只有装载primitive主数据类型的数组。
类型转换：

自动类型转换又称为：隐式类型转换
* 有多种类型的数据混合运算时，系统首先自动的将所有数据转换成容量最大的那一种数据类型，然后进行计算
>* 自动转换规则
>> boolean类型不可以转换为其他的数据类型（既不能进行自动类型的提升，也不能强制类型转换）， 否则，将编译出错。
>* 整型、字符型、浮点型的数据在混合运算中相互转换，转换时遵守以下原则：
>> byte、short、char之间不会相互转换，他们三者在计算时首先会转换为int类型
>>> char型其本身是unsigned型，同时具有两个字节，其数值范围是0 ~ 2^16-1，因为，这直接导致byte型不能自动类型提升到char，char和short直接也不会发生自动类型提升（因为负数的问题）
>>> 在Java中，整数类型（byte/short/int/long）中，对于未声明数据类型的整形，其默认类型为int型。在浮点类型（float/double）中，对于未声明数据类型的浮点型，默认为double型。

* 注意：
> int 等基本数据类型是不能够`自动转换`成String类型的,更不能强制类型转换。可以通过`valueOf()`方法进行转换。

强制类型转换又称为：显示类型转换(基本数据类型中的数值类型强制转换)
* 容量大的数据类型转换为容量小的数据类型时，需要使用强制类型转换，其格式如下：
> 变量=（目标类型）值；

* 注意：
> 在强制类型转换过程中，源类型的值可能大于目标类型，因此可能造成精度降低或溢出，使用时需注意

引用类型转换：
java的引用类型转换分为两种：
* 1、基本类型与对应包装类 ： 可以自动转换，来自 `自动装箱和自动拆箱原理`

* 2、两个引用数据类型之间的转换：
> 向上类型转换 ： 子类转换为父类
> 向下类型转换 ： 父类转换为子类`需要强制转换`（如果父类的引用实际指向的是子类对象，在运行时若不是对应的对象，会抛出ClassCastException运行时异常）

>> 向下引用转换会存在风险，但是可以利用java的instanceof关键字去解决这个问题。instanceof运算符用法：判断是一个实例对象是否属于一个类，是返回true，否则返回false。这样我们可以优化上面的代码避免强制转换类型时出现的问题：

类型主要在赋值、方法调用、算术运算上，三种情况发生的：
* 类型转换主要在在 赋值、方法调用、算术运算 三种情况下发生。

a、赋值和方法调用 转换规则：从低位类型到高位类型自动转换；从高位类型到低位类型需要强制类型转换：

（1）布尔型和其它基本数据类型之间不能相互转换；

（2）byte型可以转换为short、int、、long、float和double；

（3）short可转换为int、long、float和double；

（4）char可转换为int、long、float和double；

（5）int可转换为long、float和double；

（6）long可转换为float和double；

（7）float可转换为double；

另外还有是直接数的赋值：先通过直接数判断其类型，然后基本原则和上面谈到的赋值原则基本一致；只是直接数是整数时特殊一点，当在可表示范围内时，可以直接赋值给 byte short char三种类型；

* 算术运算 中的类型转换：1 基本就是先转换为高位数据类型，再参加运算，结果也是最高位的数据类型；2 byte short char运算会转换为Int；

（1）如操作数之一为double，则另一个操作数先被转化为double，再参与算术运算。

（2）如两操作数均不为double，当操作数之一为float，则另一操作数先被转换为float，再参与运算。

（3）如两操作数均不为double或float，当操作数之一为long，、则另一操作数先被转换为long，再参与算术运算。

（4）如两操作数均不为double、float或long，则两操作数先被转换为int，再参与运算。

特殊：

（1）如采用+=、*=等缩略形式的运算符，系统会自动强制将运算结果转换为目标变量的类型。

（2） 当运算符为自动递增运算符（++）或自动递减运算符（--）时，如果操作数为byte，short或char类型不发生改变；

运算符号
* 1、运算符号：+ - * / %

```
>* int整数类型祥运算时，结果也为整数。
>* % 取模运算，取余运算：

    int c1 = 10%3;
    int c2 = -10%3;
    int c3 = 10%-3;
    int c4 = -10%-3;
    System.out.println(c1);
    System.out.println(c2);
    System.out.println(c3);
    System.out.println(c4);
    综上所述，取模运算，运算结果符号 取决于第一个数值为（正\负）

* 2、赋值运算
>* ++e 先运算 在输出  ； 先进行加减
>* e++ 先使用 在运算  ；先使用在运算

>>* 逻辑运算符  || 或  一真则真; && 与  一假则假;  ！非  颠倒是非
>>* | 暗位或   需要两个或多个条件都为真 才可以通过
>>* != 不等于  ==等于   =赋值
>* 条件运算符   三目运算符

    boolean n = (4 < 2) ? true :false;
    System.out.println(n);
    对4 小于2语句进行判断 ？  真就为true ,否则为flase

* 3、运算符优先级运算
> 优先级由高到低

 元素符号|说明|
  |:--:|:--:|
  `var++、var--`|后置操作符
  `+、- ``++var、--var`|(后置操作符);(后置操作符)
  type|类型转换
  !|非
 ` *、/、% `|乘法、除法、求余运算
```

+、-|二元加法乘法
< 、 <+ 、>、 >+|比较操作符
== 、!=|相等操作符
^|异或
&&|条件与
`||`|条件或
` = 、+=、 -+ 、*+ 、/+、 %=`|赋值操作符

> 按操作数多少划分

一元操作符 > 二元操作符 > 三元操作符

> 按运算类型划分

算术运算符 > 关系运算符 > 逻辑运算符 > 赋值运算符

> 尽量多的使用括号

括号优先级别最高

> 一元运算符，如` -、++、- -和 ！ `

> 算术运算符，如` *、/、%、+ 和 -`

> 关系运算符，如 `>、>=、<、<=、== 和 != `

> 逻辑运算符，如` &、^、|、&&、|| `

> 条件运算符和赋值运算符，如 `? ：、=、*=、/=、+= 和 -= `

* 4、&和&&，|和||的用法区别：

&：表示`按位"与"`、&&：表示`条件与`，两者之间的区别是：&会执行两边，不管第一个条件是否成立；

&&只会执行一边，如果第一个条件为假，则不会走第二

个条件。遵循一假则假规则。

|：表示`按位"或"`、||：表示`条件或`，两者之间的区别是：||只要满足第一个条件，后面的条件就不再判断，

而|要对所有的条件进行判断。遵循一真则真原则

> 逻辑与或同按位与或是有一定区别的：

>> 相同点是：都能参与逻辑运算，按位与或完全可以代替逻辑与或。

>> 区别是：按位与或可以参与位运算，逻辑与或只能参与逻辑运算，逻辑与或可以“短路”，按位与或不能“短路”。

>> 短路运算
作用运算符：逻辑与&&，逻辑或||。

如（1）：逻辑表达式：a>b && c>d
假设a>b 为false，c>d为true，那么整个表达式结果为false;
假设a>b 为false，c>d为false，那么整个表达式结果还为false;
可见，a>b的结果已经决定了整个表达式的结果，而后面的c>d并不影响表达式的结果，可以说后半部分被“短路”了。
如（2）：逻辑表达式： a>b || c>d
假设a>b 为true，那么后半部分表达式将被“短路”;

避开关键字
* 类、方法、变量命名规则：
>* 名称必须以字母、下划线（）或$符号开头，不能用数字开头。
>* 除了第一个字符外，后面就可以用数字。反正不要用在第一个字符就行。
>* 主要符合上述两条规则，你就可以随意命名，但还要避开Java的保留字。
>* 对象的声明、创建、赋值有三个步骤。
```
```

```
    Dog d    =       new Dog();
    声明      赋值     创建
```
>* pd:转义字符：\   例如：`\\` 输出一个 `\` ;   `\"` 输出一个 `"`

引用数据类型
除上述八种primitive主数据类型之外，其余类型都称之为引用数据类型；引用数据类型，顾名思义就是：`引用`，当一个对象赋值给一个引用变量时，那么，则表明这个引用变量是指向这个对象的；一个对象可以有多个引用；一个引用同一时刻，则只能指向一个对象；
>* 事实上没有对象变量这样的东西存在
>* 只有引用(reference)到对象的变量
>* 对于引用变量保存的是存取对象的方法
>* 它并不是对象的容器，而是类似指向对象的指针，或者可以说是地址，但在Java中文名不会也不该知道引用变量中实际装载的是什么，它只是用来代表单一的对象。只有Java虚拟机才会知道如何使用引用来取得该对象。

数组
一维数组
* 定义：数组是一种数据结构，用来存储`同一类型`值的集合,一组变量的集合，属于引用数据类型。在声明数组变量时，需要指出数组类型（数组元素类型紧跟[]和数组变量的名字）。
```
1、声明并开辟数组：
    数组类型 数组名称[] = new 数组类型[长度];
    数组类型[] 数组名称  = new 数组类型[长度];
在定义数组的同时进行赋值：
    数组类型[] 数组名称  = new 数组类型[]{值，值，值.......};
eg: 定义类型为Int长度为100的数组
    int[] a = new int[100];
```

当数组开辟空间之后，就可以采用`数组名称[下标|索引]`的方式进行访问,数组长度为`数组名称.length-1`。
如果超出数组长度，就会报错：数组下标越界异常:ArrayIndexOutOfBoundsException。（一维数组一旦定义，长度固定。）

以上给出的数组定义结构使用的是动态初始化的方式。

2、分步完成：
    声明数组：数组类型 数组名称 [] = null;
    开辟数组：数组名称 = new 数组类型[长度];
```

* 初始化
其实在上面的案例中已经涉及到了数组的初始化，数组的初始化分为静态初始化、动态初始化以及默认初始化
注：个人比较喜欢简化方法

* 动态初始化就是在创建过程中只是声明数组的大小，而由系统为数组分配值
```
int [] arrNum = new int[5];
int [] arrNum = new int[5];//简化方式
1
2
3
```

* 静态初始化  就是由程序员显式的指定每个数组元素的值
```
int [] arrNum1 = new int[]{1,2,3,4,5};
int[] arrNum1 = {1,2,3,4,5};//简化方式
```

* 数组遍历：
```
for循环：
    int[] a = {1,2,5,6,49};
```
```

```
    for(int i=0;i<a.length;i++){
        syso(a[i]);
    }
更多练习在[java练习题](#https://github.com/ZHoodLum/JavaPractice/blob/master/JAVA%E7%BB%83%E4%B9%A0%E9%A2%98.md)，二维数组排序等。
```

* 总结：
>* 数组中每个元素都是变量,也可以说是8中primitive主数据类型变量中的一种，不然就是引用变量。
>* 创建一个数字数组时，所有元素都初始化为0；一旦创建数组，就不能改变它的大小（尽管可以改变每一个数组元素）。
>* boolean数组的元素会初始化为`flase`,对象数组的元素则初始化为一个特殊值`null`,这表示数组还未存放任何对象。
>* java.lang.ArrayIndexOutOfBoundsException :异常而终止执行，报这种错误，我们称之为`数组下标越界`。
>* 若想获得数组中的元素个数，可以使用array`.length`。

二维数组
二维数组本质上是以数组作为数组元素的数组，即“数组的数组”，类型说明符 数组名[常量表达式][常量表达式]。二维数组又称为矩阵，行列数相等的矩阵称为方阵。对称矩阵a[i][j] = a[j][i]，对角矩阵：n阶方阵主对角线外都是零元素。
```
动态初始化：数组类型 数组名称[][] = new 数组类型[行的个数][列的个数];
静态初始化：数组类型 数组名称[][] = new 数组类型[行的个数][列的个数]{
                                    {值，值，值...},
                                    {值，值，值...},
                                    {值，值，值...}.......
                                       };
实际上就是多个一维数组组合而成的数组。
```

* 二维数组的遍历：
```
```

```
int data [][] = new int[][] {
                {1,2,3},
                {2,3,4,5},
                {9,4,7,8}
                };
//外层循环控制数组的数据行内容
for(int x =0;x<data.length;x++){
    for(int y=0;y<data[x].length;y++){
        system.out.print(data[x][y]+"/t");
    }
    syso;
}
```

数组与方法参数的传递
在之前的所有方法传递的数据几乎都是基本数据类型，那么除了基本数据类型之外，还可以传递数组，传递数组，一定要观察内存分配图。
注：冒泡排序

实现一维数组的转置
（首位交换）
```
范例：
原始数组：1，2，3，4，5，6，7，8
转置后的数组：8，7，6，5，4，3，2，1
```

实现转置思路：
* 定义一个新的数组，将原始数组按照倒序的方式插入新的数组之中，虽有改变原始数组的引用。
```
public class ArrayDemo{
    public static void main (String args[]){
        int data[] = new int[]{1,2,3,4,5,6,7,8};
        //定义一个新的数组,数组长度与原始数组相同
        int temp[] = new int[data.length];
```

```
        //对于新的数组按照索引由小到大的顺序循环输出
        for(int x=0;x<temp.length;x++){
            temp[x] = data[foot];
            foot--;
        }
        //让data等于temp，而原始数据就会成为垃圾（被回收）
        data = temp;
        println(data);
    }

    //专门定义一个输出的功能方法
    public static void print(int temp[]){
        for(int x=0;x<temp.length;x++){
            System.out.print(temp[x]+'、');
        }
        System.out.println();
    }
}
```
上面的操作达到了转置要求，但是有垃圾。建议不采用。

* 利用算法，在一个数组上完成转置操作
```
原始数组：1，2，3，4，5，6，7，8
第一次转置：`8`，2，3，4，5，6，7，`1`
第二次转置：8，`7`，3，4，5，6，`2`，1
第三次转置：8，7，`6`，4，5，`3`，2，1
第四次转置：8，7，6，`5`，`4`，3，2，1
由此可以判断出，它是由索引控制的，一个向上加的索引，一个向下减的索引。与数组长度无关！！

转换次数=数组长度/2

实现方法：
public class ArrayDemo{
    public static void main (String args[]){
```

```
        int data[] = new int[]{1,2,3,4,5,6,7,8};
        reverse(data);
        println(data);
    }
    //专门实现数组的转置操作
    public static void reverse(int arr[]){
        //转置次数
        int len = arr.length/2;
        //头部索引
        int head = 0;
        //尾部索引
        int tail = arr.length-1;
        for(int x=0;x<len;x++){
            int temp = arr[head];
            arr[head] = arr[tail];
            arr[tail] = temp;
            head++;
            tail--;
        }
    }
    //专门定义一个输出的功能方法
    public static void print(int temp[]){
        for(int x=0;x<temp.length;x++){
            System.out.print(temp[x]+'、');
        }
        System.out.println();
    }
}
```

实现二维数组的转置
前提：行跟列完全相同得到数组
```
原始数组：
1 2 3
4 5 6
```

```
7 8 9

第一次转置：
1 `4` 3
`2` 5 6
7 8 9
第二次转置：
1 4 `7`
2 5 6
`9` 8 9
第三次转置：
1 4 7
2 5 `8`
3 `6` 9

观察转置的数组，中间对角线没有变，而且变得数据行数和列数是相同的。

实现方法：
public class ArrayDemo{
    public static void main (String args[]){
        int data[][] = new int[][]{
                    {1,2,3},
                    {4,5,6},
                    {7,8,9}
                      };
        reverse(data);
        println(data);
    }
    //专门实现数组的转置操作
    public static void reverse(int arr[][]){
        for(int x=0;x<arr.length;x++){
            for(int y=x;y<arr[x].length;y++){
                //行和列相同，进行交换
                if(x != y ){
                    int temp = arr[x][y];
                    arr[x][y] = arr[y][x];
```

```
                    arr[y][x] = temp;
                }
            }
        }
    }
    //专门定义一个输出的功能方法
    public static void print(int temp[][]){
        for(int x=0;x<temp.length;x++){
            for(int y=0;y<temp[x].length;y++){
                System.out.print(temp[x][y]+'、');
            }
        }
        System.out.println();
    }
}
```

匿名数组

实现方法：

```
public class ArrayDemo{
    public static void main (String args[]){
        int data[] = init();
        print(data);
    }
    //匿名数组
    public staticint[] init(){
        //重点关注方法的返回内容
        return new int[]{1,2,3,4,5};
    }
    public static void print(int temp[]){
        for(int x=0;x<temp.length;x++){
            System.out.print(temp[x]+'、');
        }
```

```
        System.out.println();
    }
}
```
注：

init()方法返回的是一个数组，数组可以直接使用length取值长度，返回值可以直接拿来进行操作。

数组拷贝
System.arraycopy(原数组名称，原数组拷贝开始，目标数组名称，目标数组拷贝开始索引，长度);
```
数组A：1，2，3，4，5，6，7，8，9
数组B：11，22，33，44，55，66，77，88
要求拷贝后的数组B：11，22，33，5，6，7，77，88
```

实现方法：
```
public class ArrayDemo{
    public static void main (String args[]){
        int A[] =  new int[]{1，2，3，4，5，6，7，8，9};
        int B[] =  new int[]{11，22，33，44，55，66，77，88};
        System.arraycopy(A,4,B,2,3);
    }
    public static void print(int temp[]){
        for(int x=0;x<temp.length;x++){
            System.out.print(temp[x]+'、');
        }
        System.out.println();
    }
}
```

数组的排序

java.util.Arrays.sort(数组名称)------------>升序

冒泡排序等等。

对象数组的概念
数组是引用类型，类也是引用类型，如果是对象数组的话就表示一个引用类型嵌套了其他的引用类型，之前使用的数组都属于基本数据类型的数组，但是所有引用数据类型也同样可以定义数组，这样的数组称之为对象数组。

* 动态初始化：开辟对象数组的内容都是null值
>* 声明并开辟对象数组： 类名称 对象数组名称[] = new 类名称[长度];
>* 分布完成：
>>* 声明对象数组： 类名称 对象数组名称[] = null;
>>* 开辟对象数组： 对象数组名称 = new 类名称[长度];

* 静态初始化：
>* 类名称 对象数组名称[] = new 类名称[]{实例化对象，实例化对象.......};
```
//Book类
class Book{
    private String title;
    private double price;
    //getter、setter、无参构造方法省略
    public Book(String t,double p){
        title = t;
        price = p;
    }
    public String getInfo(0{
        return "书名"+title+",价格"+price;
    }
}
```

```
//实现
public class ArrayDemo{
    public static void main (String args[]){
    //两种数组实例化的方式，使用一种即可。
        //动态实例化数组
        Book books[] = new Book[3];
        book[0] = new Book("Java",79.2);
        book[1] = new Book("PHP",59.2);
        book[2] = new Book("Android",75.2);
        //静态实例化数组
        Book books[] = new Book[]{
            new Book("Java",79.2);
            new Book("PHP",59.2);
            new Book("Android",75.2);
        }
        //输出数组
        for(x=0;x<books.length;x++){
            Ststem.out.println(books[x]);
        }

    }
}
```

对象数组就是将多个对象交给数组进行统一管理，但是数组的长度一旦定义就是固定的，所以限制的在开发中的使用。

字符串类

String类的特点
两种实例化的方式
```
直接赋值：
String 变量 = "字符串";
```

以上的代码并没有使用关键字new对象
构造方法：public String(String str); 在构造器中依然要接受一个本类对象。

利用构造方法实例化：
```
String str = new String("Hello world!");
```
String类有两种形式，建议采用第二种方式进行实例化。

字符串的比较
两种比较方式 == 和equals
```
String stra = "hello";
String strb = new String("hello");
String strc = strb;
syso(stra == strb); ----------->false
syso(stra == strc); ----------->false
syso(strb == strc); ----------->true
```
使用双等号，就要联系到内存关系，栈内存和堆内存。字符串的`==`确实进行的比较，但并不是比较字符串内容的比较，而是比较内存地址的数值。
所以：==是属于数值比较，比较的是内存地址。

那么，进行字符串内容比较要用到equals.
```
String stra = "hello";
String strb = new String("hello");
String strc = strb;
syso(stra.equals(strb));----------->true
syso(stra.equals(strc));----------->true
syso(strb.equals(strc));----------->true
```
所以说，比较字符串内容的时候，千万不要使用==，要使用equals进行判断。

字符串常量就是String的匿名对象
实际上任何语言没有提供字符串这一概念，很多语言里都是使用字符串数组来描述字符串的概念。在JAVA里也没有字符串的概念，所有开发都不可能离开字符串，在JAVA中字符串不属于基本数据类型，他将字符串作为String类的匿名对象的形式。匿名对象可以调用类中的定义方法的.
```
String stra = "hello";
syso("hello".equals());---------->true
```
匿名对象保存在堆内存当中
所谓的直接赋值就是相当于将一个匿名对象设置了一个名字，匿名对象是由系统自动生成的，不再由用户自己直接创建。
```

为了避免空指向异常的出现，可以将字符串写在前面调用
```
String stra = "hello";----假设这个内容由用户输入的
if(stra.equals("hello")){
    syso("Hello World!!!");
}
---------------------->输出Hello World!!!

但是如果由于用户输入错误，或者输入为空：
String stra = null;----假设这个内容由用户输入的
if(stra.equals("hello")){
    syso("Hello World!!!");
}
---------------------->输出报错：java.lang.NullPointerException,空指向异常
```

##### 但是如果换一种方式输入：将输入的字符串放在equals的前面，即使用户输入错误，也不会报空指向异常的错误。这个euqals处理了null，永远不可嗯那个出现空指向异常。在开发的过程用，一定要使用这种方法！！！！！！！
```
String stra = "hello";----假设这个内容由用户输入的
if("hello".equals(stra)){
```
```

```
    syso("Hello World!!!");
}
```

字符串的替换
字符串的替换后，依然是字符串，例如：替换手机号
```
replaceFirst 替换首个符合条件的内容
replaceAll 新的内容替换掉全部旧的内容

范例：
String str = "helloworld";
String resultA = str.replaceAll("1","_");
String resultB = str.repalceFirst("1","_");
syso(resultA);------------- he__owor_d
syso(resultB);--------------he_loworld
```

字符串的截取
用于截取部分字符串
```
public String `substring(int beginIndex)`
substring(int beginIndex)---->从指定索引截取到结尾

public String `substring(int beginIndex,int endIndex)`
substring(int beginIndex,int endIndex)---->从指定索引截取到结束索引，截取部分字符

范例：
String str = "helloworld";
String resultA = str.substring(5);
String resultB = str.substring(0,5);
syso(resultA);------------- world
syso(resultB);--------------hello
```

字符串的拆分
将一个完整的字符串，按照指定的内容拆分为字符串数组(对象数组，String对象)。
```
public String[] split(String regex)--------------->按照指定字符串进行全部拆分

String str = "hello world nihao mldn";
String result[] = str.split(" ");
for(int x=0;x<result.length;x++){
    syso(result[x]);
}
--------hello
    world
    nihao
    mldn

但是如果在拆分的时候写了一个空的字符串，就会按照每一个字符进行拆分


public String[] split(String regex int limit)----------->按照指定字符串进行部分拆分，由limit决定的(如果拆分的结果很多)，即前面拆，后面不拆。

String str = "hello world nihao mldn";
String result[] = str.split(" ", 2);
for(int x=0;x<result.length;x++){
    syso(result[x]);
}
limit=2的时候----->hello
                world nihao mldn
limit=3的时候----->hello
                world
           nihao mldn

如果String str = "helloworldnihaomldn";
```

那么输出的结果是：limit=3的时候----->helloworldnihaomldn，字符串没有进行拆分，不够拆分的资格。
```

那么下面我们来拆分ip4v的地址，根据`.`进行拆分
```
String str = "192.168.5.36";
String result[] = str.split(" ");
for(int x=0;x<result.length;x++){
    syso(result[x]);
}
输出结果：啥都没有，无法进行拆分
```

因为`.`无法进行拆分，那么我们在想一想Java中的通配符
```
String str = "192.168.5.36";
String result[] = str.split("\\.");
for(int x=0;x<result.length;x++){
    syso(result[x]);
}
输出结果：
192
168
5
36
```
##### 如果是一些敏感的字符，无法进行拆分的，正则标记，如果遇见无法拆分的字符串，那么我们就是用`\\`通配符进行拆分。

根据`|`进行拆分
```
String str = "张三：20|李四：30|王五：22";
String result[] = str.split("\\|");
for(int x=0;x<result.length;x++){
```
```

```
    syso(result[x]);
}
输出：
张三：20
李四：30
王五：22
```

根据`|`进行拆分后，再根据`:`进行拆分
```
String str = "张三：20|李四：30|王五：22";
String result[] = str.split("\\|");
for(int x=0;x<result.length;x++){
    String temp[] = result[x].split(":");
    syso("姓名："+temp[0]+",年龄："temp[1]+);
}
输出：
姓名：张三 年龄：20
姓名：李四 年龄：30
姓名：王五 年龄：22
```


对象与类
--

方法
--
定义
* 1、方法是完成某个功能的一组语句，通常将常用的功能写成一个方法。定义方法就是编写一段有特定功能的代码，在程序中使用同样功能的地方，没有必要重复编写

同样的代码，只要调用定义好的方法就可以。可以实现代码的重用。简化了程序的编写和维护工作。方法声明或称为定义方法。

>* 修饰符：public 、static 被称为修饰符（后续会详细讲解它们）；
>* 返回值类型：用来说明该方法运算结果的类型。如果返回其他类型，编译就可能出错；
>* 方法名：它作为调用时引用方法的标识；
>* 形参列表：在方法被调用时用于接受外部传入的变量称为形式参数简称为形参，方法的形参个数可以是0个到多个，每个参数前面要声明参数的数据类型称为参数类型；每个参数要用逗号分开。也可以一个参数都没有。
>* 方法体：它是一个语句块，执行特定的功能操作。对于有返回值类型的方法，方法体当中最后一个语句是return关键字，它的作用是把方法的执行（运算）结果返回到方法外部。
>* return 表达式：return是关键字，作用是将return后面表达式的结果作为方法的返回值。需要注意表达式的类型，必须与方法头中声明的“返回类型”相匹配。

* 2、方法分类

> 1）根据参数个数：

>> 无参方法

>> 有参方法

>>> 参数类型为基本数据类型

>>> 参数类型为引用数据类型

> 2）根据返回值类型:

>> 有返回值的方法：

>>> 返回值类型为基本数据类型

>>> 返回值类型为引用数据类型

>> 无返回值的方法

>>> 返回值类型使用void关键字

总结
1、可以将一些重复执行的代码定义在方法里面，方法（method）在有些书上也被称为函数(Function)。

2、本次方法所讲解的是有它的局限性：定义主类，并且有主方法直接调用。

3、方法返回值一旦定义了，就需要使用return返回相应数据。

4、方法重载（overloading）指的是`方法名称相同`，`参数类型`及`个数`不同，同时尽量保证`返回值类型相同`。

5、递归调用需要明确`设置一个结束条件`，否则就会出现死循环，如果处理数据量过大，就有可能出现`内存溢出`。

方法重载
方法的重载（overload），方法的重载就是在同一个类中允许同时存在一个以上同名的方法
> 方法重载的规则
>* 方法名称相同
>* 方法的参数必须`不同`:参数个数不同 或参数类型不同
>* 方法的返回值类型可以相同，也可以不同

类
* 什么是类
>> 把相似的对象划归成一个类。
>> 在软件设计中，类，就是一个模板，它定义了通用于一个特定种类的所有对象的属性（变量）和行为（方法）

对象

* 什么是对象
> 类(class) — 是对某一类事物的描述
> 对象(object) — 是实际存在的某类事物的个体,也称为实例(instance)
> 注：类是创建对象的模板，对象是类的实例。

类与对象关系
>* 类与对象是整个面向对象之中最为基础的组成单元，如果需要给出划分定义，类是共性的集合，而对象是某一个性的产物,所有类实际上都是描述出对象的结构。
>* 一类的共性的集合对象，除了具备以上特征（属性）之外，实际上还包括许多的行为（功能），所以根据此类产生出的对象都具备相同的行为。
>* 对象所能够操作的行为都是有类来决定的，超过类定义范畴的操作是不能够使用的
>* 类实际上的对象操作的模板，但是类不能够直接使用，但必须通过实例对象来使用。（先有类再有对象）
>* 类是不能直接使用的，对象是可以直接使用的，对象是可以通过类产生的。

类与对象的基本定义
* 1、如果在程序之中定义类可以使用“class 类名称”的语法结构组成，而类之中主要有两点：
>* Field(属性、成员、变量)：就是一堆变量的集合。
>* methed(方法、行为)：之前见到的方法，此时方法是由对象调用的。

* 2、对象定义格式：
>* 声明并实例化对象：类名称 对象名称 = new 类名称（）；
>>* 注：引用数据类型与基本数据类型最大的不同在于需要内存的开辟及使用，所以关键字new的主要功能就是开辟内存空间。即，只要是引用数据类型想使用，那么就必须使用关键字new来开辟空间。
* 3、内存分析，首先可以给出两块内存空间的概念：
>>* 堆内存：保存每一个对象的属性内容，堆内存需要用关键字new才可以开辟内存空间
>>* 栈内存：保存一块堆内存的地址。但是为了分析方便，栈内存保存的是对象的名字。

* 4、当一个对象实例化之后那么就可以按照如下的方式利用对象来操作类的结构
>>* 对象.属性：表示要操作类中的属性内容。
>>* 对象.方法()：表示要调用类中的方法。

* 5、当使用没有实例化对象，在程序运行时，会出现“NullPointeException”空指针指向异常，此类异常只要是引用数据类型都有可能出现。
>* 注：String 首字母大写的都是类名称，类名称都是引用类型，引用类型的默认值是空。
>* double是基本数据类型，默认值是0.0

Java面向对象的三大特征：
--
封装、继承、多态。封装和继承几乎都是为多态而准备的

封装
1、特征：
* 所谓封装是把对象的属性和行为结合在一个独立的系统单位内部
* 尽可能隐蔽对象的内部细节，只向外部提供接口
* 降低对象间的耦合度
* 封装的重要意义：
> 使对象能够集中而完整地描述并对应一个具体事物

> 体现了事物的相对独立性，使对象外部不能随意存取对象的内部数据

首先，属性能够描述事物的特征，方法能够描述事物的动作。封装就是把同一类事物的共性（包括属性和方法）归到同一类中，方便使用。封装：封装也称信息隐藏，是指利用抽象数据类型把数据和基于数据的操作封装起来，使其成为一个不可分割的整体，数据隐藏在抽象数据内部，尽可能的隐藏数据细节只保留一些接口使其与外界发生联系。也就是说用户无需知道内部的数据和方法的具体实现细节，只需根据留在外部的接口进行操作就行。

2、封装的好处：
>* 实现了专业的分工。
>* 良好的封装能够减少耦合。
>* 类内部的结构能够自有修改。
>* 可以对成员进行更精确的控制。

>* 隐藏信息，实现细节。
>* 使用公有与私有这两个存取修饰符来`隐藏数据`。
>* 封装原则：将你的实例变量标记为私有的，并提供公有的getter与setter类控制存取动作。
>* 将实例变量标记为private。将getter与setter标记为public。任何有值可以被运用到的地方，都可调用方法的方式来取得该类型的值。

3、封装的步骤
>* 修改属性的可见性来限制对属性的访问
>* 为每个属性创建一队赋值和取值方法，用于对这些属性的访问
>* 在赋值和取值方法中，加入对属性的存取限制
>* 为了实现良好的封装，我们通常将类的成员变量声明为private，在通过public方法来对这个变量来访问。对一个变量的操作，一般有读取`getter`和赋值`setter`2个操作，我们分别定义2个方法来实现这2个操作，一个是getXX（XX表示要访问的成员变量的名字）用来读取这个成员变量，另一个是setXX（）用来对这个变量赋值。
下面我们来看下这个例子：
```
    public class Husband {
    /*

     * 对属性的封装一个人的姓名、性别、年龄、妻子都是这个人的私有属性

     */
    private String name;

    private String sex;

    private int age;

    private Wife wife;

    /*

     * setter()、getter()是该对象对外开发的接口
```

```
     */

    public String getName() {

        return name;

    }
    public void setName(String name) {

        this.name = name;

    }
    public String getSex() {

        return sex;

    }
    public void setSex(String sex) {

        this.sex = sex;

    }
    public int getAge() {

        return age;

    }



    public void setAge(int age) {

        this.age = age;

    }
    public void setWife(Wife wife) {
```

```
        this.wife = wife;

    }

}
```

继承

1、Java继承

* 定义：在Java中定义一个类时，让该类通过关键字extends继承一个已有的类，这就是类的继承(泛化)。

> 被继承的类称为父类（超类，基类），新的类称为子类（派生类）。

> 子类继承父类的所有属性和方法，同时也可以增加自己的属性和方法。

Java继承是面向对象的最显著的一个特征。继承是从已有的类中派生出新的类，新的类能吸收已有类的数据属性和行为，并能扩展新的能力。JAVA不支持多继承，单继承使JAVA的继承关系很简单，一个类只能有一个父类，易于管理程序，父类是子类的一般化，子类是父类的特化（具体化)继承所表达的就是一种对象类之间的相交关系，它使得某类对象可以继承另外一类对象的数据成员和成员方法。若类B继承类A，则属于B的对象便具有类A的全部或部分性质(数据属性)和功能(操作)，我们称被继承的类A为基类、父类或超类，而称继承类B为A的派生类或子类。继承避免了对一般类和特殊类之间共同特征进行的重复描述。同时，通过继承可以清晰地表达每一项共同特征所适应的概念范围在一般类中定义的属性和操作适应于这个类本身以及它以下的每一层特殊类的全部对象。运用继承原则使得系统模型比较简练也比较清晰。

2、Java继承的特征：继承（inheritance）

--

> 也称泛化，继承性是子类自动共享父类属性和方法的机制，在定义和实现一个类的时候，可以在一个已经存在的类的基础之上来进行，把这个已经存在的类所定义的内容作为自己的内容，并加入自己若干新的内容

> 继承简化了人们对事物的认识和描述，有益于软件复用，是OO技术提高软件开发

效率的重要原因之一

> 是类之间的一种关系，一般类与特殊类之间的关系

> 继承关系的语义：“is a kind of”

>* 继承关系是传递的。若类C继承类B，类B继承类A（多继承），则类C既有从类B那里继承下来的属性与方法，也有从类A那里继承下来的属性与方法，还可以有自己新定义的属性和方法。继承来的属性和方法尽管是隐式的，但仍是类C的属性和方法。
>* 继承提供了软件复用功能。若类B继承类A，那么建立类B时只需要再描述与基类(类A)不同的少量特征(数据成员和成员方法)即可。这种做法能减小代码和数据的冗余度，大大增加程序的重用性。
>* 继承通过增强一致性来减少模块间的接口和界面，大大增加了程序的易维护性。
3、继承的规则：
* Java中只支持单继承，也就是说每个类只能有一个父类，不允许有多重继承
> 一个父类可以有多个子类
> 子类继承父类所有的属性和方法

4、继承的实例
```
class Person1 {

	public String name = "xiaomiao";

	public int age = 20;

}
class Student extends Person1 {

	void study() {

		System.out.println("I can study!");
	}

}
```

```
public class JiCheng {

	public static void main(String args[]) {

		Student stu = new Student();

		// stu.name = "zhangsan";

		// stu.age = 20;

		System.out.println("name=" + stu.name + ",,," +
"age=" + stu.age);

	}

}
```

多态
特征：
* 指同一个命名可具有不同的语义
* OO方法中，常指在一般类中定义的属性或方法被特殊类继承之后，可以具有不同的数据类型或表现出不同的行为，对于子类，可用不同的方法替代实现父类的服务的方法

>* 方法的重写、重载与动态连接构成多态性； Java之所以引入多态的概念，原因之一是它在类的继承问题上和C++不同，后者允许多继承，这确实给其带来的非常强大的功能，但是复杂的继承关系也给C++开发者带来了更大的麻烦，为了规避风险，Java只允许单继承，派生类与基类间有IS-A的关系（即“猫”is a “动物”）。这样做虽然保证了继承关系的简单明了，但是势必在功能上有很大的限制，所以，Java引入了多态性的概念以弥补这点的不足，
此外，抽象类和接口也是解决单继承规定限制的重要手段。同时，多态也是面向对象编程的精髓所在。 要理解多态性，首先要知道什么是“向上转型”。

>* 我定义了一个子类Cat，它继承了Animal类，那么后者就是前者的父类。我可以通过Cat c = new Cat();　　例化一个Cat的对象，这个不难理解。但当我这样定义时： Animal a = new Cat();这代表什么意思呢？很简单，它表示我定义了一个Animal类型的引用，指向新建的Cat类型的对象。由于Cat是继承自它的父类Animal，
所以Animal类型的引用是可以指向Cat类型的对象的。那么这样做有什么意义呢？因为子类是对父类的一个改进和扩充，所以一般子类在功能上较父类更强大，属性较父类更独特，定义一个父类类型的引用指向一个子类的对象既可以使用子类强大的功能，
又可以抽取父类的共性。所以， 父类引用只能调用父类中存在的方法和属性，不能调用子类的扩展部分；因为父类引用指向的是堆中子类对象继承的父类；（但是如果强制把超类转换成子类的话，就可以调用子类中新添加而超类没有的方法了。） 同时，父类中的一个方法只有在父类中定义而在子类中没有重写的情况下，才可以被父类类型的引用调用； 对于父类中定义的方法，如果子类中重写了该方法，那么父类类型的引用将会调用子类中的这个方法，这就是动态连接。

垃圾回收机制
* 面试容易遇到的题：

从大的方面来讲，JVM的内存模型分为两大块：永久区内存（ Permanent space ）和堆内存（heap space）。栈内存（stack space）一般都不归在JVM内存模型中，因为栈内存属于线程级别。每个线程都有个独立的栈内存空间。

> 1）java垃圾回收　是自动释放该内存快

> 2）垃圾回收机制作用于堆内存当中

> 3）没有引用指向的对象 ，会被标记为垃圾

* 两种提示垃圾回收机制：

> 1）System.gc();　提醒系统进行垃圾回收（至于效率、怎末回收，无权干涉）

> 2）.Finalize() ：当某个对象被系统收集为无用信息的时候,finalize()将被自动调用,但是jvm不保证finalize()一定被调用,也就是说,finalize()的调

用是不确定的,这也就是为什么sun不提倡使用finalize()的原因

垃圾回收的两种机制详解：
* 1、垃圾回收（Garbage Collection，GC），JDK7以后使用G1（Garbage First）机制：
当没有对象引用指向原先分配给某个对象的内存时，该内存便成为垃圾。JVM的一个系统级线程会自动释放该内存块。垃圾回收机制作用于堆内存，与栈内存无关。
垃圾回收机制是JVM内部运行的一个优先级比较低的后台线程，自动进行垃圾回收。它是保证程序健壮的主要手段，不用程序员参与，避免了由于程序员忘记回收内存而引起的内存泄漏，同时也避免了回收内存带来的代码繁琐。

* 2、对象会被回收情况：
> 1）对象的引用被赋值为
```
    null： Person p = new Person( );   p = null;
```
> 2）使用的匿名对象：
```
    new Person( ).sayHello( );
```
> 3）超出生命周期的，如：
```
    for( int i = 0; i< 100; i++){
        Person p = new Person( );
      }
```
这里，创建了100个对象，循环赋值给变量p，每结束一次循环，变量p就超出生命周期，对象变为垃圾。

* 3、System.gc（）
> (1) GC是垃圾收集的意思（Gabage Collection）,内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，J

ava提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java语言没有提供释放已分配内存的显示操作方法。

> (2) 对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。可以。程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

> (3) 垃圾回收是一种动态存储管理技术，它自动地释放不再被程序引用的对象，当一个对象不再被引用的时候,按照特定的垃圾收集算法来实现资源自动回收的功能。

> (4) System.gc();就是呼叫java虚拟机的垃圾回收器运行回收内存的垃圾。

> (5) 当不存在对一个对象的引用时，我们就假定不再需要那个对象，那个对象所占有的存储单元可以被收回，可通过System.gc()方法回收，但一般要把不再引用的对象标志为null为佳。

> (6) 每个 Java 应用程序都有一个 Runtime 类实例，使应用程序能够与其运行的环境相连接。可以通过 getRuntime 方法获取当前运行时。 Runtime.getRuntime().gc();

> (7) java.lang.System.gc()只是java.lang.Runtime.getRuntime().gc()的简写，两者的行为没有任何不同。

> (8) 唯一的区别就是System.gc()写起来比Runtime.getRuntime().gc()简单点. 其实基本没什么机会用得到这个命令, 因为这个命令只是建议JVM安排GC运行, 还有可能完全被拒绝。 GC本身是会周期性的自动运行的,由JVM决定运行的时机,而且现在的版本有多种更智能的模式可以选择,还会根据运行的机器自动去做选择,就算真的有性能上的需求,也应该去对GC的运行机制进行微调,而不是通过使用这个命令来实现性能的优化。

* 4、Finalize（）
> 注意：如果你熟悉C++，那你知道C++允许你为一个类定义一个撤消函数（destructor ），它在对象正好出作用域之前被调用。Java不支持这个想法也不提供撤消

函数。finalize() 方法只和撤消函数的功能接近。当你对Java 有丰富经验时，你将看到因为Java使用垃圾回收子系统，几乎没有必要使用撤消函数。

> finalize的工作原理应该是这样的：一旦垃圾收集器准备好释放对象占用的存储空间，它首先调用finalize()，而且只有在下一次垃圾收集过程中，才会真正回收对象的内存.所以如果使用finalize()，就可以在垃圾收集期间进行一些重要的清除或清扫工作.

> finalize()在什么时候被调用?有三种情况:

> 1.所有对象被Garbage Collection时自动调用,比如运行System.gc()的时候.

> 2.程序退出时为每个对象调用一次finalize方法。

> 3.显式的调用finalize方法

除此以外,正常情况下,当某个对象被系统收集为无用信息的时候,finalize()将被自动调用,但是jvm不保证finalize()一定被调用,也就是说,finalize()的调用是不确定的,这也就是为什么sun不提倡使用finalize()的原因

构造方法
* 1、构造方法（构造器，Constructor）
> 没有返回值　与类名相同　有重载方法 this调用
> 构造方法也是方法，但是它是特殊的方法而已
> 所有的对象都是通过构造器来创建的

* 创建对象：类名　　对象名 = new　　构造方法（）；

构造器（构造方法/构造器，Constructor）特点：
```
* 构造方法在对象实例化之后只调用一次，而普通方法再对象实例化之后可以多次调用
* 构造方法没有返回值声明
```

* 构造方法名必须与类相同的名称
* 不含返回值类型，也没有void
* 不能在方法中用return语句返回一个值
```

* 2、构造方法作用：

构造方法用来完成对象的创建，即完成对象的实例化
> 在Java中，每个类都至少要有一个构造器
> 一个类可以不写出构造方法，编译器会自动的给这个类增加一个构造器，该构造器没有任何参数，我们称之为“默认的空构造器”。
> 编程者为该类定义了构造器，系统就不再提供默认的构造器

* 3、构造方法重载：
> 构造方法可以像普通方法一样发生重载，通常一个类中不止一个构造方法，而是有多个不同参数的构造器，用户可以根据需要选择不同的构造器创建对象。
>> 一般可以使用构造器来完成对成员变量的初始化。例如：
```
public class Employee{
  int age;
  public Employee(int age){
    this.age = age;
  }
}

```
>>有带参的构造器，那么不带参的空构造器不会自动产生。为了防止在某些情况下还想使用这样的无参数默认构造器，因此最好还是养成写出这个无参数的构造器来，以备后用。
```
public class Employee{
  int age;
  public Employee(){
  }
  public Employee(int age){
    this.age = age;
```
```

```
  }
}

```

在构造方法中，可以使用this或super调用变量，参数

* this super只能出现在子类构造器中，且必须是第一行。若既没有this也没有super,则默认super();
* 创建子类时，必须调用父类构造器。

>* super()
>> 作用：调用父类的构造器
>> 只能出现在子类的构造器中，且必须是第一行
>> super()中的参数，决定了调用父类哪个构造器
>> 如果子类构造器中没有出现super，那么编译器会默认加上super()，即调用父类的空构造器，如果父类没有空构造器，编译器提示错误。

>* this()
>> 作用：调用本类的构造器
>> 只能写在构造器的第一行
>> 在同一个构造器中super()和this()不能同时出现

接下来用程序说话
```
//Book类
class Book{
    private String title;
    private double price;
    //getter setter
    public Book(String t,double p){
        title = t;
        price = p;
    }
    public String getInfo(){
         return "书名：" +this.title+",价格："+price;
```

```
    }
}

//测试
public class TestDemo{
    public static void main(String args[]){
        Book book = new Boook("java开发",89);
        syso(book.getInfo());
    }
}
输出：书名：java开发，价格：89
```

接下来我们研究一下这个方法
下面这个使用的是一个构造方法，而这个构造方法主要功能为title,price两个属性所使用，但是方法中的参数名称不好，而构造方法中的参数目的是为了类中的属性初始化，那么我们最好的做法就是将参数设置为属性名称 ，这样做事最好的。
```
那么我们将参数名称更改为对应的属性名，看一下结果
public Book(String title,double price){
    title = title;
    price = price;
}
输出：书名：null,价格：0.0
```

修改代码后，参数的作用是合理了，但是最终在构造方法传递的参数内容并没有传递到属性之中，在此构造方法中由局部变量，也有全部变量。如果现在属性名称与参数名称出现了重名的问题，没有加入任何限制，它都会取最近的“{ }”内的变量名称。所以在这种情况下，为了找到明确的变量，就要使用到`this`了。
```
修改代码，使用this
public Book(String title,double price){
    this.title = title;
    this.price = price;
}
```

输出：书名：java开发，价格：89
```
这样一来，this.title和this.price就会取类中的参数变量了。
**以后的程序开发中，只要访问类中的属性，前面最好加上“this”关键字。**

---

**在构造方法中，也可以使用this或super调用其他的构造方法**
**一个类里面方法除了普通方法之外还会包含构造方法，这个时候表示多个构造方法之间要进行互相调用，而使用的形式` this(参数，参数) `**
**代码说话，为什么需要构造方法间的互相调用**
```
定义一个book类里面三个构造方法，要求不管调用哪一个构造方法都需要输出一行提示信息“新的Book类对象产生”

```
//Book类
class Book{
    private String title;
    private double price;
    //getter setter
    public Book(){
        syso("新的Book类对象产生");//把这句话想象成多行代码
    }
    public Book(String title){
        this.title = title;
        syso("新的Book类对象产生");
    }
    public Book(String title,double price){
        syso("新的Book类对象产生");
        this.title = title;
        this.price = price;
    }
    public String getInfo(){
        return "书名：" +this.title+",价格："+price;
    }
}
```

```
//测试
public class TestDemo{
    public static void main(String args[]){
        Book book = new Boook("java开发",89);
        syso(book.getInfo());
    }
}
```

此时的代码出现大量重复代码，怎么消除消除重复代码呢？

检查方法中有哪些重复，并且可以替换的代码。

修改代码：两个有参构造方法中使用this(),来调用无参、无参构造方法。

```
//Book类
class Book{
    private String title;
    private double price;
    //getter setter
    public Book(){
        syso("新的Book类对象产生");//把这句话想象成多行代码
    }
    public Book(String title){
        this.();//调用本类的无参构造方法
        this.title = title;
    }
    public Book(String title,double price){
        this.(title);//调用本类的单参构造
        this.price = price;
    }
    public String getInfo(){
        return "书名：" +this.title+",价格："+price;
    }
}
```

```
//测试
public class TestDemo{
    public static void main(String args[]){
        Book book = new Boook("java开发",89);
        syso(book.getInfo());
    }
}
```

上面实现了构造方法间的互相调用，但是依然会存在一些限制

* 使用this()调用构造方法形式的代码只能放在构造方法的首行，放在构造方法的第一行。
* 构造方法能调用普通方法，但是普通方法无法调用构造方法。
* 进行构造方法互相调用的时候，一定要保留调用的出口。

下面观察一下错误代码：

```
//Book类
class Book{
    private String title;
    private double price;
    //getter setter
    public Book(){
        this.("HELLO",1.1);//调用双参构造
        syso("新的Book类对象产生");//把这句话想象成多行代码
    }
    public Book(String title){
        this.();//调用本类的无参构造方法
        this.title = title;
    }
    public Book(String title,double price){
        this.(title);//调用本类的单参构造
        this.price = price;
    }
    public String getInfo(){
        return "书名：" +this.title+",价格："+price;
    }
}
```

```
//测试
public class TestDemo{
    public static void main(String args[]){
        Book book = new Boook("java开发",89);
        syso(book.getInfo());
    }
}
```
此时调用语句的确是放在了构造方法首行。但是编译之后，依然还会报错**提示错误：递归构造器调用**。所以说要保诚有一个出口！！

也就是说再使用`this()`**互相调用构造的时候请至少保留一个构造没有被使用this()调用其他构造的情况。**

表示当前对象
所谓的当前对象就是当前正在调用类中的方法的对象。
```
public static void main(String args[]){
    Book booka = new Book();
    syso("booka"+booka);

    Book bookb = new Book();
    syso("bookb"+bookb);
}
输出：
booka = Book@659e0bfd
------------------------------
bookb = Book@2a139a55
两个关键字，表示开辟了两块对内存空间，存储地址也不同，输出的是存储地址。
```

接下来使用this:
```
class Book{
    public void print(){
```

```
        syso("this"+this);
    }
}
public static void main(String args[]){
    Book booka = new Book();
    syso("booka"+booka);
    syso(this.booka);

    Book bookb = new Book();
    syso("bookb"+bookb);
    syso(this.bookb);
}
输出：
booka = Book@659e0bfd
this = Book@659e0bfd
---------------------------------------
bookb = Book@2a139a55
this = Book@2a139a55

此时的this，那个对象调用了print方法，this就自动于此对象指向同一块内存地址，this就是调用当前方法的对象。
```
之前this属性，实际上就是当前对象中的属性，一定是堆内存保存的内容。

思考题
```
//类A
class A{
    private B b;
    public A(){ //2、执行A类的构造
        //3、为B类对象b进行实例化
        this.b = new B(this);   //4、这里的this就是temp

        this.b.get();   //7、调用B类的get方法
    }
    public void print(){     // 10、调用print输出
```

```
        syso("Hello World!!");
    }
}
//类B
class B{
    private A a;
    public B(A a){  //5、参数A就是temp
        this.a = a; //6、保存A对象 ，实际上就是保存temp
    }
    public void get(){  //8、调用此方法
        this.a.print();     // 9、this.a = temp
    }
}
//测试类
public class TestDemo{
    public static void main(String agrs[]){
        //1、实例化A类对象，要调用A类的无参构造，下面的代码应该从右到左执行的
        A temp = new A();
    }
}
输出：
Hello World!!
```

静态static关键字
* static关键字可以修饰的元素
> 属性
    注意：只能修饰属性，不能修饰局部变量。
> 方法
> 代码块
* static修饰属性：
> 所有对象共享，称为静态变量或类变量
> 不用static修饰的属性称为实例变量

* static修饰方法：
> 不需要实例化，可以直接访问，称为静态方法或类方法
* static修饰语句块：
> 类中由static关键字修饰的，不包含在任何方法体中的代码块，称为静态代码块

静态属性
* 用static修饰的属性，它们在类被载入时创建，只要类存在，static变量就存在。
> 静态变量和非静态变量的区别是：静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。而非静态变量是对象所拥有的，在创建对象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响。

* 两种方式访问：
> 直接访问：类名.属性；
> 实例化后访问：对象名.属性 （不建议使用）

```
class Book{
    private String title;
    private double price;

    String pub="清华大学出版社";
    public Book(String title,double price){
        this.title = title;
        this.price = price;
    }
    public String getInfo(){
        return this.title+","+this.price+","+this.pub;
    }
}

public class TestDemo{
    public static void main(String agrs[]){
        Book ba = new Book("java",10.2);
        Book bb = new Book("cc",11.2);
        Book bc = new Book("aa",12.2);
```

```
        //修改了一个属性内容
        ba.pub = "北京大学";
        syso(ba.getInfo());
        syso(bb.getInfo());
        syso(bc.getInfo());
    }
}
输出：
java,10.2,北京大学
cc,10.2,清华大学出版社
aa,10.2,清华大学出版社
```

上面的这个程序：如果现在出现了100W个Book对象，但是要求所有的对象名称更换。那么就要修改100W个对象内容，所以如果将属性定义为普通属性，最终结果就是每一块对内存空间都将要保存各自的信息，这种的结果是不方便的。
进一步将，既然所有的pub内容都应该是一样的，那么就应该将其定义一个共有的同一pub属性，那么这种情况下，就可以使用static来定义属性。

```
class Book{
    private String title;
    private double price;

    //这里使用static修饰属性
    static String pub="清华大学出版社";
    public Book(String title,double price){
        this.title = title;
        this.price = price;
    }
    public String getInfo(){
        return this.title+","+this.price+","+this.pub;
    }
}

public class TestDemo{
    public static void main(String agrs[]){
```

```
        Book ba = new Book("java",10.2);
        Book bb = new Book("cc",11.2);
        Book bc = new Book("aa",12.2);
        //修改了一个属性内容
        ba.pub = "北京大学";
        syso(ba.getInfo());
        syso(bb.getInfo());
        syso(bc.getInfo());
    }
}
输出：
java,10.2,北京大学
cc,10.2,北京大学
aa,10.2,北京大学
```
一旦再属性定义上，只要有一个对象修改了属性内容之后，所有的对象属性都会修改。
既然static是一个公共的概念，但是由一个对象可以修改所有对象的属性不太合适，所以正确的做法就是所有对象的公共的代表来进行访问，那么就是类，所以利用static定义的属性可以由类直接调用属性的。
```
//修改代码
Book.pub = "北京大学";
```

Java语言提供了很多修饰符，主要分为以下两类：

* 访问修饰符
>* 默认的，也称为default，在同一包内可见，不使用任何修饰符。
>* 私有的，以private修饰符指定，在同一类内可见。
>* 共有的，以public修饰符指定，对所有类可见。
>* 受保护的，以protected修饰符指定，对同一包内的类和所有子类可见。

* 非访问修饰符
>* static修饰符，用来创建类方法和类变量。
>* final修饰符，用来修饰类、方法和变量，final修饰的类不能够被继承，修饰

的方法不能被继承类重新定义，修饰的变量为常量，是不可修改的。
>* abstract修饰符，用来创建抽象类和抽象方法。
>* synchronized和volatile修饰符，主要用于线程的编程。

Static与非Static区别：
```
所有非Static属性必须产生实例化对象之后才可以访问，但是Static不受实例化的控制，也就是说，在没有产生实例化对象的情况下，依然可以使用Static属性。
```

Static特征：
```
虽然定义在类结构里面，但是并不受对象的控制，是独立于类存在的。
```

那么我们什么时候使用Static属性，什么时候不使用Static属性呢？
```
在编写类的过程中，你所选择首要的修饰符一定不是Static(95%情况下不写)。如果需要描述共享信息的情况下，使用Static(可以方便集体修改，可以不重复开辟内存空间的)。
```

静态方法
* 静态方法不需要实例化，可以直接访问，访问方式：
> 直接访问：类名.方法名()
> 实例化后访问：对象名.方法名() （不建议使用）

```
class Book{
    private String title;
    private double price;

    //这里使用static修饰属性
    private static String pub="清华大学出版社";
```

```
    public Book(String title,double price){
        this.title = title;
        this.price = price;
    }
    public static void setPub(String p){
        pub = p;
    }
    public String getInfo(){
        return this.title+","+this.price+","+this.pub;
    }
}

public class TestDemo{
    public static void main(String agrs[]){
        //在没有对象产生的时候进行调用操作
        Book.setPub("北京大学");
        Book ba = new Book("java",10.2);
        Book bb = new Book("cc",11.2);
        Book bc = new Book("aa",12.2);

        syso(ba.getInfo());
        syso(bb.getInfo());
        syso(bc.getInfo());
    }
}
输出：
java,10.2,北京大学
cc,10.2,北京大学
aa,10.2,北京大学
```

发现Static定义的属性和方法都不受实例化对象控制，也就说属于独立的类属性。

但是这个时候就会出现一个麻烦的问题：此时的类中的方法变成了两组：Static方法、非Static方法，两组方法间的访问也将受到限制：
* **static方法不能够直接访问属性或者方法，只能够调用static属性或方法。
**

* 为什么会存在限制问题呢？
>* 所有的非static定义结构，必须在类已经明确的产生了实例化对象才会分配空间，才可以使用。
>* 所有的static定义的结构，不受实例化对象的控制，即：可以在没有实例化对象的时候访问。

* **非static方法可以访问static的属性或方法，不说任何的限制。**

解决问题
如果一个方法定义在主类中，并且由主方法调用
```
public static 返回值类型 方法名称(参数类型 参数，...){
    [return[返回值];]
}
```

接下来编写类的时候，发现方法定义的格式改变了（方法由对象调用）
```
public 返回值类型 方法名称(参数类型 参数，...){
    [return[返回值];]
}
```

观察如下代码解决问题：
```
public class TestDemo{
    pubblic static void main(String args[]){
        fun();
    }
    public static void fun(){
        syso("Hello world!!");
    }
}
输出：Hello world!!
```
如果此时fun()方法取消了static修饰符 ，那么就成为了非static方法。所有的

非static方法必须由对象调用，此时static方法如果想要使用非static操作，必须产生对象才能进行调用。

所以代码改变成：
```
public class TestDemo{
    pubblic static void main(String args[]){
        new TestDemo.fun();
    }
    public static void fun(){
        syso("Hello world!!");
    }
}
输出：Hello world!!
```
* **与定义属性的规则一样，定义一个类的时候首先考虑非static方法，因为所有的类如果保存的信息多（属性多），那么每一一个对象执行同一个方法的时候，就可以利用static方法执行**
* 比如说一个类没有属性，产生对象就完全没有意义，所以就会使用static方法。
* 对象保存的是属性！！！

static属性保存在全局数据区
```
内存区一共有四个：栈内存、堆内存、全局数据区、全局代码区：

栈内存：存放地址；
堆内存：存放普通属性；
全局数据区：存放static数据；
全局代码区：存放方法；
```

静态变量
* 用static关键字定义的变量，与局部变量相比，static局部变量有三点不同：
> 1. 存储空间分配不同

auto类型分配在栈上，属于动态存储类别，占动态存储区空间，函数调用结束后自动释放， 而static分配在静态存储区，在程序整个运行期间都不释放，两者之间的作用域相同，但生存期不同。
> 2. static局部变量在所处模块在初次运行时进行初始化工作，且只操作一次。
> 3. 对于局部静态变量，如果不赋初值，编译期会自动赋初值0或空字符，而auto类型的初值是不确定的。(对于C++中的class对象例外， class的对象实例如果不初始化， 则会自动调用默认构造函数， 不管是否是static类型)

> 静态全局变量用来表示不能被其它文件访问的全局变量和函数。为了限制全局变量/函数的作用域，函数或变量前加static使得函数成为静态函数。但此处“static”的含义不是指存储方式，而是指对函数的作用域仅局限于本文件(所以又称内部函数)。注意：对于外部(全局)变量，不论是否有static限制，它的存储区域都是在静态存储区，生存期都是全局的。此时的static只是起作用域限制作用，限定作用域在本模块(文件)内部。

> 静态全局变量与全局变量的差别是：静态全局变量只能被同一源文件中的函数调用，其他文件中的函数不能调用静态全局变量

> 只`允许本源文件中`所有函数使用的全局变量，则该变量需要使用的存储类型是static。

静态语句块
* 一个类中由static关键字修饰的，不包含在任何方法体中的代码块
> 当类被载入时，静态代码块被执行，且只被执行一次

> 静态块经常用来进行类属性的初始化
final和static和abstract区别和使用
一、static 修饰属性、方法和代码块

* 1.static修饰属性，这个属性就可以用类名.属性名来访问，也就是使这个属性成为本类的类变量，为本类对象所共有。这个属性就是全类公有。类加载的过程，类本身保存在文件中（字节码文件保存着类的信息）的，java会通过I/O流把类的文件（字节码文件）读入JVM（java虚拟机），这个过程称为类的加载过程。JVM（java虚拟机）会通过类路径（CLASSPATH）来找字节码文件。类变量，会在加载时自动初始化，初始化规则和实例变量相同。

特别：类中的实例变量是在创建对象时被初始化的，被static修饰的属性(类变量)，是在类加载时被创建并进行初始化，类加载的过程是进行一次。也就是类变量只会被创建一次。

* 2.static修饰方法（静态方法），会使这个方法成为整个类所公有的方法，可以用类名.方法名访问。

"类名.属性名"以及"类名.方法"访问举例：

```
class Test{

    public static String TEST_IP="静态ip127.0.0.1";
    public static void testMethod(){
    System.out.println("调用静态方法成功!");

}}
```

测试类

```
public class Test2{
    public static void main(String args[]){
    //1、通过类名.属性名调用
    System.out.println("调用的属性值："+Test.TEST_IP);
    Test.TEST_IP="新的ip127.0.0.1";//重新修改地址指向的值
    System.out.println("调用的属性值："+Test.TEST_IP);
    //2、调用静态方法
    Test test=new Test();
    test.testMethod();
}}
```

* 特别：
> （1）static修饰的方法，不直接能访问本类中的非静态(static)成员（包括方法和属性），本类的非静态方法可以访问本类的静态成员（包括方法和属性）静态方法要慎重使用(原因是静态变量会一直存在，占用资源)，且在静态方法中不能出现this关键字。
> （2）父类中是静态方法，子类中不能覆盖为非静态方法；在符合覆盖规则的前提

下，在父子类中，父类中的静态方法可以被子类中的静态方法覆盖，但无多态。（在使用对象调用静态方法时，实则是调用编译时类型的静态方法）
> （3）父子类中，静态方法只能被静态方法覆盖，父子类中，非静态方法只能被非静态方法覆盖。
例：java中的main方法必须写成static的，因为在类加载时无法创建对象，静态方法可以不通过对象调用所以在类的main方法。 所在在类加载时就可以通过main方法入口来运行程序。

* 3.static修饰初始代码块，这时这个初始代码块就叫做静态初始代码块，该代码块只在类加载时被执行一次。 可以用静态初始代码块初始化一个类。动态初始代码块，写在类体中的“{}”，这个代码块是在生成对象的初始化属性是运行。这种代码块叫动态初始代码块。类在什么时候会被加载，创建对象时会加载类，调用类中静态方法或访问静态属性也是会加载类的。在加载子类时必会先加载父类，类加 载会有延迟加载原则，只有在必须加载时才会加载。

二、final修饰变量、方法和类

* 1.final修饰变量:变量被final修饰，就会变成常量（常量应大写），一旦赋值不能改变（可以在初始化时直接赋值，也可以在构造方法里也可以赋值，
只能在这两种方法里二选一，必须为常量赋值）；final的常量不会有默认初始值，对于直接在初始化是赋值时final修饰符常和static修饰符一起使用。
* 2.final修饰方法，被final修饰的方法将不能被其子类覆盖，保持方法的稳定不能被覆盖。
* 3.final修饰类，被final修饰的类将不能被继承。final类中的方法也都是final的。

> 注意一：final，不能用来修饰构造方法，在父类中如果有常量属性，在子类中使用常量属性时是不会进行父类的类加载。对象一旦创建属性就不会改变。用final修饰属性，也用final修饰类（强不变模式），用final修饰属性（弱不变模式）。
如：java.lang.String类，不变模式可以实现对象的共享（可以用一个对象实例赋值给多个对象变量）池化的思想，把需要共享的数据放在池中（节省资源空间，共享数据）
> 注意二：只有String类可以用“”中的字面值创建对象。在String类中，以字面值创建时，会到Java方法空间的串池空间中去查找，如果有就返 回串池中字符串的地址，并把这个地址付给对象变量。如果没有则会在串池里创建一个字符串对象，并返回其地址付购对象变量，当另一个以字面时，创建对象时则会重复上述过程。如果

是new在堆空间中创建String类的对象，则不会有上述的过程。String类中的intern()方法会将在堆空间中创建的String类对象中的字符串和串池中的比对，如果有相同的串就返回这个串的串池中的地址。不变模式在对于对象进行修改，添加操作是相当麻烦的，它会产生很多的中间垃圾对象。创建和销毁的资源的开销是相当大的。这就是推荐大量使用StringBuffer，而不建议使用String的原因！String类在字符串连接时会先的效率很低，就是因为它所产生的对象的属性不能被修改的，当连接字符串时也就是只能创建新的对象。造成很多无用的资源开销和浪费！

三、abstract修饰类和方法

* 1.abstract修饰类，使这个类成为一个抽象类，类将不能生成对象实例，但可以做为对象变量声明的类型，也就是编译时类型，抽象类就像当于一类的半成品，需要子类继承并覆盖其中的抽象方法。

* 2.abstract修饰方法，使这个方法变成抽象方法，即只有声明（定义）而没有实现，实现部分以";"代替。需要子类继承实现（覆盖）。

* 注意：
> （1）有抽象方法的类一定是抽象类。但是抽象类中不一定都是抽象方法，也可以全是具体方法。abstract修饰符在修饰类时必须放在类名前。 abstract修饰方法就是要求其子类覆盖（实现）这个方法。调用时可以以多态方式调用子类覆盖（实现）后的方法， 即抽象方法必须在其子类中实现，除非子类本身也是抽象类。
> （2）父类是抽象类，其中有抽象方法，那么子类继承父类，并把父类中的所有抽象方法都实现（覆盖）了，子类才有创建对象的实例的能 力，否则子类也必须是抽象类。抽象类中可以有构造方法，是子类在构造子类对象时需要调用的父类（抽象类）的构造方法。
> （3）不能放在一起的修饰符：final和abstract，private和abstract，static和abstract，因为abstract修饰的方法是必须在其子类中实现（覆盖），才能以多态方式调用，以上修饰符在修饰方法时期子类都覆盖不了这个方法，final是不可以覆盖，private是不能够继承到子类，所以也就不能覆盖，static是可以覆盖的，但是在调用时会调用编译时类型的方法，因为调用的是父类的方法，而父类的方法又是抽象的方法，又不能够调用，所以上的修饰符不能放在一起。

单例模式
这个模式主要注重懒汉模式（饿汉模式，任选其一），注册登记式单例。（熟记）
* java中单例模式是一种常见的设计模式，单例模式的写法有好几种，这里主要介绍三种：懒汉式单例、饿汉式单例、登记式单例。单例模式有以下特点：
> 1、单例类只能有一个实例。
> 2、单例类必须自己创建自己的唯一实例。
> 3、单例类必须给所有其他对象提供这一实例。
* 单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。每台计算机可以有若干个打印机，但只能有一个Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。总之，选择单例模式就是为了避免不一致状态，避免政出多头。
* 单例模式：多个内存地址只能指向同一块内存空间！
* 一、懒汉式单例
```
//懒汉式单例类.在第一次调用的时候实例化自己
/**
 * 单例模式
 * 不允许外部new多个
 *
 * //懒汉式单例模式
 * 一调用getInstance方法时  就实例化对象
 */
public class Singleton {
    private Singleton() {}
    private static Singleton single=null;
    //静态工厂方法
    public static Singleton getInstance() {
     if (single == null) {
         single = new Singleton();
     }
    return single;
    }
```

```
}
```

* Singleton通过将构造方法限定为private避免了类在外部被实例化，在同一个虚拟机范围内，Singleton的唯一实例只能通过getInstance()方法访问。（事实上，通过Java反射机制是能够实例化构造方法为private的类的，那基本上会使所有的Java单例实现失效。此问题在此处不做讨论，姑且掩耳盗铃地认为反射机制不存在。）但是以上懒汉式单例的实现没有考虑线程安全问题，它是线程不安全的，并发环境下很可能出现多个Singleton实例，要实现线程安全，有以下三种方式，都是对getInstance这个方法改造，保证了懒汉式单例的线程安全，如果你第一次接触单例模式，对线程安全不是很了解，可以先跳过下面这三小条，去看饿汉式单例，等看完后面再回头考虑线程安全的问题：

* 二、饿汉式单例

```
/**
     * 饿汉式单例模式
     * 当我一加载到静态方法区   就实例化对象
     */
//饿汉式单例类.在类初始化时，已经自行实例化
public class Singleton1 {
    private Singleton1() {}
    private static final Singleton1 single = new Singleton1
();
    //静态工厂方法
    public static Singleton1 getInstance() {
    return single;
    }
}
```

* 饿汉式在类创建的同时就已经创建好一个静态的对象供系统使用，以后不再改变，所以天生是线程安全的。

* 三、注册登记式单例（spring单例，map集合，反射机制加载）

```
//类似Spring里面的方法，将类名注册，下次从里面直接获取。
public class Singleton3 {
    private static Map<String,Singleton3> map = new HashMap<S
tring,Singleton3>();
    static{
    Singleton3 single = new Singleton3();
    map.put(single.getClass().getName(), single);
    }
    //保护的默认构造子
    protected Singleton3(){}
    //静态工厂方法,返还此类惟一的实例
    public static Singleton3 getInstance(String name) {
    if(name == null) {
        name = Singleton3.class.getName();
        System.out.println("name == null"+"--->name="+name);
    }
    if(map.get(name) == null) {
        try {
        map.put(name, (Singleton3) Class.forName(name).newIns
tance());
        } catch (InstantiationException e) {
        e.printStackTrace();
        } catch (IllegalAccessException e) {
        e.printStackTrace();
        } catch (ClassNotFoundException e) {
        e.printStackTrace();
        }
    }
    return map.get(name);
    }
    //一个示意性的商业方法
    public String about() {
    return "Hello, I am RegSingleton.";
    }
    public static void main(String[] args) {
```

```
    Singleton3 single3 = Singleton3.getInstance(null);
    System.out.println(single3.about());
    }
}
```

> 登记式单例实际上维护了一组单例类的实例，将这些实例存放在一个Map（登记薄）中，对于已经登记过的实例，则从Map直接返回，对于没有登记的，则先登记，然后返回。 这里我对登记式单例标记了可忽略，我的理解来说，首先它用的比较少，另外其实内部实现还是用的饿汉式单例，因为其中的static方法块，它的单例在类被装载的时候就被实例化了。

* 饿汉式和懒汉式区别
> 从名字上来说，饿汉和懒汉，
> 饿汉就是类一旦加载，　　就把单例初始化完成，保证getInstance的时候，单例是已经存在的了，
> 而懒汉比较懒，　　只有当调用getInstance的时候，才回去初始化这个单例。
> 另外从以下两点再区分以下这两种方式：

* 1、线程安全：
> 饿汉式天生就是线程安全的，可以直接用于多线程而不会出现问题，
> 懒汉式本身是非线程安全的，为了实现线程安全有几种写法，分别是上面的1、2、3，这三种实现在资源加载和性能方面有些区别。

* 2、资源加载和性能：
> 饿汉式在类创建的同时就实例化一个静态对象出来，不管之后会不会使用这个单例，都会占据一定的内存，但是相应的，在第一次调用时速度也会更快，因为其资源已经初始化完成，
> 而懒汉式顾名思义，会延迟加载，在第一次使用该单例的时候才会实例化对象出来，第一次调用时要做初始化，如果要做的工作比较多，性能上会有些延迟，之后就和饿汉式一样了。
* 至于1、2、3这三种实现又有些区别，
> 第1种，在方法调用上加了同步，虽然线程安全了，但是每次都要同步，会影响性能，毕竟99%的情况下是不需要同步的，
> > 第2种，在getInstance中做了两次null检查，确保了只有第一次调用单例的时候才会做同步，这样也是线程安全的，同时避免了每次都同步的性能损耗
第3种，利用了classloader的机制来保证初始化instance时只有一个线程，所

以也是线程安全的，同时没有性能损耗，所以一般我倾向于使用这一种。

* 什么是线程安全？
> 如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。
> 或者说：一个类或者程序所提供的接口对于线程来说是原子操作，或者多个线程之间的切换不会导致该接口的执行结果存在二义性,也就是说我们不用考虑同步的问题，那就是线程安全的。

抽象类和接口

方法的覆盖
* 方法的覆盖（`override`）:还可以称为重写（`rewrite`），是对从父类中继承来的方法进行改造，只有在子类继承父类时发生。
* 方法覆盖的规则
* 在子类中的覆盖方法与父类中被覆盖的方法应具有:
> 相同的方法名
> 相同的参数列表（参数数量、参数类型、参数顺序都要相同）
> 相同的返回值类型
> 子类覆盖方法的访问权限要不小于父类中被覆盖方法的访问权限
* 在重写方法上，可以使用`@Override`注解来标明是重写方法
> `@Override`表示重写（也可以省略）：编译器验证`@Override`下面的方法名是否是父类中所有的，如果没有则报错。
```
下面是一个例子：
//接口类  Condition
public interface Condition {
    public void zhileng();
    public void zhire();

}
//子类
public class CoditionA implements Condition{
```

```
    @Override
    public void zhileng() {
        // TODO Auto-generated method stub

    }

    @Override
//  public void zhire() {
//      // TODO Auto-generated method stub
//
//  }

}
```
* 抽象类的规则：
> 注意：
>> 抽象类不能够被实例化； 抽象类是不能使用`new`操作符来初始化的；
>> 包含的抽象方法必须在其子类中被实现，否则子类只能声明为`abstract`；
>> 子类可以覆盖父类方法，并将它定义为`abstract`。
>> 抽象方法不能为static; 抽象方法是非静态的。
> 在下列情况下，一个类必须声明为抽象类：
>> 当一个类的一个或多个方法是抽象方法时；
>> 当类是一个抽象类的子类，并且没有实现父类的所有抽象方法，即只实现部分；
>> 当一个类实现一个接口，并且不能为全部抽象方法都提供实现时

接口（Interface）
* 接口定义：
> 接口中只包含常量和抽象方法，而没有变量和方法的实现;接口对类来说是一套规范，是一套行为协议;接口不是一个类，不能实例化。

* 接口注意事项：
> 接口不是一个类，没有构造器，不能被实例化

> 接口使用interface关键字来定义，而不是class

> 接口默认：常量：public static final ; 抽象方法： public abstrac

t

> 接口在与多方面与类相似，但是它的目的是指`明相关`或者`不相关类的多个对象的共同行为`。

* 接口语法格式：
```
 [访问权限修饰符] `interface` 接口名 {  接口成员   }
```
* 接口和接口的继承:接口可以实现多继承
```
[修饰符]  interface   接口名 `extends` 接口1,接口2{ 接口的成员   }
```
* 接口和类的关系
> 类实现接口 — implements

>> 为了使用一个接口，你要编写实现接口的类

>> 如果一个类要实现一个接口，那么这个类就必须实现接口中所有抽象方法。否则这个类只能声明为抽象类

>> 多个无关的类可以实现一个接口，一个类可以实现多个无关的接口

>> 一个类可以在继承一个父类的同时，实现一个或多个接口

* 语法格式：
```
[修饰符]  class 类名 extends 类名 implements 接口1,接口2{  类的成员    }
```
> 接口可以实现多继承：
>> 用接口可以实现混合类型（主类型，副类型），在Java 中可以通过接口分出主次类型；

>> 主类型使用继承，副类型使用接口实现。

>> 接口可以使方法的定义和实现相分离，降低模块间或系统间的耦合性

* 接口与抽象类相同点与不同点

 |变量|构造方法|方法
|-----|:---:|:---:|:---:|
抽象类|无限制|子类通过构造方法链调用构造方法，抽象类不能用new操作符来实例化|无限制
接口|所有变量必须是public static final|没有构造方法。接口不能用new来实例化|所有方法必须是公共的抽象类实例方法

内部类

* 定义：是定义在另一个类中的类

那么我们为什么使用内部类呢？
* 内部类方法可以访问该类定义所有的作用域中的数据，包括私有的数据。也可以访问创建它的外围类对象的数据域。
* 内部类可以对同一个包中的其他类隐藏起来。
* 当想要定义一个回调函数且不想编写大量代码时，使用匿名内部类比较便捷。
特点：
* 内部类可以与外部类之间方便的进行私有属性的访问

* 内部类可以使用private声明，声明之后无法在外部类实例化内部类对象。
>* 语法：**内部类对象的实例化：外部类.内部类 对象 = new 外部类().new 内部类();**

* 使用static定义内部类就相当于是一个内部类：
>* 语法：**内部类对象的实例化：外部类.内部类 对象 = new 外部类().内部类();**

* 内部类可以在方法中定义（用的情况最多）

其实使用内部类最大的优点就在于它能够非常好的解决多重继承的问题，但是如果我

们不需要解决多重继承问题，那么我们自然可以使用其他的编码方式，但是使用内部类还能够为我们带来如下特性（摘自《Think in java》）：

Think in java:
* 1、内部类可以用多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独立。

* 2、在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类。

* 3、创建内部类对象的时刻并不依赖于外围类对象的创建。

* 4、内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。

* 5、内部类提供了更好的封装，除了该外围类，其他类都不能访问。

```
//外部类
public class Outer {
    private String msg = "Hello World";
    //内部类
    class Inner{
        public void print() {
            System.out.println(msg);
        }
    }
    public void fun() {
        new Inner().print();
    }

}
//测试类
public class Test {
```

```
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Outer Out = new Outer ();
        Out.fun();
    }

}
输出：Hello World
```

上面的这个代码没问题，但是结构有问题。虽然牺牲了一个程序的结构，但是达到了相应的目的。

内部类的外部类的属性能够相互调用、访问吗？

```
//外部类
public class Outer {
    private String msg = "Hello World";
    //内部类
    class Inner{
        //为内部类添加属性
        private String info = "你好！";
        public void print() {
            System.out.println(msg);
        }
    }
    public void fun() {
        Inner in = new Inner();
        in.print();
        System.out.println(in.info);
    }
}
//测试类
public class Test {

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

```
        // TODO Auto-generated method stub
        Outer Out = new Outer ();
        Out.fun();
    }

}
```
一旦使用了内部类，是的私有属性的访问变得非常的方便,但是有一点，访问属性不能够加上this.属性名。编译出错。
可以使用外部类。this。属性名，这样使用

添加使用this关键字
```
//外部类
public class Outer {
    private String msg = "Hello World";
    //内部类
    class Inner{
        public void print() {
            //添加使用this关键字*
            System.out.println(Outer.this.msg);
        }
    }
    public void fun() {
        Inner in = new Inner();
        in.print();
    }
}
//测试类
public class Test {
    public static void main(String[] args) {
        Outer Out = new Outer ();
        Out.fun();
    }

}
```

```
上面的代码有一个特点：通过外部类的fun()方法访问内部类的操作，那么内部类能不能像普通对象那样直接在外部产生实例化对象调用呢？

### 特点：

**这种情况需要观察.class文件**
* 内部类的class文件形式时：Outer$Inner.class。
* 所有的$ 是在文件中的命名；如果在程序中就会变成“ . ” 符号，也就是内部类的名称就是：外部类.内部类。



* **内部类对象的实例化：外部类.内部类 对象 = new 外部类().new 内部类();**
* 如果我想得到内部类的实例化对象之前，应该先实例化外部类实例化对象。
```

```
//外部类
public class Outer {
    private String msg = "Hello World";
    //内部类
    class Inner{
        public void print() {
            //添加使用this关键字*
            System.out.println(Outer.this.msg);
        }
    }

}
//测试类
public class Test {
    public static void main(String[] args) {
    //修改获取实例化对象的语法
        Outer.Inner  Out = new Outer().new Inner();
        Out.print();
    }
```

```
}
```

使用static修饰内部类
使用static定义的属性或者方法是不受类实例化对象控制的，所以如果使用static定义了一个内部类。它不会受到外部类的实例化对象控制。

如果内部类被static修饰了，那么这个内部类就会变为一个外部类，并且只能访问外部类中定义的static操作。static只能访问static修饰的。

```
//外部类
public class Outer {
    private String msg = "Hello World";
    //内部类
    static class Inner{
        public void print() {
            //添加使用this关键字*
            System.out.println(Outer.this.msg);
        }
    }

}
```
此时的这个类是无法通过编译的，如果此时想要取得内部类的实例化对象，那么应该怎么修改代码呢？

之前语法形式是：
* **内部类对象的实例化：外部类.内部类 对象 = new 外部类().new 内部类();**
更改后的语法形式是：
* **内部类对象的实例化：外部类.内部类 对象 = new 外部类().内部类();**
```
//外部类
```

```
public class Outer {
    //修改属性为staitc
    private static String msg = "Hello World";
    //内部类
    //修改属性为staitc
    static class Inner{
        public void print() {
            //添加使用this关键字*
            System.out.println(Outer.this.msg);
        }
    }

}
//测试类
public class Test {
    public static void main(String[] args) {
    //修改获取实例化对象的语法
        Outer.Inner  Out = new Outer.Inner();
        Out.print();
    }

}
```

在方法中定义内部类（常用，重点）
* 内部类可以在任意位置上定义：类中，代码块，方法里面；其中方法中定义内部类是比较常见的。

```
//外部类
public class Outer {
    private String msg = "Hello World";
    //方法
    public void fun() {
        //方法啊中定义内部类
        class Inner{
```

```
            public void print() {
                System.out.println(Outer.this.msg);
            }
        }
        new Inner().print();

    }

}
//测试类
public class Test {
    public static void main(String[] args) {
        new Outer().fun();
    }
}
```

但是方法中会接受参数，定义变量。

* **访问方法中定义的参数及变量**

```
//外部类
public class Outer {
    private String msg = "Hello World";
    //方法
    public void fun(int num) {
        //变量
        double score = 99.6;
        //方法啊中定义内部类
        class Inner{
            public void print() {
                System.out.println("属性"+Outer.this.msg);
                System.out.println("方法参数"+num;
                System.out.println("方法变量"+score);
            }
        }
```

```
        new Inner().print();

    }

}
//测试类
public class Test {
    public static void main(String[] args) {
        new Outer().fun(100);
    }
}
```

注意 **此时没有加入任何的修饰，方法中的内部类可以访问方法的参数以继定义的变量，但是这个只适用于JDK1.8之后的版本！！！**

那么之前的版本呢？ 需要使用final修饰

```
//外部类
public class Outer {
    private String msg = "Hello World";
    //方法
    // 添加final修饰
    public void fun(final int num) {
        //变量
        final double score = 99.6;
        //方法啊中定义内部类
        class Inner{
            public void print() {
                System.out.println("属性"+Outer.this.msg);
                System.out.println("方法参数"+num;
                System.out.println("方法变量"+score);
            }
        }
        new Inner().print();
```

```
    }

}
//测试类
public class Test {
    public static void main(String[] args) {
        new Outer().fun(100);
    }
}
```

异常及异常处理
* 引言：在Java中，运行时错误会作为异常抛出（而不是编译时的语法错误）。异常是一种对象，表示组织正常进行程序执行的错误或者情况。如果异常没有被处理，南无程序将会非正常终止。

* Throwable下包括：Error和Exception

> Error：Error层次结构描述了JAVA运行时系统内部错误和资源耗尽。
> Exception：有两个分支：一个分支是派生于RuntiomeException;另一个是分支包含其他异常（IOException）。
* 派生于Error或RunTimeException类的所有异常成为`未检查性(unchecked)异常`，所有其他的异常成为`已检查(checked)异常`。
~~注：图片在JAVA技术卷I P473页~~
异常类型
* 检查性异常（checked exception）

> 若系统运行时可能产生该类异常，则必须写出相应的处理代码，否则无法通过编译
> 非RuntimeException异常

~~注：　　需要进行异常处理，否则无法进行编译~~

检查异常	说明
ClassNotFoundException	无法找到想要创建对象的类文件
IOException	I/O异常的根类
FileNotFoundException	不能找到文件
EOFException	文件结束
IllegaAccessException	请求方法不存在
InterruptedException	线程中断

* 非检查性异常（unchecked exception）

> 若系统运行时可能产生该类异常，则不必在程序中声明对该类异常的处理，就可以编译执行
> RuntimeException：运行时异常

非检查异常	说明
RuntimeException	java.lang包中多数异常的基类
ArithmeticException	算术错误，如除以0
IllegalArgumentException	方法收到非法参数
ArrayIndexOutOfBoundsException	数组下标越界
NullPointerException	试图访问null对象引用

* 规则：
> 有程序错误导致的异常属于RuntimeException；而程序本身没有问题，但由于I/O错误这类问题导致的异常属于其他异常。

> 如果出现RunTimeException异常，那么就一定是你的问题。很有道理的定理。

编译时异常和运行时异常的区别
最简单的说法：

* javac出来的异常就是编译时异常，就是说把源代码编译成字节码（class）文件时报的异常，一般如果用Eclispe,你敲完代码保存的时候就是编译的时候。

* java出来的异常就是运行时异常

Java异常可分为3种：

* (1)编译时异常:Java.lang.Exception

* (2)运行期异常:Java.lang.RuntimeException

* (3)错误:Java.lang.Error

>* Java.lang.Exception和Java.lang.Error继承自Java.lang.Throwable;

>* Java.lang.RuntimeException继承自Java.lang.Exception.

编译时异常： 程序正确，但因为外在的环境条件不满足引发。例如：用户错误及I/O问题----程序试图打开一个并不存在的远程Socket端口。这不是程序本身的逻辑错误，而很可能是远程机器名字错误(用户拼写错误)。对商用软件系统，程序开发者必须考虑并处理这个问题。Java编译器强制要求处理这类异常，如果不捕获这类异常，程序将不能被编译。

运行期异常： 这意味着程序存在bug，如数组越界，0被除，入参不满足规范.....这类异常需要更改程序来避免，Java编译器强制要求处理这类异常。运行时异常是java,lang.RuntimeException,出现这种异常，那么就一定是你的错误。

系统错误： 一般很少见，也很难通过程序解决。它可能源于程序的bug，但一般更可能源于环境问题，如内存耗尽或者内部错误。错误在程序中无须处理，而有运行环境处理。

* 顺便说一下：编译期和运行期的区别：

编译期和运行期进行的操作是不相同的，编译器只是进行语法的分析，分析出来的错误也只是语法上的错误，而运行期在真正在分配内存..

比如说你写一个while循环，一直往栈里写，编译器是不会出错的，可是运行期就会

出现栈满的错误··

声明异常
为了在方法中声明异常，就要在方法头中使用关键字throws, 若是多个异常，异常间用逗号隔开。
```
 public void Mwthod1() throws Exception1,Exception2   {   }
```
声明异常关键字是：`throws`

抛出异常
检测到错误的长须可以创建一个合适的异常类型的实例并抛出他，成为抛出异常
```
throw new IllegaAccessException("Wrong Arfument 出现异常啦！！");
```
抛出异常关键字是：`throw` 。

> throws声明异常 有可能出现问题的异常类型

> 次方法中有异常 所以要向上抛 谁调用我 谁需要解决异常

捕获异常
* 处理异常两种方式：

> 自行处理：可能引发异常的语句封入在 try 块内，而处理异常的相应语句则封入在 catch 块内。

> 回避异常：在方法声明中包含 throws 子句，通知潜在调用者，如果发生了异常，必须由调用者处理。

* 异常处理机制

> 每次try块有异常抛出，系统会试着从上到下往每个catch块中传参，直到遇到一个类型匹配的catch块为止。

> 如上示例中最后一个catch块中的形参为Exception类型，它是所有异常类的父类，任何异常都可以传参到该块中，该块可以处理任何类型的异常。因此，这个catch块只能放到最后面，否则所有的异常都被它处理了，其他的catch块就不能分门别类的起作用了。

> 一般一个catch块中是专门处理异常的代码，在程序中这里还可以是记录日志的语句，当发生异常时记录该日志，无异常时将不会记录。

> 如果编写过程中我们违背了异常继承顺序，会产生编译错误

```
catch (ArrayIndexOutOfBoundsException e) {
   System.out.println(“Out of Bounds!”);
}
catch (RuntimeException e) {
   System.out.println(“Runtime Exception!”);
}
catch (Exception e) {
   System.out.println(“Exception!”);
}
```

* 规则

> 如果程序抛出多个不同类型的异常，我们需要多个catch()语句来处理。catch块，是用来捕获并处理try块抛出的异常的代码块。

```
try{}表示可能发生异常的语句
catch( )内的参数异常类对象的声明
catch{}内的语句是对异常的处理
```

> 和特殊异常类相关联的catch()块必须写在和普通异常类相关联的catch()之前。

* finally子句

> 无论异常是否产生，finally子句总是会被执行的。

> 通常在finally语句中可以进行资源的清除操作，如：关闭打开文件、删除临时文件

> 对应finally代码中的语句，即使try代码块和catch代码块中使用了return语句退出当前方法或般若break跳出某个循环，相关的finally代码块都有执行。

> 当try或catch代码块中执行了System.exit(0)时，finally代码块中的内容不被执行

```
public static void main(String[] args) {
        //异常：运行时  出现的错误，而不是编译时出现的错误
        /**
         * 不需要对异常做处理
         * eg:数组下标越界。。。。等
         *
         * 需要对异常做处理 解决异常报错
         * Exception in thread "main" java.lang.StackOverflow
Error
         */
        int[] b;
        try {
            b = new int[5];
            //数组下标越界
            System.out.println(b[6]);
        }
        catch (NullPointerException e) {
//          e.printStackTrace();
            System.err.println("空指针异常");
        }
        //自行捕获异常 并处理  继续向下运行
        catch (ArrayIndexOutOfBoundsException e) {
//          e.printStackTrace();
            System.err.println("数组下标越界异常");
        }
        catch(Exception e){
```

```
            System.out.println("出现异常！");
        }
        finally{
            System.out.println("不管try是否有异常，不论catch到异
常与否，都会会执行finally方法！");
        }
        System.out.println("----");
    }

    static void  method1(){
        method1();
    }
    void  method2() throws IOException{

        //出现检查性异常 必须解决解决方式：1、try catch  2、声明thro
ws异常
        System.out.println("IOException");
    }
    void  method3(){
        try {
            method2();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    void method4() throws IOException{
        method2();
    }
/**
 * throw抛出异常
 * throws声明异常  有可能出现问题的异常类型
 */
    //次方法中有异常 所以要向上抛  谁调用我 谁需要解决异常
    void method5()throws IOException{
        throw new ArrayIndexOutOfBoundsException();//属于非检查
```

性异常
//   throw new IOException; //属于检查性异常
    }
```

#### 自定义异常
如果Java提供的异常类型不能满足程序设计的需要，我们可以定义自己的异常类型。用户自定义的异常类应为 Exception 类（或者Exception 类的子类）的子类
```
public class Exception_sample_5 {
  public static void main(String[] args) {
    try {
      int a = new Scanner(System.in).nextInt();
      int b = new Scanner(System.in).nextInt();
      add(a,b);
    } catch (MyException e) {
      System.out.println("输入错误!");
    }
  }
  public static void add(int a,int b)throws MyException{
    if(a<0&&b<0){
      throw new MyException();
    }else{
      System.out.println(a+b);
    }
  }
}

```

---
### 工具类
#### 常用类
* java.lang
> Java语言包，包含String、StringBuffer、Integer、Math、System。任何类中，该包中的类都会被自动导入。在java.lang不用引包.
* java.util
```

```
> 包含一些实用的工具类（ 包含list, calendar, date等类）
* java.io
> 提供多种输入/输出功能的类。
* java.net
> 提供网络应用功能的类。

 |java.lang包下的类| 
:----|:----|:----
Boolean|Object|Error
Byte|String|Throwable
Character|StringBuffer|Exception
Double|StringBuilder|ClassNotFoundException
Float|System|NullPointerException
Integer|Math|NumberFormatException
Long|Runnable(接口)|RuntimeException
Short|Thread|ArithmeticException

 |java.util包下的类| 
:----|:----|:----
Collection(接口)|Arrays|Calender
Iterator(接口)|Set(接口)|Date
ListIterator|HashSet|Random
List(接口)|TreeSet|Scanner
ArrayList|Map(接口)|Collections
LinkedList|HashMap| 
Vector|Hashtable| 
Stack|TreeMap| 

#### Object类
* Object介绍：
* 1、bject类是所有类的超类。
> Object是Java语言中唯一一个没有父类的类。(一个没有爸爸的孩子！！)

* 2、一个类可以不是Object类的直接子类，但一定是Object类的子类，Java中的每一个类都是从Object扩展来的。
```

* 3、在Object类中定义的方法，在所有类中都可以使用。

> 1)比较两个对象引用的值是否相等（比较哈希地址）
```
public boolean equals(Object obj)

比较两个对象引用的值是否相等（比较地址）。指出调用该方法的对象是否与obj对象相等。即地址是否相等。
```

> 2)返回十进制整数，唯一标识一个对象
```
public int hashCode()

该方法返回对象的哈希码，哈希码是一个代表对象的十六进制整数，比作对象的身份证号。在程序运行期间，每次

调用同一个对象的hashCode()返回的哈希码必定相同，但是多次执行同一个程序，

程序的一次执行和下一次执行期间同一个对象的哈希码不一定相同。

实际上默认的哈希码是将对象的内存地址通过某种转换得到的，所以不同对象会有不同的哈希码。
```

> 3)返回　类名@hashcode
```
public String toString()

返回　类名@hashcode ；事实上返回这样的字符串没有什么实际的意义。一般子类都会覆盖该方法，让它返回有意义的文本。
```

* equals()方法判断相等

* 理解equals()方法和==运算符的区别是非常重要的。

> 默认情况下（即没有被重写时）equals()只能比较引用类型，"=="既能比较引用类型又能比较基本类型。
> equals()方法从Object类继承，即比较对象引用的值
> 一般都被子类方法覆盖，不再比较引用的值
* "=="运算符:
> 比较基本数据类型：相当于算术等号
> 比较引用数据类型：比较引用的值，不能被覆盖。

* 通常情况，子类要重写equals( )，改变它的含义。所以有的类中equals( )是比较地址，有的类中该方法就不比较地址，具体的，就看子类新定义的该方法的规定。看看包装类中的equals()方法？
* 在java中有个规定：如果equals( )返回两个对象是相等的，那这两个对象上调用hashCode( )返回的整数必须相等。否则在使用Hash类型集合时就会产生错误。
>* 注意：覆盖equals( )方法同时，还要记得覆盖hashCode(）方法。需要说明，如果equals( )返回两个对象不等，它们的hashCode( )也可以返回相同的整数。但是最好让它们的hashCode( )返回不同的整数，这有利于提高Hash类型集合的性能。

* 1、重写equals方法时，一定要重写hashcode（）方法吗？
> hashcode的调用的条件：
>> 想往map里面放一个类作为map的键值，这个类又是自己设计的；
>> 虽然类不是自己写的，但是你修改了这个类的equals方法；
* 2、如果满足上述调用条件，就要注意重写hashcode方法。 这样 当你往map里放值得时候，系统会调用这个对象的.hashcode()方法来生成相应的hash值，来映射相应的对象。
> 如果同一个类的两个对象的属性值相等，那么他们的hashcode一定相等吗？
这个要看你具体如何实现你的hashcode，如果你希望他们的值一样hashcode也一样，你就可以这样实现。 但是hashcode的实现，一般要满足几个特征，比如 自反性，传递性什么的。

包装类
* Everything is object.
> Java编程语言不把基本数据类型看作对象。Java 编程语言提供包装类来将基本数据类型看作对象。

> 在功能上包装类能够完成数据类型之间（除boolean）的相互转换，尤其是基本数据类型和String类型的转换。
> 包装类中包含了对应基本数据类型的值，封装了String和基本数据类型之间相互转换的方法，还有一些处理这些基本数据类型时非常有用的属性和方法

* 基本数据类型和对应包装类

数据类型	包装类
boolean	Boolean
byte	Byte
char	Char
double	Double
float	Float
int	Int
long	Long
short	Short

* 包装类常用方法和属性

类型	最大值	最小值
byte	Byte.MAX_VALUE 127	Byte.MIN_VALUE -128
short	Short.MAX_VALUE 32767	Short.MIN_VALUE -32767
int	Int.MAX_VALUE 0x7fffffff	Int.MIN_VALUE 0x80000000
long	Long.MAX_VALUE 0x7fffffffffffffffL	Long.MIN_VALUE 0x8000000000000000L

* 字符串与基本数据类型、包装类型转换图

> * `字符串类型(String)`-----使用包装类构造器valueOf()------->`包装类Wrapper`
> * `包装类Wrapper`------包装类xxxValue()方法--------->`基本数据

类型`
> * `基本数据类型`---------String类的valueOf方法，用连字符"+"------>`字符串类型(String)`
> * `基本数据类型`---------包装类通过构造器.valueOf()-------->`包装类Wrapper`
> * `字符串类型(String)`---------包装类的parsexxx(方法)------->`基本数据类型`

数值-字符串转换（String转Double）
```
String ss = "3.141592653";
double value = Double.valueOf(ss.toString());
```

Double.parseDouble()与Double.valueOf()区别
* Double.parseDouble方法是把数字类型的字符串，转换成double类型
* Double.valueOf方法是把数字类型的字符串，转换成Double类型
```
        double a = 0.009;
        String b = "1993";

        double parseDouble = Double.parseDouble(b);
        System.out.println(parseDouble);
        Double valueOf = Double.valueOf(a); //1993.0
        Double valueOf2 = Double.valueOf(b);
        System.out.println(valueOf);//0.009
        System.out.println(valueOf2);//1993.0
结果是：
1993.0
0.009
1993.0
```

可以看出大体是没有任何区别，就是返回值的类型区别！

* Double.parseDouble()方法的源码：

```
public static double parseDouble(String s) throws NumberForma
tException {
        return FloatingDecimal.readJavaFormatString(s).double
Value();
}
```

* Double.valueOf()方法的源码：
```
  public static Double valueOf(String s) throws NumberFormatE
xception {
          return new Double(FloatingDecimal.readJavaFormatStr
ing(s).doubleValue());
  }
```

* valueOf方法在返回的时候new Double()，所以返回的是Double。

```
public static void main(String[] args) {
        //如果忘记包装类型长度，试着输出一下最大值与最小值
        System.out.println(Byte.MAX_VALUE);
        System.out.println(Byte.MIN_VALUE);

        System.out.println(Short.MAX_VALUE);

        System.out.println(Integer.MIN_VALUE);

        System.out.println(Long.MAX_VALUE);
        System.out.println(Long.MIN_VALUE);

        //类型转换
        int a=10;
```

```
        Integer a1 = new Integer(a);
        System.out.println(a1);

        Integer a2 = new Integer("9999");
        //Integer 的valueof 方法可以将其他数据类型     转换为自己的I
nteger数据类型
        Integer a4 = Integer.valueOf("99");
        System.out.println(a4);

        //Integer的intValue()或者是longValue方法可以转换为   将自己
转换为对应的数据类型
        int a6 = a2.intValue();
        long a7 = a2.longValue();
        System.out.println(a6+",,,,"+a7);

        //
        int b1 = 10;
        /**
         * Integer b2=b1;
         * 从基本数据类型    直接转换为包装数据类型   叫做自动装箱
         *
         * int b3=b2;
         * 从包装数据类型    直接转换为基本数据类型    叫做自动拆箱
         */
        Integer b2=b1;
        int b3=b2;

    }
```

String字符串

* 1、字符串是我们在编程中最常使用的一种数据类型，它的表现形式可以分为两种：
> String
> StringBuffer。
* 字符串不属于8种基本的数据类型，而是一种引用类型。
* String对象代表一组不可改变的Unicode字符序列，对它的任何修改实际上又

产生一个新的字符串，String类对象的内容一旦被初始化就不能再改变。
* StringBuffer对象代表一组可改变的Unicode字符序列。
* String类是final类型的类。

* 2、String的创建：有两种方式:
> 静态方式（常用）。`String 变量 = "字符串";`像是给变量直接赋值一样来使用。如：String s1 = “abc”; String s2 = “abc”;

> 动态方式。动态的内存分配，`使用new运算符进行`。如：String s3= new String(“abc”); String s4= new String(“abc”);

* 3、两种方式创建的字符串区别：
> 使用静态方式创建的字符串，在方法区的常量池中只会产生唯一一个字符串对象，使用该方式产生同样一个字符串时，内存中不再开辟另外一块空间，而是两个引用变量指向同一个字符串对象。

> 使用动态方式创建的字符串，在堆内存中会产生出不同的对象

* 4、连接操作符
> 使用concat方法连接两个字符串。`String s3 = s1.concat(s2);`
> 使用(+)号连接两个或者多个字符串。 `String s3 = s1 + s2;`

分割符号
```
Scanner sc = new Scanner(System.in);
syso("请输入数字用,分割")
String s = sc.next();//控制台输入
String[] sarray = s.split(",");
```

* 5、字符串比较

> 任何一个String对象在创建之后都不能对它的内容作出任何改变（immutable）

> 连接、获得子串和改变大小写等操作，如果返回值同原字符串不同，实际上是产

生了一个新的String对象

> 在程序的任何地方，相同的字符串字面常量都是同一个对象

> String类重置了equals方法，用于比较两个字符串的内容

* 字符串`==`与`equals`区别：
```
String str1 = “hello ”;
String str2 =  new String(“hello”);
System.out.println(str1==str2);---------->false
System.out.println(str1.equals(str2));---------->true
```
上面这个实例：str1创建字符串hello,str2使用new关键字一个字符又创建一个字符串hello，可以说实在堆内存当中分配了一快内存空间，此时两个字符串的内存地址是不同的，但字符串内容是相同的。

总结：操作符`==`只能检测两个字符串`是否只想同一个对象`（即内存地址是否相同），但他不会告诉你两个字符串的内容是否相同。若想检查`字符串内容是否相同`，可以使用`equals()方法`进行检测。

* 字符串`compareTo`：
如果两个字符串相等，equals方法返回true;如果他们不相等，返回false。compareTo方法会根据一个字符串是否等于、大于、小于另一个字符串，分贝返回0、正整数、负整数。
```
输入两个城市，然后以字母表顺序进行显示
public static void main(String[] args) {
    Scanner s = new Scanner(System.in);

    System.out.println("请输入第一个城市：");
    String city1 = s.nextLine();

    System.out.println("请输入第二个城市：");
    String city2 = s.nextLine();
```

```
    if(city1.compareTo(city2)>0){
        System.out.println("第一个城市"+city1+" "+city2);
    }else{
        System.out.println("第一个城市"+city2+" "+city1);
    }
}
```
调用city1.compareTo(city2)比较两个字符串city1和city2，返回一个负值表明city1小于city2.

* 注： 使用`.nextLine()`为了能够输入一个包含空格的字符串。
```

* 6、String对象的比较方法

方法|描述
:--|:--|
equals(s1)|如果该字符串等于字符串s1，返回值为true
equalsIgnoreCase(s1)|如果该字符串等于字符串s1，返回值为true,不区分大小写
compareTo(s1)|返回一个大于0、等于0、小于0的整数，表明一个字符串是否大于等于小于s1
compareToIgnoreCase(s1)|和compareTo一样，除了比较是区分大小写的之外
startsWith(prefix)|如果字符串以特定的前缀开始，返回值为true
endsWith(suffix)|如果字符串以特定的后缀结束，返回值为true
contains(s1)|如果s1是该字符串的子字符串，返回值为true

* String类中常用的方法

方法|含义
:--|:--|
String substring(int begin)|返回一个新字符串，该字符串是从begin开始的字符串的内容
String trim( )|返回新的字符串，忽略前导空白和尾部空白
int   length( )|返回此字符串的长度
&amp;nbsp;|&amp;nbsp;
String toLowerCase( )|将String对象中的所有字符都转换为小写
```

boolean equals(String)|判断两个字符串对象的内容是否相等
boolean equalsIgnoreCase(String)比较两个字符串的内容是否相等，忽略大小写
String substring(int begin,int end)|返回一个新字符串，该字符串是从begin开始到end-1结束的字符串的内容
int   indexOf/lastIndexOf(char)|返回指定字符在此字符串中第一次/最后一次出现处的索引。
int   indexOf/lastIndexOf(char,int)|从指定的索引开始搜索，返回在此字符串中第一次/最后一次出现指定字符处的索引
int   indexOf/lastIndexOf(String)|返回第一次出现的指定子字符串在此字符串中的索引
int   indexOf/lastIndexOf(String,int)|从指定的索引开始搜索，返回在此字符串中第一次/最后一次出现指定字符串处的索引

* 数组length属性：array.length;-------------------->没有括号
* String对象length属性：string.length();------------->有括号
* 判断字符串是否为空：

```
String str = "hello";
syso(str.isEmpty());-----------false
syso("".isEmpty());----------->true
```

如果觉得`isEmpty()`使用不方便，可以使用`"".equals(str)`来进行判断。

String类虽然提供了大量的支持方法，但是却少了一个重要的方法，initcap()方法，首字母大写，而这样的功能只能自己实现。

```
public static void main(String args[]){
    String str = "HELLO";
    syso(initcap(str));
}
public static String initcap(String temp){
    //return temp.substring(0,1).toUpperCase()+temp.substring
(1).toLowerCase();
    return temp.substring(0,1).toUpperCase()+temp.substring
(1);
```

```
}
输出：Hello
```

虽然Java中没有这样的功能，但是一些第三方的组件包会提供，例如：apache的commons会提供，他会封装好供我们使用。

StringBuffer类

* 1、StringBuffer类用于内容可以改变的字符串
> 可以使用StringBuffer来对字符串的内容进行动态操作，不会产生额外的对象

* 2、StringBuffer对象的创建
> 构造一个其中不带字符的字符串缓冲区，其初始容量为 16 个字符
StringBuffer 变量名 = new StringBuffer();
> 构造一个不带字符，但具有指定初始容量的字符串缓冲区。
StringBuffer 变量名 = new StringBuffer(int capacity);
> 构造一个字符串缓冲区，并将其内容初始化为指定的字符串内容
StringBuffer 变量名 = new StringBuffer(String value);

* 3、StringBuilder常用方法

方法	含义
int capacity( )	返回当前容量
int length( )	返回长度（字符数）
StringBuilder reverse( )	将此字符序列用其反转形式取代
void setCharAt(int index,char ch)	将给定索引index处的字符设置为ch
StringBuilder delete(int begin,int end)	移除此序列的子字符串中的字符
char charAt(int index)	返回此序列中指定索引处的 char 值
String toString( )	将StringBuilder对象转换成相应的String
StringBuilder append(String str)	将指定的字符串追加到此字符序列
StringBuilder append(int num)	将 int 参数的字符串表示形式追加到此

```
序列
StringBuilder append(Object o)|追加 Object 参数的字符串表示形式
StringBuilder insert(int index,String str)|将字符串插入此字符序列中
StringBuilder insert(int index,char ch)|将字符插入此字符序列中
StringBuilder insert(int index,Object o)|将 Object 参数的字符串表示形式插入此字符序列中

* 4、StringBuffer常用方法

方法|含义
:--|:--
int  capacity( )|返回当前容量
int  length( )|返回长度（字符数）
StringBuffer  reverse( )|将此字符序列用其反转形式取代
void  setCharAt(int,char)|将给定索引index处的字符设置为 ch
StringBuffer   delete(int begin,int end)|移除此序列的子字符串中的字符
char  charAt(int)|返回此序列中指定索引处的 char 值
String  toString( )|将StringBuilder对象转换成相应的String
StringBuffer  append(String str)|将指定的字符串追加到此字符序列
StringBuffer append(int num)|将 int 参数的字符串表示形式追加到此序列
StringBuffer append(Object o)|追加 Object 参数的字符串表示形式
StringBuffer insert(int index,String str)|将字符串插入此字符序列中
StringBuffer insert(int index,char ch)|将字符插入此字符序列中
StringBuffer insert(int index,Object o)|将 Object 参数的字符串表示形式插入此字符序列中

### String类与StringBuilder类的比较
* Java中定义了String与StringBuffer两个类来封装对
> 字符串的各种操作
* String类与StringBuffer类都被放到了java.lang包中
* 两者的主要区别在于
> String类对象中的内容初始化不可以改变
```

> StringBuffer类对象中的内容可以改变
* StringBuffer和StringBuilder都是长度可变的字符串。
* 两者的操作基本相同。
* 两者的主要区别在于
> StringBuffer类是线程安全的；
> StringBuilder类是线程不安全的。
> StringBuffer在JDK1.0中就有，而StringBuilder是在JDK5.0后才出现的。
> StringBuilder的一些方法实现要比StringBuffer快些。

```
输入一个手机号码，将中间四位使用星号替代。例如：输入：13312349876,  输出：133****9876
public static void main(String[] args) {
    // TODO Auto-generated method stub
    String s = "13588889999";
    String a = s.substring(0, 3)+"****"+s.substring(7,s.length());
    System.out.println(a);
}
```

Math

* Math类
> Math类提供了大量用于数学运算的方法
> Math类是final类，因此不能从Math类继承
> Math类中的方法都是static方法，因此不必创建Math类的对象就可以直接使用该类的方法
* Math类中的常量

```
public static final double PI
public static final double E
```

* Math类中的常用方法

方法|含义

```
:--|:--
static int abs(int)|返回参数的绝对值，返回值类型与参数类型相同
static double abs(double)|返回参数的绝对值
static double ceil(double)|返回大于所给参数的最小的整数值
static double floor(double)|返回不大于所给参数的最大的整数值
static int   max(int a,int b)|返回两个int值中较大的一个
static double max(double,double)|返回两个double值中较大的一个
static int min(int a,int b)|返回两个int值中较小的一个
static double min(double,double)|返回两个double值中较小的一个
static double random( )|返回在0.0~1.0之间的随机的double值
static int round(double)|返回同所给值最接近的整数，采用4舍5入法
static double sin/cos/tan(double)|返回给定的弧度值对应的三角函数值
static double sqrt(double)|返回所给值的平方根，若所给值为负数则返回NaN

#### Date

* 1、Date 类
> java.util.Date。
> Date 类表示指定的时间信息，可以精确到毫秒。
> 不支持国际化。
> 该类的许多方法已过时。
> 获取当前系统时间：

* 注意：现在我们更应该多使用 Calendar 类实现日期和时间字段之间转换，使用 DateFormat 类来格式化和分析日期字符串；Date 中的相应方法已废弃

* 2、Date类常用方法

方法|含义
:--|:--
boolean after(Date when)|测试此日期是否在指定日期之后
boolean before(Date when)|测试此日期是否在指定日期之前
int compareTo(Date anotherDate)|比较两个日期的顺序。如果参数Date等于此 Date，则返回值0；如果此Date在 Date参数之前，则返回小于0 的值；如果此Date在Date参数之后，则返回大于0 的值。
```

````
boolean equals(Object obj)|比较两个日期的相等性。

* 3、 Calendar类
> java.util.Calendar。
>> java.util.GregorianCalendar
> 常用的日期处理的对象。可以设置自己的时区和国际化格式。
> 是一个抽象类。
> Calendar 抽象类定义了足够的方法，让我们能够表述日历的规则 。
>* 获取Calendar对象的实例：
```
Calendar c = Calendar.getInstance();
```
>* 设置Calendar实例所表示的时间：
```
c.set(2016,9,9);
```
>* 获取指定的时间属性
```
c.get(Calendar.YEAR);
```
* Calendar类常用属性

属性|含义
:--|:--
static int HOUR|小时时间
static int MINUTE|分时间
static int SECOND |秒时间
static int DATE |日期的Date部分
static int MONTH |日期的Month部分
static int YEAR |日期的年部分
```
public static void main(String[] args) {
    Calendar c1 = Calendar.getInstance();
    c1.set(2012, 2,8);
    System.out.print(c1.get(1));
    //瞬间可用毫秒值来表示，它是距历元（即格林威治标准时间 1970 年 1
````
````

月 1 日的 00:00:00.000，格里高利历）的偏移量。
    //获取对应数据
    //获得指定年月日时当年的第几天
    System.out.print(c1.get(Calendar.DAY_OF_YEAR));
}
```
* SimpleDateFormat类

>* SimpleDateFormat类 ：java.text.SimpleDateFormat
> 一个以与语言环境相关的方式来格式化和分析日期的具体类。是抽象类java.text.DateFormat类的子类。
> SimpleDateFormat使得可以选择任何用户定义的日期-时间格式的模式。
>* SimpleDateFormat类的使用
获取SimpleDateFormat的实例
```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
```
将日期格式成指定的字符串
```
sdf.format(new Date());
```
将格式化的字符串转换成日期对象
```
sdf.parse("2018-9-9");
```

* SimpleDateFormat格式说明

字母|日期或时间元素
:--|:--
y |年
M |年中的月份
d |月份中的天数
E |星期中的天数
a |Am/pm标记
```

```
H |一天中的小时数（0-23）
h |am/pm中的小时数（1-12）
m |小时中的分钟数
s |分钟中的秒数
S |毫秒数
```

```
public static void main(String[] args) throws ParseException{
//      当前系统时间
        Date d = new Date();
        System.out.println(d);
        Date d1 = new Date(3,3,2033);
        //当前时间是否在d1指定时间之后
        System.out.println(d.after(d1));
        System.out.println(d.before(d1));
        //大小
        System.out.println(d.compareTo(d1));

        System.out.println(d.getTime());
        System.out.println(d.getDate());

        d.setTime(01);
        //当前系统时间中国是UTC+8所开始的时间  1970.1.1 08：00：00
        //当前系统时间为UTC  世界标准时间  1970.1.1 08：00：00
        System.out.println(d);
        //设置的long值实际上是距离 1970.1.1 08：00：00  的毫秒数
        d.setTime(10001);
        System.out.println(d);

        /**
         * CHina
         * date转换为String类型
         */
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss:SS zz");//时间格式
        Date nowDate = new Date();//得到当前时间
        String date = sdf.format(nowDate );
```

```
        System.out.println(date);

        /**
         * String转换为date类型
         */
        SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM
-dd");//时间格式
        String str="2008-7-7";
        Date d2 = sdf1.parse(str);
        System.out.println(str);

    }
```

集合类
* 注：数组预习：
```
/**
  *数组拷贝
  */
public static void main(String[] args) {
    int[] a = {12,23,34,53,6};
    //所谓的数组就是新分配一块内存空间
    System.out.println("old  "+a.length);
    System.out.println(a);

    int[] b = {12,23,34,53,6};
    System.out.println(b.length);

    a = Arrays.copyOf(a, 10);
    System.out.println("new  "+a.length);
    System.out.println(a);
    /**
     * 数组中拷贝方法
     * a 源数组 原数组中的开始位置
```

```
     * b 目标数组   目标数组的开始位置
     * 拷贝长度
     */
    System.arraycopy(a, 0, b, 0, 5);

    for(int n:a){
        System.out.println(n);
    }
}
/**
  *使用方法TreeSet方法
  */
public static void main(String[] args) throws ParseException
{
/*  有2个多维数组分别是 2 3 4   和  1 5 2 8
                4 6 8       5 9 10 -3
                      2 7 -5 -18
按照如下方式进行运算。生成一个2行4列的数组。此数组的第1行1列是2*1+3*5+4*2第1行2列是2*5+3*9+4*7  第2行1列是4*1+6*5+8*2 依次类推。（知识点：多维数组定义和创建、数组遍历、数组元素访问） [选作题]
*/
//  将一个数组中的重复元素保留一个其他的清零 次方法只适用于String类型数组
    String[] a ={"1","2","2","3","3","3","6"};
    TreeSet ts = new TreeSet();
    for(String s:a) {
        ts.add(s);
    }
    String[] array = (String[])ts.toArray(new String[]{});
    System.out.println(Arrays.toString(array));

}
```

Collection接口

* 1、一组称为元素的对象
* 2、一个Collection中可以放不同类型的数据

```
* 3、是Set接口和List接口的父类
* 4、是否有特定的顺序以及是否允许重复，取决于它的实现
> Set — 无序的集合；不允许重复
>* HashSet
>* TreeSet

> List — 有序的集合；允许重复
>* ArrayList
>* LinkedList

#### Collection接口常用方法

方法|含义
:--|:--
boolean add(Object) |集合中加入一个对象，成功时返回true
boolean addAll(Collection) |集合中加入另外一个集合对象
int size() |集合内容纳的元素数量
boolean isEmpty() |集合是否为空
boolean contains(Object) |集合内是否含有参数对象
Iterator iterator() |产生一个迭代器
Object[] toArray() |返回一个包含所有元素的对象数组
boolean remove(Object) |从集合中删除对象
boolean removeAll(Collection) |清空指定集合
boolean containsAll(Collection) |判断集合内是否包含子集
boolean retainAll(Collection) |仅保留此 collection 中那些也包含在指定 collection 的元素
void clear() |清空集合
```

```
public static void main(String[] args) {
    //集合中可以自动扩容
    Collection c1 = new ArrayList();
    /**
     * Object是所有类的超类
     * 也是包装数据类型的超类
     * 因为 基本数据类型，性质：自动装箱：
     * 所以int 类型的10   转换为integer类型的10   int 10  ----->
```

```
integer  10
     * 因为包装数据类型的超类，所以integer 10  相当于变为Object类型的父类引用指向子类对象的引用  形式
     * 所以10  可以放到方法的参数当中
     */
    c1.add(10);
    c1.add(20);
    c1.add(30);
    c1.add(40);
    c1.add(50);
    c1.size();
    System.out.println(c1.size());
    System.out.println(c1.toArray());

    Object[] o1array = c1.toArray();
    /**
     * 遍历数组
     * 为甚麽可以遍历Object类型的数组？
     * 在syso当中输出一个类相当于自动调用toString方法
     * 因为有了，父类引用指向子类对象的形式，Object o = new Integer(10);
     * 所以  调用toString方法是发生重写，实际调用的是Integer当中的toString方法  即输出数字10
     */
    for(Object o:o1array){
        System.out.println(o);
    }
}
```

Collections类
* Collections类是类似于Arrays类的公用工具类 ,它提供了一些static方法供集合类使用或操作集合类 。
* Collections类中的方法

Collections类中的方法

方法含义
Object max(Collection c,Comparator comp) |max算法采用Comparator比较算法
Object max(collection c) |返回集合中的最大元素，需要考虑比较接口的实现
Object min(Collection c) |返回集合中的最小元素
void reverse(Collection c) |把集合中的元素顺序反转
void copy(List dest,List src) |src集合中元素复制到dest集合
void fill(List list,Object o) |填充list集合，填充元素为o
int binarySearch(List list,Object key) |对排序后的集合list进行查询元素操作
void sort(List list) |对一种List做排序

总结：collection与collections区别：
* collection是集合；
* collections是操作集合的工具类

Set接口
* 1、Collection的子接口
* 2、用来包含一组 无序无重复 的对象
> 无序 — 是指元素存入顺序和集合内存储的顺序不同；
> 无重复 — 两个对象e1和e2，如果e1.equals(e2)返回true，则认为e1和e2重复，在set中只保留一个。

* 3、Set接口的实现类
HashSet — HashSet的特性在于其内部对象的散列存取，即采用哈希技术
>* 无序 不可重复　　输出时数据无序，此类允许使用 null 元素。
>* 如果equles值相等　那么Hashcode值一定相等
>* 注：　　凡是Hash　都是无序的

* 此类实现 Set 接口，由哈希表（实际上是一个 HashMap 实例）支持。它不保证 set 的迭代顺序；特别是它不保证该顺序恒久不变。
* 此类为基本操作提供了稳定性能，这些基本操作包括 add、remove、contain

s 和 size，假定哈希函数将这些元素正确地分布在桶中。
* 底层 HashMap 实例的默认初始容量是 16，加载因子是 0.75。

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    /**
     * TreeSet
     *
     * HashSet    无序 不可重复    输出时数据无序   允许使用main值
     * 如果equles值相等   那么Hashcode值一定相等
     * 凡是Hash   都是无序的
     */
    Set s = new HashSet();
    s.add("aaa");
    s.add("bbb");
    s.add("ccc");
    s.add("ddd");
    Iterator i = s.iterator();
    while(i.hasNext()){
        System.out.println(i.next());
    }
}
```

* TreeSet — TreeSet存入的顺序跟存储的顺序不同，但是存储是按照排序存储的，
* 它是无序的。输出的时候，是已经排序好的。
* 不可重复
* 是根据二叉树的结构进行自然顺序排序，基于红黑树进行排序的

* 红黑树性质：
* 节点颜色是红、黑色
* 根节点是黑色的
* 每个叶节点（NIL节点，空节点）是黑色的
* 每个红色节点的两个子节点都是黑色的（从每个叶子到根的所有路径不能有两个连续的红色节点）

* 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。
```
public static void main(String[] args) {
    /**
     * 无序  输入顺序与输出顺序不同  所以无序  不可重复
     *  树状结构
     */
    TreeSet t = new TreeSet();
    t.add(2);
    t.add(22);
    t.add(23);
    t.add(28);
    t.add(92);
    Iterator i = t.iterator();
    while(i.hasNext()){
        System.out.println(i.next());
    }
}
```
* 4、使用foreach方式遍历Set集合

注：扩展：：：：：：
为甚麽可以遍历Object类型的数组？
* 在syso当中输出一个类相当于自动调用toString方法
* 因为有了，父类引用指向子类对象的形式，Object o = new Integer(10);
* 所以 调用toString方法是发生重写，实际调用的是Integer当中的toString方法 即输出数字10。

Object是所有类的超类
* 也是包装数据类型的超类
* 因为 基本数据类型，性质：自动装箱：
* 所以int 类型的10 转换为integer类型的10 int 10 -----> integer 10
* 因为包装数据类型的超类，所以integer 10 相当于变为Object类型 的父类引用指向子类对象的引用 形式
* 所以10 可以放到方法的参数当中

List接口
有序 可重复，存入的顺序与取出的顺序相同

* list 有序可重复 初始值10 每次扩容1.5倍 被称为是序列
* list接口是java.util.collection接口下的一个子接口，可以通过new实现类来使用list集合。

* 1、List接口
> Collection的子接口
> 用来包含一组 `有序有重复 `的对象
> List中的元素都对应一个整数型的序号，记载其在容器中的位置，可以根据序号存取容器中的元素
> List有两种主要的集合实现类：
>> ArrayList
>> LinkedList

* 2、两个实现类的区别：
> ArrayList
>> ArrayList是线性顺序存储的，是一种线性表
>> 它的特性和数组很接近，数组大小是不变的，而ArrayList的大小是可以动态改变的
> LinkedList
>> 是数据结构中链表的java实现
>> 相对于List来说，LinkedList最主要的功能方面的增强是可以在List的头部和尾部添加、删除、取得元素，直接提供了这些方法的实现。所以它可以非常方便的实现我们数据结构中的常见的Stack(栈)、queue(队列)等

List接口常用用法

方法含义
:--|:--
void add(int index,Object element) |在列表中的index位置，添加element元素
Object get(int index) |返回列表中指定位置的元素

```
int indexOf(Object o) |在list中查询元素的索引值，如不存在，返回－1。
int lastIndexOf(Object o) |List中如果存在多个重复元素，indexOf()方法返回第一个匹配元素的index。lastIndexOf(o)是返回最后一个匹配元素的index.
ListIterator listIterator() |返回列表中元素的列表迭代器
Object remove(int index) |移除列表中指定位置的元素
Object set(int index,Object element) |用指定元素替换列表中指定位置的元素
```

```
public static void main(String[] args) {
    /**
     * 数组要有统一的数据类型
     * List接口<>泛型， 存取顺序一致，------》有序，可重复
     *
     * 初始值10    每次扩容1.5倍    查找快 增删慢   数组结构
     */
    List list = new ArrayList();
    //创建一个int类型的数组
    //int[] a = new int[10];

    Integer[] a = new Integer[10];
    String[] b = new String[10];

    a[0] = 1;
    System.out.println(a[0]);
    b[0] = "aaaa";
    System.out.println(b[0]);

    //int ---->String
    b[0] = 1 + "";

    //Object是所有类的父类
    //显然 这样的数据是没有灵魂的   我要装相同数据类型的数据，这时候就要用到泛型
    Object[] c = new Object[10];
    c[0] = 111;
```

```
    c[1] = "aaa";
    System.out.println(c[0]+" "+c[1]);
    /*
    * 使用泛型 <>    里面定义数据类型
    * 使用集合 应学会如何去遍历数组
     */
    List<String> list1 = new ArrayList<>();
    list1.add("古天乐");
    list1.add("渣渣会");
    list1.add("爱迪生");
    //删除数据  数据位置
    list1.remove(1);
    //根据数组下标进行修改
    list1.set(1, "想改我渣渣辉！");
    //查询第一次出现指定元素的索引  如果不包含该元素  则返回-1
    System.out.println(list1.indexOf("HHH"));
    System.out.println("------------------------");

    //对数组进行遍历 数组遍历方法
    //1、List(arrayList)  把集合转换为数组 再遍历数组

    Object[] o = list1.toArray();
    for(Object n:o){
        System.out.println(n);
    }
    System.out.println("------------------------");
    //2、迭代器  iterator 使用泛型  保证数据为String类型数据
    ListIterator<String> i1 = list.listIterator();
//      Iterator<String> i1 = list1.iterator();
    while(i1.hasNext()){//判断 不移动指针

        String a1 = i1.next();//判断 移动指针
        if(a1 == "孙红雷"){
            i1.remove(); //使用迭代器来操作
            i1.add("黛埃迪");            //  c.Integer  没有add
方法，再操作list集合想使用add方法需要使用ListIterator
```

```
        }
        System.out.println(a1);
    }
    System.out.println(list1);
}
```

实现类的初始化

* ArrayList的构造方法

```
ArrayList 变量名 = new ArrayList();
ArrayList 变量名 = new ArrayList(int capacity);
ArrayList 变量名 = new ArrayList(Collection c);
```

* LinkedList类的构造方法

```
LinkedList  变量名 = new  LinkedList() ;
LinkedList 变量名 = new LinkedList(Collection c) ;
```

ArrayList和LinkedList

* LinkList增加方法

方法	含义
void addFirst(Object o)	将给定元素插入此列表的开头
void addLast(Object o)	将给定元素追加到此列表的结尾
Object getFirst()	返回此列表的第一个元素
Object getLast()	返回此列表的最后一个元素
Object removeFirst()	移除并返回此列表的第一个元素
ObjectremoveLast()	移除并返回此列表的最后一个元素

* List接口的实现类

> ArrayList与LinkedList的比较

>>* 1、存储结构

>> ArrayList是线性顺序存储 -------------- 数组结构

>> LinkedList对象间彼此串连起来的一个链表----------链表结构
>>* 2、操作性能
>> ArrayList适合随机查询的场合，查找快，增删慢
>> LinkedList元素的插入和删除操作性高，查找慢，增删块
>>* 3、从功能上，LinkedList要多一些

Vector类
使用elements进行遍历、Iterator也可以。
```
        Vector<String> v = new Vector<String>();
        v.addElement("aaa");
        v.addElement("bbb");
        v.addElement("vvv");
        //返回此向量的组件的枚举
        Enumeration<String> e =  v.elements();
        while(e.hasMoreElements()){//是否有更多的元素，指针不动
            String s = e.nextElement();//指针移动，跟ListIterat
or一样
            System.out.println(s);
        }
        Iterator<String> i = v.iterator();
        while(i.hasNext()) {
            String a = i.next();
            System.out.println(a);
        }
        Iterator<String> i2 = v.listIterator();
        while(i2.hasNext()) {
            String a2 = i2.next();
            System.out.println(a2);
        }

```

链表
* 每一个链表实际上就是由多个接待你组成的。开头节点为`root(根)`,结尾节点指向`null`

```
> data----->保存数据。
> next------>要保存的下一个节点。


### Map集合
* HashMap 的实例有两个参数影响其性能：初始容量 和加载因子。
>* 容量 是哈希表中桶的数量，初始容量只是哈希表在创建时的容量。
>* 加载因子 是哈希表在其容量自动增加之前可以达到多满的一种尺度。
>* 默认加载因子：0.75

* 当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 rehash 操作（即重建内部数据结构），从而哈希表将具有大约两倍的桶数。

* 基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。
* （除了非同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

* 此实现假定哈希函数将元素适当地分布在各桶之间，可为基本操作（get 和 put）提供稳定的性能。迭代 collection 视图所需的时间与 HashMap 实例的“容量”（桶的数量）及其大小（键-值映射关系数）成比例。所以，如果迭代性能很重要，则不要将初始容量设置得太高（或将加载因子设置得太低）。

#### 特点：
* 使用put属性添加元素
* 使用get属性获取元素
* key作为键，不可以重复的，若出现重复后者代替前者；比如说，谢霆锋女友张柏芝分手，王菲后来居上与峰哥领证，柏芝不哭(只支持一个人只能有一张结婚证)
* value作为值，是可以重复的

**Map两个实现类：**
* HashMap  可以存放空值
* TreeMap  不可以存放空值
* HashSet与TreeSet是一个道理。

**总结：**
```

* Map 主要用于存储键(key)值(value)对，根据键得到值，因此键不允许键重复,但允许值重复。
* Map是不允许key重复的，所以如果有key重复的话，那么前面的value会被后面的覆盖了 （后者会覆盖前者）

* **map映射表用来存放键值对的**
* 创建集合时，与前两个集合一样，父类引用执行子类对象。

```
public static void main(String[] args) {
        Map<String,String> map=new  HashMap<String,String>();
        //map<key,value>  通过key值找value值
        //一夫------>   一妻制
//          key值不能重复，如果重复看最后的-----》 领证时 ，后面的证有法律意义
        map.put("峰哥", "柏芝");
        map.put("峰哥", "菲姐");
        map.put("墨", "黑");
        map.put("宝宝","蓉蓉");
        map.put("喆喆","蓉蓉");
        // 一夫一妻前提下，   蓉蓉（妻子）可以搞破鞋
//      value值重复可以存在的--------------》蓉蓉（妻子）可以搞破鞋
//      System.out.println(map.get("朱"));
        System.out.println("宝宝："+map.get("宝宝"));
        System.out.println("喆喆："+map.get("喆喆"));
        System.out.println(map.get("峰哥"));
    }
```

**那么我们如何去遍历Map集合呢？
```
public static void main(String[] args) {
    HashMap h1=new HashMap();
    h1.put(1111, "11111111111");
    h1.put(2222, "11111111112");
    h1.put(3333, "11111111113");
```

```
    h1.put(4444, "11111111114");
    h1.put(5555, "11111111115");
    h1.put(6666, "11111111116");
//遍历map集合思路 有两大类
//  思路1 ：把key 装到set集合当中，再利用set几何的iterator 进行遍历
        Set<Integer> s1=h1.keySet();
        Iterator i1=s1.iterator();
        while(i1.hasNext()){
            int key=(int) i1.next();
            System.out.println(h1.get(key));
        }

//      思路2  把key，value 看成一个整体   Map.Entry
        Set<Map.Entry<Integer,String>> s2=   h1.entrySet();
        terator<Entry<Integer, String>> i2=s2.iterator();
        while(i2.hasNext()){
            Map.Entry<Integer,String> me=i2.next();
            System.out.println(me.getKey()+":"+me.getValue
());
        }

        //思路3 只拿value值
        Collection s3=   h1.values();
        terator<String>i3=  s3.iterator();
        while(i3.hasNext()){
                    System.out.println( i3.next());
        }

    }
```

总结：
通过使用set集合进行遍历，先转换为set集合，再对set集合进行遍历
* 1）k值当作set集合进行遍历
* 2）获得value当作set集合进行遍历
* 3）k,value当作set集合进行遍历，Map.Entry

Map.Entry<K,V>
public static interface Map.Entry<K,V>
映射项（键-值对）。Map.entrySet 方法返回映射的 collection 视图，其中的元素属于此类。获得映射项引用的唯一 方法是通过此 collection 视图的迭代器来实现。这些 Map.Entry 对象仅 在迭代期间有效；更确切地讲，如果在迭代器返回项之后修改了底层映射，则某些映射项的行为是不确定的，除了通过 setValue 在映射项上执行操作之外。

Iterator接口
* 1、Iterator对象称作迭代器，用来方便的实现对容器内的元素进行遍历操作
* 2、所有实现了Collection接口的集合类都有一个iterator( )方法，返回一个实现了Iterator接口的对象
* 3、Iterator对象实现了统一的一个用来遍历Collection中对象的方法
* 4、Iterator是为遍历而设计，能够从集合中取出元素和删除元素，但是没有添加元素的功能
* 5、Iterator的功能上比较简单，
* 6、使用中，只能单向移动
* 可以遍历所有的集合

Iterator接口方法

方法	含义
Object next()	返回游标右边的元素并将游标移动到下一个位置
boolean hasNext()	判断游标右边是否有元素
void remove()	删除游标左边的元素，在执行完next之后，该操作只能执行一次

ListIterator迭代器
* ListIterator迭代器只能遍历List或其子类的集合。
* 有添加删除的方法

* 有hastNext()方法和next() 方法，实现顺序向后遍历
* hasPrevious()和previous() 方法,可以实现逆序向前遍历。previous返回表中前一个元素
foreach和iterator区别：
* for循环一般用来处理比较简单的有序的，可预知大小的集合或数组
* foreach可用于遍历任何集合或数组，而且操作简单易懂，但是需要了解集合内部类型。
* iterator是最强大的，他可以随时修改或者删除集合内部的元素，并且是在不需要知道元素和集合的类型的情况下进行的，当你需要对不同的容器实现同样的遍历方式时，迭代器是最好的选择！

总结：迭代器有哪些特点：
* Java中的Iterator功能比较简单，并且只能单向移动：
* (1) 使用方法iterator()要求容器返回一个Iterator。第一次调用Iterator的next()方法时，它返回序列的第一个元素。
>* 注意：iterator()方法是java.lang.Iterable接口,被Collection继承。
* (2) 使用next()获得序列中的下一个元素。 Iterator中的next()方法采用的是顺序访问方法
* (3) 使用hasNext()检查序列中是否还有元素。
* (4) 使用remove()将迭代器新返回的元素删除。
>* 提供一种方法对一个容器对象中的各个元素进行访问，把访问逻辑从不同类型的集合类中抽取出来，从而避免向外部暴露集合的内部结构。

注：
* Java集合框架的集合类，我们有时候称之为容器。容器的种类有很多种，比如ArrayList、LinkedList、HashSet...，每种容器都有自己的特点，ArrayList底层维护的是一个数组；LinkedList是链表结构的；HashSet依赖的是哈希表，每种容器都有自己特有的数据结构。

文件管理
概述
* java中的对文件管理，通过java.io包中的`File`类实现

* java中文件的管理，主要是针对文件或是目录路径名的管理
> 文件的属性信息
> 文件的检查
> 文件的删除等
> 不包括文件的访问
File类
* File类的构造方法：
* File 变量名 = new File(String pathname) ;
> 通过将给定路径名字符串转换成抽象路径名来创建一个新 File 实例: File f1 = new File (“d:/temp/abc.txt”);

* File 变量名 = new File(URI uri) ;
> 通过将给定File的uri转换成抽象路径名来创建一个新 File 实例 : File f2 = new File(“abc.txt”);

* File 变量名 = new File(String parent,Stirng child) ;
> 根据 parent 路径名字符串和 child 路径名字符串创建一个新 File 实例 : File f3 = new File(“d:/temp”,“abc.txt”);

* File 变量名 = new File(File parent,Stirng child) ;
> 根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例 : File f = new File(“d:/temp”);File f4 = new File(f,“abc.txt”);

方法	含义
boolean createNewFile()	当且仅当不存在具有此抽象路径名指定的名 称的文件时，原子地创建由此抽象路径名指 定的一个新的空文件。
static File createTempFile(String prefix,String suffix)	在默认临时文件目录中创建一个空文件，使 用给定前缀和后缀生成其名称
static File createTempFile(String prefix,Stirng suffix,File directory)	在指定目录中创建一个新的空文件，使用给 定的前缀和后缀字符串生成其名称
boolean exists( )	测试此抽象路径名表示的文件或目录是否存在
boolean delete( )	删除此抽象路径名表示的文件或目录
boolean equals(Object obj)	测试此抽象路径名与给定对象是否相等
boolean canRead()	测试应用程序是否可以读取此抽象路径名表示 的文件

boolean canWrite() |测试应用程序是否可以修改此抽象路径名表示 的文件
String[] list( ) |返回由此抽象路径名所表示的目录中的文件和 目录的名称所组成字符串数
String getAbsolutePath( ) |返回抽象路径名的绝对路径名字符串
String getName( ) |返回由此抽象路径名表示的文件或目录的名 称，不包括路径名称
String getPath( ) |将此抽象路径名转换为一个路径名字符串
File[] listFiles() |返回一个抽象路径名数组，这些路径名表示 此抽象路径名所表示目录中的文件
boolean renameTo(File dest) |重新命名此抽象路径名表示的文件
long length( ) |返回由此抽象路径名表示的文件的大小，以 byte为单位
boolean mkdir( ) |创建此抽象路径名指定的目录
boolean mkdirs( ) |创建此抽象路径名指定的目录，包括创建必 需但不存在的父目录。注意，如果此操作失 败，可能已成功创建了一些必需的父目录

流的概念及API

1、流（Stream)的概念代表的是程序中数据的流通
* 数据流是一串连续不断的数据的集合
* 在Java程序中，对于数据的输入/输出操作是以流(Stream)的方式进行的
> 输入流 — 流入程序的数据
> 输出流 — 流出程序的数据
> 在java程序中，从输入流读取数据（读到内存中），而从输出 流输出数据（从内存存储到文件或显示到屏幕上）

2、流的分类
>* 按流的方向不同
>> 输入流、输出流
>* 按处理数据的单位不同
>> 字节流、字符流
>* 按功能不同
>> 节点流、处理流
* Java语言中，控制数据流的 类都放在java.io包中
字节流 字符流 输入流 InputStream Reader 输出流 OutputStream Writer 。java.io包中有两大继承体系：以byte处理为主的Stream类， 他们的命名方式是XXXStream：以字符处理为主的Reader / Writer类，他们的命名方式 X

XXReader或XXXWriter：InputStream、OutputStream、Reader、 Writer这四个类，是这两大继承体系的父类

 |字节流|字符流
:--|:--|:--
输入流|InputStream|Reader
输出流|OutputStream|Writer

3、字节输入流的主要方法
> 此抽象类是表示输入字节流的所有类的超类
> `InputStream`常用的方法
方法|含义
int read( ) |一次读取一个byte的数据，并以int类型把数据返 回来，如果没有数据可以读了，会返回”-1”
int read(byte[] buffer) |把所读取到的数据放在这个byte数组中，返回一 个int型的数据，这个int型数据存储了返回的真正 读取到的数据byte数
int read(byte[] buffer,int offset,int length) |读取length个字节，并存储到一个字节数组buffer 中，并从offset位置开始返回实际读取的字节数
void close( ) |关闭此输入流并释放与该流关联的所有系统资源

4、字节输出流的主要方法
> 此抽象类是表示输出字节流的所有类的超类
> `OutputStream`常用的方法

方法|含义
void write(byte[] buffer) |将要输出的数组先放在一个byte数组中，然后用 这个方法一次把一组数据输出出去
void write(byte[] buffer,int off,int len) |将指定字节数组中从偏移量 off 开始的 len 个字节 写入此输出流
abstract void write(int b) |将指定的字节写入此输出流
void close( ) |关闭此输出流并释放与此流有关的所有系统资源
void flush( ) |刷新此输出流并强制写出所有缓冲的输出字节

5、字符输入流的主要方法
> 用于输入字符流的抽象类

* `Reader`常用的方法 :
> Reader是输入字符数据用的类，它所提供的方法和InputStream 类一样，差别在于InputStream类中用的是byte类型，而Reader 类中用的是char类型。
> 注： Reader类中没有available方法，取而代之的是”ready” 方法，这个方法会去检查Reader对象是否已经准备好输 入数据了，如果是返回true，反之返回false。

方法|含义
：--|：--
int read( )| 一次读取一个char的数据，并以int类型把数据返 回来，如果没有数据可以读了，会返回”-1”
int read(char[] cbuffer) |把所读取到的数据放在这个char数组中，返回一 个int型的数据，这个int型数据存储了返回的真 正读取到的数据char数
int read(char[] cbuffer,int offset,int length) |读取length个字符，并存储到一个字节数组 cbuffer中，并从offset位置开始返回实际读取的字符数
void close( ) |关闭此Reader并释放与其关联的所有系统资源

6、字符输出流的主要方法
> 输出字符流的抽象类
* Writer常用的方法
> Writer类是输出字符数据的类，同样地，提供的方法和 OutputStream类中的方法类似，将OutputStream类中用到的 byte类型，换成char类型就可。
> 注： Writer类另外提供了两个writer方法，所以Writer类有5 个writer方法，多出来的两个只是把char数据换成String 对象而已，方便输出字符的数据

方法|含义
void write(char[] cbuffer) |将要输出的数组先放在一个char数组中，然后用这 个方法一次把一组数据输出出去
void write(char[] cbuffer,int off,int len) |将指定字符数组中从偏移量 off 开始的 len 个字符 写入此输出流
int write(int b) |将指定的字符写入此输出流 void write(String str) 写入字符串
void write(String str, int off,int len) |将指定字符串中从偏移量 off 开始的 len 个字符写 入此输出流
void close( ) |关闭此输出流并释放与此流有关的所有系统资源

void flush( ) |刷新此输出流并强制写出所有缓冲的输出字节

7、节点流与处理流的使用
什么是节点流
> 节点流：从一个特定的数据源（节点）读写数据（如：文件、内存） 的类叫做节点流类
> 这些节点类跟数据源或数据目的地做直接连接用的 。 在java.io包中，字节继承体系有三种节点类，而字符继承体系有四 种节点类

类型	字节流	字符流
File	FileInputStream、FileOutputStream	FileReader、FileWriter
Memory Array	ByteArrayInputStream ByteArrayOutputStream	CharArrayReader CharArrayWriter
Memory String	:	StringReader、StringWriter
Piped	PipedInputStream PipedOutputStream	PipedReader PipedWriter

8、节点流的方法
> 节点流的方法 — InputStream

方法	含义
int read( )	这个方法没有参数，一次读取 一个byte的数据，并以int类型把数据返回来，如果没有数据可 以读了，会返回”-1”。
int read(byte[] b)	这个方法有一个byte数据类型的 参数，这个方法会把所读取到 的数据放在这个byte数组中，返 回一个int型的数据，这个int型 数据存储了返回的真正读取到 的数据byte数。
int read(byte[] b,int off,int len)	将输入流中最多 len 个数据字节读入字节,返回值同上
void close( )	关闭此输入流并释放与该流关 联的所有系统资源。
int available( )	获取这个流中还有多少个byte 的数据可以读取。返回值告诉 我们还有多少个byte的数据可 以读取。 注：这个方法会产生 IOException 异常，另外如果 InputStream对象调用这个方法 的话，它只会返回0，这个方法必须由继承InputStream类的 子类对象调用才有作用。

long skip(long n) |跳过和放弃此输入流中的 n 个数据字节。返回值返回真 正跳过的字节数

9、节点流的方法 — OutputStream
* 注意：
> 使用write方法输出数据时，有些数据并不会马上输出到我们指 定的目的，通常会在内存中有个暂存区，有些输出的数据会暂 时存放在这里，如果我们想要立刻把数据输出到目的地，不要 放在暂存区中时，可以调用”flush”这个方法来对暂存区做清 除的动作。

> 同样，数据输出完后，记得把它”close”，在调用close这个方 法时，会先调用flush这个方法，以确保所有的数据都已经输出 到目的地了。
方法	含义
void write(byte[] b)	将要输出的数组先放在一个byte 数组中，然后用这个方法一次 把一组数据输出出去。
void write(byte[] b, int off, int len)	将指定字节数组中从偏移量 off 开始的 len 个字节写入此输出流。
void write(int b)	将要输出的byte数据传给这个方 法就可。
void close()	关闭此输出流并释放与此流有 关的所有系统资源
void flush()	刷新此输出流并强制写出所有 缓冲的输出字节

处理流
* 只用字节或是字符为单位来对数据做输入输出是不够的，有时候 我们需要一行一行的读数据，有时我们需要读取特定格式的数据， 因此Java提供了这样的机制，能把数据流作连接(chain)，让原本没 有特殊访问方法的流，通过连接到特殊的流后，变成可以用特定 的方法来访问数据
* “连接”在已存在的流（节点流或处理流）之上，通过对数据的 处理为程序提供更为强大的读写功能
* 处理流类的构造函数中，都必须接收另外一个流对象作为参数

常见的处理流类

种类\继承体系|字节 |字符
:--|:--
缓冲(Buffered) |BufferedInputStream, BuueredOutputStream | Buf

```
feredReader, BufferedWriter
字符和字节转换 | |InputStreamReader, OutputStreamWriter
对象序列化| ObjectInputStream, ObjectOutputStream |  
特定数据类型访问| DataInputStream,  DataOutputStream |  
计数| LineNumberInputStream |  
重复 |PushbackInputStream |  
打印 |PrintStream | PrintWriter

##### 缓冲流（Buffered）
* 缓冲流对读写的数据提供了缓冲的功能，提高了读写的效率，同 时增加了一些新的方

构造方法|含义
:--|:--
BufferedInputStream(InputStream in) |创建了一个带有32字节缓冲区的缓冲输 入流
BufferedInputStream(InputStream in,  int size) |创建了一个带有size大小缓冲区的缓冲 输入流
BufferedOutputStream(OutputStream out) |创建了一个带有32字节缓冲区的缓冲输 出流
BufferedOutputStream(OutputStream out，int size) |创建了一个带有size大小缓冲区的缓冲 输出流

* Java提供了四种缓冲流，其构造方法

构造方法|含义
:--|:--
BufferedReader(Reader in) |创建一个使用默认大小输入缓冲区的 缓冲字符输入流
BufferedReader(Reader in,int size) |创建一个使用size大小输入缓冲区的 缓冲字符输入流
BufferedWriter(Writer out) |创建一个使用默认大小输入缓冲区的 缓冲字符输出流
BufferedWriter(Writer out,int size) |创建一个使用size大小输入缓冲区的 缓冲字符输出流
```

* 缓冲流中的方法

> BufferedInputStream支持其父类的mark和reset方法

> BufferedWriter提供了readLine方法用于读取一行字符串(以\r或 \n分隔)

> BufferedWriter提供了newLine方法用于写入一个行分隔符
对于BufferedOutputStream和BufferdWriter，写出的数据会先在 内存中缓存，使用flush()方法将使内存中的数据立刻写出

对象序列化
对象序列化概述

* 通过使用ObjectInputStream和ObjectOutputStream类保存和读取对 象的机制叫做序列化机制
* 对象(Object)序列化是指将对象转换为字节序列的过程 ， 反序列化则是根据字节序列恢复对象的过程
* 序列化一般用于以下场景：
> 永久性保存对象，保存对象的字节序列到本地文件中
> 通过序列化对象在网络中传递对象
> 通过序列化在进程间传递对象

支持序列化的接口和类

* 序列化的过程，是将任何实现了Serializable接口或Externalizable接 口的对象通过ObjectOutputStream类提供的相应方法转换为连续的 字节数据，这些数据以后仍可通过ObjectInputStream类提供的相应 方法被还原为原来的对象状态，这样就可以将对象完成的保存在本 地文件中，或在网络和进程间传递
* 支持序列化的接口和类
> Serializable接口
>>* 只有一个实现Serializable接口的对象可以被序列化工具存储和恢复
>>* Serializable接口没有定义任何属性或方法。它只用来表示一个类可以被序列化。如果一个类可以序列化，它的所有子类都可 以序列化

> Externalizable接口
>>* 可以让需要序列化的类实现Serializable接口的子接口 Externalizable

>>* Externalizable接口表示实现该接口的类在序列化中由该类本身 来控制信息的写出和读入

> ObjectInputStream
>>* ObjectOutputStream类继承OutputStream类，并实现了 ObjectOutput接口。它负责向流写入对象
>>* 构造方法 ： ObjectOutputStream(OutputStream out
>>* 主要方法 ： writeObject(Object obj) 向指定的OutputStream中写入对象obj

> ObjectOutputStream
>>* ObjectInputStream类继承InputStream类，并实现了ObjectInput 接口。它负责从流中读取对象
>>* 构造方法 ： ObjectInputStream(InputStream in)
>>* 主要方法 ： readObject(Object obj) 从指定的InputStream中读取对象

对象序列化的条件

* 该对象类必须实现Serializable接口
* 如果该类有直接或者间接的不可序列化的基类，那么该基类必须 有一个默认的构造器。该派生类需要负责将其基类中的数据写入 流中。
* 建议所有可序列化类都显式声明 serialVersionUID 值。
> serialVersionUID在反序列化过程中用于验证序列化对象的发 送者和接收者是否为该对象加载了与序列化兼容的类。
> 如果接收者加载的该对象的类的 serialVersionUID 与对应的发 送者的类的版本号不同，则反序列化将会导致 InvalidClassException

transient关键字
* transient修饰的属性不进行序列化的操作，起到一定消息屏蔽的效果
* 被transient修饰的属性可以正确的创建，但被系统赋为默认值。 即int类型为0，String类型为null
> 注：ObjectInputStream和ObjectOutputStream类不会保存和读 写对象中的transient和static类型的成员变量

多线程
线程与进程概念、区别
* 概念：
> 进程 计算机在执行的程序 的实体
> 线程 一个程序内部 的顺序控制流
* 注： 一个进程中可以包含 一个或多个线程，一个 线程就是一个程序内部 的一条执行线

* 区别：
> 每个进程都有独立的代码和数据空间，进程的切换 会有很大的开销
> 同一类线程共享代码和数据空间，每个线程有独立 运行的栈和程序计数器，线程切换的开销

>* 多进程 ：在操作系统中能同时运行多个任务（程序）
>* 多线程 ：在同一应用程序中有多个顺序流同时进行
```
进程包含线程 有一个进程至少有一个线程。
```
多线程实现方式
* 创建线程的两种方式 — 线程类
>* 继承Thread类 —— java.lang.Thread
>* 实现Runnable接口 —— java.lang.Runnable

引用传递
引用传递的核心意义：同一块堆内存空间可以被不同的栈内存所指向，不同栈内存可以对统一堆内存的内容进行修改。想想栈堆内存分配图
范例一：
```
class Message{
    private int num = 10;
    public Message(int num){
        this.num = num;
    }
    public void setNum(int num){
```

```
        this.num = num;
    }
    public int getNum{
        return this.num;
    }
}
public static TestDemo{
    public static void main(String agrs[]){
        Message msg = new Message(30);
        fun(msg);//引用传递
        syso(msg.getNum());

    }
    public static void fun(Message temp){
        temp.setNum(100);
    }
}
输出：100
```

范例二：

```
public static TestDemo{
    public static void main(String agrs[]){
        String msg = "Hello";
        fun(msg);
        syso(msg.getNum());

    }
    public static void fun(String temp){
        temp = "World";
    }
}
输出：Hello
```

**解决思路：String类对象的内容一旦声明就不可改变，对象内容的改变依靠的是

引用地址的改变。**

范例三：
```
class Message{
    private String info = "nihao";
    public Message(String info){
        this.info = info;
    }
    public void setInfo(String info){
        this.info = info;
    }
    public String getInfo(){
        return this.info;
    }
}
public static TestDemo{
    public static void main(String agrs[]){
        Message msg = new Message("Hello");
        fun(msg);
        syso(msg.getInfo());
    }
    public static void fun(Message temp){
        temp.setInfo("World");
    }
}
```
输出：HWorld

此范例可以想象成为：
```
int x = 19;
iny y = x;
y = 12;
```
总结：虽然String属于类，属于引用类型，但是由于内容不可改变的特点，很多的时候，就把String当成基本数据类型使用，也就是说：每一个String变量只能保

存一个数据（改变了就会把之前的数据替换掉了。）

Java：遍历Map/HashMap的各种方法

在遍历Map集合之前首先先定义一个Map对象：
```
Map<String, String> map = new LinkedHashMap<String, String>
();
map.put("1", "one");
map.put("2", "two");
map.put("3", "three");
map.put("4", "fore");
map.put("5", "five");
```

这个地方使用的是LinkedHashMap，主要是为了确保让map中的元素是按照插入的顺序存放的。
1. 使用keySet()方法遍历
使用keyset方法遍历，是先取出map的key组成的Set集合，通过对Set集合的遍历，然后使用map.get(key)方法取出value值。
```
for (String key : map.keySet()) {
    System.out.println(key + " : " + map.get(key));
}
```

2. 使用map的values()方法遍历集合的values
```
map.values()返回的是由map的值组成的Collection，这个方法只能遍历map的所有value，不能得到map的key。
for (String value : map.values()) {
    System.out.println(value);
}
```

3. 使用map的entrySet()方法遍历
使用map的entrySet()方法返回一个以Entry为元素的Set集合，然后对Set集合进行遍历。
```
for (Entry<String, String> entry : map.entrySet()) {
System.out.println(entry.getKey()+ " : " +entry.getValue());
}
```

4. 通过keySet()返回的集合的iterator遍历
由于map.keySet()返回的是一个Set集合，所以通过它的iterator()方法返回一个迭代器，通过迭代器遍历map。
```
Iterator<String> it = map.keySet().iterator();
while(it.hasNext()) {
    String key = it.next();
    System.out.println(key + " : " + map.get(key));
}
```

5.通过values()返回的Collection的iterator遍历
map.values()方法返回的是一个Collection对象，这个集合对象可以使用iterator方法访问。
```
Iterator<String> it = map.values().iterator();
while(it.hasNext()) {
    String key = it.next();
    System.out.println(key + " : " + map.get(key));
}
```

6.通过entrySet()返回的Set的iterator遍历
同上，map.entrySet()方法返回的是一个Set<Entry<String, String»类型的集合，可以使用iterator来访问该集合。
```
```

```
Iterator<Entry<String, String>> it =    map.entrySet().iterat
or();
while(it3.hasNext()) {
    Entry<String, String> entry = it3.next();
System.out.println(entry.getKey() + " : " +entry.getValue());
}
```

以上总结了对map集合的集中遍历方式，根据自身需要灵活选择使用哪种方式。
实例：

```
public static void main(String[] args) {
  Map<String, String> map = new HashMap<String, String>();
map.put("1", "value1");
map.put("2", "value2");
map.put("3", "value3");
  //第一种：普遍使用，二次取值
  System.out.println("通过Map.keySet遍历key和value：");
  for (String key : map.keySet()) {
System.out.println("key= "+ key + " and value= " + map.get(ke
y));
  }
  //第二种
System.out.println("通过Map.entrySet使用iterator遍历key和valu
e：");
Iterator<Map.Entry<String, String>> it = map.entrySet().itera
tor();
  while (it.hasNext()) {
   Map.Entry<String, String> entry = it.next();
System.out.println("key="+entry.getKey() +
"and value= " +entry.getValue());
  }
  //第三种：推荐，尤其是容量大时
  System.out.println("通过Map.entrySet遍历key和value");
  for (Map.Entry<String, String> entry : map.entrySet()) {
   System.out.println("key= " + entry.getKey() +
```

```
" and value= " + entry.getValue());
  }
  //第四种
System.out.println("通过Map.values()遍历所有的value，但不能遍历key");
  for (String v : map.values()) {
   System.out.println("value= " + v);
  }
 }
```

总结

如果仅需要键(keys)或值(values)使用方法二。如果你使用的语言版本低于java 5，或是打算在遍历时删除entries，必须使用iterator遍历。