## java语言在调用函数时，只能传递值给函数，不能传递变量。

JRE(Java Runtime Enviroment)是Java的运行环境。

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

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)。

public static void main

//这是java程序的入口地址

public:表示的

static:表明方法是静态的,不依赖类的对象的,是属于类的,在类加载的时候main()方法也随着加载到内存中去

void:main()方法是不需要返回值的

main:约定俗成,规定的

ART+/

//自动推荐出未输入完整的公示

CTRL+/

//将所选段落前添加//变为注释

### 命名规则

变量命名只能使用字母 数字 $ \_

变量第一个字符 只能使用 字母 $ \_

变量第一个字符 不能使用数字

注：\_ 是下划线，不是-减号或者—— 破折号

System.***out***.println

//输出

JAVA只能用英文分号，不能用中文分号。

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

//固定语法，使代码能够接受输入

**int** num = in.nextInt();

//将输入的值转化为整数变量

**Final int** a = 100

//固定a=100不可更改

JAVA中，整数的运算结果只能是整数

System.***out***.println(1.2-1.1);结果为0.09999999999999987，浮点数是有误差的

单目和赋值运算规则是自右向左

System.***out***.println((**int**)(2.3+4.4));结果为6，对数据int的格式与python有区别

**int** a = (int)(100+0.1); 结果为100，对数据int的格式与python有区别

APPLICATION程序

要点：class是主体

Public类名与文件同名

Main（）写法固定

System.out.print及println及printf

Applet程序 #小应用程序，不能独立运行，要加网页代码

Import表示导入

Extends JApplet表示继承

有paint()方法，表示如何绘制

没有main（）方法

Math.abs(a-b) 函数a-b的绝对值

JAVA中，if后不用加引号：

switch(控制表达式){

case 常量:

语句

case 常量:

语句

default：

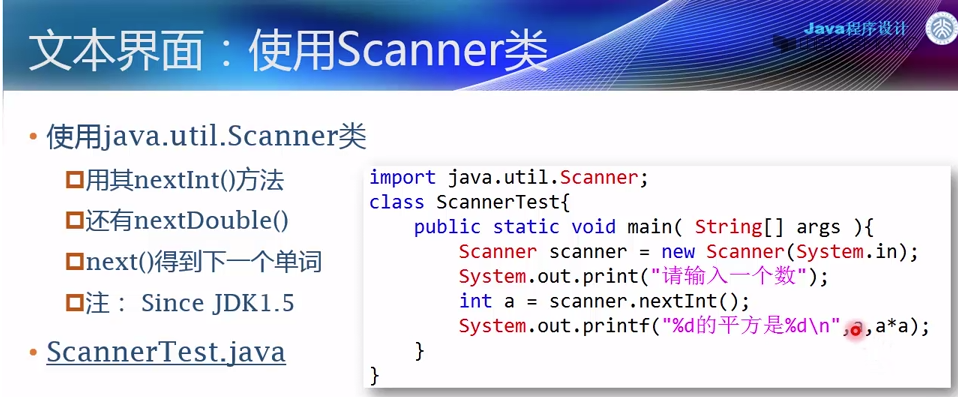
语句

}

//控制表达式只能是整数型结果，根据控制表达式决定的值决定从那个case开始执行，且若语句后没有break会继续执行。

应用程序(Java Application)的输入输出可以是文本界面，也可以是图形界面。

小程序(Java Applet）则只能是图形界面。



while()

}

//先确定是否循环再做

do{

}while();

//先做再确定是否循环

可在循环前添加标签和冒号，这个标签就指代了之后的循环。

例：OUT:

for(){

for(){

break OUT;

}

}

可直接跳出OUT指代的大循循环

String //定义字符串

数组定义

<类型>[]<名字>=new <类型>[元素个数]

例：double[] name = new double[20];

数组变量是数组的管理者不是拥有者，当改变数组时，所有数组变量对应的值也改变

Last time:11月6号

For-each 循环

例：for （int k : [数组]） //将数组中每一个值单独拿出来作为k循环

只适用于知道数组中是否存在该值，不知道该值位置

**Boolean初始值默认为false**

JAVA的二维数组格式为a[i][j],不同于Python的a[i,j]

创建二维数组时可直接用如：

Int[][] a = {

{1,2,3,4},

{1,2,3},

};

的方式。

说明：编译器来数数

每行一个{}，逗号分隔

最后的逗号可以存在，有古老的传统

如果省略，表示补零

### 数组

排序：**选择法排序**：先遍历数组，**只和最前面的数比较**，将最小的放在最前，再遍历数组，将第二小的放在第二位。这样双重for循环排序.

**冒泡排序法**：遍历数组，**将相邻两个进行比较**，大的会一直往后，再遍历数组，第二大的一直往后。这样双重for循环

**二维数组排序**：Arrays.sort(int[][], (a,b) -> (a[0]-b[0]));//a，b为二维数组的行，按每行的第一列进行排序

**数组复制：**System.arraycopy(src, srcPos, dest, destPos, length)

        //src: 源数组

        //srcPos: 从源数组复制数据的起始位置

        //dest: 目标数组

        //destPos: 复制到目标数组的启始位置

        //length: 复制的长度

数组方法：

copyOfRange 数组复制 int[] b = Arrays.copyOfRange(a, 0, 3);// 第一个参数表示源数组，第二个参数表示开始位置(取得到)， 第三个参数表示结束位置(取不到)

toString() 转换为字符串 Arrays.toString(a);

sort 排序 Arrays.sort(a);

binarySearch 搜索 Arrays.binarySearch(a, 62)；//数字62在a中出现的下标

注：使用binarySearch之前，必须先使用sort进行排序

equals 判断是否相同 Arrays.equals(a, b)

fill 填充 使用同一个值，填充整个数组 Arrays.fill(a, 5);

### 数字与字符串

JAVA使用的Unicode字符，能支持包括汉字在内的多种语言

单个字符用char定义，小写字母顺序码在大写字母后面

逃逸字符（不能印出来的控制字符或特殊字符）

|  |  |  |
| --- | --- | --- |
| 基本类型 | 包裹类型 | bit |
| boolean | Boolean | 1 |
| char | Character | 16 |
| byte | Byte | 8 |
| short | Short | 16 |
| int | Integer | 32 |
| long | Long | 32 |
| float | Float | 32 |
| double | Double | 64 |

int i = 5;

**//基本类型转换成封装类型**

**Integer it = new Integer(i);**

//转为封装类型即将数据转化为对象，可对对象直接进行一些数据不能进行的操作处理，

int i = 5;

Integer it = new Integer(i);

**//封装类型转换成基本类型**

**int i2 = it.intValue();**

不需要调用构造方法，通过=符号自动把 基本类型 转换为 类类型 就叫装箱；不需要调用Integer的intValue方法，通过=就自动转换成int类型，就叫拆箱

**//自动转换就叫拆箱**

**int i3 = it;**

int size = Integer.SIZE; // 获取 int 类型的二进制位（32）

long val = 26L; //以L结尾的字面值表示long型

int decVal = 26; //默认就是int型

int hexVal = 0x1a; //16进制

int oxVal = 032; //8进制

int binVal = 0b11010; //2进制

int i= Integer.parseInt(str);

final 修饰的变量在方法中，可以先初始化再赋值。

但是如果是全局变量，必须在初始化的时赋值，不然会报错。

#### 字符串

**Java的字符串还是一种特殊的“不可变”对象，所有的字符串操作都是产生一个新的字符串，而不是对原来的字符串的修改。对这一点的理解颇为重要。**

字符串String（首字母大写）只是对象的调用者，不能修改对象（字符串），不能被继承。用+加号进行字符串拼接会创建新的字符串对象。

输入字符串：in.next();//读取一个单词，单词的标志是空格（回车，tab,space都算空格）

in.nextLine();//读取一整行

比较字符串是否相等不能用‘==’（此时只是比较两方是否为同一对象），而应该用（String.equals(“对应字符串”）来比较。

对字符串的操作都是通过方法使用的，如比较大小的方法（只能比较字符串编码位之和的大小）Sring1.compareTo(Sring2);结果大于则输出1反之输出-1，相等输出0。

字符串的方法

String str = String.valueOf(i);//将数字转换为字符串

charAt 获取字符 str1.charAt（index） 输出index位上的字符

toCharArray 获取对应的字符数组

subString 截取子字符串 str1. subString（n）//得到从n号位置到末尾的全部内容

str1. subString（b,e）//得到从b号位置到e号位置之前的内容

split 分隔 sentence.split(",");//根据“，”进行分隔

trim 去掉首尾空格 sentence.trim()；

toLowerCase 小写

toUpperCase 大写

indexOf sentence.indexOf('8')// 字符第一次出现的位置 sentence.indexOf("88")//字符串第一次出现的位置 sentence.indexOf(',',5)// 从位置5开始，出现的第一次,的位置

lastIndexOf sentence.lastIndexOf("88")//字符串最后出现的位置

contains 定位 sentence.contains("击杀")//是否包含字符串"击杀"

replaceAll sentence.replaceAll("击杀", "被击杀");//替换所有的击杀为被击杀

replaceFirst 替换第一个

str1.equals(str2);//完全一样返回true

str1.equalsIgnoreCase(str3);//忽略大小写的比较，返回true|false

str1.startsWith(str2);//是否以str2 ..开始

str1.endsWith(str3);//是否以str3.. 结束

**StringBuffer是可变长的字符串**

和String内部是一个字符数组一样，StringBuffer也维护了一个字符数组。 但是，这个字符数组，留有冗余长度，长度不够时自动多分配一个数组

创建：StringBuffer sb = new StringBuffer(str1); //根据str1创建一个StringBuffer对象

方法：append追加

delete 删除 sb.delete(4, 10);//删除下标4-10之间的字符

insert 插入 sb.insert(4, "there ");//在4这个位置插入 there

reverse 反转 sb.reverse(); //反转

属性：sb.length(); //内容长度

sb.capacity();//分配的总空间

i++; 先取值输出，再运算

++i; 先运算，再取值输出

### 接口与继承

接口（interface）：类实现了接口就必须使用接口中的方法，一个类可实现多个接口,但一个类只能继承一个类。

接口只能继承接口，不能实现，继承接口后可以增加新的方法

同包不同类之间调用方法直接类.方法()

Java值传递：当变量作为参数传递到方法内部时，过程执行的是拷贝，不论方法对形参如何修改，都不会影响实参数据。当变量为类类型时传递，实质传递的是该引用的地址，内部对形参修改影响的只是引用的地址新指向，除非修改对象.属性值，才能真正影响外部实参。

使用private修饰属性  
自身：是可以访问的  
同包子类：不能继承  
不同包子类：不能继承  
同包类：不能访问  
其他包类：不能访问



复制数组

System.arraycopy(src, srcPos, dest, destPos, length)

src: 源数组  
srcPos: 从源数组复制数据的起始位置  
dest: 目标数组  
destPos: 复制到目标数组的起始位置  
length: 复制的长度

枚举enum是一种特殊的类(还是类)，使用枚举可以很方便的定义常量

比如设计一个枚举类型 季节，里面有4种常量

继承：

Hero h = new Hero()；

Int a = 12;

注释：h 为Hero类型，a为Interger类型；等式右边创建了一个Hero对象和12整数对象；h,a都只是引用对象，即记录的它们的内存地址。当h,a被声明时它们的类型就固定了。

**抽象类：**

类中声明一个方法，这个方法没有实现体，是一个“空”方法

这样的方法就叫抽象方法，使用修饰符“abstract"

当一个类有抽象方法的时候，该类必须被声明为**抽象类**

**内部类分为四种：**

**非静态内部类**

在一个类里面声明一个非静态内部类

语法: new 外部类().new 内部类()

作为非静态内部类，是可以直接访问外部类的private实例属性的

**静态内部类**

在一个类里面声明一个静态内部类

与非静态内部类不同，静态内部类的实例化 不需要一个外部类的实例为基础，可以直接实例化

语法：new 外部类.静态内部类();

因为没有一个外部类的实例，所以在静态内部类里面不可以访问外部类的实例属性和方法

除了可以访问外部类的私有静态成员外，静态内部类和普通类没什么大的区别

**匿名类**

匿名类指的是在声明一个类的同时实例化它，使代码更加简洁精练

通常情况下，要使用一个接口或者抽象类，都必须创建一个子类

有的时候，为了快速使用，直接实例化一个抽象类，并“当场”实现其抽象方法。

既然实现了抽象方法，那么就是一个新的类，只是这个类，没有命名。

这样的类，叫做匿名类

在匿名类中使用外部的局部变量，外部的局部变量必须修饰为final。

在jdk8中，已经不需要强制修饰成final了

**本地类**

语法: new 外部类().new 内部类()

作为Hero的非静态内部类，是可以直接访问外部类的private实例属性name的

与非静态内部类不同，静态内部类水晶类的实例化 不需要一个外部类的实例为基础，可以直接实例化

语法：new 外部类.静态内部类();

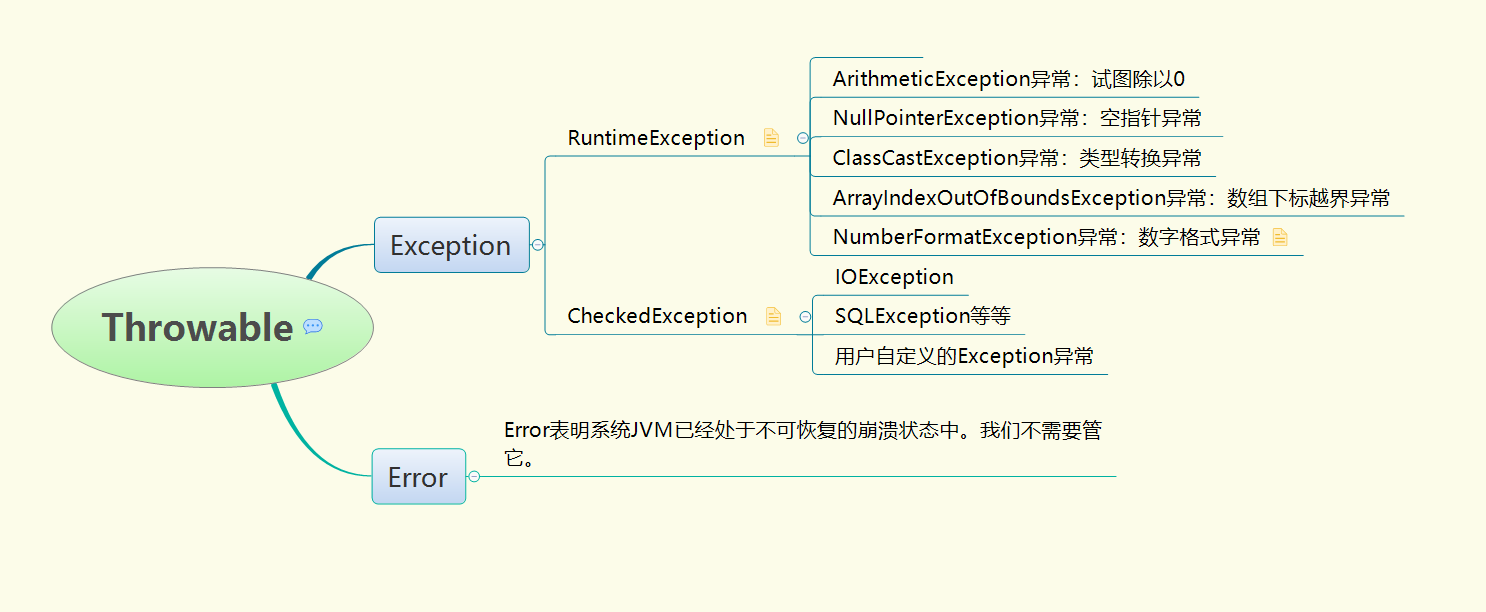
因为没有一个外部类的实例，所以在静态内部类里面不可以访问外部类的实例属性和方法

除了可以访问外部类的私有静态成员外，静态内部类和普通类没什么大的区别

# JAVA中级

# exception：

**文件不存在异常FileNotFoundException**



e.printStackTrace(); 会打印出方法的调用痕迹，如此例，会打印出异常开始于TestException的第16行，这样就便于定位和分析到底哪里出了异常

FileNotFoundException是Exception的子类，使用Exception也可以catch住FileNotFoundException

无论是否出现异常，finally中的代码都会被执行

throws可将异常抛给后面处理

private static void method2() throws FileNotFoundException {}

throws与throw这两个关键字接近，不过意义不一样，有如下区别：  
1. throws 出现在方法声明上，而throw通常都出现在方法体内。  
2. throws 表示出现异常的一种可能性，并不一定会发生这些异常；throw则是抛出了异常，执行throw则一定抛出了某个异常对象。

## 异常分类

**可查异常**： CheckedException  
可查异常即**必须进行处理的异常**，要么try catch住,要么往外抛，谁调用，谁处理，比如 FileNotFoundException  
如果不处理，编译器，就不让你通过

**运行时异常RuntimeException指：** **不是必须进行try catch的异常**  
**常见运行时异常:**  
除数不能为0异常:ArithmeticException  
下标越界异常:ArrayIndexOutOfBoundsException  
空指针异常:NullPointerException

与可查异常不同之处在于，**即便不进行try catch，也不会有编译错误**

**错误Error**，指的是**系统级别的异常**，通常是内存用光了  
在**默认设置下**，一般java程序启动的时候，最大可以使用16m的内存

ArithmeticException——由于除数为0引起的异常  
 ArrayStoreException——由于数组存储空间不够引起的异常  
 ClassCastException—一当把一个对象归为某个类，但实际上此对象并不是由这个类 创建的，也不是其子类创建的，则会引起异常  
 IllegalMonitorStateException——监控器状态出错引起的异常  
 NegativeArraySizeException—一数组长度是负数，则产生异常  
 NullPointerException—一程序试图访问一个空的数组中的元素或访问空的对象中的 方法或变量时产生异常  
 OutofMemoryException——用new语句创建对象时，如系统无法为其分配内存空 间则产生异常  
 SecurityException——由于访问了不应访问的指针，使安全性出问题而引起异常  
 IndexOutOfBoundsExcention——由于数组下标越界或字符串访问越界引起异常  
 IOException——由于文件未找到、未打开或者I/O操作不能进行而引起异常  
 ClassNotFoundException——未找到指定名字的类或接口引起异常  
 CloneNotSupportedException——一程序中的一个对象引用Object类的clone方法，但 此对象并没有连接Cloneable接口，从而引起异常  
 InterruptedException—一当一个线程处于等待状态时，另一个线程中断此线程，从 而引起异常，有关线程的内容，将在下一章讲述  
 NoSuchMethodException一所调用的方法未找到，引起异常  
 Illega1AccessExcePtion—一试图访问一个非public方法  
 StringIndexOutOfBoundsException——访问字符串序号越界，引起异常  
 ArrayIdexOutOfBoundsException—一访问数组元素下标越界，引起异常  
 NumberFormatException——字符的UTF代码数据格式有错引起异常  
 IllegalThreadException—一线程调用某个方法而所处状态不适当，引起异常  
 FileNotFoundException——未找到指定文件引起异常  
 EOFException——未完成输入操作即遇文件结束引起异常。

## **I/O流**

把流定义在try()里,try,catch或者finally结束的时候，会自动关闭

这种编写代码的方式叫做 try-with-resources， 这是从JDK7开始支持的技术

## 文件常用方法：

绝对路径：

File f = **new** File("d:/LOLFolder/LOL.exe");

// 相对路径,相对于工作目录，如果在eclipse中，就是项目目录

  File f2 = **new** File("LOL.exe");

// 把f1作为父目录创建文件对象

        File f3 = **new** File(f1, "LOL.exe");

"判断是否存在："+f.exists()

"判断是否是文件夹："+f.isDirectory()

"判断是否是文件："+f.isFile()

"获取文件的长度："+f.length()

 //文件绝对路径获取

f.getAbsolutePath();

 //文件最后修改时间

**long** time = f.lastModified();

        Date d = **new** Date(time);

        System.out.println("获取文件的最后修改时间："+d);

        //设置文件修改时间为1970.1.1 08:00:00

        f.setLastModified(0);

//文件重命名

        File f2 =**new** File("d:/LOLFolder/DOTA.exe");

        f.renameTo(f2);

// 以字符串数组的形式，返回当前文件夹下的所有文件（不包含子文件及子文件夹）

        f.list();

// 以文件数组的形式，返回当前文件夹下的所有文件（不包含子文件及子文件夹）

        File[]fs= f.listFiles();

        // 以字符串形式返回获取所在文件夹

        f.getParent();

        // 以文件形式返回获取所在文件夹

        f.getParentFile();

        // 创建文件夹，如果父文件夹skin不存在，创建就无效

        f.mkdir();

        // 创建文件夹，如果父文件夹skin不存在，就会创建父文件夹

        f.mkdirs();

        // 创建一个空文件,如果父文件夹skin不存在，就会抛出异常

        f.createNewFile();

        // 所以创建一个空文件之前，通常都会创建父目录

        f.getParentFile().mkdirs();

        // 列出所有的盘符c: d: e: 等等

        f.listRoots();

        // 刪除文件

        f.delete();

        // JVM结束的时候，刪除文件，常用于临时文件的删除

        f.deleteOnExit();

## 编码方式

ISO-8859-1 包含 ASCII  
GB2312 是简体中文，BIG5是繁体中文，GBK同时包含简体和繁体以及日文。  
UNICODE 包括了所有的文字，无论中文，英文，藏文，法文，世界所有的文字都包含其中

UNICODE的**各种减肥子编码**：UTF-8对数字和字母就使用一个字节，而对汉字就使用3个字节

用FileInputStream 字节流正确读取中文

为了能够正确的读取中文内容  
1. 必须了解文本是以哪种编码方式保存字符的  
2. 使用字节流读取了文本后，再使用对应的**编码方式去识别这些数字**，得到正确的字符  
如本例，一个文件中的内容是字符**中**，编码方式是GBK，那么读出来的数据一定是D6D0。  
再使用GBK编码方式识别D6D0，就能正确的得到字符**中**

以字节为单位进行编码，所以用字符读取时FileReader是不能手动设置编码方式的，为了使用其他的编码方式，只能使用InputStreamReader来代替，像这样：

new InputStreamReader(new FileInputStream(f),Charset.forName("UTF-8"));

## 字节流

InputStream是字节输入流，同时也是抽象类，只提供方法声明，不提供方法的具体实现。

FileInputStream 是InputStream子类，以FileInputStream 为例进行文件读取

FileInputStream fis =**new** FileInputStream(f);

fis.read(all);//all为字节组

fis.close();

OutputStream是字节输出流，同时也是抽象类，只提供方法声明，不提供方法的具体实现。

FileOutputStream 是OutputStream子类，以FileOutputStream 为例向文件写出数据

FileOutputStream fos = **new** FileOutputStream(f);

fos.write(data);

## 字符流

FileReader 是Reader子类，以FileReader 为例进行文件读取

FileReader fr = **new** FileReader(f)；

fr.read(all);//all为字符组

FileWriter 是Writer的子类，以FileWriter 为例把字符串写入到文件

FileWriter fr = **new** FileWriter(f)；

fr.write(cs);

## 缓存流

以介质是硬盘为例，字节流和字符流的弊端：

在每一次读写的时候，都会访问硬盘。 如果读写的频率比较高的时候，其性能表现不佳。

缓存流在读取的时候，会一次性读较多的数据到缓存中，以后每一次的读取，都是在缓存中访问，直到缓存中的数据读取完毕，再到硬盘中读取。

缓存流在写入数据的时候，会先把数据写入到缓存区，直到缓存区达到一定的量，才把这些数据，一起写入到硬盘中去。按照这种操作模式，就不会像字节流，字符流那样每写一个字节都访问硬盘，从而减少了IO操作

**缓存字符输入流 BufferedReader 可以一次读取一行数据**

 // 缓存流必须建立在一个存在的流的基础上

FileReader fr = **new** FileReader(f);

BufferedReader br = **new** BufferedReader(fr);

// 一次读一行

 String line = br.readLine();

**PrintWriter 缓存字符输出流， 可以一次写出一行数据**

FileWriter fw = **new** FileWriter(f);

PrintWriter pw = **new** PrintWriter(fw);

pw.println("xxxxxx");

有的时候，需要立即把数据写入到硬盘，而不是等缓存满了才写出去。 这时候就需要用到**flush**

pw.println("xxxxxx");

//强制把缓存中的数据写入硬盘，无论缓存是否已满

pw.flush();

## 数据流

使用数据流的writeUTF()和readUTF() 可以进行数据的格式化顺序读写

用DataInputStream 读取一个文件，这个文件必须是由DataOutputStream 写出的，否则会出现EOFException，因为DataOutputStream 在写出的时候会做一些特殊标记，只有DataInputStream 才能成功的读取。

FileOutputStream fos  = **new** FileOutputStream(f);

DataOutputStream dos =**new** DataOutputStream(fos);

dos.writeBoolean(**true**);

dos.writeInt(300);

dos.writeUTF("123 this is gareen");

FileInputStream fis  = **new** FileInputStream(f);

 DataInputStream dis =**new** DataInputStream(fis)

**boolean** b= dis.readBoolean();

**int** i = dis.readInt();

String str = dis.readUTF();

## 对象流

对象流指的是可以直接把一个对象以流的形式传输给其他的介质，比如硬盘

一个对象以流的形式进行传输，叫做序列化。

该对象所对应的类，必须是实现Serializable接口

//先把Hero类实现Serializable接口

**import** java.io.Serializable;

**public** **class** Hero **implements** Serializable{....}

 //创建对象输出流

FileOutputStream fos = **new** FileOutputStream(f);

ObjectOutputStream oos =**new** ObjectOutputStream(fos);

//创建对象输入流

FileInputStream fis = **new** FileInputStream(f);

ObjectInputStream ois =**new** ObjectInputStream(fis);

oos.writeObject(h);

 Hero h2 = (Hero) ois.readObject();

## System.in

System.out 是常用的在控制台输出数据的

System.in 可以从控制台输入数据

InputStream is = System.in;

使用System.in.read虽然可以读取数据，但是很不方便

使用Scanner就可以逐行读取了

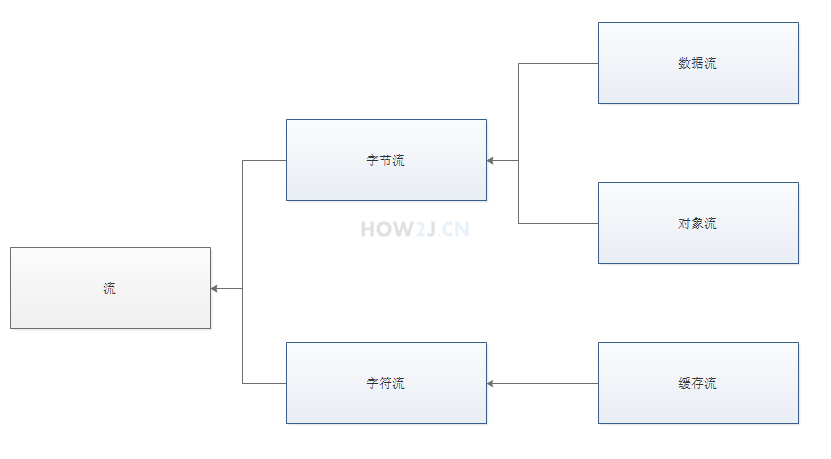
Scanner s = **new** Scanner(System.in);

**while**(**true**){

                String line = s.nextLine();

                System.out.println(line);

            }



InputStream下面有很多的子类。 这些子类不需要立即掌握，他们大体上用法是差不多的，只是在一些特殊场合下用起来更方便，在工作中用到的时候再进行学习就行了。

# 集合框架

## ArrayList

为了解决数组的局限性，引入容器类的概念。 最常见的容器类就是

ArrayList.容器的容量"capacity"会随着对象的增加，自动增长,不用担心会出现数组的边界问题.

**import** java.util.ArrayList;

ArrayList heros = **new** ArrayList();

方法

1. 增加

heros.add(new Hero("hero " + i));//第一种是直接add对象，把对象加在最后面

heros.add(3, specialHero);//第二种是在指定位置加对象

2.通过方法contains 判断一个对象是否在容器中

其原理是在内部进行for循环

判断标准： 是否是同一个对象，而不是name是否相同

heros.contains(specialHero)//返回布尔值

3.通过get获取指定位置的对象，如果输入的下标越界，一样会报错

heros.get(5)

4.**定位** indexOf用于判断一个对象在ArrayList中所处的位置

与contains一样，判断标准是对象是否相同，而非对象的name值是否相等

heros.indexOf(specialHero)

5.**删除** remove用于把对象从ArrayList中删除

remove可以根据下标删除n’dList的元素

heros.remove(2);

也可以根据对象删除

heros.remove(specialHero);

当对象是整数型时注意区别

时间复杂度O(n)

1. **替换** set用于替换指定位置的元素

heros.set(5, **new** Hero("hero 5"));

1. **获取大小 size 用于获取ArrayList的大小**

heros.size()//返回int 类似于length

1. **转换数组** toArray可以把一个ArrayList对象转换为数组。

需要注意的是，如果要转换为一个Hero数组，那么需要传递一个Hero数组类型的对象给toArray()，这样toArray方法才知道，你希望转换为哪种类型的数组，否则只能转换为Object数组

Hero hs[] = (Hero[])heros.toArray(**new** Hero[]{});

1. addAll 把另一个容器所有对象都加进来

ArrayList anotherHeros = **new** ArrayList();

heros.addAll(anotherHeros);

1. clear  清空一个ArrayList

heros.clear();

实现List接口

**import** java.util.List;

List heros = **new** ArrayList();

List的方法都可以在ArrayList中实现

## 泛型 Generic

不指定泛型的容器，可以存放任何类型的元素

指定了泛型的容器，只能存放指定类型的元素以及其子类

//声明容器的时候，就指定了这种容器，只能放Hero及其子类，放其他的就会出错，

List<Hero> genericheros = **new** ArrayList<Hero>();

//可简写为

List<Hero> genericheros2 = new ArrayList<>();

## 遍历

Iterator<Hero> it= heros.iterator();

        //从最开始的位置判断"下一个"位置是否有数据

        //如果有就通过next取出来，并且把指针向下移动

        //直到"下一个"位置没有数据

**while**(it.hasNext()){

            Hero h = it.next();

            System.out.println(h);

        }

//迭代器的for写法

        System.out.println("--------使用for的iterator-------");

**for** (Iterator<Hero> iterator = heros.iterator(); iterator.hasNext();) {

            Hero hero = (Hero) iterator.next();

            System.out.println(hero);

        }

序列分先进先出FIFO,先进后出FILO

FIFO在Java中又叫Queue 队列

FILO在Java中又叫Stack 栈

### LinkedList

与ArrayList一样，LinkedList也实现了List接口

除了实现了List接口外，LinkedList还实现了双向链表结构Deque，可以很方便的在头尾插入删除数据。

什么是链表结构: 与数组结构相比较，数组结构，就好像是电影院，每个位置都有标示，每个位置之间的间隔都是一样的。 而链表就相当于佛珠，每个珠子，只连接前一个和后一个，不用关心除此之外的其他佛珠在哪里。

LinkedList<Hero> ll =**new** LinkedList<Hero>();

 //在最后插入新的英雄

ll.addLast(**new** Hero("hero1"));

//在最前面插入新的英雄

ll.addFirst(**new** Hero("heroX"));

//查看最前面的英雄

ll.getFirst();

//取出最前面的英雄

//取出会导致英雄被删除

ll.removeFirst();

LinkedList 除了实现了List和Deque外，还实现了Queue接口(队列)。

Queue是先进先出队列 FIFO，常用方法：

offer 在最后添加元素

poll 取出第一个元素

peek 查看第一个元素

**import** java.util.Queue;

Queue<Hero> q= **new** LinkedList<Hero>();

q.offer(**new** Hero("Hero1"));

//取出第一个Hero，FIFO 先进先出

Hero h = q.poll();

//把第一个拿出来看一看，但是不取出来

h=q.peek();

FILO先入后出称为栈Stack

根据接口Stack ：实现类：MyStack

public class MyStack implements Stack

栈的结构，就像给弹夹添加子弹一样，先添加的子弹，就放在了最下面，打手枪的时候，只能从最上面取子弹。

### 二叉树

二叉树由各种节点组成

二叉树特点：

每个节点都可以有左子节点，右子节点，每一个节点都有一个值

package collection;

public class Node {

// 左子节点

public Node leftNode;

// 右子节点

public Node rightNode;

// 值

public Object value;

// 插入 数据

public void add(Object v) {

// 如果当前节点没有值，就把数据放在当前节点上

if (null == value)

value = v;

// 如果当前节点有值，就进行判断，新增的值与当前值的大小关系

else {

// 新增的值，比当前值小或者相同

if ((Integer) v -((Integer)value) <= 0) {

if (null == leftNode)

leftNode = new Node();

leftNode.add(v);

}

// 新增的值，比当前值大

else {

if (null == rightNode)

rightNode = new Node();

rightNode.add(v);

}

}

}

public static void main(String[] args) {

int randoms[] = new int[] { 67, 7, 30, 73, 10, 0, 78, 81, 10, 74 };

Node roots = new Node();

for (int number : randoms) {

roots.add(number);

}

}

}

// 中序遍历所有的节点

**public** List<Object> values() {

        List<Object> values = **new** ArrayList<>();

        // 左节点的遍历结果

**if** (**null** != leftNode)

            values.addAll(leftNode.values());

        // 当前节点

        values.add(value);

        // 右节点的遍历结果

**if** (**null** != rightNode)

            values.addAll(rightNode.values());

**return** values;

    }

## Hashmap

HashMap储存数据的方式是—— 键值对

对于HashMap而言，key是唯一的，不可以重复的。

所以，以相同的key 把不同的value插入到 Map中会导致旧元素被覆盖，只留下最后插入的元素。

不过，同一个对象可以作为值插入到map中，只要对应的key不一样

**import** java.util.HashMap;

HashMap<String,Hero> heroMap = **new** HashMap<String,Hero>();

//清空map

heroMap.clear();

 //同一个对象可以作为值插入到map中，只要对应的key不一样

heroMap.put("hero1", gareen);

heroMap.put("hero2", gareen);

 System.out.println(heroMap);

## HASHSET

Set中的元素，不能重复

HashSet没有自身的实现，而是里面封装了一个HashMap

**import** java.util.HashSet;

HashSet<String> names = **new** HashSet<String>();

names.add("gareen");

Set中的元素，没有顺序。

严格的说，是没有按照元素的插入顺序排列

HashSet的具体顺序，既不是按照插入顺序，也不是按照hashcode的顺序

不保证Set的迭代顺序; 确切的说，在不同条件下，元素的顺序都有可能不一样

换句话说，同样是插入0-9到HashSet中， 在JVM的不同版本中，看到的顺序都是不一样的。 所以在开发的时候，不能依赖于某种臆测的顺序，这个顺序本身是不稳定的

Set不提供get()来获取指定位置的元素

所以遍历需要用到迭代器，或者增强型for循环

### Collection

Collection是 Set，List，Queue，和Deque的接口

Queue: 先进先出队列

Deque: 双向链表

注：Collection和Map之间没有关系，Collection是放一个一个对象的，Map 是放键值对的

注：Deque 继承 Queue,间接的继承了 Collection

### Collections

Collections是一个类，容器的工具类,就如同Arrays是数组的工具类

reverse 使List中的数据发生翻转

Collections.reverse(List);

shuffle 混淆List中数据的顺序

Collections.shuffle(List);

sort 对List中的数据进行排序

 Collections.sort(List);

swap 交换两个数据的位置

 Collections.swap(List,0,5);//交换下标位置0和5的数据

rotate 把List中的数据，向右滚动指定单位的长度

Collections.rotate(List,2);//把集合向右滚动2个单位

synchronizedList 把非线程安全的List转换为线程安全的List。

List<Integer>synchronizedNumbers=(List<Integer>)Collections.synchronizedList(numbers);

ArrayList 插入，删除数据慢

LinkedList， 插入，删除数据快

ArrayList是顺序结构，所以定位很快，指哪找哪。

LinkedList 是链表结构，必须得一个一个的数过去，所以定位慢.

## HashMap和Hashtable都实现了Map接口，都是键值对保存数据的方式

区别1：

HashMap可以存放 null

Hashtable不能存放null

区别2：

HashMap不是线程安全的类

Hashtable是线程安全的类

HashSet： 无序

LinkedHashSet： 按照插入顺序

TreeSet： 从小到大排序

Hashcode原理

所有的对象，都有一个特殊算法计算出对应的hashcode（散列值）

如果两个对象有相同的散列值，则在该处创建一个链表，并先后存入。

查找时根据算法计算出散列值，再根据散列值下标到数组中定位查找（根据数组下标进行定位，是非常快速的）。若该处有链表则再用equals逐一比较

这是一种用空间换时间的思维方式

下是Java API提供的String的hashcode生成办法；

s[0]\*31^(n-1) + s[1]\*31^(n-2) + ... + s[n-1]

按照指定属性进行排序

**import** java.util.Comparator;

Comparator<Hero> c = **new** Comparator<Hero>() {

//写比较的方法

}

//按照规则C对heros进行排序

Collections.sort(heros,c);

ArrayList heroList<? extends Hero> 表示这是一个Hero泛型或者其子类泛型

所以 可以确凿的是，从heroList取出来的对象，一定是可以转型成Hero

但是，不能往里面放东西

ArrayList heroList<? super Hero> 表示这是一个Hero泛型或者其父类泛型

heroList的泛型可能是Hero

heroList的泛型可能是Object

可以往里面插入Hero以及Hero的子类

但是取出来有风险，因为不确定取出来是Hero还是Object

**如果希望只取出，不插入，就使用? extends Hero**

**如果希望只插入，不取出，就使用? super Hero**

**如果希望，又能插入，又能取出，就不要用通配符？**

# 多线程

进程：启动一个LOL.exe就叫一个进程。 接着又启动一个DOTA.exe，这叫两个进程。  
**线程（thread）：线程是在进程内部同时做的事情**，比如在LOL里，有很多事情要同时做，比如"盖伦” 击杀“提莫”，同时“赏金猎人”又在击杀“盲僧”，这就是由多线程来实现的。

### 设计一个类KillThread 继承Thread，并且重写run方法

启动线程办法： 实例化一个KillThread对象（一个该对象代表一个线程），并且**调用其start方法**就可以观察到 赏金猎人攻击盲僧的同时，盖伦也在攻击提莫。

KillThread killThread1 = **new** KillThread(gareen,teemo);

killThread1.start();

### Runnable接口简介

Runnable接口是线程辅助类，仅定义了一个方法run()方法，用来辅助实现多线程。

**优点：**  
灵活：Runnable可以继承其他类实现对Runnable实现类的增强，避免了Thread类由于继承Thread类而无法继承其他类的问题  
共享资源:Runnable接口的run()方法可以被多个线程共享，适用于多个进程处理一种资源的问题

**使用方法：**

(1) 实现Runnable接口

(2) 重写run()方法

(3) 创建runnable实例

(4) 创建Thread实例

(5) 将Runnable实例放入Thread实例中

(6) 通过线程实例控制线程的行为(运行，停止)，在运行时会调用Runnable接口中的run方法。

**new** Thread(newclass).start();//class实现Runnable接口后再实例化。

### 实现 Callable接口或Future

JDK1.5后才更新的功能，Callable<V>返回v类型的值，用~.call()函数启动

1. 创建 Callable 接口的实现类，并实现 call() 方法，该 call() 方法将作为线程执行体，并且有返回值。

2. 创建 Callable 实现类的实例，使用 FutureTask 类来包装 Callable 对象，该 FutureTask 对象封装了该 Callable 对象的 call() 方法的返回值。

3. 使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。

4. 调用 FutureTask 对象的 get() 方法来获得子线程执行结束后的返回值。

**多线程创建-匿名类**

使用[匿名类](https://how2j.cn/k/interface-inheritance/interface-inheritance-inner-class/322.html#step687)，继承Thread,重写run方法，直接在run方法中写业务代码  
匿名类的一个好处是可以很方便的访问外部的局部变量。  
前提是外部的局部变量需要被声明为final。(JDK7以后就不需要了)

Thread t1= **new** Thread(){

**public** **void** run(){

**while**(!teemo.isDead()){

      gareen.attackHero(teemo);

      }

    }

}

### 常见线程方法

1. **暂停线程Thread.sleep(1000);** 表示当前线程暂停1000毫秒 ，其他线程不受影响；会抛出InterruptedException 中断异常，因为当前线程sleep的时候，有可能被停止，这时就会抛出 InterruptedException
2. **加入到当前线程中**

所有进程，至少会有一个线程即主线程，即main方法开始执行，就会有一个看不见的主线程存在。  
执行t.join();，即表明在主线程中加入该线程。  
主线程会等待该线程结束完毕， 才会往下运行。

1. **线程优先级**

当线程处于竞争关系的时候，优先级高的线程会有更大的几率获得CPU资源

 t1.setPriority(Thread.MAX\_PRIORITY);

 t2.setPriority(Thread.MIN\_PRIORITY);

1. **临时暂停该线程**

当前线程，临时暂停，使得其他线程可以有更多的机会占用CPU资源

 Thread.yield();

1. **守护线程**

守护线程的概念是： 当一个进程里，所有的线程都是守护线程的时候，结束当前进程。

守护线程通常会被用来做日志，性能统计等工作。

t1.setDaemon(**true**);

### 多线程的同步

多线程的同步问题指的是多个线程同时修改一个数据的时候，可能导致的问题  
多线程的问题，又叫Concurrency 问题

由于多线程同时修改数据导致的错误数据在业务上又叫做**脏数据**

总体解决思路是： 在增加线程访问同一数据期间，其他线程不可以访问该数据

**synchronized 同步对象概念**

synchronized表示当前线程，独占 对象 someObject

当前线程独占 了对象someObject，如果有其他线程试图占有对象someObject，就会等待，直到当前线程释放对someObject的占用。

someObject 又叫同步对象，所有的对象，都可以作为同步对象

为了达到同步的效果，必须使用同一个同步对象

释放同步对象的方式： synchronized 块自然结束，或者有异常抛出

**synchronized** (someObject) { }

该块也可以直接加在方法里，此时每个调用该方法的函数都自动同步

如果一个类，其方法都是有synchronized修饰的，那么该类就叫做线程安全的类

若用synchronized修饰类，则该类中所有对象方法都含有同步（即使用该类时就进行同步）

### 线程安全的类

HashMap和Hashtable都实现了Map接口，都是键值对保存数据的方式

区别1：

HashMap可以存放 null

Hashtable不能存放null

区别2：

HashMap不是线程安全的类

Hashtable是线程安全的类

StringBuffer 是线程安全的

StringBuilder 是非线程安全的

所以当进行大量字符串拼接操作的时候，如果是单线程就用StringBuilder会更快些，如果是多线程，就需要用StringBuffer 保证数据的安全性

Vector是线程安全的类，而ArrayList是非线程安全的。、

把非线程安全的集合转换为线程安全

ArrayList是非线程安全的，换句话说，多个线程可以同时进入一个ArrayList对象的add方法。借助**Collections.synchronizedLis**t，可以把ArrayList转换为线程安全的List。  
与此类似的，还有HashSet,LinkedList,HashMap等等非线程安全的类，都通过**[工具类Collections](https://how2j.cn/k/collection/collection-collections/369.html)转换**为线程安全的

### 死锁

1. 线程1 首先占有对象1，接着试图占有对象2

2. 线程2 首先占有对象2，接着试图占有对象1

3. 线程1 等待线程2释放对象2

4. 与此同时，线程2等待线程1释放对象1

### 交互

this.wait()表示 让占有this的线程等待，并临时释放占有

this.notify() 表示通知那些等待在this的线程，可以苏醒过来了。

这样能释放内存，提升性能

需要强调的是，wait方法和notify方法，并不是Thread线程上的方法，它们是Object上的方法。

wait()的意思是： 让占用了这个同步对象的线程，临时释放当前的占用，并且等待。 所以调用wait是有前提条件的，一定是在synchronized块里，否则就会出错。

notify() 的意思是，通知**一个**等待在这个同步对象上的线程，你可以苏醒过来了，有机会重新占用当前对象了。

notifyAll() 的意思是，通知**所有的**等待在这个同步对象上的线程，你们可以苏醒过来了，有机会重新占用当前对象了。

### 线程池

**每一个线程的启动和结束都是比较消耗时间和占用资源的。**

如果在系统中用到了很多的线程，大量的启动和结束动作会导致系统的性能变卡，响应变慢。

线程池设计思路

1. 准备一个任务容器

2. 一次性启动10个 消费者线程

3. 刚开始任务容器是空的，所以线程都wait在上面。

4. 直到一个外部线程往这个任务容器中扔了一个“任务”，就会有一个消费者线程被唤醒notify

5. 这个消费者线程取出“任务”，并且执行这个任务，执行完毕后，继续等待下一次任务的到来。

6. 如果短时间内，有较多的任务加入，单个线程无法及时完成，那么就会有多个线程被唤醒，去执行这些任务。

java提供自带的线程池，而不需要自己去开发一个自定义线程池了。  
线程池类ThreadPoolExecutor在包java.util.concurrent下

ThreadPoolExecutor threadPool= new ThreadPoolExecutor(10, 15, 60, TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());

第一个参数10 表示这个线程池初始化了10个线程在里面工作

第二个参数15 表示如果10个线程不够用了，就会自动增加到最多15个线程

第三个参数60 结合第四个参数TimeUnit.SECONDS，表示经过60秒，多出来的线程还没有接到活儿，就会回收，最后保持池子里就10个

第四个参数TimeUnit.SECONDS 如上

第五个参数 new LinkedBlockingQueue() 用来放任务的集合

execute方法用于添加新的任务

### Lock对象

与synchronized类似的，lock也能够达到同步的效果

Lock是一个接口，为了使用一个Lock对象，需要用到

Lock lock = new ReentrantLock();

与 synchronized 不同的是，一旦synchronized 块结束，就会自动释放对someObject的占用。 **lock却必须调用unlock方法进行手动释放**，为了保证释放的执行，往往会把unlock() 放在finally中进行。

Lock锁定时到底锁住了什么对象（锁住了线程内lock之后的所有对象直到unlock释放）

**synchronized 是不占用到手不罢休的，会一直试图占用下去**。（自旋锁）

Lock接口提供了一个**trylock方法**，trylock会在指定时间范围内试图占用，占成功了，就执行。 如果时间到了，还占用不成功，直接跳过。

注意： 因为使用trylock有可能成功，有可能失败，所以后面unlock释放锁的时候，需要判断是否占用成功了，如果没占用成功也unlock,就会抛出异常

lock.tryLock(1,TimeUnit.SECONDS)//尝试占用1s

使用synchronized方式进行线程交互，用到的是同步对象的wait,notify和notifyAll方法  
Lock也提供了类似的解决办法，首先通过lock对象得到一个Condition对象，然后分别调用这个Condition对象的：**await, signal,signalAll 方法**  
注意： 不是Condition对象的wait,nofity,notifyAll方法,是await,signal,signalAll

**总结：**

1. Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现，Lock是代码层面的实现。

2. Lock可以选择性的获取锁，如果一段时间获取不到，可以放弃。synchronized不行，会一根筋一直获取下去。 借助Lock的这个特性，就能够规避死锁，synchronized必须通过谨慎和良好的设计，才能减少死锁的发生。

3. synchronized在发生异常和同步块结束的时候，会自动释放锁。而Lock必须手动释放， 所以如果忘记了释放锁，一样会造成死锁。

**原子访问**

所谓的原子性操作即不可中断的操作，比如赋值操作

**原子性操作本身是线程安全的**  
但是 i++ 这个行为，事实上是有3个原子性操作组成的。

步骤 1. 取 i 的值；步骤 2. i + 1；步骤 3. 把新的值赋予i

这三个步骤，每一步都是一个原子操作，但是合在一起，就不是原子操作。就不是线程安全的。换句话说，一个线程在步骤1 取i 的值结束后，还没有来得及进行步骤2，另一个线程也可以取 i的值了。

**AtomicInteger**

JDK6 以后，新增加了一个包java.util.concurrent.atomic，里面有各种原子类，比如AtomicInteger。  
而AtomicInteger提供了各种自增，自减等方法，这些方法都是原子性的。 换句话说，自增方法 incrementAndGet 是线程安全的，同一个时间，只有一个线程可以调用这个方法。

常用方法int i = atomicI.decrementAndGet();//i--

        int j = atomicI.incrementAndGet();//j++

        int k = atomicI.addAndGet(3);//k+=3

# Lambda

与匿名类 概念相比较，

Lambda 其实就是匿名方法，这是一种把方法作为参数进行传递的编程思想。

把参数类型和圆括号去掉(只有一个参数的时候，才可以去掉圆括号).

只保留方法参数和方法体，参数和方法体之间加上符号 ->

ilter(heros, h -> h.hp > 100 && h.damage < 50);

引入Lambda表达式，会使得代码更加紧凑，而不是各种接口和匿名类到处飞。

缺点：

1. 可读性差，与啰嗦的但是清晰的匿名类代码结构比较起来，Lambda表达式一旦变得比较长，就难以理解

2. 不便于调试，很难在Lambda表达式中增加调试信息，比如日志

3. 版本支持，Lambda表达式在JDK8版本中才开始支持，如果系统使用的是以前的版本，考虑系统的稳定性等原因，而不愿意升级，那么就无法使用。

Lambda比较适合用在简短的业务代码中，并不适合用在复杂的系统中，会加大维护成本。

**Lambda引用静态方法**

System.out.println("使用Lambda表达式");

filter(heros, h->h.hp>100 && h.damage<50);

System.out.println("在Lambda表达式中使用静态方法");

filter(heros, h -> TestLambda.testHero(h) );

System.out.println("直接引用静态方法");

filter(heros, TestLambda::testHero);//方法需额外在TestLambda类下构造。

**Lambda引用对象方法**

与引用静态方法很类似，只是传递方法的时候，需要一个对象的存在

TestLambda testLambda = new TestLambda();

filter(heros, testLambda::testHero);

**Lambda引用容器中的对象方法**

在Lambda表达式中调用容器中的对象Hero的方法matched

filter(heros,h-> h.matched() );

matched恰好就是容器中的对象Hero的方法，那就可以进一步改写为

filter(heros, Hero::matched);

**Lambda聚合操作**

Stream 和Collection结构化的数据不一样，Stream是一系列的元素，就像是生产线上的罐头一样，一串串的出来。

管道指的是一系列的聚合操作。管道又分3个部分

管道源：在这个例子里，源是一个List

中间操作： 每个中间操作，又会返回一个Stream，比如.filter()又返回一个Stream, 中间操作是“懒”操作，并不会真正进行遍历。

结束操作：当这个操作执行后，流就被使用“光”了，无法再被操作。所以这必定是流的最后一个操作。 结束操作不会返回Stream，但是会返回int、float、String、 Collection或者像forEach，什么都不返回, 结束操作才进行真正的遍历行为，在遍历的时候，才会去进行中间操作的相关判断

把Collection切换成管道源很简单，调用stream()就行了。

heros.stream()

但是数组却没有stream()方法，需要使用

Arrays.stream(hs)或者Stream.of(hs)

**中间操作**

每个中间操作，又会返回一个Stream，比如.filter()又返回一个Stream, 中间操作是“懒”操作，并不会真正进行遍历。  
中间操作比较多，主要分两类。对元素进行筛选 和 转换为其他形式的流  
对元素进行筛选：  
filter 匹配

.filter(h->h.hp>100&&h.damage<50)  
distinct 去除重复(根据equals判断)

.distinct()  
sorted 自然排序

.sorted((h1,h2)->h1.hp>=h2.hp?1:-1)  
sorted(Comparator<T>) 指定排序  
limit 保留

.limit(3)//从前往后保留  
skip 忽略

.skip(3)//  
转换为其他形式的流  
mapToDouble 转换为double的流  
map 转换为任意类型的流

结束操作才真正进行遍历行为，前面的中间操作也在这个时候，才真正的执行。

常见结束操作如下：

forEach() 遍历每个元素

toArray() 转换为数组

min(Comparator<T>) 取最小的元素

max(Comparator<T>) 取最大的元素

count() 总数

findFirst() 第一个元素

# Mysql

创建数据库

**create database** how2java

创建表

**CREATE TABLE** hero (

id int(11) **AUTO\_INCREMENT,**

name varchar(30) ,

hp float ,

damage int(11) ,

**PRIMARY KEY (id)**

) **DEFAULT CHARSET=utf8;**

插入数据

**insert into** hero **values** (null, '盖伦', 616, 100)

查询所有数据

**select \* from** hero

统计表中数据量

**select count(\*) from** hero

显示前5条数据

**select \* from** hero **limit** 0,5

修改数据

**update** hero **set** hp = 818 **where** id = 1

删除数据

**delete from** hero **where** id = 1

停止数据

net stop mysql