Java 基础知识总结

写代码：

1，明确需求。我要做什么？
2，分析思路。我要怎么做？1,2,3。
3，确定步骤。每一个思路部分用到哪些语句，方法，和对象。
4，代码实现。用具体的 java 语言代码把思路体现出来。

学习新技术的四点：

1，该技术是什么？
2，该技术有什么特点(使用注意)：
3，该技术怎么使用。demo
4，该技术什么时候用？test。

一：java 概述：

1991 年 Sun 公司的 James Gosling 等人开始开发名称为 Oak 的语言，希望用于控制嵌入在有线电视交换盒、PDA 等的微处理器；

1994 年将 Oak 语言更名为 Java；

Java 的三种技术架构：

JAVAEE：Java Platform Enterprise Edition，开发企业环境下的应用程序，主要针对 web 程序开发；

JAVASE：Java Platform Standard Edition，完成桌面应用程序的开发，是其它两者的基础；

JAVAME：Java Platform Micro Edition，开发电子消费产品和嵌入式设备，如手机中的程序；

1，JDK：Java Development Kit，java 的开发和运行环境，java 的开发工具和 jre。

2，JRE：Java Runtime Environment，java 程序的运行环境，java 运行的所需的类库+JVM(java 虚拟机)。

3，配置环境变量：让 java jdk\bin 目录下的工具，可以在任意目录下运行，原因是，将该工具所在目录告诉了系统，当使用该工具时，由系统帮我们去找指定的目录。

环境变量的配置：

1)：永久配置方式：JAVA_HOME=%安装路径%\Java\jdk
path=%JAVA_HOME%\bin

2)：临时配置方式：set path=%path%;C:\Program Files\Java\jdk\bin

特点：系统默认先去当前路径下找要执行的程序，如果没有，再去 path 中设置的路径下找。

classpath 的配置：

1)：永久配置方式：classpath=.;c:\;e:\

2)：临时配置方式：set classpath=.;c:\;e:\

注意：在定义 classpath 环境变量时，需要注意的情况

如果没有定义环境变量 classpath，java 启动 jvm 后，会在当前目录下查找要运行的类文件；

如果指定了 classpath，那么会在指定的目录下查找要运行的类文件。

还会在当前目录找吗？两种情况：

1)：如果 classpath 的值结尾处有分号，在具体路径中没有找到运行的类，会默认在当前目录再找一次。

2)：如果 classpath 的值结果出没有分号，在具体的路径中没有找到运行的类，不会再当前目

录找。

一般不指定分号，如果没有在指定目录下找到要运行的类文件，就报错，这样可以调试程序。

4，javac 命令和 java 命令做什么事情呢？

要知道 java 是分两部分的：一个是编译，一个是运行。

javac：负责的是编译的部分，当执行 javac 时，会启动 java 的编译器程序。对指定扩展名的.java 文件进行编译。 生成了 jvm 可以识别的字节码文件。也就是 class 文件，也就是 java 的运行程序。

java：负责运行的部分.会启动 jvm.加载运行时所需的类库,并对 class 文件进行执行.

一个文件要被执行,必须要有一个执行的起始点,这个起始点就是 main 函数.

5,虚拟机

当我在虚拟机中进行软件评测时，可能系统一样会崩溃，但是，崩溃的只是虚拟机上的操作系统，而不是物理计算机上的操作系统，并且，使用虚拟机的“Undo”（恢复）功能，我可以马上恢复虚拟机到安装软件之前的状态。

二：java 语法基础：

1，关键字：其实就是某种语言赋予了特殊含义的单词。

保留字：其实就是还没有赋予特殊含义，但是准备日后要使用过的单词。

2，标示符：其实就是在程序中自定义的名词。比如类名，变量名，函数名。包含 0-9、a-z、$、_ ；

注意：

1），数字不可以开头。

2），不可以使用关键字。

3，常量：是在程序中的不会变化的数据。

4，变量：其实就是内存中的一个存储空间，用于存储常量数据。

作用：方便于运算。因为有些数据不确定。所以确定该数据的名词和存储空间。

特点：变量空间可以重复使用。

什么时候定义变量？只要是数据不确定的时候，就定义变量。

变量空间的开辟需要什么要素呢？

1，这个空间要存储什么数据？数据类型。

2，这个空间叫什么名字啊？变量名称。

3，这个空间的第一次的数据是什么？ 变量的初始化值。

变量的作用域和生存期：

变量的作用域：

作用域从变量定义的位置开始，到该变量所在的那对大括号结束；

生命周期：

变量从定义的位置开始就在内存中活了；

变量到达它所在的作用域的时候就在内存中消失了；

数据类型：

1）：基本数据类型：byte、short、int、long、float、double、char、boolean

2）：引用数据类型：数组、类、接口。

级别从低到高为：byte,char,short(这三个平级)-->int-->float-->long-->double
自动类型转换：从低级别到高级别，系统自动转的；
强制类型转换：什么情况下使用?把一个高级别的数赋给一个别该数的级别低的变量；

运算符号：

1)、算术运算符。

+ - * / %　　**%:**任何整数模 2 不是 0 就是 1，所以只要改变被模数就可以实现开关运算。

+:连接符。

++,--

2)、赋值运算符。

= += -= *= /= %=

3)、比较运算符。

特点：该运算符的特点是：运算完的结果，要么是 true，要么是 false。

4)、逻辑运算符。

& | ^ ! && ||

逻辑运算符除了 ！ 外都是用于连接两个 boolean 类型表达式。

&: 只有两边都为 true 结果是 true。否则就是 false。

|:只要两边都为 false 结果是 false，否则就是 true

^:异或：和或有点不一样。

两边结果一样，就为 false。

两边结果不一样，就为 true.

& 和 &&区别： & ：无论左边结果是什么，右边都参与运算。

&&:短路与，如果左边为 false，那么右边不参数与运算。

| 和|| 区别：|：两边都运算。

||：短路或，如果左边为 true，那么右边不参与运算。

5)、位运算符:用于操作二进制位的运算符。

& | ^

<< >> >>>(无符号右移)

练习：对两个变量的数据进行互换。不需要第三方变量。

```
int a = 3,b = 5;-->b = 3,a = 5;
a = a + b; a = 8;
b = a - b; b = 3;
a = a - b; a = 5;
a = a ^ b;//
b = a ^ b;//b = a ^ b ^ b = a
a = a ^ b;//a = a ^ b ^ a = b;
```

练习：高效的算出 2*8 = 2<<3;

5，语句。

If switch do while while for

这些语句什么时候用?

1)、当判断固定个数的值的时候，可以使用 if，也可以使用 switch。

但是建议使用 switch，效率相对较高。

```
switch(变量){
  case 值:要执行的语句;break;
  …
  default:要执行的语句;
}
```

工作原理：用小括号中的变量的值依次和 case 后面的值进行对比，和哪个 case 后面的值相同了就执行哪个 case 后面的语句，如果没有相同的则执行 default 后面的语句；

细节：1)：break 是可以省略的，如果省略了就一直执行到遇到 break 为止；

2)：switch 后面的小括号中的变量应该是 byte,char,short,int 四种类型中的一种；

3)：default 可以写在 switch 结构中的任意位置；**如果将 default 语句放在了第一行，则不管 expression 与 case 中的 value 是否匹配，程序会从 default 开始执行直到第一个 break 出现。**

2)、当判断数据范围，获取判断运算结果 boolean 类型时，需要使用 if。

3)、当某些语句需要执行很多次时，就用循环结构。

while 和 for 可以进行互换。

区别在于：如果需要定义变量控制循环次数。建议使用 for。因为 for 循环完毕，变量在内存中释放。

break:作用于 switch ，和循环语句，用于跳出，或者称为结束。

break 语句单独存在时，下面不要定义其他语句，因为执行不到，编译会失败。当循环嵌套时，break 只跳出当前所在循环。要跳出嵌套中的外部循环，只要给循环起名字即可，这个名字称之为**标号**。

continue:只作用于循环结构，继续循环用的。

作用：结束本次循环，继续下次循环。该语句单独存在时，下面不可以定义语句，执行不到。

6，函 数：为了提高代码的复用性，可以将其定义成一个单独的功能，该功能的体现就是 java 中的函数。函数就是体现之一。

java 中的函数的定义格式：

```
修饰符 返回值类型 函数名(参数类型 形式参数 1，参数类型 形式参数 1，…){
    执行语句；
    return 返回值；
}
```

当函数没有具体的返回值时，返回的返回值类型用 void 关键字表示。

如果函数的返回值类型是 void 时，return 语句可以省略不写的，系统会帮你自动加上。

return 的作用：结束函数。结束功能。

如何定义一个函数？

函数其实就是一个功能，定义函数就是实现功能，通过两个明确来完成：

1)、明确该功能的运算完的结果，其实是在明确这个函数的返回值类型。

2)、在实现该功能的过程中是否有未知内容参与了运算，其实就是在明确这个函数的参数列表(参数类型&参数个数)。

函数的作用：

1)、用于定义功能。

2)、用于封装代码提高代码的复用性。

注意：函数中只能调用函数，不能定义函数。

主函数：

1)、保证该类的独立运行。

2)、因为它是程序的入口。

3)、因为它在被 jvm 调用。

函数定义名称是为什么呢？

答：1)、为了对该功能进行标示，方便于调用。

2)、为了通过名称就可以明确函数的功能，为了增加代码的阅读性。

重载的定义是：在一个类中，如果出现了两个或者两个以上的同名函数，只要它们的参数的个数，或者参数的类型不同，即可称之为该函数重载了。
如何区分重载：当函数同名时，只看参数列表。和返回值类型没关系。

7，数 组：用于存储同一类型数据的一个容器。**好处：**可以对该容器中的数据进行编号，从 0 开始。数组用于封装数据，就是一个具体的实体。

如何在 java 中表现一个数组呢？两种表现形式。

1)、元素类型[] 变量名 = new 元素类型[元素的个数];

2)、元素类型[] 变量名 = {元素 1，元素 2...};

元素类型[] 变量名 = new 元素类型[]{元素 1，元素 2...};

//二分查找法。必须有前提：数组中的元素要有序。

```
public static int halfSeach_2(int[] arr,int key){
    int min,max,mid;
    min = 0;
    max = arr.length-1;
    mid = (max+min)>>1; //(max+min)/2;
    while(arr[mid]!=key){
        if(key>arr[mid]){
            min = mid + 1;
        }
        else if(key<arr[mid])
            max = mid - 1;
        if(max<min)
            return -1;
        mid = (max+min)>>1;
    }
    return mid;
}
```

---

java 分了 5 片内存。

1：寄存器。2：本地方法区。3：方法区。4：栈。5：堆。

栈：存储的都是局部变量 （ 函数中定义的变量，函数上的参数，语句中的变量 ）；

只要数据运算完成所在的区域结束，该数据就会被释放。

堆：用于存储数组和对象，也就是**实体**。啥是实体啊？就是用于封装多个数据的。

1：每一个实体都有内存首地址值。

2：堆内存中的变量都有默认初始化值。因为数据类型不同，值也不一样。

3：垃圾回收机制。

三：面向对象：★★★★★

特点：**1：将复杂的事情简单化。**

2：面向对象将以前的过程中的执行者，变成了指挥者。

3：面向对象这种思想是符合现在人们思考习惯的一种思想。

过程和对象在我们的程序中是如何体现的呢？**过程**其实就是函数;**对象**是将函数等一些内容进行了封装。

匿名对象使用场景：
1：当对方法只进行一次调用的时候，可以使用匿名对象。
2：当对象对成员进行多次调用时，不能使用匿名对象。必须给对象起名字。

在类中定义其实都称之为成员。成员有两种：
1：**成员变量**：其实对应的就是事物的属性。
2：**成员函数**：其实对应的就是事物的行为。

所以，其实定义类，就是在定义成员变量和成员函数。但是在定义前，必须先要对事物进行属性和行为的分析，才可以用代码来体现。

```
private int age;//私有的访问权限最低，只有在本类中的访问有效。
```
注意：私有仅仅是封装的一种体现形式而已。

私有的成员：其他类不能直接创建对象访问，所以只有通过本类对外提供具体的访问方式来完成对私有的访问，可以通过对外提供函数的形式对其进行访问。
好处：可以在函数中加入逻辑判断等操作，对数据进行判断等操作。

总结：开发时，记住，属性是用于存储数据的，直接被访问，容易出现安全隐患，所以，类中的属性通常被私有化，并对外提供公共的访问方法。
这个方法一般有两个，规范写法：对于属性 xxx，可以使用 setXXX(),getXXX()对其进行操作。

类中怎么没有定义主函数呢？
注意：主函数的存在，仅为该类是否需要独立运行，如果不需要，主函数是不用定义的。
主函数的解释：保证所在类的独立运行，是程序的入口，被 jvm 调用。

成员变量和局部变量的区别：
1：**成员变量直接定义在类中。**
局部变量定义在方法中，参数上，语句中。
2：**成员变量在这个类中有效。**
局部变量只在自己所属的大括号内有效，大括号结束，局部变量失去作用域。
3：**成员变量存在于堆内存中，随着对象的产生而存在，消失而消失。**
局部变量存在于栈内存中，随着所属区域的运行而存在，结束而释放。

构造函数：用于给对象进行初始化，是给与之对应的对象进行初始化，它具有针对性，函数中的一种。
特点：
1：该函数的名称和所在类的名称相同。
2：不需要定义返回值类型。
3：该函数没有具体的返回值。
记住：所有对象创建时，都需要初始化才可以使用。

注意事项：一个类在定义时，如果没有定义过构造函数，那么该类中会自动生成一个空参数的构造函数，为了方便该类创建对象，完成初始化。如果在类中自定义了构造函数，那么默认的构造函

数就没有了。

一个类中，可以有多个构造函数，因为它们的函数名称都相同，所以只能通过参数列表来区分。所以，**一个类中如果出现多个构造函数。它们的存在是以重载体现的。**

构造函数和一般函数有什么区别呢？

1：两个函数定义格式不同。

2：构造函数是在对象创建时，就被调用，用于初始化，**而且初始化动作只执行一次。**

一般函数，是对象创建后，需要调用才执行，可以被调用多次。

什么时候使用构造函数呢？

分析事物时，发现具体事物一出现，就具备了一些特征，那就将这些特征定义到构造函数内。

构造代码块和构造函数有什么区别？

构造代码块：是给所有的对象进行初始化，也就是说，所有的对象都会调用一个代码块。只要对象一建立。就会调用这个代码块。

构造函数：是给与之对应的对象进行初始化。它具有针对性。

```
Person p = new Person();
```

创建一个对象都在内存中做了什么事情？

1：先将硬盘上指定位置的 Person.class 文件加载进内存。

2：执行 main 方法时，在栈内存中开辟了 main 方法的空间(压栈-进栈)，然后在 main 方法的栈区分配了一个变量 p。

3：在堆内存中开辟一个实体空间，分配了一个内存首地址值。new

4：在该实体空间中进行属性的空间分配，并进行了默认初始化。

5：对空间中的属性进行显示初始化。

6：进行实体的构造代码块初始化。

7：调用该实体对应的构造函数，进行构造函数初始化。()

8：将首地址赋值给 p ，p 变量就引用了该实体。(指向了该对象)

封 装（面向对象特征之一）：是指隐藏对象的属性和实现细节，仅对外提供公共访问方式。

好处：将变化隔离；便于使用；提高重用性；安全性。

封装原则：将不需要对外提供的内容都隐藏起来，把属性都隐藏，提供公共方法对其访问。

this:**代表对象。就是所在函数所属对象的引用。**

this 到底代表什么呢？**哪个对象调用了 this 所在的函数，this 就代表哪个对象，就是哪个对象的引用。**

开发时，什么时候使用 this 呢？

在定义功能时，如果该功能内部使用到了调用该功能的对象，这时就用 this 来表示这个对象。

this 还可以用于构造函数间的调用。

调用格式：this(实际参数)；

this 对象后面跟上 . 调用的是成员属性和成员方法(一般方法)；

this 对象后面跟上 () 调用的是本类中的对应参数的构造函数。

注意：用 this 调用构造函数，必须定义在构造函数的第一行。因为构造函数是用于初始化的，所

以初始化动作一定要执行。否则编译失败。

static：★★★ **关键字，是一个修饰符，用于修饰成员(成员变量和成员函数)。**
特点：
1，**想要实现对象中的共性数据的对象共享。可以将这个数据进行静态修饰。**
2，**被静态修饰的成员，可以直接被类名所调用。**也就是说，静态的成员多了一种调用方式。**类名.静态方式**。
3，**静态随着类的加载而加载。而且优先于对象存在。**

弊端：
1，有些数据是对象特有的数据，是不可以被静态修饰的。因为那样的话，特有数据会变成对象的共享数据。这样对事物的描述就出了问题。所以，在定义静态时，必须要明确，这个数据是否是被对象所共享的。
2，**静态方法只能访问静态成员，不可以访问非静态成员。**
(这句话是针对同一个类环境下的，比如说，一个类有多个成员（属性，方法，字段），静态方法A，那么可以访问同类名下其他静态成员，你如果访问非静态成员就不行)

因为静态方法加载时，优先于对象存在，所以没有办法访问对象中的成员。
3，静态方法中不能使用 this，super 关键字。
因为 this 代表对象，而静态在时，有可能没有对象，所以 this 无法使用。
4，主函数是静态的。

什么时候定义静态成员呢？或者说：定义成员时，到底需不需要被静态修饰呢？
成员分两种：
1，**成员变量。**（**数据共享时静态化**）
该成员变量的数据是否是所有对象都一样：
如果是，那么该变量需要被静态修饰，因为是共享的数据。
如果不是，那么就说这是对象的特有数据，要存储到对象中。
2，**成员函数。**（方法中没有调用特有数据时就定义成静态）
如果判断成员函数是否需要被静态修饰呢？
只要参考，该函数内是否访问了对象中的特有数据：
如果有访问特有数据，那方法不能被静态修饰。
如果没有访问过特有数据，那么这个方法需要被静态修饰。

成员变量和静态变量的区别：
1，成员变量所属于对象。所以也称为实例变量。
静态变量所属于类。所以也称为类变量。
2，成员变量存在于堆内存中。
静态变量存在于方法区中。
3，成员变量随着对象创建而存在。随着对象被回收而消失。
静态变量随着类的加载而存在。随着类的消失而消失。
4，成员变量只能被对象所调用 。
静态变量可以被对象调用，也可以被类名调用。
所以，成员变量可以称为对象的特有数据，静态变量称为对象的共享数据。

静态的注意：静态的生命周期很长。
静态代码块：就是一个有静态关键字标示的一个代码块区域。定义在类中。

作用：可以完成类的初始化。静态代码块随着类的加载而执行，而且只执行一次（new 多个对象就只执行一次）。如果和主函数在同一类中，优先于主函数执行。

Public：访问权限最大。
static：不需要对象，直接类名即可。
void：主函数没有返回值。
Main：主函数特定的名称。
(String[] args)：主函数的参数，是一个字符串数组类型的参数，jvm 调用 main 方法时，传递的实际参数是 new String[0]。

jvm 默认传递的是长度为 0 的字符串数组，我们在运行该类时，也可以指定具体的参数进行传递。可以在控制台，运行该类时，在后面加入参数。参数之间通过空格隔开。jvm 会自动将这些字符串参数作为 args 数组中的元素，进行存储。

静态代码块、构造代码块、构造函数同时存在时的执行顺序：静态代码块 → 构造代码块 → 构造方法；

生成 Java 帮助文档：命令格式：javadoc -d 文件夹名 -auther -version *.java

```
/**      //格式
*类描述
*@author 作者名
*@version 版本号
*/
/**
*方法描述
*@param  参数描述
*@return  返回值描述
*/
```

设计模式：解决问题最行之有效的思想。是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。

java 中有 23 种设计模式：
单例设计模式：★★★★★
解决的问题：**保证一个类在内存中的对象唯一性。**
比如：多程序读取一个配置文件时，建议配置文件封装成对象。会方便操作其中数据，又要保证多个程序读到的是同一个配置文件对象，就需要该配置文件对象在内存中是唯一的。

Runtime()方法就是单例设计模式进行设计的。

如何保证对象唯一性呢？
思想：
1，不让其他程序创建该类对象。
2，在本类中创建一个本类对象。
3，对外提供方法，让其他程序获取这个对象。

步骤：

1，因为创建对象都需要构造函数初始化，只要将本类中的构造函数私有化，其他程序就无法再创建该类对象；

2，就在类中创建一个本类的对象；

3，定义一个方法，返回该对象，让其他程序可以通过方法就得到本类对象。（作用：可控）

代码体现：

1，私有化构造函数；

2，创建私有并静态的本类对象；

3，定义公有并静态的方法，返回该对象。

```
//饿汉式
class Single{
    private Single(){} //私有化构造函数。
     private static Single s = new Single(); //创建私有并静态的本类对象。
    public static Single getInstance(){ //定义公有并静态的方法，返回该对象。
        return s;
    }
}
```

```
//懒汉式:延迟加载方式。
class Single2{
    private Single2(){}
     private static Single2 s = null;
    public static Single2 getInstance(){
        if(s==null)
            s = new Single2();
        return s;
    }
}
```

继 承（面向对象特征之一）

好处：

1：提高了代码的复用性。

2：让类与类之间产生了关系，提供了另一个特征多态的前提。

父类的由来：其实是由多个类不断向上抽取共性内容而来的。

java 中对于继承，java 只支持单继承。java 虽然不直接支持多继承，但是保留了这种多继承机制，进行改良。

单继承：一个类只能有一个父类。

多继承：一个类可以有多个父类。

为什么不支持多继承呢？

因为当一个类同时继承两个父类时，两个父类中有相同的功能，那么子类对象调用该功能时，运行哪一个呢？因为父类中的方法中存在方法体。

但是 java 支持多重继承。A 继承 B　B 继承 C　C 继承 D。

多重继承的出现，就有了继承体系。体系中的顶层父类是通过不断向上抽取而来的。它里面定义的该体系最基本最共性内容的功能。

所以，一个体系要想被使用，直接查阅该系统中的父类的功能即可知道该体系的基本用法。那么想要使用一个体系时，需要建立对象。建议建立最子类对象，因为最子类不仅可以使用父类中的功能。还可以使用子类特有的一些功能。

简单说：对于一个继承体系的使用，查阅顶层父类中的内容，创建最底层子类的对象。

子父类出现后，类中的成员都有了哪些特点：

1：成员变量。

当子父类中出现一样的属性时，子类类型的对象，调用该属性，值是子类的属性值。

如果想要调用父类中的属性值，需要使用一个关键字：super

This：**代表是本类类型的对象引用。**

Super：**代表是子类所属的父类中的内存空间引用。**

注意：子父类中通常是不会出现同名成员变量的，因为父类中只要定义了，子类就不用在定义了，直接继承过来用就可以了。

2：成员函数。

当子父类中出现了一模一样的方法时，建立子类对象会运行子类中的方法。好像父类中的方法被覆盖掉一样。所以这种情况，是函数的另一个特性：覆盖(复写，重写)

什么时候使用覆盖呢？当一个类的功能内容需要修改时，可以通过覆盖来实现。

3：构造函数。

发现子类构造函数运行时，先运行了父类的构造函数。为什么呢?

原因：子类的**所有构造函数中的第一行**，其实都有一条隐身的语句 super();

super()：表示父类的构造函数，并会调用于参数相对应的父类中的构造函数。而 super():是在调用父类中空参数的构造函数。

为什么子类对象初始化时，都需要调用父类中的函数？(为什么要在子类构造函数的第一行加入这个 super()?)

因为子类继承父类，会继承到父类中的数据，所以必须要看父类是如何对自己的数据进行初始化的。所以子类在进行对象初始化时，先调用父类的构造函数，这就是**子类的实例化过程。**

注意：子类中所有的构造函数都会默认访问父类中的空参数的构造函数，因为每一个子类构造内第一行都有默认的语句 super();

如果父类中没有空参数的构造函数，那么子类的构造函数内，必须通过 super 语句指定要访问的父类中的构造函数。

如果子类构造函数中用 this 来指定调用子类自己的构造函数，那么被调用的构造函数也一样会访问父类中的构造函数。

问题：super()和 this()是否可以同时出现的构造函数中。

两个语句只能有一个定义在第一行，所以只能出现其中一个。

super()或者 this():为什么一定要定义在第一行？

因为 super()或者 this()都是调用构造函数，构造函数用于初始化，所以初始化的动作要先完成。

继承的细节：

什么时候使用继承呢？

当类与类之间存在着所属关系时，才具备了继承的前提。a 是 b 中的一种。a 继承 b。狼是犬科中的一种。
英文书中，所属关系：″ is a ″
注意：不要仅仅为了获取其他类中的已有成员进行继承。

所以判断所属关系，可以简单看，如果继承后，被继承的类中的功能，都可以被该子类所具备，那么继承成立。如果不是，不可以继承。

细节二：
在方法覆盖时，注意两点：
1：子类覆盖父类时，必须要保证，子类方法的权限必须大于等于父类方法权限可以实现继承。否则，编译失败。
2：覆盖时，要么都静态，要么都不静态。（静态只能覆盖静态，或者被静态覆盖）

继承的一个弊端：打破了封装性。对于一些类，或者类中功能，是需要被继承，或者复写的。
这时如何解决问题呢？介绍一个关键字，final:最终。

final 特点：
1：这个关键字是一个修饰符，可以修饰类，方法，变量。
2：被 final 修饰的类是一个最终类，不可以被继承。
3：被 final 修饰的方法是一个最终方法，不可以被覆盖。
4：被 final 修饰的变量是一个常量，只能赋值一次。

其实这样的原因的就是给一些固定的数据起个阅读性较强的名称。
不加 final 修饰不是也可以使用吗？那么这个值是一个变量，是可以更改的。加了 final，程序更为严谨。常量名称定义时，有规范，所有字母都大写，如果由多个单词组成，中间用 _ 连接。

抽象类：abstract
抽象：不具体，看不明白。抽象类表象体现。
在不断抽取过程中，将共性内容中的方法声明抽取，但是方法不一样，没有抽取，这时抽取到的方法，并不具体，需要被指定关键字 abstract 所标示，声明为抽象方法。
抽象方法所在类一定要标示为抽象类，也就是说该类需要被 abstract 关键字所修饰。

抽象类的特点：
1：抽象方法只能定义在抽象类中，抽象类和抽象方法必须由 abstract 关键字修饰（可以描述类和方法，不可以描述变量）。
2：抽象方法只定义方法声明，并不定义方法实现。
3：抽象类不可以被创建对象(实例化)。
4：只有通过子类继承抽象类并覆盖了抽象类中的所有抽象方法后，该子类才可以实例化。否则，该子类还是一个抽象类。

抽象类的细节：
1：抽象类中是否有构造函数？有，用于给子类对象进行初始化。
2：抽象类中是否可以定义非抽象方法？
可以。其实，抽象类和一般类没有太大的区别，都是在描述事物，只不过抽象类在描述事物时，有些功能不具体。所以抽象类和一般类在定义上，都是需要定义属性和行为的。只不过，比一般类多了一个抽象函数。而且比一般类少了一个创建对象的部分。

3：抽象关键字 abstract 和哪些不可以共存？final ， private ， static

4：抽象类中可不可以不定义抽象方法？可以。抽象方法目的仅仅为了不让该类创建对象。

模板方法设计模式：

解决的问题：当功能内部一部分实现时确定，一部分实现是不确定的。这时可以把不确定的部分暴露出去，让子类去实现。

```
abstract class GetTime{
    public final void getTime(){ //此功能如果不需要复写，可加 final 限定
        long start = System.currentTimeMillis();
        code(); //不确定的功能部分，提取出来，通过抽象方法实现
        long end = System.currentTimeMillis();
        System.out.println("毫秒是："+(end-start));
    }
    public abstract void code(); //抽象不确定的功能，让子类复写实现
}
class SubDemo extends GetTime{
    public void code(){ //子类复写功能方法
        for(int y=0; y<1000; y++){
            System.out.println("y");
        }
    }
}
```

接 口：★★★★★

1：是用关键字 interface 定义的。

2：接口中包含的成员，最常见的有全局常量、抽象方法。

注意：接口中的成员都有固定的修饰符。

成员变量：public static final

成员方法：public abstract

```
interface Inter{
    public static final int x = 3;
    public abstract void show();
}
```

3：接口中有抽象方法，说明**接口不可以实例化。接口的子类必须实现了接口中所有的抽象方法后，该子类才可以实例化。否则，该子类还是一个抽象类。**

4：类与类之间存在着继承关系，类与接口中间存在的是实现关系。

继承用 extends ；实现用 implements ；

5：接口和类不一样的地方，就是，**接口可以被多实现**，这就是多继承改良后的结果。java 将多继承机制通过多现实来体现。

6：一个类在继承另一个类的同时，还可以实现多个接口。所以**接口的出现避免了单继承的局限性**。还可以将类进行功能的扩展。

7：其实 java 中是有多继承的。接口与接口之间存在着继承关系，**接口可以多继承接口**。

接口都用于设计上，设计上的特点：（可以理解主板上提供的接口）

1：接口是对外提供的规则。

2：接口是功能的扩展。
3：接口的出现降低了耦合性。

抽象类与接口：
抽象类：一般用于描述一个体系单元，将一组共性内容进行抽取，特点：可以在类中定义抽象内容让子类实现，可以定义非抽象内容让子类直接使用。它里面定义的都是一些体系中的基本内容。
接口：一般用于定义对象的扩展功能，是在继承之外还需这个对象具备的一些功能。

抽象类和接口的共性：都是不断向上抽取的结果。

抽象类和接口的区别：
1：抽象类只能被继承，而且只能单继承。
　　接口需要被实现，而且可以多实现。
2：抽象类中可以定义非抽象方法，子类可以直接继承使用。
　　接口中都是抽象方法，需要子类去实现。
3：抽象类使用的是　is a 关系。
　　接口使用的 like a 关系。
4：抽象类的成员修饰符可以自定义。
　　接口中的成员修饰符是固定的。全都是 public 的。

在开发之前，先定义规则，A 和 B 分别开发，A 负责实现这个规则，B 负责使用这个规则。至于 A 是如何对规则具体实现的，B 是不需要知道的。这样这个接口的出现就降低了 A 和 B 直接耦合性。

多　态★★★★★（面向对象特征之一）：函数本身就具备多态性，某一种事物有不同的具体的体现。

体现：父类引用或者接口的引用指向了自己的子类对象。//Animal a = new Cat();父类可以调用子类中覆写过的（父类中有的方法）
多态的好处：提高了程序的扩展性。继承的父类或接口一般是类库中的东西，（如果要修改某个方法的具体实现方式）只有通过子类去覆写要改变的某一个方法，这样在通过将父类的应用指向子类的实例去调用覆写过的方法就行了！
多态的弊端：当父类引用指向子类对象时，虽然提高了扩展性，但是只能访问父类中具备的方法，不可以访问子类中特有的方法。(前期不能使用后期产生的功能，即访问的局限性)
多态的前提：
1：必须要有关系，比如继承、或者实现。
2：通常会有覆盖操作。

多态的出现思想上也做着变化：以前是创建对象并指挥对象做事情。有了多态以后，我们可以找到对象的共性类型，直接操作共性类型做事情即可，这样可以指挥一批对象做事情，即通过操作父类或接口实现。

```
class 毕姥爷{
    void 讲课(){
        System.out.println("企业管理");
```

```
    }
    void 钓鱼(){
        System.out.println("钓鱼");
    }
}
class 毕老师 extends 毕姥爷{
    void 讲课(){
        System.out.println("JAVA");
    }
    void 看电影(){
        System.out.println("看电影");
    }
}
class {
    public static void main(String[] args) {
        毕姥爷 x = new 毕老师(); //毕老师对象被提升为了毕姥爷类型。
//      x.讲课();
//      x.看电影();  //错误.
        毕老师 y = (毕老师)x; //将毕姥爷类型强制转换成毕老师类型。
        y.看电影();//在多态中，自始自终都是子类对象在做着类型的变化。
    }
}
```

如果想用子类对象的特有方法，如何判断对象是哪个具体的子类类型呢？

可以可以通过一个关键字 instanceof ;//判断对象是否实现了指定的接口或继承了指定的类

格式：<对象 instanceof 类型> ，判断一个对象是否所属于指定的类型。

Student instanceof Person = true;//student 继承了 person 类

多态在子父类中的成员上的体现的特点：

1，成员变量：在多态中，子父类成员变量同名。

在编译时期：参考的是**引用型变量所属的类**中是否有调用的成员。（**编译时不产生对象，只检查语法错误**）

运行时期：也是参考**引用型变量所属的类**中是否有调用的成员。

简单一句话：无论编译和运行，成员变量参考的都是引用变量所属的类中的成员变量。

再说的更容易记忆一些：**成员变量 --- 编译运行都看 = 左边。**

2，成员函数。

编译时期：参考引用型变量所属的类中是否有调用的方法。

运行事情：参考的是对象所属的类中是否有调用的方法。

为什么是这样的呢？**因为在子父类中，对于一模一样的成员函数，有一个特性：覆盖。**

简单一句：成员函数，编译看引用型变量所属的类，运行看对象所属的类。

更简单：成员函数 --- 编译看 = 左边，运行看 = 右边。

3，静态函数。

编译时期：参考的是引用型变量所属的类中是否有调用的成员。

运行时期：也是参考引用型变量所属的类中是否有调用的成员。

为什么是这样的呢？**因为静态方法，其实不所属于对象，而是所属于该方法所在的类。**

调用静态的方法引用是哪个类的引用调用的就是哪个类中的静态方法。

简单说：静态函数 --- 编译运行都看 = 左边。

------java.lang.Object

Object：所有类的直接或者间接父类，Java 认为所有的对象都具备一些基本的共性内容，这些内容可以不断的向上抽取，最终就抽取到了一个最顶层的类中的，该类中定义的就是所有对象都具备的功能。

具体方法：

1，boolean equals(Object obj)：用于比较两个对象是否相等，**其实内部比较的就是两个对象地址。如果根据 equals(Object) 方法，两个对象是相等的，那么对这两个对象中的每个对象调用 hashCode 方法都必须生成相同的整数结果；**

而根据对象的属性不同，判断对象是否相同的具体内容也不一样。所以在定义类时，一般都会复写 equals 方法，建立本类特有的判断对象是否相同的依据。

```
public boolean equals(Object obj){
    if(!(obj instanceof Person))
        return false;
    Person p = (Person)obj;
    return this.age == p.age;
}
```

2，String toString()：将对象变成字符串；默认返回的格式：类名@哈希值 = getClass().getName() + '@' + Integer.toHexString(hashCode())

为了对象对应的字符串内容有意义，可以通过复写，建立该类对象自己特有的字符串表现形式。

```
public String toString(){
    return "person : "+age;
}
```

3，Class getClass()：获取任意对象运行时的所属字节码文件对象。

4，int hashCode()：返回该对象的哈希码值。支持此方法是为了提高哈希表的性能。将该对象的内部地址转换成一个整数来实现的。

通常 equals，toString，hashCode，在应用中都会被复写，建立具体对象的特有的内容。

内部类：如果 A 类需要直接访问 B 类中的成员，而 B 类又需要建立 A 类的对象。这时，为了方便设计和访问，直接将 A 类定义在 B 类中。就可以了。A 类就称为内部类。内部类可以直接访问外部类中的成员。而外部类想要访问内部类，必须要建立内部类的对象。

```
class Outer{
    int num = 4;
    class  Inner {
        void show(){
            System.out.println("inner show run "+num);
        }
    }
    public void method(){
        Inner in = new Inner();//创建内部类的对象。
```

```
        in.show();//调用内部类的方法。  //内部类直接访问外部类成员，用自己的实例对象；
    }                                    //外部类访问内部类要定义内部类的对象；
}
```

当内部类定义在外部类中的成员位置上，可以使用一些成员修饰符修饰 private、static。

1：默认修饰符。

直接访问内部类格式：外部类名.内部类名 变量名 = 外部类对象.内部类对象;

Outer.Inner in = new Outer.new Inner();//这种形式很少用。

但是这种应用不多见，因为内部类之所以定义在内部就是为了封装。**想要获取内部类对象通常都通过外部类的方法来获取。这样可以对内部类对象进行控制。**

2：私有修饰符。

通常内部类被封装，都会被私有化，因为封装性不让其他程序直接访问。

3：静态修饰符。

如果内部类被静态修饰，相当于外部类，会出现访问局限性，只能访问外部类中的静态成员。

注意；**如果内部类中定义了静态成员，那么该内部类必须是静态的。**

内部类编译后的文件名为：“外部类名$内部类名.java”；

为什么内部类可以直接访问外部类中的成员呢？

那是因为内部中都持有一个外部类的引用。这个是引用是 外部类名.this

内部类可以定义在外部类中的成员位置上，也可以定义在外部类中的局部位置上。

当内部类被定义在局部位置上，只能访问局部中被 final 修饰的局部变量。

匿名内部类（对象）：没有名字的内部类。就是内部类的简化形式。一般只用一次就可以用这种形式。匿名内部类其实就是一个**匿名子类对象**。想要定义匿名内部类：需要前提，内部类必须继承一个类或者实现接口。

匿名内部类的格式：new 父类名&接口名(){ 定义子类成员或者覆盖父类方法 }.方法。

匿名内部类的使用场景：

当函数的参数是接口类型引用时，如果接口中的方法不超过 3 个。可以通过匿名内部类来完成参数的传递。

其实就是在创建匿名内部类时，该类中的封装的方法不要过多，最好两个或者两个以内。

```
//面试
        //1
        new Object(){
            void show(){
                System.out.println("show run");
            }
        }.show();                                   //写法和编译都没问题
        //2
        Object obj = new Object(){
            void show(){
                System.out.println("show run");
            }
        };
```

```
        obj.show();                                    //写法正确，编译会报错
```

1 和 2 的写法正确吗？有区别吗？说出原因。

写法是正确，1 和 2 都是在通过匿名内部类建立一个 Object 类的子类对象。

区别：

第一个可是编译通过，并运行。

第二个编译失败，因为匿名内部类是一个子类对象，当用 Object 的 obj 引用指向时，就被提升为了 Object 类型，而编译时检查 Object 类中是否有 show 方法，所以编译失败。

```
class InnerClassDemo6 {
    +（static）class Inner{
        void show(){}
    }
    public void method(){
        this.new Inner().show();//可以
    }
    public static void main(String[] args) {//static 不允许 this
        This.new Inner().show();//错误，Inner 类需要定义成 static
    }
}
```

```
interface Inter{
    void show();
}
class Outer{//通过匿名内部类补足 Outer 类中的代码。
    public static Inter method(){          //返回 Inter 类型的变量；
        return new Inter(){
            public void show(){}
        };
    }
}
class InnerClassDemo7 {
    public static void main(String[] args) {
        Outer.method().show();
    /*
        Outer.method():意思是：Outer 中有一个名称为 method 的方法，而且这个方法是静态的。
        Outer.method().show():当 Outer 类调用静态的 method 方法运算结束后的结果又调用了 show 方法，意味着：method()方法运算完一个是对象，而且这个对象是 Inter 类型的。
    */
        function (new Inter(){
            public void show(){}
        }); //匿名内部类作为方法的参数进行传递。
    }
    public static void function(Inter in){
        in.show();
    }
}
```

异 常：★★★★

异常：就是不正常。程序在运行时出现的不正常情况。其实就是程序中出现的问题。这个问题按照面向对象思想进行描述，并封装成了对象。因为问题的产生有产生的原因、有问题的名称、有问题的描述等多个属性信息存在。当出现多属性信息最方便的方式就是将这些信息进行封装。异常就是 java 按照面向对象的思想将问题进行对象封装。这样就方便于操作问题以及处理问题。

出现的问题有很多种，比如角标越界，空指针等都是。就对这些问题进行分类。而且这些问题都有共性内容比如：每一个问题都有名称，同时还有问题描述的信息，问题出现的位置，所以可以不断的向上抽取。形成了**异常体系**。

--------java.lang.Throwable：

Throwable：可抛出的。

|--Error：错误，一般情况下，不编写针对性的代码进行处理，通常是 jvm 发生的，需要对程序进行修正。

|--Exception：异常，可以有针对性的处理方式

无论是错误还是异常，它们都有具体的子类体现每一个问题，它们的子类都有一个共性，就是都**以父类名才作为子类的后缀名**。

这个体系中的所有类和对象都具备一个独有的特点；就是可抛性。

可抛性的体现：就是这个体系中的类和对象都可以被 throws 和 throw 两个关键字所操作。

```
class  ExceptionDemo{
    public static void main(String[] args) {
//      byte[] buf = new byte[1024*1024*700];//java.lang.OutOfMemoryError 内存溢出错误
    }
}
```

在开发时，如果定义功能时，发现该功能会出现一些问题，**应该将问题在定义功能时标示出来**，这样调用者就可以在使用这个功能的时候，预先给出处理方式。

如何标示呢？通过 throws 关键字完成，**格式：throws 异常类名,异常类名...**

这样标示后，调用者，在使用该功能时，就必须要处理，否则编译失败。

处理方式有两种：1、捕捉；2、抛出。

对于捕捉：java 有针对性的语句块进行处理。

```
try {
    需要被检测的代码;
}
catch(异常类 变量名){
    异常处理代码;
}
fianlly{
    一定会执行的代码;
}
```

```
-------------------------------------------------------------
catch (Exception e) { //e 用于接收 try 检测到的异常对象。
    System.out.println("message:"+e.getMessage());//获取的是异常的信息。
    System.out.println("toString:"+e.toString());//获取的是异常的名字+异常的信息。
    e.printStackTrace();//打印异常在堆栈中信息； 异常名称+异常信息+异常的位置。
}
-------------------------------------------------------------
```

异常处理原则：功能抛出几个异常，功能调用如果进行 try 处理，需要与之对应的 catch 处理代码块，这样的处理有针对性，抛几个就处理几个。

特殊情况：try 对应多个 catch 时，如果有父类的 catch 语句块，一定要放在下面。

throw 和 throws 关键字的区别：

throw 用于抛出异常对象，后面跟的是异常对象；throw 用在函数内。

throws 用于抛出异常类，后面跟的异常类名，可以跟多个，用逗号隔开。throws 用在函数上。

通常情况：函数内容如果有 throw，抛出异常对象，并没有进行处理，那么函数上一定要声明，否则编译失败。但是也有特殊情况。

异常分两种：

1：编译时被检查的异常，只要是 Exception 及其子类都是编译时被检测的异常。

2：运行时异常，其中 Exception 有一个特殊的子类 RuntimeException，以及 RuntimeException 的子类是运行异常，也就说这个异常是编译时不被检查的异常。

编译时被检查的异常和运行时异常的区别：

编译被检查的异常在函数内被抛出，函数必须要声明，否编译失败。

声明的原因：是需要调用者对该异常进行处理。

运行时异常如果在函数内被抛出，在函数上不需要声明。

不声明的原因：不需要调用者处理，运行时异常发生，已经无法再让程序继续运行，所以，不让调用处理的，直接让程序停止，由调用者对代码进行修正。

定义异常处理时，什么时候定义 try，什么时候定义 throws 呢？

功能内部如果出现异常，如果内部可以处理，就用 try；

如果功能内部处理不了，就必须声明出来，让调用者处理。使用 throws 抛出，交给调用者处理。谁调用了这个功能谁就是调用者；

自定义异常：当开发时，项目中出现了 java 中没有定义过的问题时，这时就需要我们按照 java 异常建立思想，将项目的中的特有问题也进行对象的封装。这个异常，称为自定义异常。

对于除法运算，0 作为除数是不可以的。java 中对这种问题用 ArithmeticException 类进行描述。对于这个功能，在我们项目中，除数除了不可以为 0 外，还不可以为负数。可是负数的部分 java 并没有针对描述。所以我们就需要自定义这个异常。

自定义异常的步骤：

1：定义一个子类继承 Exception 或 RuntimeException，让该类具备可抛性(既可以使用 throw 和 throws 去调用此类)。

2：通过 throw 或者 throws 进行操作。

异常的转换思想：当出现的异常是调用者处理不了的，就需要将此异常转换为一个调用者可以处理的异常抛出。

try catch finally 的几种结合方式：

1，	2，	3，
try	try	try
catch	catch	finally
finally		

这种情况，如果出现异常，并不处理，但是资源一定关闭，所以 try finally 集合只为关闭资源。
记住：finally 很有用，主要用户关闭资源。无论是否发生异常，资源都必须进行关闭。
System.exit(0)； //退出 jvm，只有这种情况 finally 不执行。

当异常出现后，在子父类进行覆盖时，有了一些新的特点：

1：当子类覆盖父类的方法时，如果父类的方法抛出了异常，那么子类的方法要么不抛出异常要么抛出父类异常或者该异常的子类，不能抛出其他异常。

2：如果父类抛出了多个异常，那么子类在覆盖时只能抛出父类的异常的子集。

注意：

如果父类或者接口中的方法没有抛出过异常，那么子类是不可以抛出异常的，如果子类的覆盖的方法中出现了异常，只能 try 不能 throws。

如果这个异常子类无法处理，已经影响了子类方法的具体运算，这时可以在子类方法中，通过 throw 抛出 RuntimeException 异常或者其子类，这样，子类的方法上是不需要 throws 声明的。

常见异常：

1、脚标越界异常（IndexOutOfBoundsException）包括数组、字符串；
空指针异常（NullPointerException）

2、类型转换异常：ClassCastException

3、没有这个元素异常：NullPointerException

4、不支持操作异常；

异常要尽量避免，如果避免不了，需要预先给出处理方式。比如家庭备药，比如灭火器。

包：定义包用 package 关键字。

1：对类文件进行分类管理。

2：给类文件提供多层名称空间。

如果生成的包不在当前目录下，需要最好执行 classpath，将包所在父目录定义到 classpath 变量中即可。

一般在定义包名时，因为包的出现是为了区分重名的类。所以包名要尽量唯一。怎么保证唯一性呢？可以使用 **url 域名**来进行包名称的定义。

package pack;//定义了一个包，名称为 pack。 **注意：包名的写法规范：所有字母都小写。**

//package cn.itcast.pack.demo;

类的全名称是 **包名.类名**

编译命令：javac -d 位置（.当前路径） java 源文件 **（就可以自动生成包）**

包是一种封装形式，用于封装类，想要被包以外的程序访问，该类必须 public；
类中的成员，如果被包以外访问，也必须 public；

包与包之间访问可以使用的权限有两种：

1：public
2：protected：只能是不同包中的子类可以使用的权限。

总结 java 中的四种权限：

范围	public	protected	default	private
同一个类中	ok	ok	ok	ok
同一包中	ok	ok	ok	
子类	ok			
不同包中	ok			

Import - 导入：类名称变长，写起来很麻烦。为了简化，使用了一个关键字：import，可以使用这个关键字导入指定包中的类。记住：实际开发时,到的哪个类就导入哪个类，不建议使用*.

```
import packa.*;//这个仅仅是导入了 packa 当前目录下的所有的类。不包含子包。
import packa.abc.*;//导入了 packa 包中的子包 abc 下的当前的所有类。
```

如果导入的两个包中存在着相同名称的类。这时如果用到该类，必须在代码中指定包名。

常见的软件包:

java.lang ：language java 的核心包，Object System String Throwable jdk1.2 版本后，该包中的类自动被导入。
java.awt ： 定义的都是用于 java 图形界面开发的对象。
javax.swing：提供所有的 windows 桌面应用程序包括的控件,比如:Frame , Dialog, Table, List 等等,就是 java 的图形界面库。
java.net ： 用于 java 网络编程方面的对象都在该包中。
java.io ：input output 用于操作设备上数据的对象都在该包中。比如：读取硬盘数据，往硬盘写入数据。
java.util ： java 的工具包，时间对象，集合框架。
java.applet：application+let 客户端 java 小程序。server+let --> servlet 服务端 java 小程序。

jar ：java 的压缩包，主要用于存储类文件，或者配置文件等。

命令格式：jar -cf 包名.jar 包目录
解压缩：jar -xvf 包名.jar
将 jar 包目录列表重定向到一个文件中：jar -tf 包名.jar >c:\1.txt

多线程：★★★★

进程：正在进行中的程序。其实进程就是一个应用程序运行时的内存分配空间。
线程：其实就是进程中一个程序执行控制单元，一条执行路径。进程负责的是应用程序的空间的标示。线程负责的是应用程序的执行顺序。

一个进程至少有一个线程在运行，当一个进程中出现多个线程时，就称这个应用程序是多线程应用程序，每个线程在栈区中都有自己的执行空间，自己的方法区、自己的变量。
jvm 在启动的时，首先有一个**主线程**，负责程序的执行，调用的是 main 函数。主线程执行的代码都在 main 方法中。
当产生垃圾时，收垃圾的动作，是不需要主线程来完成，因为这样，会出现主线程中的代码执行会停止，会去运行垃圾回收器代码，效率较低，所以由单独一个线程来负责垃圾回收。

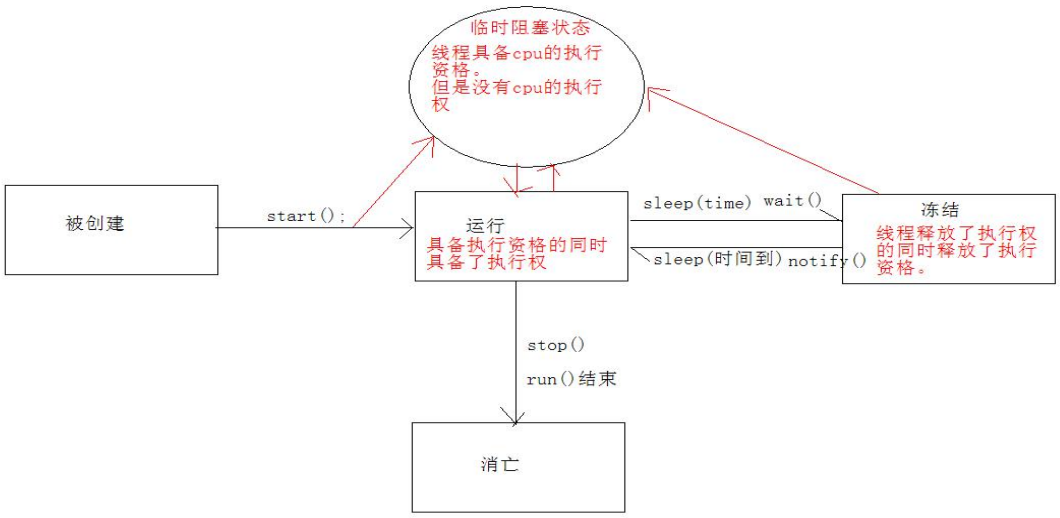
随机性的原理：因为 cpu 的快速切换造成，哪个线程获取到了 cpu 的执行权，哪个线程就执行。

返回当前线程的名称：Thread.currentThread().getName()
线程的名称是由：Thread-编号定义的。编号从 0 开始。
线程要运行的代码都统一存放在了 run 方法中。

线程要运行必须要通过类中指定的方法开启。start 方法。（启动后，就多了一条执行路径）
start 方法：1)、启动了线程；2)、让 jvm 调用了 run 方法。

创建线程的第一种方式：继承 Thread ，由子类复写 run 方法。
步骤：
1，定义类继承 Thread 类；
2，目的是复写 run 方法，将要让线程运行的代码都存储到 run 方法中；
3，通过创建 Thread 类的子类对象，创建线程对象；
4，调用线程的 start 方法，开启线程，并执行 run 方法。

线程状态：
被创建：start()
运行：具备执行资格，同时具备执行权；
冻结：sleep(time),wait()—notify()唤醒；线程释放了执行权，同时释放执行资格；
临时阻塞状态：线程具备 cpu 的执行资格，没有 cpu 的执行权；
消亡：stop()



创建线程的第二种方式：实现一个接口 Runnable。
步骤：
1，定义类实现 Runnable 接口。

2，覆盖接口中的 run 方法（用于封装线程要运行的代码）。
3，通过 Thread 类创建线程对象；
4，**将实现了 Runnable 接口的子类对象作为实际参数传递给 Thread 类中的构造函数。**
为什么要传递呢？因为要让线程对象明确要运行的 run 方法所属的对象。
5，调用 Thread 对象的 start 方法。开启线程，并运行 Runnable 接口子类中的 run 方法。

```
Ticket t = new Ticket();
/*
直接创建 Ticket 对象，并不是创建线程对象。
因为创建对象只能通过 new Thread 类，或者 new Thread 类的子类才可以。
所以最终想要创建线程。既然没有了 Thread 类的子类，就只能用 Thread 类。
*/
Thread t1 = new Thread(t); //创建线程。
/*
只要将 t 作为 Thread 类的构造函数的实际参数传入即可完成线程对象和 t 之间的关联
为什么要将 t 传给 Thread 类的构造函数呢？其实就是为了明确线程要运行的代码 run 方法。
*/
t1.start();
```

为什么要有 Runnable 接口的出现？

1：通过继承 Thread 类的方式，可以完成多线程的建立。但是这种方式有一个局限性，如果一个类已经有了自己的父类，就不可以继承 Thread 类，因为 java 单继承的局限性。

可是该类中的还有部分代码需要被多个线程同时执行。这时怎么办呢？

只有对该类进行额外的功能扩展，java 就提供了一个接口 Runnable。这个接口中定义了 run 方法，其实 run 方法的定义就是为了存储多线程要运行的代码。

所以，通常创建线程都用第二种方式。

因为实现 Runnable 接口可以避免单继承的局限性。

2：其实是将不同类中需要被多线程执行的代码进行抽取。将多线程要运行的代码的位置单独定义到接口中。为其他类进行功能扩展提供了前提。

所以 Thread 类在描述线程时，内部定义的 run 方法，也来自于 Runnable 接口。

实现 Runnable 接口可以避免单继承的局限性。而且，继承 Thread，是可以对 Thread 类中的方法，进行子类复写的。但是不需要做这个复写动作的话，只为定义线程代码存放位置，实现 Runnable 接口更方便一些。所以 **Runnable 接口将线程要执行的任务封装成了对象。**

```
//面试
new Thread(new Runnable(){  //匿名
    public void run(){
        System.out.println("runnable run");
    }
})
{
    public void run(){
        System.out.println("subthread run");
    }
}.start();  //结果：subthread run
```

```
Try {
    Thread.sleep(10);
}catch(InterruptedException e){}// 当刻意让线程稍微停一下，模拟 cpu    切换情况。
```

多线程安全问题的原因：

通过图解：发现一个线程在执行多条语句时，并运算同一个数据时，在执行过程中，其他线程参与进来，并操作了这个数据。导致到了错误数据的产生。

涉及到两个因素：

1，**多个线程在操作共享数据。**

2，**有多条语句对共享数据进行运算。**

原因：这多条语句，在某一个时刻被一个线程执行时，还没有执行完，就被其他线程执行了。

解决安全问题的原理：

只要将操作共享数据的语句在某一时段让一个线程执行完，在执行过程中，其他线程不能进来执行就可以解决这个问题。

如何进行多句操作共享数据代码的封装呢？

java **中提供了一个解决方式：就是同步代码块。**

格式：

```
synchronized(对象) {  // 任意对象都可以。这个对象就是锁。
    需要被同步的代码；
}
```

同步：★★★★★ **//就是在操作共享数据代码时，访问时只能让一个线程进去访问，此线程执行完退出后，别的线程才能再对此共享数据代码进行访问。**

好处：解决了线程安全问题。Synchronized

弊端：相对降低性能，因为判断锁需要消耗资源，产生了死锁。

定义同步是有前提的：

1，必须要有两个或者两个以上的线程，才需要同步。

2，多个线程必须保证使用的是同一个锁。

同步的第二种表现形式： **//对共享资源的方法定义同步**

同步函数：其实就是将同步关键字定义在函数上，让函数具备了同步性。

同步函数是用的哪个锁呢？ **//synchronized(this)用以定义需要进行同步的某一部分代码块**

通过验证，函数都有自己所属的对象 this，所以同步函数所使用的锁就是 this 锁。This.方法名

当同步函数被 static 修饰时，这时的同步用的是哪个锁呢？

静态函数在加载时所属于类，这时有可能还没有该类产生的对象，但是该类的字节码文件加载进内存就已经被封装成了对象，这个对象就是**该类的字节码文件对象**。

所以静态加载时，只有一个对象存在，那么静态同步函数就使用的这个对象。

这个对象就是 类名.class

同步代码块和同步函数的区别？

同步代码块使用的锁可以是任意对象。

同步函数使用的锁是 this，静态同步函数的锁是该类的字节码文件对象。

在一个类中只有一个同步的话，可以使用同步函数。如果有多同步，必须使用同步代码块，来确定不同的锁。所以同步代码块相对灵活一些。

★考点问题：请写一个延迟加载的单例模式？写懒汉式；当出现多线程访问时怎么解决？加同步，解决安全问题；效率高吗？不高；怎样解决？通过双重判断的形式解决。

//懒汉式：延迟加载方式。

当多线程访问懒汉式时，因为懒汉式的方法内对共性数据进行多条语句的操作。所以容易出现线程安全问题。为了解决，加入同步机制，解决安全问题。但是却带来了效率降低。

为了效率问题，通过双重判断的形式解决。

```
class Single{
    private static Single s = null;
    private Single(){}
    public static Single getInstance(){ //锁是谁？字节码文件对象；
        if(s == null){
            synchronized(Single.class){
                if(s == null)
                    s = new Single();
            }
        }
        return s;
    }
}
```

同步死锁：通常只要将同步进行嵌套，就可以看到现象。同步函数中有同步代码块，同步代码块中还有同步函数。

线程间通信：思路：多个线程在操作同一个资源，但是操作的动作却不一样。

1：将资源封装成对象。

2：将线程执行的任务(任务其实就是 run 方法。)也封装成对象。

等待唤醒机制：涉及的方法：

wait:将同步中的线程处于冻结状态。释放了执行权，释放了资格。同时将线程对象存储到线程池中。

notify：唤醒线程池中某一个等待线程。

notifyAll:唤醒的是线程池中的所有线程。

注意：

1：**这些方法都需要定义在同步中。**

2：因为这些方法必须要标示所属的锁。

你要知道 A 锁上的线程被 wait 了，那这个线程就相当于处于 A 锁的线程池中，只能 A 锁的 notify 唤醒。

3：这三个方法都定义在 Object 类中。为什么操作线程的方法定义在 Object 类中？

因为这三个方法都需要定义同步内，并标示所属的同步锁，既然被锁调用，而锁又可以是任意对象，那么能被任意对象调用的方法一定定义在 Object 类中。

wait 和 sleep 区别： 分析这两个方法：从执行权和锁上来分析：

wait：可以指定时间也可以不指定时间。不指定时间，只能由对应的 notify 或者 notifyAll 来唤醒。

sleep：必须指定时间，时间到自动从冻结状态转成运行状态(临时阻塞状态)。

wait：线程会释放执行权，而且线程会释放锁。

Sleep：线程会释放执行权，但不是不释放锁。

线程的停止：通过 stop 方法就可以停止线程。但是这个方式过时了。

停止线程：原理就是：让线程运行的代码结束，也就是结束 run 方法。

怎么结束 run 方法？一般 run 方法里肯定定义循环。所以只要结束循环即可。

第一种方式：**定义循环的结束标记。**

第二种方式：如果线程处于了冻结状态，是不可能读到标记的，这时就需要通过 Thread 类中的 interrupt 方法，将其冻结状态强制清除。让线程恢复具备执行资格的状态，让线程可以读到标记，并结束。

---------< java.lang.Thread >----------

interrupt()：中断线程。

setPriority(int newPriority)：更改线程的优先级。

getPriority()：返回线程的优先级。

toString()：返回该线程的字符串表示形式，包括线程名称、优先级和线程组。

Thread.yield()：暂停当前正在执行的线程对象，并执行其他线程。

setDaemon(true)：将该线程标记为守护线程或用户线程。将该线程标记为守护线程或用户线程。当正在运行的线程都是守护线程时，Java 虚拟机退出。该方法必须在启动线程前调用。

join：临时加入一个线程的时候可以使用 join 方法。

当 A 线程执行到了 B 线程的 join 方式。A 线程处于冻结状态，释放了执行权，B 开始执行。A 什么时候执行呢？只有当 B 线程运行结束后，A 才从冻结状态恢复运行状态执行。

LOCK 的出现替代了同步：lock.lock();……lock.unlock();

Lock 接口：**多线程在 JDK1.5 版本升级时，推出一个接口 Lock 接口。**

解决线程安全问题使用同步的形式，(同步代码块，要么同步函数)其实最终使用的都是锁机制。

到了后期版本，直接将锁封装成了对象。线程进入同步就是具备了锁，执行完，离开同步，就是释放了锁。

在后期对锁的分析过程中，发现，获取锁，或者释放锁的动作应该是锁这个事物更清楚。所以将这些动作定义在了锁当中，并把锁定义成对象。

所以**同步是隐示的锁操作，而 Lock 对象是显示的锁操作**，它的出现就替代了同步。

在之前的版本中使用 Object 类中 wait、notify、notifyAll 的方式来完成的。那是因为同步中的锁是任意对象，所以操作锁的等待唤醒的方法都定义在 Object 类中。

而现在锁是指定对象 Lock。所以查找等待唤醒机制方式需要通过 Lock 接口来完成。而 Lock 接口中并没有直接操作等待唤醒的方法，而是将这些方式又单独封装到了一个对象中。这个对象就是 Condition，将 Object 中的三个方法进行单独的封装。并提供了功能一致的方法 await()、

signal()、signalAll()体现新版本对象的好处。

< java.util.concurrent.locks > Condition 接口：await()、signal()、signalAll()；

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();
   final Object[] items = new Object[100];
   int putptr, takeptr, count;
   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     }
    finally {
       lock.unlock();
     }
   }
   public Object take() throws InterruptedException {
     lock.lock();
     try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       if (++takeptr == items.length) takeptr = 0;
       --count;
       notFull.signal();
       return x;
     }
     finally {
       lock.unlock();
     }
   }
 }
```

API：（Application Programming Interface,应用程序编程接口）是一些预先定义的函数，目的是提供应用程序与开发人员基于某软件或硬件的以访问一组例程的能力，而又无需访问源码，或理解内部工作机制的细节。

--< java.lang >-- String 字符串：★★★☆

java 中用 String 类进行描述。对字符串进行了对象的封装。这样的好处是可以对字符串这种常见数据进行方便的操作。对象封装后，可以定义 N 多属性和行为。

如何定义字符串对象呢？String s = "abc";只要是双引号引起的数据都是字符串对象。

特点：字符串一旦被初始化，就不可以被改变，存放在方法区中的常量池中。

```
String s1 = "abc"; // s1 指向的内存中只有一个对象 abc。
String s2 = new String("abc"); // s2 指向的内容中有两个对象 abc、new 。

System.out.println(s1==s2);//false
System.out.println(s1.equals(s2));//true ,字符串中 equals 比较的是字符串内容是否相同。
```

字符串的方法：

1：构造方法：将字节数组或者字符数组转成字符串。

```
String s1 = new String();//创建了一个空内容的字符串。
String s2 = null;//s2 没有任何对象指向，是一个 null 常量值。
String s3 = "";//s3 指向一个具体的字符串对象，只不过这个字符串中没有内容。
//一般在定义字符串时，不用 new。
String s4 = new String("abc");
String s5 = "abc"; 一般用此写法
new String(char[]);//将字符数组转成字符串。
new String(char[],offset,count);//将字符数组中的一部分转成字符串。
```

2：一般方法：

按照面向对象的思想：

2.1 获取：

2.1.1：获取字符串的长度。length();

2.1.2：指定位置的字符。char charAt(int index);

2.1.3：获取指定字符的位置。如果不存在返回-1，所以可以通过返回值-1 来判断某一个字符不存在的情况。

```
int indexOf(int ch);//返回第一次找到的字符角标
int indexOf(int ch,int fromIndex); //返回从指定位置开始第一次找到的角标
int indexOf(String str); //返回第一次找到的字符串角标
int indexOf(String str,int fromIndex);

int lastIndexOf(int ch);
int lastIndexOf(int ch,int fromIndex);
int lastIndexOf(String str);
int lastIndexOf(String str,int fromIndex);
```

2.1.4：获取子串。

```
String substring(int start);//从 start 位开始，到 length()-1 为止.
String substring(int start,int end);//从 start 开始到 end 为止。//包含 start 位，不包含 end 位。
```

substring(0,str.length());//获取整串

2.2 判断：

2.2.1：字符串中包含指定的字符串吗？

boolean **contains**(String substring);

2.2.2：字符串是否以指定字符串开头啊？

boolean **startsWith**(string);

2.2.3：字符串是否以指定字符串结尾啊？

boolean **endsWith**(string);

2.2.4：判断字符串是否相同

boolean equals(string);//覆盖了 Object 中的方法，判断字符串内容是否相同。

2.2.5：判断字符串内容是否相同，忽略大小写。

boolean **equalsIgnoreCase**(string) ;

2.3 转换：

2.3.1：通过构造函数可以将字符数组或者字节数组转成字符串。

2.3.2：可以通过字符串中的静态方法，将字符数组转成字符串。

static String **copyValueOf**(char[]);

static String copyValueOf(char[],int offset,int count);

static String **valueOf**(char[]);

static String valueOf(char[],int offset,int count);

2.3.3：将基本数据类型或者对象转成字符串。

static String valueOf(char);

static String valueOf(boolean);

static String valueOf(double);

static String valueOf(float);

static String valueOf(int);

static String valueOf(long);

static String valueOf(Object);

2.3.4：将字符串转成大小写。

String **toLowerCase**();

String **toUpperCase**();

2.3.5：将字符串转成数组。

char[] **toCharArray**();//转成字符数组。

byte[] **getBytes**();//可以加入编码表。转成字节数组。

2.3.6：将字符串转成字符串数组。切割方法。

String[] split(分割的规则-字符串);

2.3.7：将字符串进行内容替换。**注意：修改后变成新字符串，并不是将原字符串直接修改。**

String **replace**(oldChar,newChar);

String replace(oldstring,newstring);

2.3.8： String **concat**(string); //对字符串进行追加。

String trim();//去除字符串两端的空格

int compareTo();//如果参数字符串等于此字符串，则返回值 0；如果此字符串按字典顺序小于字符串参数，则返回一个小于 0 的值；如果此字符串按字典顺序大于字符串参数，则返回一个大于 0 的值。

--< java.lang >-- StringBuffer 字符串缓冲区：★★★☆

构造一个其中不带字符的字符串缓冲区，初始容量为 16 个字符。

特点：

1：可以对字符串内容进行修改。

2：是一个容器。

3：是可变长度的。

4：缓冲区中可以存储任意类型的数据。

5：最终需要变成字符串。

容器通常具备一些固定的方法：

1，添加。

StringBuffer **append**(data):在缓冲区中追加数据。追加到尾部。

StringBuffer **insert**(index,data):在指定位置插入数据。

2，删除。

StringBuffer **delete**(start,end);删除从 start 至 end-1 范围的元素

StringBuffer deleteCharAt(index);删除指定位置的元素

//sb.delete(0,sb.length());//清空缓冲区。

3，修改。

StringBuffer **replace**(start,end,string);将 start 至 end-1 替换成 string

void **setCharAt**(index,char);替换指定位置的字符

void **setLength**(len);将原字符串置为指定长度的字符串

4，查找。（查不到返回-1）

int **indexOf**(string); 返回指定子字符串在此字符串中第一次出现处的索引。

int indexOf(string,int fromIndex);从指定位置开始查找字符串

int **lastIndexOf**(string); 返回指定子字符串在此字符串中最右边出现处的索引。

int lastIndexOf(string,int fromIndex); 从指定的索引开始反向搜索

5，**获取子串。**

string **substring**(start); 返回 start 到结尾的子串

string substring(start,end); 返回 start 至 end-1 的子串

6，反转。

StringBuffer **reverse**();字符串反转

--< java.lang >-- StringBuilder 字符串缓冲区：★★★☆

JDK1.5 出现 StringBuiler；构造一个其中不带字符的字符串生成器，初始容量为 16 个字符。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。

方法和 StringBuffer 一样；

StringBuffer 和 StringBuilder 的区别：

StringBuffer 线程安全。

StringBuilder 线程不安全。

单线程操作，使用 StringBuilder 效率高。

多线程操作，使用 StringBuffer 安全。

```
StringBuilder sb = new StringBuilder("abcdefg");
sb.append("ak");  //abcdefgak
```

```
        sb.insert(1,"et");//aetbcdefg
        sb.deleteCharAt(2);//abdefg
        sb.delete(2,4);//abefg
        sb.setLength(4);//abcd
        sb.setCharAt(0,'k');//kbcdefg
        sb.replace(0,2,"hhhh");//hhhhcdefg

        //想要使用缓冲区，先要建立对象。
        StringBuffer sb = new StringBuffer();
        sb.append(12).append("haha");//方法调用链。
        String s = "abc"+4+'q';
        s = new StringBuffer().append("abc").append(4).append('q').toString();
```

```
class  Test{
    public static void main(String[] args) {
        String s1 = "java";
        String s2 = "hello";
        method_1(s1,s2);
        System.out.println(s1+"...."+s2); //java....hello

        StringBuilder s11 = new StringBuilder("java");
        StringBuilder s22 = new StringBuilder("hello");
        method_2(s11,s22);
        System.out.println(s11+"-----"+s22); //javahello-----hello
    }
    public static void method_1(String s1,String s2){
        s1.replace('a','k');
        s1 = s2;
    }
    public static void method_2(StringBuilder s1,StringBuilder s2){
        s1.append(s2);
        s1 = s2;
    }
}
```

基本数据类型对象包装类：是按照面向对象思想将基本数据类型封装成了对象。

好处：

1：可以通过对象中的属性和行为操作基本数据。

2：可以实现基本数据类型和字符串之间的转换。

关键字	对应的类名	
byte	Byte	
short	Short	paserShort(numstring);
int	Integer	静态方法：parseInt(numstring)
long	Long	
float	Float	
double	Double	

char Character
Boolean Boolean

基本数据类型对象包装类：都有 XXX parseXXX 方法
只有一个类型没有 parse 方法：Character ；

Integer 对象： ★★★☆
数字格式的字符串转成基本数据类型的方法：
1：将该字符串封装成了 Integer 对象，并调用对象的方法 intValue();
2：使用 Integer.parseInt(numstring)—>**类.方法名**:不用建立对象，直接类名调用；

将基本类型转成字符串：
1：Integer 中的静态方法 String toString(int);
2：int+"";

将一个十进制整数转成其他进制：
转成二进制：toBinaryString
转成八进制：toOctalString
转成十六进制：toHexString
toString(int num,int radix);

将其他进制转换十进制：
parseInt(string,radix); //将给定的数转成指定的基数进制；

在 jdk1.5 版本后，对基本数据类型对象包装类进行升级。在升级中，使用基本数据类型对象包装类可以像使用基本数据类型一样，进行运算。

```
Integer i = new Integer(4); //1.5 版本之前的写法；
Integer i = 4; //自动装箱，1.5 版本后的写法；
i = i + 5;
```

//i 对象是不能直接和 5 相加的，其实底层先将 i 转成 int 类型，在和 5 相加。而转成 int 类型的操作是隐式的。**自动拆箱：拆箱的原理就是 i.intValue();**i+5 运算完是一个 int 整数。如何赋值给引用类型 i 呢？其实有对结果进行装箱。

```
Integer c = 127;
Integer d = 127;
System.out.println(c == d); //true
```

//**在装箱时，如果数值在 byte 范围之内，那么数值相同，不会产生新的对象，也就是说多个数值相同的引用指向的是同一个对象。**

集合框架：★★★★★，用于存储数据的容器。
特点：
1：对象封装数据，对象多了也需要存储。**集合用于存储对象。**
2：对象的个数确定可以使用数组，但是不确定怎么办？可以用集合。因为**集合是可变长度的。**

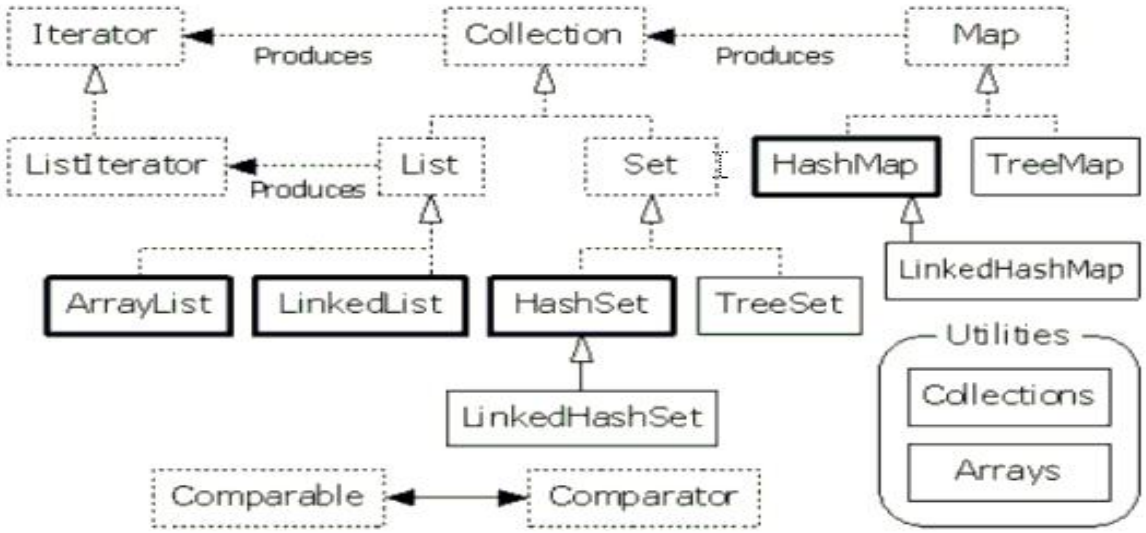
集合和数组的区别：
1：数组是固定长度的；集合可变长度的。
2：数组可以存储基本数据类型，也可以存储引用数据类型；**集合只能存储引用数据类型。**

3：数组存储的元素必须是同一个数据类型；集合存储的对象可以是不同数据类型。

数据结构：就是容器中存储数据的方式。

对于集合容器，有很多种。因为每一个容器的自身特点不同，其实原理在于每个容器的内部数据结构不同。
集合容器在不断向上抽取过程中。出现了集合体系。
在使用一个体系时，原则：参阅顶层内容。建立底层对象。

java 中集合类的关系图



--< java.util >-- Collection 接口：
Collection：
|--List：有序(元素存入集合的顺序和取出的顺序一致)，元素都有索引。元素可以重复。
|--Set：无序(存入和取出顺序有可能不一致)，不可以存储重复元素。必须保证元素唯一性。

1，添加：
add(object)：添加一个元素
addAll(Collection) ：添加一个集合中的所有元素。
2，删除：
clear()：将集合中的元素全删除，**清空集合**。
remove(obj) ：删除集合中指定的对象。**注意：删除成功，集合的长度会改变。**
removeAll(collection) ：删除部分元素。部分元素和传入 Collection 一致。
3，判断：
boolean contains(obj) ：集合中是否包含指定元素 。
boolean containsAll(Collection) ：集合中是否包含指定的多个元素。
boolean isEmpty()：集合中是否有元素。
4，获取：
int size()：集合中有几个元素。
5，取交集：
boolean retainAll(Collection) ：对当前集合中保留和指定集合中的相同的元素。如果两个集合元素相同，返回 flase；如果 retainAll 修改了当前集合，返回 true。

6，获取集合中所有元素：

Iterator iterator()：迭代器

7，将集合变成数组：

toArray();

--< java.util >-- Iterator 接口：

迭代器：是一个接口。作用：用于取集合中的元素。

boolean	**hasNext**() 如果仍有元素可以迭代，则返回 true。
E	**next**() 返回迭代的下一个元素。
void	**remove**() 从迭代器指向的 collection 中移除迭代器返回的最后一个元素（可选操作）。

每一个集合都有自己的数据结构(就是容器中存储数据的方式),都有特定的取出自己内部元素的方式。为了便于操作所有的容器，取出元素。将容器内部的取出方式按照一个统一的规则向外提供，这个规则就是 Iterator 接口。

也就说，只要通过该接口就可以取出 Collection 集合中的元素，至于每一个具体的容器依据自己的数据结构，如何实现的具体取出细节，这个不用关心，这样就降低了取出元素和具体集合的耦合性。

Iterator it = coll.iterator();//获取容器中的迭代器对象，至于这个对象是是什么不重要。这对象肯定符合一个规则 Iterator 接口。

```
public static void main(String[] args) {
    Collection coll = new ArrayList();
    coll.add("abc0");
    coll.add("abc1");
    coll.add("abc2");
    //-------------方式 1---------------------
    Iterator it = coll.iterator();
    while(it.hasNext()){
        System.out.println(it.next());
    }
    //---------------方式 2 用此种----------------------
    for(Iterator it = coll.iterator();it.hasNext(); ){
        System.out.println(it.next());
    }
}
```

--< java.util >-- List 接口：

List 本身是 Collection 接口的子接口，具备了 Collection 的所有方法。现在学习 List 体系特有的共性方法，查阅方法发现 List 的特有方法都有索引，这是该集合最大的特点。

List：有序(元素存入集合的顺序和取出的顺序一致)，元素都有索引。元素可以重复。

|--ArrayList：底层的数据结构是数组,线程不同步，ArrayList 替代了 Vector，查询元素的速度非常快。

|--LinkedList：底层的数据结构是链表，线程不同步，增删元素的速度非常快。

|--Vector：底层的数据结构就是数组，线程同步的，Vector 无论查询和增删都巨慢。

1，添加：
add(index,element) ：在指定的索引位插入元素。
addAll(index,collection) ：在指定的索引位插入一堆元素。
2，删除：
remove(index) ：删除指定索引位的元素。 返回被删的元素。
3，获取：
Object get(index) ：通过索引获取指定元素。
int indexOf(obj) ：获取指定元素第一次出现的索引位，如果该元素不存在返回-1；
所以，通过-1，可以判断一个元素是否存在。
int lastIndexOf(Object o) ：反向索引指定元素的位置。
List subList(start,end) ：获取子列表。
4，修改：
Object set(index,element) ：对指定索引位进行元素的修改。
5，获取所有元素：
ListIterator listIterator()：list 集合特有的迭代器。

List 集合支持对元素的增、删、改、查。

List 集合因为角标有了自己的获取元素的方式： 遍历。

```
for(int x=0; x<list.size(); x++){
  sop("get:"+list.get(x));
}
```

在进行 list 列表元素迭代的时候，如果想要在迭代过程中，想要对元素进行操作的时候，比如满足条件添加新元素。会发生.ConcurrentModificationException 并发修改异常。

导致的原因是：

集合引用和迭代器引用在同时操作元素，通过集合获取到对应的迭代器后，在迭代中，进行集合引用的元素添加，迭代器并不知道，所以会出现异常情况。

如何解决呢？

既然是在迭代中对元素进行操作，找迭代器的方法最为合适.可是 Iterator 中只有 hasNext,next,remove 方法.通过查阅的它的子接口,ListIterator,发现该列表迭代器接口具备了对元素的增、删、改、查的动作。

ListIterator 是 List 集合特有的迭代器。

```
ListIterator it = list.listIterator;//取代 Iterator it = list.iterator;
```

方法摘要	
void	**add**(E e) 将指定的元素插入列表（可选操作）。
boolean	**hasNext**() 以正向遍历列表时，如果列表迭代器有多个元素，则返回 true（换句话说，如果 next 返回一个元素而不是抛出异常，则返回 true）。
boolean	**hasPrevious**() 如果以逆向遍历列表，列表迭代器有多个元素，则返回 true。
E	**next**() 返回列表中的下一个元素。
int	**nextIndex**() 返回对 next 的后续调用所返回元素的索引。
E	**previous**() 返回列表中的前一个元素。
int	**previousIndex**() 返回对 previous 的后续调用所返回元素的索引。

void	remove() 从列表中移除由 next 或 previous 返回的最后一个元素（可选操作）。
void	set(E e) 用指定元素替换 next 或 previous 返回的最后一个元素（可选操作）。

可变长度数组的原理：
当元素超出数组长度，会产生一个新数组，将原数组的数据复制到新数组中，再将新的元素添加到新数组中。
ArrayList：是按照原数组的 50%延长。构造一个初始容量为 10 的空列表。
Vector：是按照原数组的 100%延长。

注意：对于 list 集合，底层判断元素是否相同，其实用的是元素自身的 equals 方法完成的。所以建议元素都要复写 equals 方法，建立元素对象自己的比较相同的条件依据。

LinkedList：的特有方法。
addFirst();
addLast();
在 jdk1.6 以后。
offerFirst();
offerLast();

getFirst():获取链表中的第一个元素。如果链表为空，抛出 NoSuchElementException;
getLast();获取链表中的最后一个元素。如果链表为空，抛出 NoSuchElementException;
在 jdk1.6 以后。
peekFirst();获取链表中的第一个元素。如果链表为空，返回 null。
peekLast();

removeFirst()：获取链表中的第一个元素，但是会删除链表中的第一个元素。如果链表为空，抛出 NoSuchElementException
removeLast();
在 jdk1.6 以后。
pollFirst();获取链表中的第一个元素，但是会删除链表中的第一个元素。如果链表为空，返回 null。
pollLast();

<阅读到此>--< java.util >-- Set 接口：
数据结构：数据的存储方式；
Set 接口中的方法和 Collection 中方法一致的。Set 接口取出方式只有一种，**迭代器**。
|--HashSet：底层数据结构是哈希表，线程**是不同步的**。无序，高效；
HashSet 集合保证元素唯一性：通过元素的 hashCode 方法，和 equals 方法完成的。
当元素的 hashCode 值相同时，才继续判断元素的 equals 是否为 true。
如果为 true，那么视为相同元素，不存。如果为 false，那么存储。
如果 hashCode 值不同，那么不判断 equals，从而提高对象比较的速度。
|--LinkedHashSet：有序，hashset 的子类。
|--TreeSet：对 Set 集合中的元素的进行指定顺序的排序。**不同步**。TreeSet 底层的数据结构就是二叉树。

哈希表的原理：

1，对对象元素中的关键字(对象中的特有数据)，进行哈希算法的运算，并得出一个具体的算法值，这个值 称为**哈希值**。
2，哈希值就是这个元素的位置。
3，如果哈希值出现冲突，再次判断这个关键字对应的对象是否相同。如果对象相同，就不存储，因为元素重复。如果对象不同，就存储，在原来对象的哈希值基础 +1 顺延。
4，存储哈希值的结构，我们称为哈希表。
5，既然哈希表是根据哈希值存储的，为了提高效率，最好保证对象的关键字是唯一的。
这样可以尽量少的判断关键字对应的对象是否相同，提高了哈希表的操作效率。

对于 ArrayList 集合，判断元素是否存在，或者删元素底层依据都是 equals 方法。
对于 HashSet 集合，判断元素是否存在，或者删除元素，底层依据的是 hashCode 方法和 equals 方法。

TreeSet：
用于对 Set 集合进行元素的指定顺序排序，排序需要依据元素自身具备的比较性。
如果元素不具备比较性，在运行时会发生 ClassCastException 异常。
所以需要元素**实现 Comparable 接口**，强制让元素具备比较性，**复写 compareTo 方法**。
依据 compareTo 方法的返回值，确定元素在 TreeSet 数据结构中的位置。
TreeSet **方法保证元素唯一性的方式：**就是参考比较方法的结果是否为 0，如果 return 0，视为两个对象重复，不存。

注意：在进行比较时，如果判断元素不唯一，比如，同姓名，同年龄，才视为同一个人。
在判断时，需要分主要条件和次要条件，当主要条件相同时，再判断次要条件，按照次要条件排序。

TreeSet 集合排序有两种方式，Comparable 和 Comparator 区别：
1：让元素自身具备比较性，需要元素对象实现 Comparable 接口，覆盖 compareTo 方法。
2：让集合自身具备比较性，需要定义一个实现了 Comparator 接口的比较器，并覆盖 compare 方法，并将该类对象作为实际参数传递给 TreeSet 集合的构造函数。
第二种方式较为灵活。

Map 集合：
|--Hashtable：底层是哈希表数据结构，是线程同步的。不可以存储 null 键，null 值。
|--HashMap：底层是哈希表数据结构，是线程不同步的。可以存储 null 键，null 值。替代了 Hashtable.
|--TreeMap：底层是二叉树结构，可以对 map 集合中的键进行指定顺序的排序。

Map 集合存储和 Collection 有着很大不同：
Collection 一次存一个元素；Map 一次存一对元素。
Collection 是单列集合；Map 是双列集合。
Map 中的存储的一对元素：一个是键，一个是值，**键与值之间有对应(映射)关系。**
特点：要保证 map 集合中键的唯一性。

1，添加。
put(key,value)：当存储的键相同时，新的值会替换老的值，并将老值返回。如果键没有重复，返回 null。

void **putAll**(Map);

2，删除。

void **clear**()：清空

value **remove**(key) ：删除指定键。

3，判断。

boolean **isEmpty**()：

boolean **containsKey**(key)：是否包含 key

boolean **containsValue**(value) ：是否包含 value

4，取出。

int **size**()：返回长度

value **get**(key) ：通过指定键获取对应的值。如果返回 null，可以判断该键不存在。当然有特殊情况，就是在 hashmap 集合中，是可以存储 null 键 null 值的。

Collection **values**()：**获取 map 集合中的所有的值。**

5，想要获取 map 中的所有元素：

原理：map 中是没有迭代器的，collection 具备迭代器，只要将 map 集合转成 Set 集合，可以使用迭代器了。之所以转成 set，是因为 map 集合具备着键的唯一性，其实 set 集合就来自于 map，set 集合底层其实用的就是 map 的方法。

★ 把 map 集合转成 set 的方法：

```
Set keySet();
Set entrySet();//取的是键和值的映射关系。
```

Entry 就是 Map 接口中的内部接口；

为什么要定义在 map 内部呢？entry 是访问键值关系的入口，是 map 的入口，访问的是 map 中的键值对。

取出 map 集合中所有元素的方式一：keySet()方法。

可以将 map 集合中的键都取出存放到 set 集合中。对 set 集合进行迭代。迭代完成，再通过 get 方法对获取到的键进行值的获取。

```
Set keySet = map.keySet();
Iterator it = keySet.iterator();
while(it.hasNext()) {
    Object key = it.next();
    Object value = map.get(key);
    System.out.println(key+":"+value);
}
```

取出 map 集合中所有元素的方式二：entrySet()方法。

```
Set entrySet = map.entrySet();
Iterator it = entrySet.iterator();
while(it.hasNext()) {
    Map.Entry  me = (Map.Entry)it.next();
    System.out.println(me.getKey()+"::::"+me.getValue());
}
```

使用集合的技巧：

看到 Array 就是数组结构，有角标，查询速度很快。

看到 link 就是链表结构：增删速度快，而且有特有方法。addFirst； addLast； removeFirst()；removeLast()； getFirst()； getLast()；

看到 hash 就是哈希表，就要想要哈希值，就要想到唯一性，就要想到存入到该结构的中的元素必须覆盖 hashCode，equals 方法。

看到 tree 就是二叉树，就要想到排序，就想要用到比较。

比较的两种方式：

一个是 Comparable：覆盖 compareTo 方法；

一个是 Comparator：覆盖 compare 方法。

LinkedHashSet，LinkedHashMap:这两个集合可以保证哈希表有存入顺序和取出顺序一致，保证哈希表有序。

集合什么时候用？

当存储的是一个元素时，就用 Collection。当存储对象之间存在着映射关系时，就使用 Map 集合。

保证唯一，就用 Set。不保证唯一，就用 List。

Collections：它的出现给集合操作提供了更多的功能。这个类不需要创建对象，内部提供的都是静态方法。

静态方法：

```
Collections.sort(list);//list 集合进行元素的自然顺序排序。
  Collections.sort(list,new ComparatorByLen());//按指定的比较器方法排序。
class ComparatorByLen implements Comparator<String>{
  public int compare(String s1,String s2){
      int temp = s1.length()-s2.length();
      return temp==0?s1.compareTo(s2):temp;
  }
}
Collections.max(list); //返回 list 中字典顺序最大的元素。
int index = Collections.binarySearch(list,"zz");//二分查找，返回角标。
Collections.reverseOrder();//逆向反转排序。
Collections.shuffle(list);//随机对 list 中的元素进行位置的置换。
```

将非同步集合转成同步集合的方法：Collections 中的 `XXX synchronizedXXX(XXX);`

```
List synchronizedList(list);
Map synchronizedMap(map);
```

原理：定义一个类，将集合所有的方法加同一把锁后返回。

Collection 和 Collections 的区别：

Collections 是个 java.util 下的类，是针对集合类的一个工具类,提供一系列静态方法,实现对集合的查找、排序、替换、线程安全化（将非同步的集合转换成同步的）等操作。

Collection 是个 java.util 下的接口，它是各种集合结构的父接口，继承于它的接口主要有 Set 和 List,提供了关于集合的一些操作,如插入、删除、判断一个元素是否其成员、遍历等。

Arrays：

用于操作数组对象的工具类，里面都是静态方法。

asList 方法：将数组转换成 list 集合。

```
String[] arr = {"abc","kk","qq"};
```

List<String> list = Arrays.asList(arr);//将 arr 数组转成 list 集合。

将数组转换成集合，有什么好处呢？用 aslist 方法，将数组变成集合；

可以通过 list 集合中的方法来操作数组中的元素：isEmpty()、contains、indexOf、set；

注意（局限性）：数组是固定长度，不可以使用集合对象增加或者删除等，会改变数组长度的功能方法。比如 add、remove、clear。（会报不支持操作异常 UnsupportedOperationException）；

如果数组中存储的引用数据类型，直接作为集合的元素可以直接用集合方法操作。

如果数组中存储的是基本数据类型，asList 会将数组实体作为集合元素存在。

集合变数组：用的是 Collection 接口中的方法：toArray();

如果给 toArray 传递的指定类型的数据长度小于了集合的 size，那么 toArray 方法，会自定再创建一个该类型的数据，长度为集合的 size。

如果传递的指定的类型的数组的长度大于了集合的 size，那么 toArray 方法，就不会创建新数组，直接使用该数组即可，并将集合中的元素存储到数组中，其他为存储元素的位置默认值 null。

所以，在传递指定类型数组时，最好的方式就是指定的长度和 size 相等的数组。

将集合变成数组后有什么好处？限定了对集合中的元素进行增删操作，只要获取这些元素即可。

Jdk5.0 新特性：

Collection 在 jdk1.5 以后，有了一个父接口 Iterable，这个接口的出现的将 iterator 方法进行抽取，提高了扩展性。

增强 for 循环：**foreach 语句，**foreach 简化了迭代器。

格式：// 增强 for 循环括号里写两个参数，第一个是声明一个变量，第二个就是需要迭代的容器

```
for( 元素类型 变量名 : Collection 集合 & 数组 ) {
  …
}
```

高级 for 循环和传统 for 循环的区别：

高级 for 循环在使用时，必须要明确被遍历的目标。这个目标，可以是 Collection 集合或者数组，如果遍历 Collection 集合，在遍历过程中还需要对元素进行操作，比如删除，**需要使用迭代器。**

如果遍历数组，还需要对数组元素进行操作，建议用传统 for 循环因为可以定义角标通过角标操作元素。如果只为遍历获取，可以简化成高级 for 循环，它的出现为了简化书写。

高级 for 循环可以遍历 map 集合吗？不可以。但是可以将 map 转成 set 后再使用 foreach 语句。

1)、作用：对存储对象的容器进行迭代： 数组 collection map

2)、增强 for 循环迭代数组：

```
String [] arr = {"a", "b", "c"};//数组的静态定义方式，只试用于数组首次定义的时候
for(String s : arr) {
  System.out.println(s);
}
```

3)、单列集合 Collection：

```
List list = new ArrayList();
list.add("aaa");
// 增强 for 循环，没有使用泛型的集合能不能使用增强 for 循环迭代？能
for(Object obj : list) {
```

```
    String s = (String) obj;
    System.out.println(s);
  }
```

4)、双列集合 Map：

```
  Map map = new HashMap();
  map.put("a", "aaa");
  // 传统方式：必须掌握这种方式
  Set entrys = map.entrySet(); // 1.获得所有的键值对 Entry 对象
  iter = entrys.iterator(); // 2.迭代出所有的 entry
  while(iter.hasNext()) {
    Map.Entry entry = (Entry) iter.next();
    String key = (String) entry.getKey(); // 分别获得 key 和 value
    String value = (String) entry.getValue();
    System.out.println(key + "=" + value);
  }
```

// 增强 for 循环迭代：原则上 map 集合是无法使用增强 for 循环来迭代的，因为增强 for 循环只能针对实现了 Iterable 接口的集合进行迭代；Iterable 是 jdk5 中新定义的接口，就一个方法 iterator 方法，只有实现了 Iterable 接口的类，才能保证一定有 iterator 方法，java 有这样的限定是因为增强 for 循环内部还是用迭代器实现的，而实际上，我们可以通过某种方式来使用增强 for 循环。

```
  for(Object obj : map.entrySet()) {
    Map.Entry entry = (Entry) obj;  // obj 依次表示 Entry
    System.out.println(entry.getKey() + "=" + entry.getValue());
  }
```

5)、集合迭代注意问题：在迭代集合的过程中，不能对集合进行增删操作（会报并发访问异常）；可以用迭代器的方法进行操作（子类 listIterator：有增删的方法）。

6)、增强 for 循环注意问题：在使用增强 for 循环时，不能对元素进行赋值；

```
  int[] arr = {1,2,3};
  for(int num : arr) {
    num = 0; //不能改变数组的值
  }
  System.out.println(arr[1]); //2
```

可变参数（...）：用到函数的参数上，当要操作的同一个类型元素个数不确定的时候，可是用这个方式，这个参数可以接受任意个数的同一类型的数据。

和以前接收数组不一样的是：

以前定义数组类型，需要先创建一个数组对象，再将这个数组对象作为参数传递给函数。现在，直接将数组中的元素作为参数传递即可。底层其实是将这些元素进行数组的封装，而这个封装动作，是在底层完成的，被隐藏了。所以简化了用户的书写，少了调用者定义数组的动作。

如果在参数列表中使用了可变参数，**可变参数必须定义在参数列表结尾(也就是必须是最后一个参数，否则编译会失败。)。**

如果要获取多个 int 数的和呢？可以使用将多个 int 数封装到数组中，直接对数组求和即可。

静态导入：导入了类中的所有静态成员，简化静态成员的书写。

```
import static java.util.Collections.*;  //导入了 Collections 类中的所有静态成员
```

枚举：关键字 enum

问题：对象的某个属性的值不能是任意的，必须为固定的一组取值其中的某一个；

解决办法：

1）、在 setGrade 方法中做判断，不符合格式要求就抛出异常；

2）、直接限定用户的选择，通过自定义类模拟枚举的方式来限定用户的输入，写一个 Grade 类，私有构造函数，对外提供 5 个静态的常量表示类的实例；

3)、jdk5 中新定义了枚举类型，专门用于解决此类问题；

4)、枚举就是一个特殊的 java 类，可以定义属性、方法、构造函数、实现接口、继承类；

自动拆装箱：java 中数据类型分为两种 ： 基本数据类型 引用数据类型(对象)

在 java 程序中所有的数据都需要当做对象来处理，针对 8 种基本数据类型提供了包装类，如下：

int --> Integer

byte --> Byte

short --> Short

long --> Long

char --> Character

double --> Double

float --> Float

boolean --> Boolean

jdk5 以前基本数据类型和包装类之间需要互转：

基本---引用 Integer x = new Integer(x);

引用---基本 int num = x.intValue();

1)、Integer x = 1; x = x + 1; 经历了什么过程？**装箱 → 拆箱 → 装箱**；

2)、为了优化，虚拟机为包装类提供了缓冲池，**Integer 池的大小 -128~127 一个字节的大小**；

3)、**String 池：**Java 为了优化字符串操作 提供了一个缓冲池；

泛型：jdk1.5 版本以后出现的一个安全机制。表现格式：< >

好处：

1：将运行时期的问题 ClassCastException 问题转换成了编译失败，体现在编译时期，程序员就可以解决问题。

2：避免了强制转换的麻烦。

只要带有<>的类或者接口，都属于带有类型参数的类或者接口，在使用这些类或者接口时，必须给<>中传递一个具体的引用数据类型。

泛型技术：其实应用在编译时期，是给编译器使用的技术，到了运行时期，泛型就不存在了。

为什么？因为**泛型的擦除**：也就是说，编辑器检查了泛型的类型正确后，在生成的类文件中是没有泛型的。

在运行时，如何知道获取的元素类型而不用强转呢？

泛型的补偿：因为存储的时候，类型已经确定了是同一个类型的元素，所以在运行时，只要获取到该元素的类型，在内部进行一次转换即可，所以使用者不用再做转换动作了。

什么时候用泛型类呢？

当类中的操作的**引用数据类型不确定**的时候，以前用的 Object 来进行扩展的，现在可以用泛型来表示。这样可以避免强转的麻烦，而且将运行问题转移到的编译时期。

泛型在程序定义上的体现：

//泛型类：将泛型定义在类上。

```
class Tool<Q> {
    private Q obj;
    public  void setObject(Q obj) {
        this.obj = obj;
    }
    public Q getObject() {
        return obj;
    }
}
```

//当方法操作的引用数据类型不确定的时候，可以将泛型定义在方法上。

```
    public <W> void method(W w) {
        System.out.println("method:"+w);
    }
```

//静态方法上的泛型：静态方法无法访问类上定义的泛型。如果静态方法操作的引用数据类型不确定的时候，必须要将泛型定义在方法上。

```
    public static <Q> void function(Q t) {
        System.out.println("function:"+t);
    }
```

//泛型接口.

```
interface Inter<T> {
    void show(T t);
}
class InterImpl<R> implements Inter<R> {
    public void show(R r) {
        System.out.println("show:"+r);
    }
}
```

泛型中的通配符：可以解决当具体类型不确定的时候，这个通配符就是 ？ ；当操作类型时，不需要使用类型的具体功能时，只使用 Object 类中的功能。那么可以用 ？ 通配符来表未知类型。

泛型限定：

上限：？ extends E：表示这个对象的实例，可以接收 E 类型或者 E 的子类型对象。

下限：？ super E：可以接收 E 类型或者 E 的父类型对象。

上限什么时候用：往集合中添加元素时，既可以添加 E 类型对象，又可以添加 E 的子类型对象。为什么？因为取的时候，E 类型既可以接收 E 类对象，又可以接收 E 的子类型对象。

下限什么时候用：当从集合中获取元素进行操作的时候，可以用当前元素的类型接收，也可以用当前元素的父类型接收。

泛型的细节：

1)、泛型到底代表什么类型取决于调用者传入的类型，如果没传，默认是 Object 类型；

2)、使用带泛型的类创建对象时，等式两边指定的泛型必须一致；

原因：编译器检查对象调用方法时只看变量，然而程序运行期间调用方法时就要考虑对象具体类型了；

3)、等式两边可以在任意一边使用泛型，在另一边不使用(考虑向后兼容)；

```
ArrayList<String> al = new ArrayList<Object>();  //错
//要保证左右两边的泛型具体类型一致就可以了，这样不容易出错。
ArrayList<? extends Object> al = new ArrayList<String>();
al.add("aa");  //错，不能加 String 类型的对象
//因为集合具体对象中既可存储 String，也可以存储 Object 的其他子类，所以添加具体的类型对象不合适，类型检查会出现安全问题。 ? extends Object 代表 Object 的子类型不确定，怎么能添加具体类型的对象呢？
public static void method(ArrayList<? extends Object> al) {
    al.add("abc");  //错
  //只能对 al 集合中的元素调用 Object 类中的方法，具体子类型的方法都不能用，因为子类型不确定。
}
```

API--- java.lang.System: 属性和行为都是静态的。

long currentTimeMillis(); // 返回当前时间毫秒值

exit(); // 退出虚拟机

Properties getProperties() ; // 获取当前系统的属性信息

Properties prop = System.getProperties(); **//获取系统的属性信息，并将这些信息存储到 Properties 集合中。**

System.setProperty("myname","毕老师"); **//给系统属性信息集添加具体的属性信息**

//临时设置方式：运行 jvm 时，可以通过 jvm 的参数进行系统属性的临时设置，可以在 java 命令的后面加入 –D<name>=<value> 用法：java –Dmyname=小明 类名。

String name = System.getProperty("os.name");**//获取指定属性的信息**

```
//想要知道该系统是否是该软件所支持的系统中的一个。
  Set<String> hs = new HashSet<String>();
  hs.add("Windows XP");
  hs.add("Windows 7");
  if(hs.contains(name))
    System.out.println("可以支持");
  else
    System.out.println("不支持");
```

API--- java.lang.Runtime: 类中没有构造方法，不能创建对象。

但是有非静态方法。说明该类中应该定义好了对象，并可以通过一个 static 方法获取这个对象。用这个对象来调用非静态方法。这个方法就是 static Runtime getRuntime();

这个 Runtime 其实使用单例设计模式进行设计。

```
class    RuntimeDemo {
    public static void main(String[] args) throws Exception {
        Runtime r = Runtime.getRuntime();
        Process p = r.exec("notepad.exe SystemDemo.java");     //运行指定的程序
        Thread.sleep(4000);
        p.destroy();    //杀掉进程
    }
}
```

API--- java.util.Math: 用于数学运算的工具类，属性和行为都是静态的。该类是 final 不允许继承。

```
static double ceil(double a) ; //返回大于指定数值的最小整数
static double floor(double a) ; //返回小于指定数值的最大整数
static long round(double a) ; //四舍五入成整数
static double pow(double a, double b) ; //a 的 b 次幂
static double random(); //返回 0~1 的伪随机数

    public static void main(String[] args) {
        Random r = new Random();
        for(int x=0; x<10; x++) {
            //double d = Math.floor(Math.random()*10+1);
            //int d    = (int)(Math.random()*10+1);
            int d = r.nextInt(10)+1;
            System.out.println(d);
        }
    }
```

API--- java.util.Date： 日期类，月份从 0-11；

```
    /*
    日期对象和毫秒值之间的转换。
    1，日期对象转成毫秒值。Date 类中的 getTime 方法。
    2，如何将获取到的毫秒值转成具体的日期呢？
        Date 类中的 setTime 方法。也可以通过构造函数。
    */
        //日期对象转成毫秒值
        Date d = new Date();
        long time1 = d.getTime();
        long time2 = System.currentTimeMillis(); / /毫秒值。

        //毫秒值转成具体的日期
        long time = 1322709921312l;
        Date d = new Date();
        d.setTime(time);
    /*
    将日期字符串转换成日期对象：使用的就是 DateFormat 方法中的    Date parse(String source) ；
    */
```

```
public static void method() throws Exception {
    String str_time = "2011/10/25";
    DateFormat df = new SimpleDateFormat("yyyy/MM/dd"); //SimpleDateFormat 作为可以指定用户自定义的格式来完成格式化。
    Date d = df.parse(str_time);
}
/*
如果不需要使用特定的格式化风格，完全可以使用 DateFormat 类中的静态工厂方法获取具体的已经封装好风格的对象。getDateInstance();getDateTimeInstance();
*/
    Date d = new Date();
    DateFormat df = DateFormat.getDateInstance(DateFormat.LONG);
    df = DateFormat.getDateTimeInstance(DateFormat.LONG,DateFormat.LONG);
    String str_time = df.format(d);

//将日期对象转换成字符串的方式：DateFormat 类中的 format 方法。
    //创建日期格式对象。
    DateFormat df = new SimpleDateFormat(); //该对象的建立内部会封装一个默认的日期格式。11-12-1 下午 1:48
    //如果想要自定义日期格式的话。可使用 SimpleDateFormat 的构造函数。将具体的格式作为参数传入到构造函数中。如何表示日期中年的部分呢？可以必须要参与格式对象文档。
    df = new SimpleDateFormat("yyyy 年 MM 月 dd 日  HH:mm:ss");
    //调用 DateFormat 中的 format 方法。对已有的日期对象进行格式化。
    String str_time = df.format(d);
```

API--- java.util. Calendar：日历类

```
public static void method(){
    Calendar c = Calendar.getInstance();
    System.out.println(c.get(Calendar.YEAR)+"年"+(c.get(Calendar.MONTH)+1)+"月"
                       +getNum(c.get(Calendar.DAY_OF_MONTH))+"日"
                       +"星期"+getWeek(c.get(Calendar.DAY_OF_WEEK)));
}
public static String getNum(int num){
    return num>9 ? num+"" : "0"+num;
}
public static String getWeek(int index){
/*
查表法：建立数据的对应关系.
最好：数据个数是确定的，而且有对应关系。如果对应关系的一方，是数字，而且可以作为角标，那么可以通过数组来作为表。
*/
    String[] weeks = {"","日","一","二","三","四","五","六"};
    return weeks[index];
}
```

IO 流：★★★★★，用于处理设备上数据。在流中一般以字节的形式存放着数据！

流：可以理解数据的流动，就是一个数据流。IO 流最终要以对象来体现，对象都存在 IO 包中。
流也进行分类：
1：输入流（读）和输出流（写）。
2：因为处理的数据不同，分为字节流和字符流。

字节流：处理字节数据的流对象。设备上的数据无论是图片或者 dvd，文字，它们都以二进制存储的。二进制的最终都是以一个 8 位为数据单元进行体现，所以计算机中的最小数据单元就是字节。意味着，字节流可以处理设备上的所有数据，所以字节流一样可以处理字符数据。

那么为什么要有字符流呢？因为字符每个国家都不一样，所以涉及到了字符编码问题，那么 GBK 编码的中文用 unicode 编码解析是有问题的，所以需要获取中文字节数据的同时+ 指定的编码表才可以解析正确数据。为了方便于文字的解析，所以将字节流和编码表封装成对象，这个对象就是**字符流**。只要操作字符数据，优先考虑使用字符流体系。

注意：**流的操作只有两种：读和写。**

流的体系因为功能不同，但是有共性内容，不断抽取，形成继承体系。该体系一共有四个基类，而且都是抽象类。

字节流：InputStream　OutputStream
字符流：Reader　Writer

在这四个系统中，它们的子类，都有一个共性特点：子类名后缀都是父类名，前缀名都是这个子类的功能名称。

```
public static void main(String[] args) throws IOException { //读、写都会发生 IO 异常
    /*
    1：创建一个字符输出流对象，用于操作文件。该对象一建立，就必须明确数据存储位置，是一个文件。
    2：对象产生后，会在堆内存中有一个实体，同时也调用了系统底层资源，在指定的位置创建了一个存储数据的文件。
    3：如果指定位置，出现了同名文件，文件会被覆盖。
    */
    FileWriter fw = new FileWriter("demo.txt"); // FileNotFoundException
    /*
    调用 Writer 类中的 write 方法写入字符串。字符串并未直接写入到目的地中，而是写入到了流中，(其实是写入到内存缓冲区中)。怎么把数据弄到文件中？
    */
    fw.write("abcde");
    fw.flush(); // 刷新缓冲区，将缓冲区中的数据刷到目的地文件中。
    fw.close(); // 关闭流，其实关闭的就是 java 调用的系统底层资源。在关闭前，会先刷新该流。
}
```

close()和 flush()的区别：
flush()：将缓冲区的数据刷到目的地中后，流可以使用。
close()：将缓冲区的数据刷到目的地中后，流就关闭了，该方法主要用于结束调用的底层资源。这个动作一定做。

---

io 异常的处理方式：io 一定要写 finally；

FileWriter 写入数据的细节：
1：window 中的换行符：\r\n 两个符号组成。 linux：\n。
2：**续写数据，只要在构造函数中传入新的参数 true。**
3：目录分割符：window \\ /

```
public static void main(String[] args) {
        FileWriter fw = null;
        try {
            fw = new FileWriter("demo.txt",true);
            fw.write("abcde");
        }
        catch (IOException e ){
            System.out.println(e.toString()+"....");
        }
        finally{
            if(fw!=null)
                try{
                    fw.close();
                }
                catch (IOException e){
                    System.out.println("close:"+e.toString());
                }
        }
}
```

FileReader：使用 Reader 体系，读取一个文本文件中的数据。**返回 -1 ，标志读到结尾。**

```
import java.io.*;
class   FileReaderDemo {
    public static void main(String[] args) throws IOException {
        /*
        创建可以读取文本文件的流对象，FileReader 让创建好的流对象和指定的文件相关联。
        */
        FileReader fr = new FileReader("demo.txt");
        int ch = 0;
        while((ch = fr.read())!= -1) { //条件是没有读到结尾
            System.out.println((char)ch); //调用读取流的 read 方法，读取一个字符。
            read()方法一次读一个字节的二进制数据—是 int 型的！
        }
        fr.close();
    }
}
```

读取数据的第二种方式：第二种方式较为高效，**自定义缓冲区。**

```
import java.io.*;
```

```
class FileReaderDemo2 {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("demo.txt"); //创建读取流对象和指定文件关联。
        //因为要使用 read(char[])方法，将读取到字符存入数组。所以要创建一个字符数组，一般数组的长度都是 1024 的整数倍。
        char[] buf = new char[1024];//读取的字符数组长度是 1024
        int len = 0;
        while(( len=fr.read(buf)) != -1) {
            System.out.println(new String(buf,0,len));//将 char 类型的数据从 0 到 len 转换成 String
        }
        fr.close();
    }
}
```

IO 中的使用到了一个设计模式：装饰设计模式。

装饰设计模式解决：对一组类进行功能的增强。

包装：写一个类(包装类)对被包装对象进行包装；

* 1、包装类和被包装对象要实现同样的接口；
* 2、包装类要持有一个被包装对象；
* 3、包装类在实现接口时，大部分方法是靠调用被包装对象来实现的，对于需要修改的方法我们自己实现；

字符流：

Reader：用于读取字符流的抽象类。子类必须实现的方法只有 read(char[], int, int) 和 close()。

|---BufferedReader：从字符输入流中读取文本，缓冲各个字符，从而实现字符、数组和行的高效读取。 可以指定缓冲区的大小，或者可使用默认的大小。大多数情况下，默认值就足够大了。

|---LineNumberReader：跟踪行号的缓冲字符输入流。此类定义了方法 setLineNumber(int) 和 getLineNumber()，它们可分别用于设置和获取当前行号。

|---InputStreamReader：是字节流通向字符流的桥梁：它使用指定的 **charset 读取字节并将其解码为字符**。它使用的字符集可以由名称指定或显式给定，或者可以接受平台默认的字符集。

|---FileReader：用来读取字符文件的便捷类。此类的构造方法假定默认字符编码和默认字节缓冲区大小都是适当的。要自己指定这些值，可以先在 FileInputStream 上构造一个 InputStreamReader。

|---CharArrayReader：

|---StringReader：

Writer：写入字符流的抽象类。子类必须实现的方法仅有 write(char[], int, int)、flush() 和 close()。

|---BufferedWriter：将文本写入字符输出流，缓冲各个字符，从而提供单个字符、数组和字符串的高效写入。

|---OutputStreamWriter：是字符流通向字节流的桥梁：可使用指定的 charset **将要写入流中的字符编码成字节**。它使用的字符集可以由名称指定或显式给定，否则将接受平台默认的字符集。

|---FileWriter：用来写入字符文件的便捷类。此类的构造方法假定默认字符编码和默认字节缓冲区大小都是可接受的。要自己指定这些值，可以先在 FileOutputStream 上构造一个 OutputStreamWriter。

|---PrintWriter：

|---CharArrayWriter：

|---StringWriter：

字节流：

InputStream：是表示字节输入流的所有类的超类。

|--- FileInputStream：从文件系统中的某个文件中获得输入字节。哪些文件可用取决于主机环境。FileInputStream 用于读取诸如图像数据之类的原始字节流。要读取字符流，请考虑使用 FileReader。

|--- FilterInputStream：包含其他一些输入流，它将这些流用作其基本数据源，它可以直接传输数据或提供一些额外的功能。

|--- BufferedInputStream：该类实现缓冲的输入流。

|--- Stream：

|--- ObjectInputStream：

|--- PipedInputStream：

OutputStream：此抽象类是表示输出字节流的所有类的超类。

|--- FileOutputStream：文件输出流是用于将数据写入 File 或 FileDescriptor 的输出流。

|--- FilterOutputStream：此类是过滤输出流的所有类的超类。

|--- BufferedOutputStream：该类实现缓冲的输出流。

|--- PrintStream：

|--- DataOutputStream：

|--- ObjectOutputStream：

|--- PipedOutputStream：

缓冲区是提高效率用的，给谁提高呢？

BufferedWriter(将流和缓冲区结合)：是给字符输出流提高效率用的，那就意味着，缓冲区对象建立时，必须要先有流对象。明确要提高具体的流对象的效率。

```
FileWriter fw = new FileWriter("bufdemo.txt");
BufferedWriter bufw = new BufferedWriter(fw);//让缓冲区和指定流相关联。
for(int x=0; x<4; x++){
    bufw.write(x+"abc");
    bufw.newLine(); //写入一个换行符，这个换行符可以依据平台的不同写入不同的换行符。
    bufw.flush();//对缓冲区进行刷新，可以让数据到目的地中。
}
bufw.close();//关闭缓冲区，其实就是在关闭具体的流。
```

BufferedReader：

```
FileReader fr = new FileReader("bufdemo.txt");
BufferedReader bufr  = new BufferedReader(fr);
String line = null;
while((line=bufr.readLine())!=null){  //readLine 方法返回的时候是不带换行符的。
    System.out.println(line);
}
bufr.close();
```

```
//记住，只要一读取键盘录入，就用这句话。
BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));
    //将读取到的从键盘输入的字节转化成字符存在流中，并将其放入缓冲区
BufferedWriter bufw = new BufferedWriter(new OutputStreamWriter(System.out));//输出到
```

控制台

```
String line = null;
while((line=bufr.readLine())!=null){
    if("over".equals(line))
        break;
    bufw.write(line.toUpperCase());//将输入的字符转成大写字符输出
    bufw.newLine();//换行
    bufw.flush();
    }
bufw.close();
bufr.close();
```

流对象：其实很简单，就是读取和写入。但是因为功能的不同，流的体系中提供 N 多的对象。那么开始时，到底该用哪个对象更为合适呢？这就需要明确流的操作规律。

流的操作规律：

1，明确源和目的。

数据源：就是需要读取，可以使用两个体系：InputStream、Reader；

数据汇：就是需要写入，可以使用两个体系：OutputStream、Writer；

2，操作的数据是否是纯文本数据？

如果是：数据源：Reader

数据汇：Writer

如果不是：数据源：InputStream

数据汇：OutputStream

3，虽然确定了一个体系，但是该体系中有太多的对象，到底用哪个呢？

明确操作的数据设备。

数据源对应的设备：硬盘(File)，内存(数组)，键盘(System.in)

数据汇对应的设备：硬盘(File)，内存(数组)，控制台(System.out)。

4，需要在基本操作上附加其他功能吗？比如缓冲。

如果需要就进行装饰。以提高效率！

转换流特有功能：转换流可以将字节转成字符，原因在于，将获取到的字节通过查编码表获取到指定对应字符。

转换流的最强功能就是基于 字节流 + 编码表 。没有转换，没有字符流。

发现转换流有一个子类就是操作文件的字符流对象：

InputStreamReader

|--FileReader

OutputStreamWriter

|--FileWrier

想要操作文本文件，必须要进行编码转换，而编码转换动作转换流都完成了。所以操作文件的流对象只要继承自转换流就可以读取一个字符了。

但是子类有一个局限性，就是子类中使用的编码是固定的，是本机默认的编码表，对于简体中文版的系统默认码表是 GBK。

FileReader fr = new FileReader("a.txt");
InputStreamReader isr = new InputStreamReader(new FileInputStream("a.txt"),"gbk");
以上两句代码功能一致，
如果仅仅使用平台默认码表，就使用 FileReader fr = new FileReader("a.txt"); //因为简化。

如果需要制定码表，必须用转换流。
转换流 = 字节流+编码表。
转换流的子类 File = 字节流 + 默认编码表。

凡是操作设备上的文本数据，涉及编码转换，必须使用转换流。

File 类：将文件系统中的文件和文件夹封装成了对象。提供了更多的属性和行为可以对这些文件和文件夹进行操作。这些是流对象办不到的，因为流只操作数据。

File 类常见方法：

1：创建。

boolean createNewFile()：在指定目录下创建文件，如果该文件已存在，则不创建。而对操作文件的输出流而言，输出流对象已建立，就会创建文件，如果文件已存在，会覆盖。除非续写。

boolean mkdir()：创建此抽象路径名指定的目录。

boolean mkdirs()：创建多级目录。

2：删除。

boolean delete()：删除此抽象路径名表示的文件或目录。

void deleteOnExit()：在虚拟机退出时删除。

注意：在删除文件夹时，必须保证这个文件夹中没有任何内容，才可以将该文件夹用 delete 删除。

window 的删除动作，是从里往外删。**注意：**java 删除文件不走回收站。要慎用。

3：获取.

long length()：获取文件大小。

String getName()：返回由此抽象路径名表示的文件或目录的名称。

String getPath()：将此抽象路径名转换为一个路径名字符串。

String getAbsolutePath()：返回此抽象路径名的绝对路径名字符串。

String getParent()：返回此抽象路径名父目录的抽象路径名，如果此路径名没有指定父目录，则返回 null。

long lastModified()：返回此抽象路径名表示的文件最后一次被修改的时间。

File.pathSeparator：返回当前系统默认的路径分隔符，windows 默认为 “;”。

File.Separator：返回当前系统默认的目录分隔符，windows 默认为 “\”。

4：判断：

boolean exists()：判断文件或者文件夹是否存在。

boolean isDirectory()：测试此抽象路径名表示的文件是否是一个目录。

boolean isFile()：测试此抽象路径名表示的文件是否是一个标准文件。

boolean isHidden()：测试此抽象路径名指定的文件是否是一个隐藏文件。

boolean isAbsolute()：测试此抽象路径名是否为绝对路径名。

5：重命名。

boolean renameTo(File dest)：**可以实现移动的效果。剪切+重命名。**

String[] list()：列出指定目录下的当前的文件和文件夹的名称。包含隐藏文件。

如果调用 list 方法的 File 对象中封装的是一个文件，那么 list 方法返回数组为 null。如果封装的对象不存在也会返回 null。**只有封装的对象存在并且是文件夹时，这个 list()方法才有效。**

递归：就是函数自身调用自身。

什么时候用递归呢？

当一个功能被重复使用，而每一次使用该功能时的参数不确定，都由上次的功能元素结果来确定。

简单说：**功能内部又用到该功能，但是传递的参数值不确定。**(每次功能参与运算的未知内容不确定)。

递归的注意事项：

1：一定要定义递归的条件。

2：递归的次数不要过多。容易出现 StackOverflowError **栈内存溢出错误。**

其实递归就是在栈内存中不断的加载同一个函数。

Java.util.**Properties：**一个可以将键值进行持久化存储的对象。Map--Hashtable 的子类。

Map

|--Hashtable

|--Properties：用于属性配置文件，键和值都是字符串类型。

特点：1：可以持久化存储数据。2：键值都是字符串。3：一般用于配置文件。

|-- load()：将流中的数据加载进集合。

原理：其实就是将读取流和指定文件相关联，并读取一行数据，因为数据是规则的 key=value，所以获取一行后，通过 = 对该行数据进行切割，左边就是键，右边就是值，将键、值存储到 properties 集合中。

|-- store()：写入各个项后，刷新输出流。

|-- list()：将集合的键值数据列出到指定的目的地。

以下介绍 IO 包中扩展功能的流对象：基本都是装饰设计模式。

Java.io.outputstream.**PrintStream：**打印流

1：提供了更多的功能，比如打印方法。可以直接打印任意类型的数据。

2：它有一个自动刷新机制，创建该对象，指定参数，对于指定方法可以自动刷新。

3：它使用的本机默认的字符编码.

4：该流的 print 方法不抛出 IOException。

该对象的构造函数。

PrintStream(File file) ：创建具有指定文件且不带自动行刷新的新打印流。

PrintStream(File file, String csn) ：创建具有指定文件名称和字符集且不带自动行刷新的新打印流。

PrintStream(OutputStream out) ：创建新的打印流。

PrintStream(OutputStream out, boolean autoFlush) ：创建新的打印流。

PrintStream(OutputStream out, boolean autoFlush, String encoding) ：创建新的打印流。

PrintStream(String fileName) ：创建具有指定文件名称且不带自动行刷新的新打印流。

PrintStream(String fileName, String csn)

PrintStream 可以操作目的：1：File 对象。2：字符串路径。3：字节输出流。
前两个都 JDK1.5 版本才出现。而且在操作文本文件时，可指定字符编码了。

当目的是一个字节输出流时，如果使用的 println 方法，可以在 printStream 对象上加入一个 true 参数。这样对于 println 方法可以进行自动的刷新，而不是等待缓冲区满了再刷新。最终 print 方法都将具体的数据转成字符串，而且都对 IO 异常进行了内部处理。

既然操作的数据都转成了字符串，那么使用 PrintWriter 更好一些。因为 PrintWrite 是字符流的子类，可以直接操作字符数据，同时也可以指定具体的编码。

PrintWriter：具备了 PrintStream 的特点同时，还有自身特点：
该对象的目的地有四个：1：File 对象。2：字符串路径。3：字节输出流。4：字符输出流。

开发时尽量使用 PrintWriter。

方法中直接操作文件的第二参数是编码表。
直接操作输出流的，第二参数是自动刷新。

```
//读取键盘录入将数据转成大写显示在控制台.
BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));//源：键盘输入
//目的：把数据写到文件中，还想自动刷新。
PrintWriter out = new PrintWriter(new FileWriter("out.txt"),true);//设置 true 后自动刷新
String line = null;
while((line=bufr.readLine())!=null){
    if("over".equals(line))
        break;
    out.println(line.toUpperCase());//转大写输出
}
    //注意：System.in，System.out 这两个标准的输入输出流，在 jvm 启动时已经存在了。随时可以使用。当 jvm 结束了，这两个流就结束了。但是，当使用了显示的 close 方法关闭时，这两个流在提前结束了。
out.close();
bufr.close();
```

SequenceInputStream：序列流，作用就是将多个读取流合并成一个读取流。实现数据合并。

表示其他输入流的逻辑串联。它从输入流的有序集合开始，并从第一个输入流开始读取，直到到达文件末尾，接着从第二个输入流读取，依次类推，直到到达包含的最后一个输入流的文件末尾为止。

这样做，可以更方便的操作多个读取流，其实这个序列流内部会有一个有序的集合容器，用于存储多个读取流对象。

该对象的构造函数参数是枚举，想要获取枚举，需要有 Vector 集合，但不高效。需用 ArrayList，但 ArrayList 中没有枚举，只有自己去创建枚举对象。

但是方法怎么实现呢？因为枚举操作的是具体集合中的元素，所以无法具体实现，但是枚举和

迭代器是功能一样的，所以，可以用迭代替代枚举。

合并原理：多个读取流对应一个输出流。三一
切割原理：一个读取流对应多个输出流。一三

```
import java.io.*;
import java.util.*;
class  SplitFileDemo{
   private static final String CFG = ".properties";
   private static final String SP = ".part";
   public static void main(String[] args) throws IOException{
      File file = new File("c:\\0.bmp");
      File dir = new File("c:\\partfiles");
      meger(dir);
   }
   //数据的合并。
   public static void meger(File dir)throws IOException{
      if(!(dir.exists() && dir.isDirectory()))
         throw new RuntimeException("指定的目录不存在，或者不是正确的目录");
      File[] files = dir.listFiles(new SuffixFilter(CFG));
      if(files.length==0)
         throw new RuntimeException("扩展名.proerpties 的文件不存在");
      //获取到配置文件
      File config = files[0];
      //获取配置文件的信息。
      Properties prop = new Properties();
      FileInputStream fis = new FileInputStream(config);
      prop.load(fis);
      String fileName = prop.getProperty("filename");
      int partcount = Integer.parseInt(prop.getProperty("partcount"));
      //--------------------------
      File[] partFiles = dir.listFiles(new SuffixFilter(SP));
      if(partFiles.length!=partcount)
         throw new RuntimeException("缺少碎片文件");
      //---------------------
      ArrayList<FileInputStream> al = new ArrayList<FileInputStream>();
      for(int x=0; x<partcount; x++){
         al.add(new FileInputStream(new File(dir,x+SP)));
      }
      Enumeration<FileInputStream> en = Collections.enumeration(al);
      SequenceInputStream sis = new SequenceInputStream(en);
      File file = new File(dir,fileName);
      FileOutputStream fos = new FileOutputStream(file);
      byte[] buf = new byte[1024];
      int len = 0;
      while((len=sis.read(buf))!=-1){
         fos.write(buf,0,len);
```

```
            }
            fos.close();
            sis.close();
        }
        //带有配置信息的数据切割。
        public static void splitFile(File file)throws IOException{
            //用一个读取流和文件关联。
            FileInputStream fis = new FileInputStream(file);
            //创建目的地。因为有多个。所以先创建引用。
            FileOutputStream fos = null;
            //指定碎片的位置。
            File dir = new File("c:\\partfiles");
            if(!dir.exists())
                dir.mkdir();
            //碎片文件大小引用。
            File f = null;
            byte[] buf = new byte[1024*1024];
            //因为切割完的文件通常都有规律的。为了简单标记规律使用计数器。
            int count = 0;
            int len = 0;
            while((len=fis.read(buf))!=-1){
                f = new File(dir,(count++)+".part");
                fos = new FileOutputStream(f);
                fos.write(buf,0,len);
                fos.close();
            }
            //碎片文件生成后，还需要定义配置文件记录生成的碎片文件个数。以及被切割文件的名称。
            //定义简单的键值信息，可是用 Properties。
            String filename = file.getName();
            Properties prop = new Properties();
            prop.setProperty("filename",filename);
            prop.setProperty("partcount",count+"");
            File config = new File(dir,count+".properties");
            fos = new FileOutputStream(config);
            prop.store(fos,"");
            fos.close();
            fis.close();
        }
    }
    class SuffixFilter implements FileFilter{
        private String suffix;
        SuffixFilter(String suffix){
            this.suffix  = suffix;
        }
        public boolean accept(File file){
            return  file.getName().endsWith(suffix);
```

```
        }
    }
```

RandomAccessFile:

特点：

1：**该对象即可读取，又可写入。**

2：该对象中的定义了一个大型的 byte 数组，通过定义指针来操作这个数组。

3：可以通过该对象的 getFilePointer()获取指针的位置，通过 seek()方法设置指针的位置。

4：该对象操作的源和目的必须是文件。

5：其实该对象内部封装了字节读取流和字节写入流。

注意：实现随机访问，最好是数据有规律。

```
class RandomAccessFileDemo{
    public static void main(String[] args) throws IOException{
        write();
        read();
        randomWrite();
    }
    //随机写入数据，可以实现已有数据的修改。
    public static void randomWrite()throws IOException{
        RandomAccessFile raf = new RandomAccessFile("random.txt","rw");
        raf.seek(8*4);
        System.out.println("pos :"+raf.getFilePointer());
        raf.write("王武".getBytes());
        raf.writeInt(102);
        raf.close();
    }
    public static void read()throws IOException{
        RandomAccessFile raf = new RandomAccessFile("random.txt","r");//只读模式。
        //指定指针的位置。
        raf.seek(8*1);//实现随机读取文件中的数据。注意：数据最好有规律。
        System.out.println("pos1 :"+raf.getFilePointer());
        byte[] buf = new byte[4];
        raf.read(buf);
        String name = new String(buf);
        int age = raf.readInt();
        System.out.println(name+"::"+age);
        System.out.println("pos2 :"+raf.getFilePointer());
        raf.close();
    }
    public static void write()throws IOException{
        //rw：当这个文件不存在，会创建该文件。当文件已存在，不会创建。所以不会像输出流一样覆盖。
        RandomAccessFile raf = new RandomAccessFile("random.txt","rw");//rw 读写模式
        //往文件中写入人的基本信息，姓名，年龄。
        raf.write("张三".getBytes());
```

```
        raf.writeInt(97);
        raf.close();
    }
}
```

管道流：管道读取流和管道写入流可以像管道一样对接上，管道读取流就可以读取管道写入流写入的数据。

注意：需要加入多线程技术，因为单线程，先执行 read，会发生死锁，因为 read 方法是阻塞式的，没有数据的 read 方法会让线程等待。

```
public static void main(String[] args) throws IOException{
    PipedInputStream pipin = new PipedInputStream();
    PipedOutputStream pipout = new PipedOutputStream();
    pipin.connect(pipout);
    new Thread(new Input(pipin)).start();
    new Thread(new Output(pipout)).start();
}
```

对象的序列化：目的：将一个具体的对象进行持久化，写入到硬盘上。

注意：静态数据不能被序列化，因为静态数据不在堆内存中，是存储在静态方法区中。

如何将非静态的数据不进行序列化？用 transient 关键字修饰此变量即可。

Serializable：用于启动对象的序列化功能，可以强制让指定类具备序列化功能，该接口中没有成员，这是一个标记接口。这个标记接口用于给序列化类提供 UID。这个 uid 是依据类中的成员的数字签名进行运行获取的。如果不需要自动获取一个 uid，可以在类中，手动指定一个名称为 serialVersionUID id 号。依据编译器的不同，或者对信息的高度敏感性。最好每一个序列化的类都进行手动显示的 UID 的指定。

```
import java.io.*;
class ObjectStreamDemo {
    public static void main(String[] args) throws Exception{
        writeObj();
        readObj();
    }
    public static void readObj()throws Exception{
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream("obj.txt"));
        Object obj = ois.readObject();//读取一个对象。
        System.out.println(obj.toString());
    }
    public static void writeObj()throws IOException{
        ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("obj.txt"));
        oos.writeObject(new Person("lisi",25)); //写入一个对象。
        oos.close();
    }
```

```
}
class Person implements Serializable{
    private static final long serialVersionUID = 42L;
    private transient String name;//用 transient 修饰后 name 将不会进行序列化
    public int age;
    Person(String name,int age){
        this.name = name;
        this.age = age;
    }
    public String toString(){
        return name+"::"+age;
    }
}
```

DataOutputStream、DataInputStream： 专门用于操作基本数据类型数据的对象。

```
    DataOutputStream dos =  new DataOutputStream(new FileOutputStream("data.txt"));
    dos.writeInt(256);
    dos.close();

    DataInputStream dis = new DataInputStream(new FileInputStream("data.txt"));
    int num = dis.readInt();
    System.out.println(num);
    dis.close();
```

ByteArrayInputStream：源：内存

ByteArrayOutputStream：目的：内存。

这两个流对象不涉及底层资源调用，操作的都是内存中数组，所以不需要关闭。

直接操作字节数组就可以了，为什么还要把数组封装到流对象中呢？因为数组本身没有方法，只有一个 length 属性。为了便于数组的操作，将数组进行封装，对外提供方法操作数组中的元素。

对于数组元素操作无非两种操作：设置（写）和获取（读），而这两操作正好对应流的读写操作。这两个对象就是使用了流的读写思想来操作数组。

```
    //创建源：
    ByteArrayInputStream bis = new ByteArrayInputStream("abcdef".getBytes());
    //创建目的：
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    int ch = 0;
    while((ch=bis.read())!=-1){
        bos.write(ch);
    }
    System.out.println(bos.toString());
```

网络编程：

端口：

物理端口：
逻辑端口：用于标识进程的逻辑地址，不同进程的标识；有效端口：0~65535，其中 0~1024 系统使用或保留端口。

java 中 ip 对象：InetAddress.

```
import java.net.*;
class  IPDemo{
    public static void main(String[] args) throws UnknownHostException{
        //通过名称(ip 字符串 or 主机名)来获取一个 ip 对象。
        InetAddress ip =
InetAddress.getByName("www.baidu.com");//java.net.UnknownHostException
        System.out.println("addr:"+ip.getHostAddress());
        System.out.println("name:"+ip.getHostName());
    }
}
```

Socket：★★★★，套接字，通信的端点。
就是为网络服务提供的一种机制，通信的两端都有 Socket，网络通信其实就是 Socket 间的通信，数据在两个 Socket 间通过 IO 传输。

UDP 传输：
1，只要是网络传输，必须有 socket 。
2，数据一定要封装到数据包中，数据包中包括目的地址、端口、数据等信息。

直接操作 udp 不可能，对于 java 语言应该将 udp 封装成对象，易于我们的使用，这个对象就是 DatagramSocket. 封装了 udp 传输协议的 socket 对象。

因为数据包中包含的信息较多，为了操作这些信息方便，也一样会将其封装成对象。这个数据包对象就是：DatagramPacket.通过这个对象中的方法，就可以获取到数据包中的各种信息。

DatagramSocket 具备发送和接受功能，在进行 udp 传输时，需要明确一个是发送端，一个是接收端。

udp 的发送端：
1，建立 udp 的 socket 服务，创建对象时如果没有明确端口，系统会自动分配一个未被使用的端口。
2，明确要发送的具体数据。
3，将数据封装成了数据包。
4，用 socket 服务的 send 方法将数据包发送出去。
5，关闭资源。

```
import java.net.*;
class  UdpSend{
    public static void main(String[] args)throws Exception {
//      1，建立 udp 的 socket 服务。
        DatagramSocket ds = new DatagramSocket(8888);//指定发送端口，不指定系统会随机分配。
//      2，明确要发送的具体数据。
```

```
        String text = "udp 传输演示 哥们来了";
        byte[] buf = text.getBytes();
//      3，将数据封装成了数据包。
        DatagramPacket dp = new DatagramPacket(buf,
                                buf.length,InetAddress.getByName("10.1.31.127"),10000);
//      4，用 socket 服务的 send 方法将数据包发送出去。
        ds.send(dp);
//      5，关闭资源。
        ds.close();
    }
}
```

---

udp 的接收端：

1，创建 udp 的 socket 服务，必须要明确一个端口，作用在于，只有发送到这个端口的数据才是这个接收端可以处理的数据。
2，定义数据包，用于存储接收到数据。
3，通过 socket 服务的接收方法将收到的数据存储到数据包中。
4，通过数据包的方法获取数据包中的具体数据内容，比如 ip、端口、数据等等。
5，关闭资源。

```
class UdpRece {
    public static void main(String[] args) throws Exception{
//      1，创建 udp 的 socket 服务。
        DatagramSocket ds = new DatagramSocket(10000);
//      2，定义数据包，用于存储接收到数据。先定义字节数组，数据包会把数据存储到字节数组中。
        byte[] buf = new byte[1024];
        DatagramPacket dp = new DatagramPacket(buf,buf.length);
//      3，通过 socket 服务的接收方法将收到的数据存储到数据包中。
        ds.receive(dp);//该方法是阻塞式方法。
//      4，通过数据包的方法获取数据包中的具体数据内容，比如 ip，端口，数据等等。
        String ip = dp.getAddress().getHostAddress();
        int port = dp.getPort();
        String text = new String(dp.getData(),0,dp.getLength());//将字节数组中的有效部分转成字符串。
        System.out.println(ip+":"+port+"--"+text);
//      5，关闭资源。
        ds.close();
    }
}
```

TCP 传输：两个端点的建立连接后会有一个传输数据的通道，这通道称为流，而且是建立在网络基础上的流，称之为 socket 流。该流中既有读取，也有写入。

tcp 的两个端点：一个是客户端，一个是服务端。
客户端：对应的对象，Socket
服务端：对应的对象，ServerSocket

TCP 客户端：

1，建立 tcp 的 socket 服务，最好明确具体的地址和端口。这个对象在创建时，就已经可以对指定 ip 和端口进行连接(三次握手)。

2，如果连接成功，就意味着通道建立了，socket 流就已经产生了。只要获取到 socket 流中的读取流和写入流即可，只要通过 getInputStream 和 getOutputStream 就可以获取两个流对象。

3，关闭资源。

```
import java.net.*;
import java.io.*;
//需求：客户端给服务器端发送一个数据。
class  TcpClient{
    public static void main(String[] args) throws Exception{
        Socket s = new Socket("10.1.31.69",10002);
        OutputStream out = s.getOutputStream();//获取了 socket 流中的输出流对象。
        out.write("tcp 演示，哥们又来了!".getBytes());
        s.close();
    }
}
```

TCP 服务端：

1，创建服务端 socket 服务，并监听一个端口。

2，服务端为了给客户端提供服务，获取客户端的内容，可以通过 accept 方法获取连接过来的客户端对象。

3，可以通过获取到的 socket 对象中的 socket 流和具体的客户端进行通讯。

4，如果通讯结束，关闭资源。注意：要先关客户端，再关服务端。

```
class  TcpServer{
    public static void main(String[] args) throws Exception{
        ServerSocket ss = new ServerSocket(10002);//建立服务端的 socket 服务
        Socket s = ss.accept();//获取客户端对象
        String ip = s.getInetAddress().getHostAddress();
        System.out.println(ip+".....connected");
//      可以通过获取到的 socket 对象中的 socket 流和具体的客户端进行通讯。
        InputStream in = s.getInputStream();//读取客户端的数据，使用客户端对象的 socket
读取流
        byte[] buf = new byte[1024];
        int len = in.read(buf);
        String text = new String(buf,0,len);
        System.out.println(text);
//      如果通讯结束，关闭资源。注意：要先关客户端，在关服务端。
        s.close();
        ss.close();
    }
}
```

反射技术：其实就是动态加载一个指定的类，并获取该类中的所有的内容。而且将字节码文件封

装成对象，并将字节码文件中的内容都封装成对象，这样便于操作这些成员。简单说：**反射技术可以对一个类进行解剖。**

反射的好处：大大的增强了程序的扩展性。

反射的基本步骤：

1、获得 Class 对象，就是获取到指定的名称的字节码文件对象。

2、实例化对象，获得类的属性、方法或构造函数。

3、访问属性、调用方法、调用构造函数创建对象。

获取这个 Class 对象，有三种方式：

1：通过每个对象都具备的方法 getClass 来获取。弊端：必须要创建该类对象，才可以调用 getClass 方法。

2：每一个数据类型(基本数据类型和引用数据类型)都有一个**静态的属性 class**。弊端：必须要先明确该类。

前两种方式不利于程序的扩展，因为都需要在程序使用具体的类来完成。

3：使用的 Class 类中的方法，**静态的 forName 方法**。

指定什么类名，就获取什么类字节码文件对象，这种方式的扩展性最强，只要将类名的字符串传入即可。

```
// 1. 根据给定的类名来获得   用于类加载
  String classname = "cn.itcast.reflect.Person";// 来自配置文件
  Class clazz = Class.forName(classname);// 此对象代表 Person.class
// 2. 如果拿到了对象，不知道是什么类型    用于获得对象的类型
  Object obj = new Person();
  Class clazz1 = obj.getClass();// 获得对象具体的类型
// 3. 如果是明确地获得某个类的 Class 对象   主要用于传参
  Class clazz2 = Person.class;
```

反射的用法：

1)、需要获得 java 类的各个组成部分，首先需要获得类的 Class 对象，获得 Class 对象的三种方式：

Class.forName(classname)　用于做类加载

obj.getClass()　　用于获得对象的类型

类名.class　　用于获得指定的类型，传参用

2)、反射类的成员方法：

```
Class clazz = Person.class;
Method method = clazz.getMethod(methodName, new Class[]{paramClazz1, paramClazz2});
method.invoke();
```

3)、反射类的构造函数：

```
Constructor con = clazz.getConstructor(new Class[]{paramClazz1, paramClazz2,...})
con.newInstance(params...)
```

4)、反射类的属性：

```
Field field = clazz.getField(fieldName);
field.setAccessible(true);
field.setObject(value);
```

获取了字节码文件对象后，最终都需要创建指定类的对象：

创建对象的两种方式(其实就是对象在进行实例化时的初始化方式)：

1，调用空参数的构造函数：使用了 Class 类中的 newInstance()方法。

2，调用带参数的构造函数：先要获取指定参数列表的构造函数对象，然后通过该构造函数的对象的 newInstance(实际参数) 进行对象的初始化。

综上所述，第二种方式，必须要先明确具体的构造函数的参数类型，不便于扩展。所以一般情况下，被反射的类，内部通常都会提供一个公有的空参数的构造函数。

```
// 如何生成获取到字节码文件对象的实例对象。
    Class clazz = Class.forName("cn.itcast.bean.Person");//类加载

     // 直接获得指定的类型

    clazz = Person.class;

    // 根据对象获得类型

    Object obj = new Person("zhangsan", 19);
    clazz = obj.getClass();


    Object obj = clazz.newInstance();//该实例化对象的方法调用就是指定类中的空参数构造函数，给创建对象进行初始化。当指定类中没有空参数构造函数时，该如何创建该类对象呢？请看 method_2();
public static void method_2() throws Exception {
    Class clazz = Class.forName("cn.itcast.bean.Person");
    //既然类中没有空参数的构造函数,那么只有获取指定参数的构造函数,用该函数来进行实例化。
    //获取一个带参数的构造器。
    Constructor constructor = clazz.getConstructor(String.class,int.class);
    //想要对对象进行初始化，使用构造器的方法 newInstance();
    Object obj = constructor.newInstance("zhagnsan",30);
    //获取所有构造器。
    Constructor[] constructors = clazz.getConstructors();//只包含公共的
    constructors = clazz.getDeclaredConstructors();//包含私有的
    for(Constructor con : constructors) {
        System.out.println(con);
    }
}
```

反射指定类中的方法：

```
//获取类中所有的方法。
public static void method_1() throws Exception {
    Class clazz = Class.forName("cn.itcast.bean.Person");
    Method[] methods = clazz.getMethods();//获取的是该类中的公有方法和父类中的公有方法。
    methods = clazz.getDeclaredMethods();//获取本类中的方法，包含私有方法。
    for(Method method : methods) {
        System.out.println(method);
```

```
        }
    }
    //获取指定方法；
    public static void method_2() throws Exception {
        Class clazz = Class.forName("cn.itcast.bean.Person");
        //获取指定名称的方法。
        Method method = clazz.getMethod("show", int.class,String.class);
        //想要运行指定方法，当然是方法对象最清楚，为了让方法运行，调用方法对象的 invoke 方法即可，但是方法运行必须要明确所属的对象和具体的实际参数。
        Object obj = clazz.newInstance();
        method.invoke(obj, 39,"hehehe");//执行一个方法
    }
    //想要运行私有方法。
    public static void method_3() throws Exception {
        Class clazz = Class.forName("cn.itcast.bean.Person");
        //想要获取私有方法。必须用 getDeclearMethod();
        Method method = clazz.getDeclaredMethod("method", null);
        // 私有方法不能直接访问，因为权限不够。非要访问，可以通过暴力的方式。
        method.setAccessible(true);//一般很少用，因为私有就是隐藏起来，所以尽量不要访问。
    }
    //反射静态方法。
    public static void method_4() throws Exception {
        Class clazz = Class.forName("cn.itcast.bean.Person");
        Method method = clazz.getMethod("function",null);
        method.invoke(null,null);
    }
```

正则表达式：★★★☆，其实是用来操作字符串的一些规则。

好处：正则的出现，对字符串的复杂操作变得更为简单。

特点：将对字符串操作的代码用一些符号来表示。只要使用了指定符号，就可以调用底层的代码对字符串进行操作。符号的出现，简化了代码的书写。

弊端：符号的出现虽然简化了书写，但是却降低了阅读性。

其实更多是用正则解决字符串操作的问题。

组：用小括号标示，每定义一个小括号，就是一个组，而且有自动编号，从 1 开始。

只要使用组，对应的数字就是使用该组的内容。别忘了，数组要加\\。

(aaa(wwww(ccc))(eee))技巧，从左括号开始数即可。有几个左括号就是几组。

常见操作：

1，匹配：其实用的就是 String 类中的 matches 方法。

```
String reg = "[1-9][0-9]{4,14}";
boolean b = qq.matches(reg);//将正则和字符串关联对字符串进行匹配。
```

2，切割：其实用的就是 String 类中的 split 方法。

3，替换：其实用的就是 String 类中的 replaceAll();

4，获取：

1），先要将正则表达式编译成正则对象。使用的是 Pattern 中静态方法 compile(regex);

2），通过 Pattern 对象获取 Matcher 对象。

Pattern 用于描述正则表达式，可以对正则表达式进行解析。

而将规则操作字符串，需要从新封装到匹配器对象 Matcher 中。

然后使用 Matcher 对象的方法来操作字符串。

如何获取匹配器对象呢？

通过 Pattern 对象中的 matcher 方法。该方法可以正则规则和字符串想关联。并返回匹配器对象。

3），使用 Matcher 对象中的方法即可对字符串进行各种正则操作。
