Java中级目录 【异常处理+I/O+集合框架】

[1. 异常处理 2](#_Toc5721)

[2. Throwable 4](#_Toc16801)

[3. 自定义异常 4](#_Toc14973)

[4. 文件IO 5](#_Toc4222)

[5. 文件常用方法 6](#_Toc31410)

[6. 字节流 7](#_Toc11678)

[7. 关闭流的三种方式 11](#_Toc23984)

[8. 字符流 13](#_Toc23207)

[9. 读取中文问题 14](#_Toc17705)

[10. 缓存流 15](#_Toc1693)

[11. 数据流 17](#_Toc4037)

[12. 对象流 18](#_Toc23153)

[13. System.in练习 （给定模板替换生成类） 19](#_Toc7155)

[14. 文件练习（很实用!!） 21](#_Toc14098)

[15. lO流关系图 25](#_Toc14897)

[16. ArrayList存放对象 （相当于C++ vector 如果不指定泛型什么类型都可以放） 25](#_Toc6413)

[17. 泛型Generic 27](#_Toc21188)

[18. LinkedList 27](#_Toc13806)

[19. HashMap 28](#_Toc25073)

[20. HashSet （作为HashMapd的key存在） 29](#_Toc2574)

[21. Collections 工具类 30](#_Toc18230)

[22. 比较器 Comparator 30](#_Toc369)

[23. 接口Comparable 31](#_Toc1814)

[24.泛型 32](#_Toc31961)

[25.通配符 33](#_Toc23663)

1. 异常处理
   1. try catch

(1)将可能抛出异常的代码放在try里

(2)如果无异常，就会顺序往下执行，并且不执行catch块中的代码

(3)如果出现异常，try里的代码会立即终止，程序流程会运行到对应的catch块中

(4)e.printStackTrace(); 会打印出方法的调用痕迹。

**import** java.io.File;  
**import** java.io.FileInputStream;  
**import** java.io.FileNotFoundException;  
  
**public class** Main{  
 **public static void** main(String args[]){  
 File f = **new** File(**"d:LOL.exe"**);  
  
 **try**{  
 System.***out***.println(**"try to open file"**);  
 **new** FileInputStream(f);  
 System.***out***.println(**"success to open file"**);  
 }**catch**(FileNotFoundException e){  
 System.***out***.println(**"cannot open file"**);  
 e.printStackTrace();  
 }  
 }  
}

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

* 1. 多异常捕捉

1. 多个catch
2. 放在一个catch中统一捕捉，要区分使用instanceof区分

**import** java.io.File;  
**import** java.io.FileInputStream;  
**import** java.io.FileNotFoundException;  
**import** java.text.ParseException;  
**import** java.text.SimpleDateFormat;  
**import** java.util.Date;  
  
**public class** Main{  
 **public static void** main(String args[]){  
 File f = **new** File(**"d:/LOL.exe"**);  
 **try**{  
 System.***out***.println(**"try to open the file"**);  
 **new** FileInputStream(f);  
 System.***out***.println(**"success to open file"**);  
 SimpleDateFormat sdf = **new** SimpleDateFormat(**"yyyy-MM-dd"**);  
 Date d = sdf.parse(**"2020-01-27"**);  
 } **catch** (FileNotFoundException | ParseException e) {  
 **if**(e **instanceof** FileNotFoundException) System.***out***.println(**"file is not exist"**);  
 **if**(e **instanceof** ParseException) System.***out***.println(**"date error"**);  
 e.printStackTrace();  
 }  
 }  
}

* 1. finally 无论是否出现异常，finally中的代码都会执行。
  2. Throws

考虑如下情况:

主方法调用method1

method1调用method2

method2中打开文件

method2中需要进行异常处理

但是method2不打算处理，而是把这个异常通过throws抛出去

那么method1就会接到该异常。处理办法也是两种，要么是try catch处理掉，要么也是抛出去

method1选择本地try catch住，一旦try catch住了，就相当于把这个异常消化掉了，主方法在调用method1的时候，就不需要进行异常处理了。

**import** java.io.File;  
**import** java.io.FileInputStream;  
  
**public class** Main{  
 **public static void** main (String args[]){  
 *method1*();  
 }  
 **public static void** method1(){  
 **try** {  
 *method2*();  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
 **public static void** method2() **throws** Exception{  
 File f = **new** File(**"d:/LOL.exe"**);  
 System.***out***.println(**"try to open the file"**);  
 **new** FileInputStream(f);  
 System.***out***.println(**"success to open file"**);  
 }  
}

* 1. throw和throws的区别

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。
3. Throwable

Throwable是类，Exception和Error都继承了该类

**import** java.io.File;  
**import** java.io.FileInputStream;  
**public class** Main{  
 **public static void** main(String[] args) {  
 File f = **new** File(**"d:/LOL.exe"**);  
 **try** {  
 **new** FileInputStream(f); *//使用Throwable进行异常捕捉* } **catch** (Throwable t) {  
 t.printStackTrace();  
 }  
 }  
}

1. 自定义异常

·创建一个自定义的异常：一个英雄攻击另一个英雄的时候，如果发现另一个英雄已经挂了，就会抛出EnemyHeroIsDeadException

·创建一个类EnemyHeroIsDeadException,并继承Exception

需要提供两个构造方法()：无参的构造方法；带参的构造方法，并调用父类对应的构造方法。

·在Hero的attack方法中，当发现敌方英雄的血量为0的时候，抛出该异常

1. 创建一个EnemyHeroIsDeadException实例
2. 通过throw抛出该异常
3. 当前方法通过throws抛出该异常

在外部调用attack方法的时候，就需要进行捕捉，并且捕捉的时候，可以通过e.getMessage() 获取当时出错的具体原因

**public class** Hero {  
 **public** String **name**;  
 **protected float hp**;  
 Hero(String name, **float** hp){  
 **this**.**name** = name;  
 **this**.**hp** = hp;  
 }  
 **class** EnemyHeroIsDeadException **extends** Exception{  
 **public** EnemyHeroIsDeadException(){}  
 **public** EnemyHeroIsDeadException(String msg){  
 **super**(msg);  
 }  
 }  
  
 **public void** attack(Hero h) **throws** EnemyHeroIsDeadException {  
 **if**(h.**hp** == 0){  
 **throw new** EnemyHeroIsDeadException(h.**name** + **"已经挂了，无需再释放技能。"**);  
 }  
 }  
 **public static void** main(String args[]){  
 Hero garen = **new** Hero(**"garen"**,616);  
 Hero temmo = **new** Hero(**"temmo"**, 0);  
 **try**{  
 garen.attack(temmo);  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
}

1. 文件IO

**import** java.io.File;  
**public class** Main {  
 **public static void** main(String args[]){  
 File f1 = **new** File(**"d:/testa.txt"**);  
 File f2 = **new** File(**"d:/testb.txt"**);  
 System.***out***.println(f1.getAbsolutePath()); *//d:\testa.txt* System.***out***.println(f2.getAbsolutePath()); *//d:\testb.txt* }  
}

1. 文件常用方法

(1).常用方法

**public class** Main {  
 **public static void** main(String args[]){  
 File f = **new** File(**"d:/testa.txt"**);  
 System.***out***.println(**"当前文件是："**+ f);  
 System.***out***.println(**"判断是否存在"** + f.exists());  
 System.***out***.println(**"判断是否为文件夹"** + f.isDirectory());  
 System.***out***.println(**"判断是否为文件"** + f.isFile());  
 System.***out***.println(**"获取文件的长度"** + f.length());  
  
 **long** time = f.lastModified();  
 Date d = **new** Date(time);  
 System.***out***.println(**"获取文件的最后修改时间："**+ d);  
 f.setLastModified(0);  
  
 *//文件重命名* File f2 = **new** File(**"d:/testb.txt"**);  
 f.renameTo(f2);  
 }  
}

(2).f.list()

File f = **new** File(**"D:/My\_Java/Test"**);  
*// 以字符串的形式，返回当前文件夹下的所有文件（不包括子文件和子文件夹）*String str[] = f.list();  
**for**(String it : str){  
 System.***out***.println(it);  
}

输出:

.project

bin

src

(3).

**public static void** main(String args[]){  
 File f = **new** File(**"D:/My\_Java/Test"**);  
 *//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();  
  
 String fa\_file = f.getParent();  
 System.***out***.println(fa\_file); *//以字符串形式返回获取所在文件夹  
 // 以文件形式返回获取所在文件夹* File fa = f.getParentFile();  
}

1. 字节流

InputStream字节输入流

OutputStream 字节输出流

用于以字节的形式读取和写入数据

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。

1). 读文件：

*//FileInputStream 读文件示例***public class** Main {  
 **public static void** main(String args[]){  
 **try** {  
 File f = **new** File(**"d:/testb.txt"**);  
 FileInputStream fis = **new** FileInputStream(f);  
  
 **byte**[] all = **new byte**[(**int**)f.length()];  
 fis.read(all);  
 **for**(**byte** b : all){  
 System.***out***.print((**char**)b);  
 }  
 fis.close();  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
}

2). 写文件

*//FileOutputStream 写文件示例***public class** Main {  
 **public static void** main(String args[]){  
 **try** {  
 File f = **new** File(**"d:/testb.txt"**);  
 FileOutputStream fos = **new** FileOutputStream(f);  
 **byte**[] data = {**'X'**, **'Y'**};  
 fos.write(data);  
 fos.close();  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
}

1. . 自动创建目录

练习：[以字节流的形式向文件写入数据](https://how2j.cn/k/io/io-bytestream/340.html" \l "step2414) 中的例子，当testb.txt不存在的时候，是会自动创建testb.txt文件的。  
但是，如果是写入数据到d:/xyz/testb.txt，而目录xyz又不存在的话，就会抛出异常。  
那么怎么自动创建xyz目录？  
如果是多层目录 d:/xyz/abc/def/testb.txt 呢？

代码： 优点（增强了健壮性）

*//FileOutputStream 写文件示例***public class** Main {  
 **public static void** main(String args[]){  
 **try** {  
 File f = **new** File(**"d:/xyz/abc/def/testb.txt"**);  
 *//因为默认情况下，文件系统中不存在 d:\xyz\abc\def，所以输出会失败  
 //首先获取文件所在的目录* File dir = f.getParentFile();  
 *//如果该目录不存在，则创建该目录* **if**(!dir.exists()){  
*// dir.mkdir(); //使用mkdir会抛出异常，因为该目录的父目录也不存在* dir.mkdirs();*//使用mkdirs则会把不存在的目录都创建好* }  
 FileOutputStream fos = **new** FileOutputStream(f);  
 **byte**[] data = {**'X'**, **'Y'**};  
 fos.write(data);  
 fos.close();  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
}

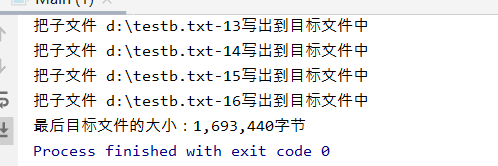
1. . 文件拆分

*//拆分文件***public class** Main {  
 **public static void** main(String args[]){  
 **int** eachSize = 100 \* 1024; *// 100k* File srcFile = **new** File(**"d:/testb.txt"**);  
 *splitFile*(srcFile, eachSize);  
 }  
 */\*\*  
 \* 拆分的思路，先把源文件的所有内容读取到内存中，然后从内存中挨个分到子文件里  
 \** ***@param srcFile*** *要拆分的源文件  
 \** ***@param eachSize*** *按照这个大小，拆分  
 \*/* **private static void** splitFile(File srcFile, **int** eachSize){  
 **if**(srcFile.length() == 0)  
 **throw new** RuntimeException(**"文件长度为0，不可拆分"**);  
  
 **byte**[] fileContent = **new byte**[(**int**) srcFile.length()];  
 *//1.先把文件读到数组中* **try**{  
 FileInputStream fis = **new** FileInputStream(srcFile);  
 fis.read(fileContent);  
 fis.close();  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 *//2.计算需要被划分成多少份子文件* **int** fileNumber;  
 **if**(fileContent.**length** % eachSize == 0)  
 fileNumber = (**int**) (fileContent.**length** / eachSize);  
 **else** fileNumber = (**int**) (fileContent.**length** / eachSize) + 1;  
  
 **for**(**int** i = 0; i < fileNumber; i ++){  
 String eachFileName = srcFile.getName() + **'-'** + i;  
 File eachFile = **new** File(srcFile.getParent(), eachFileName);  
 **byte**[] eachContent;  
  
 *//3. 从源文件的内容里，复制部分数据到子文件  
 // 除开最后一个文件， 其它文件大小都是eachSize  
 // 最后一个文件的大小可能不足eachSize* **if**(i != fileNumber - 1) *//不是最后一个* eachContent = Arrays.*copyOfRange*(fileContent, eachSize \* i, eachSize \* (i + 1));  
 **else** eachContent = Arrays.*copyOfRange*(fileContent, eachSize \* i, fileContent.**length**);  
 **try**{ *//4. 写出去* FileOutputStream fos = **new** FileOutputStream(eachFile);  
 fos.write(eachContent);  
 fos.close();  
 System.***out***.printf(**"输出子文件%s，其大小是 %d字节%n"**,eachFile.getAbsoluteFile(),eachFile.length());  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}



1. . 文件合并

**public class** Main {  
 **public static void** main(String args[]){  
 *mergeFile*(**"d:/"**,**"testb.txt"**);  
 }  
 */\*\*  
 \* 合并的思路，就是从testb.txt-0开始，读取到一个文件，就开始写出到testb.txt中，直到没有文件可以读  
 \** ***@param folder*** *需要合并的文件所处于的目录  
 \** ***@param fileName*** *需要合并的文件的名称  
 \*/* **private static void** mergeFile(String folder, String fileName){  
 **try**{  
 *//合并的目标文件* File destFile = **new** File(folder, fileName);  
 FileOutputStream fos = **new** FileOutputStream(destFile);  
 **int** index = 0;  
 **while**(**true**){  
 *//子文件* File eachFile = **new** File(folder, fileName + **"-"** + index++);  
 *//如果子文件不存在了就结束* **if**(!eachFile.exists())  
 **break**;  
  
 *//读取子文件的内容* FileInputStream fis = **new** FileInputStream(eachFile);  
 **byte**[] eachContent = **new byte**[(**int**) eachFile.length()];  
 fis.read(eachContent);  
 fis.close();  
  
 *//把子文件的内容写出去* fos.write(eachContent);  
 fos.flush(); *//清空缓存区，即直接将缓存的内容写入文件中* System.***out***.printf(**"把子文件 %s写出到目标文件中%n"**,eachFile);  
 }  
  
 fos.close(); *//所有输出结束后再关闭输出流* System.***out***.printf(**"最后目标文件的大小：%,d字节"**,destFile.length());  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}



1. 关闭流的三种方式
2. 在try中关闭（不健壮，在关闭流前的代码出现异常会无法关闭流）

在try的作用域里关闭文件输入流，在前面的示例中都是使用这种方式，这样做有一个弊端；  
如果文件不存在，或者读取的时候出现问题而抛出异常，那么就不会执行这一行关闭流的代码，存在巨大的资源占用隐患。 不推荐使用

**public static void** main(String[] args) {  
 **try** {  
 File f = **new** File(**"d:/lol.txt"**);  
 FileInputStream fis = **new** FileInputStream(f);  
 **byte**[] all = **new byte**[(**int**) f.length()];  
 fis.read(all);  
 **for** (**byte** b : all) {  
 System.***out***.println(b);  
 }  
 *// 在try 里关闭流* fis.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
}

1. 在finally中关闭（较健壮，但是可读性差）

这是标准的关闭流的方式  
1. 首先把流的引用声明在try的外面，如果声明在try里面，其作用域无法抵达finally.  
2. 在finally关闭之前，要先判断该引用是否为空  
3. 关闭的时候，需要再一次进行try catch处理  
这是标准的严谨的关闭流的方式，但是看上去很繁琐

**public static void** main(String[] args) {  
 File f = **new** File(**"d:/lol.txt"**);  
 FileInputStream fis = **null**;  
 **try** {  
 fis = **new** FileInputStream(f);  
 **byte**[] all = **new byte**[(**int**) f.length()];  
 fis.read(all);  
 **for** (**byte** b : all) {  
 System.***out***.println(b);  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } **finally** {  
 *// 在finally 里关闭流* **if** (**null** != fis)  
 **try** {  
  
 fis.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

1. 使用try()的方式（健壮 + 代码可读性强）

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  
所有的流，都实现了一个接口叫做 **AutoCloseable**，任何类实现了这个接口，都可以在try()中进行实例化。 并且在try, catch, finally结束的时候自动关闭，回收相关资源。

**public class** Main {  
 **public static void** main(String[] args) {  
 File f = **new** File(**"d:/lol.txt"**);  
  
 *//把流定义在try()里,try,catch或者finally结束的时候，会自动关闭* **try** (FileInputStream fis = **new** FileInputStream(f)) {  
 **byte**[] all = **new byte**[(**int**) f.length()];  
 fis.read(all);  
 **for** (**byte** b : all) {  
 System.***out***.println(b);  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

1. 字符流

Reader字符输入流

Writer字符输出流

专门用于字符的形式读取和写入数据

1. 使用字符流读取文件

**public class** Main{  
 **public static void** main(String args[]){  
 File f = **new** File(**"d:/testb.txt"**);  
 **try**(FileReader fr = **new** FileReader(f)){  
 **char** all[] = **new char**[(**int**) f.length()];  
 fr.read(all);  
 **for**(**char** b: all){  
 System.***out***.println(b);  
 }  
 }**catch** (IOException e){  
 e.printStackTrace();  
 }  
 }  
}

1. 使用字符流把字符串写入到文件

**public class** Main{  
 **public static void** main(String args[]){  
 File f = **new** File(**"d:/testb.txt"**);  
 **try**(FileWriter fw = **new** FileWriter(f)){  
 String data = **"abcdadsfad1234"**;  
 **char**[] cs = data.toCharArray();  
 fw.write(cs);  
 }**catch** (IOException e){  
 e.printStackTrace();  
 }  
 }  
}

1. 读取中文问题

(1).用FileInputStream字节流正确读取中文

*//用FileInputStream字节流正确读取中文***public class** Main {  
 **public static void** main(String[] args) {  
 File f = **new** File(**"d:\\testb.txt"**);  
 **try** (FileInputStream fis = **new** FileInputStream(f);) {  
 **byte**[] all = **new byte**[(**int**) f.length()];  
 fis.read(all);  
 String str = **new** String(all,**"UTF-8"**); *//对应testb.txt的编码* System.***out***.println(str);  
 } **catch** (IOException e) {  
 *//* ***TODO Auto-generated catch block*** e.printStackTrace();  
 }  
 }  
}

(2).用FileReader字符流正确读取中文

FileReader得到的是字符，所以一定是已经把字节****根据某种编码识别成了字符****了  
而FileReader使用的编码方式是Charset.defaultCharset()的返回值，如果是中文的操作系统，就是GBK  
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"));

代码：

**public class** Main {  
  
 **public static void** main(String[] args) **throws** UnsupportedEncodingException, FileNotFoundException {  
 File f = **new** File(**"D:/testb.txt"**);  
 System.***out***.println(**"默认编码方式:"**+Charset.*defaultCharset*());  
 *//FileReader得到的是字符，所以一定是已经把字节根据某种编码识别成了字符了  
 //而FileReader使用的编码方式是Charset.defaultCharset()的返回值，如果是中文的操作系统，就是GBK 我的电脑是UTF-8...* **try** (FileReader fr = **new** FileReader(f)) {  
 **char**[] cs = **new char**[(**int**) f.length()];  
 fr.read(cs);  
 System.***out***.printf(**"FileReader会使用默认的编码方式%s,识别出来的字符是：%n"**,Charset.*defaultCharset*());  
 System.***out***.println(**new** String(cs));  
 } **catch** (IOException e) {  
 *//* ***TODO Auto-generated catch block*** e.printStackTrace();  
 }  
 *//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  
 //并且使用new InputStreamReader(new FileInputStream(f),Charset.forName("UTF-8")); 这样的形式* **try** (InputStreamReader isr = **new** InputStreamReader(**new** FileInputStream(f),Charset.*forName*(**"UTF-8"**))) {  
 **char**[] cs = **new char**[(**int**) f.length()];  
 isr.read(cs);  
 System.***out***.printf(**"InputStreamReader 指定编码方式UTF-8,识别出来的字符是：%n"**);  
 System.***out***.println(**new** String(cs));  
 } **catch** (IOException e) {  
 *//* ***TODO Auto-generated catch block*** e.printStackTrace();  
 }  
  
 }  
}

1. 缓存流
2. 使用缓存流读取数据

缓存字符输入流BufferedReader可以一次读取一行数据

**public class** Main{  
 **public static void** main(String args[]){  
 File f = **new** File(**"d:/testb.txt"**);  
 *// 创建文件字符流  
 // 缓存流必须建立在一个存在的流的基础上* **try** (  
 FileReader fr = **new** FileReader(f);  
 BufferedReader br = **new** BufferedReader(fr);  
 )  
 {  
 **while**(**true**){  
 *//一次读一行* String line = br.readLine();  
 **if**(line == **null**) **break**;  
 System.***out***.println(line);  
 }  
 }**catch** (IOException e){  
 e.printStackTrace();  
 }  
 }  
}

1. 使用缓存流写出数据

PrintWriter 缓存字符输出流， 可以一次写出一行数据

**public class** Main {  
 **public static void** main(String[] args) {  
 *// 向文件testb.txt中写入三行语句* File f = **new** File(**"d:/testb.txt"**);  
 **try** (  
 *// 创建文件字符流* FileWriter fw = **new** FileWriter(f);  
 *// 缓存流必须建立在一个存在的流的基础上* PrintWriter pw = **new** PrintWriter(fw);  
 ) {  
 pw.println(**"garen kill teemo"**);  
 pw.println(**"teemo revive after 1 minutes"**);  
 pw.println(**"teemo try to garen, but killed again"**);  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

1. Flush

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

**public class** Main {  
 **public static void** main(String[] args) {  
 File f = **new** File(**"d:/testb.txt"**);  
 **try** (FileWriter fw = **new** FileWriter(f); PrintWriter pw = **new** PrintWriter(fw)) {  
 pw.println(**"garen kill teemo"**);  
 pw.flush(); *//强制把缓存中的数据写入硬盘，无论缓存是否已满* pw.println(**"teemo revive after 1 minutes"**);  
 pw.flush();  
 pw.println(**"teemo try to garen, but killed again"**);  
 pw.flush();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

1. 数据流

DataInputStream数据输入流

DataOutputStream数据输出流

使用数据流的writeUTF()和readUTF() 可以进行数据的格式化顺序读写  
如本例，通过DataOutputStream 向文件顺序写出 布尔值，整数和字符串。 然后再通过DataInputStream 顺序读入这些数据。

****注：**** 要用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。

**import** java.io.\*;  
**public class** Main{  
 **public static void** main(String args[]){  
 *write*();  
 *read*();  
 }  
 **private static void** read(){  
 File f = **new** File(**"d:/testb.txt"**);  
 **try**(FileInputStream fis = **new** FileInputStream(f); DataInputStream dis = **new** DataInputStream(fis)){  
 **boolean** b = dis.readBoolean();  
 **int** i = dis.readInt();  
 String str = dis.readUTF();  
  
 System.***out***.println(**"读取到布尔值:"**+b);  
 System.***out***.println(**"读取到整数:"**+i);  
 System.***out***.println(**"读取到字符串:"**+str);  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
 **private static void** write(){  
 File f = **new** File(**"d:/testb.txt"**);  
 **try**(FileOutputStream fos = **new** FileOutputStream(f); DataOutputStream dos = **new** DataOutputStream(fos)){  
 dos.writeBoolean(**true**);  
 dos.writeInt(300);  
 dos.writeUTF(**"123 this is garen"**);  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
}

1. 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它的介质，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，必须实现Serializable接口。

Hero.java

**package** fondamental;  
**import** java.io.Serializable;  
**public class** Hero **implements** Serializable {  
 *//表示这个类当前的版本，如果有了变化，比如新设计了属性，就应该修改这个版本号* **private static final long *serialVersionUID*** = 1L;  
 **public** String **name**;  
 **protected float hp**;  
 Hero(String name, **float** hp){  
 **this**.**name** = name;  
 **this**.**hp** = hp;  
 }  
}

Main.java

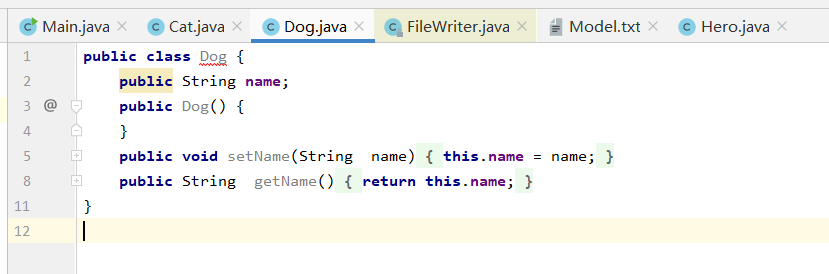
**package** IO;  
**import** fondamental.Hero;  
**import** java.io.\*;  
  
**public class** Main{  
 **public static void** main(String args[]){  
 *//要把Hero对象直接保存在文件上，务必让Hero类实现Serializable接口* Hero h = **new** Hero(**"garen"**, 616);  
 File f = **new** File(**"d:/testb.txt"**);  
 **try**(*//创建对象输出流* 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.***out***.println(h2.**name**);  
 System.***out***.println(h2.**hp**);  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
}

1. System.in练习 （给定模板替换生成类）



**import** java.io.File;  
**import** java.io.FileReader;  
**import** java.io.FileWriter;  
**import** java.util.Scanner;  
  
**public class** Main{  
 **public static void** main(String args[]){  
 *//1. 接收输入* Scanner s = **new** Scanner(System.***in***);  
 System.***out***.println(**"请输入类的名称："**);  
 String className = s.nextLine();  
 System.***out***.println(**"请输入属性的类型："**);  
 String type = s.nextLine();  
 System.***out***.println(**"请输入类型的名称："**);  
 String property = s.nextLine();  
 String Uproperty = *toUpperFirstLetter*(property);  
  
 *//2. 读取模板文件* File modelFile = **new** File(**"D:\\java\_fondamental\\src\\IO\\Model.txt"**);  
 String modelContent = **null**;  
 **try**(FileReader fr = **new** FileReader(modelFile)){  
 **char** cs[] = **new char**[(**int**) modelFile.length()];  
 fr.read(cs);  
 modelContent = **new** String(cs);  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
  
 *//3.替换* String fileContent = modelContent.replaceAll(**"@class@"**, className);  
 fileContent = fileContent.replaceAll(**"@type@"**, type);  
 fileContent = fileContent.replaceAll(**"@property@"**, property);  
 fileContent = fileContent.replaceAll(**"@Uproperty@"**, Uproperty);  
 String fileName = className + **".java"**;  
  
 *//4.替换后的内容* System.***out***.println(**"替换后的内容："**);  
 System.***out***.println(fileContent);  
 File file = **new** File(**"D:\\java\_fondamental\\src\\IO"**, fileName);  
 **try**(FileWriter fw = **new** FileWriter(file)){  
 fw.write(fileContent);  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
  
 System.***out***.println(**"文件保存在:"** + file.getAbsolutePath());  
 }  
  
 **private static** String toUpperFirstLetter(String property) {  
 **char** upperCaseFirst = Character.*toUpperCase*(property.charAt(0));  
 String rest = property.substring(1);  
 **return** upperCaseFirst + rest;  
 }  
}

运行后：



1. 文件练习（很实用!!）
2. 复制文件

实现复制源文件srcFile到目标文件destFile

public static void copyFile(String srcFile, String destFile){}

需要留意的是，read会返回实际的读取数量，有可能实际的读取数量小于缓冲的大小，那么把缓冲中的数据写出到目标文件的时候，就只应该写出部分数据。

**import** java.io.File;  
**import** java.io.FileInputStream;  
**import** java.io.FileOutputStream;  
**import** java.io.IOException;  
  
**public class** CopyFileTest {  
 */\*\*  
 \** ***@param srcPath*** *源文件  
 \** ***@param destPath*** *目标文件  
 \*/* **public static void** copyFile(String srcPath, String destPath){  
 File srcFile = **new** File(srcPath);  
 File destFile = **new** File(destPath);  
  
 *//缓存区，一次性读取1024个字节* **byte**[] buffer = **new byte**[1024];  
 **try**(  
 FileInputStream fis = **new** FileInputStream(srcFile);  
 FileOutputStream fos = **new** FileOutputStream(destFile);  
 ){  
  
 **while**(**true**){  
 *//实际读取的长度是 actuallyReaded, 有可能小于1024* **int** actuallyReaded = fis.read(buffer);  
 *//-1表示没有可读的内容了* **if**(actuallyReaded == -1)  
 **break**;  
 fos.write(buffer, 0, actuallyReaded);  
 fos.flush();  
 }  
 }**catch** (IOException e){  
 e.printStackTrace();  
 }  
 }  
  
 **public static void** main(String args[]){  
 *copyFile*(**"d:/testio.txt"**, **"d:/testio2.txt"**);  
 }  
}

1. 复制文件夹

把源文件夹下所有的文件 复制到目标文件夹下(包括子文件夹)

public static void copyFolder(String srcFolder, String destFolder){}

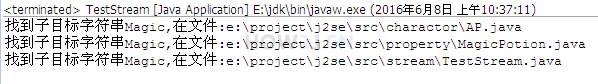
**public class** CopyFileTest {  
 */\*\*  
 \** ***@param srcPath*** *源文件  
 \** ***@param destPath*** *目标文件  
 \*/* **public static void** copyFile(String srcPath, String destPath){  
 File srcFile = **new** File(srcPath);  
 File destFile = **new** File(destPath);  
 *//缓存区，一次性读取1024个字节* **byte**[] buffer = **new byte**[1024];  
 **try**(  
 FileInputStream fis = **new** FileInputStream(srcFile);  
 FileOutputStream fos = **new** FileOutputStream(destFile);  
 ){  
  
 **while**(**true**){  
 *//实际读取的长度是 actuallyReaded, 有可能小于1024* **int** actuallyReaded = fis.read(buffer);  
 *//-1表示没有可读的内容了* **if**(actuallyReaded == -1)  
 **break**;  
 fos.write(buffer, 0, actuallyReaded);  
 fos.flush();  
 }  
 }**catch** (IOException e){  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \*  
 \** ***@param srcPath*** *源文件夹  
 \** ***@param destPath*** *目标文件夹  
 \*/* **public static void** copyFolder(String srcPath, String destPath){  
 File srcFolder = **new** File(srcPath);  
 File destFolder = **new** File(destPath);  
 *//源文件夹不存在* **if**(!srcFolder.exists()) **return**;  
 *//源文件夹不是一个文件夹* **if**(!srcFolder.isDirectory()) **return**;  
 *//目标文件夹不存在* **if**(!destFolder.exists()) destFolder.mkdirs();  
 *//遍历文件夹* File[] files = srcFolder.listFiles();  
 **for**(File srcFile : files){  
 *//如果是文件，就复制* **if**(srcFile.isFile()){  
 File newDestFile = **new** File(destFolder , srcFile.getName());  
 *copyFile*(srcFile.getAbsolutePath(),newDestFile.getAbsolutePath());  
 }  
 *//如果是文件夹，就递归* **if**(srcFile.isDirectory()){  
 File newDestFolder = **new** File(destFolder, srcFile.getName());  
 *copyFolder*(srcFile.getAbsolutePath(), newDestFolder.getAbsolutePath());  
 }  
 }  
 }  
 **public static void** main(String args[]){  
 *copyFolder*(**"d:/JAVAIOTest"**, **"d:/TestFile"**);  
 }  
}

1. 查找文件内容

public static void search(File folder, String search);

要求：

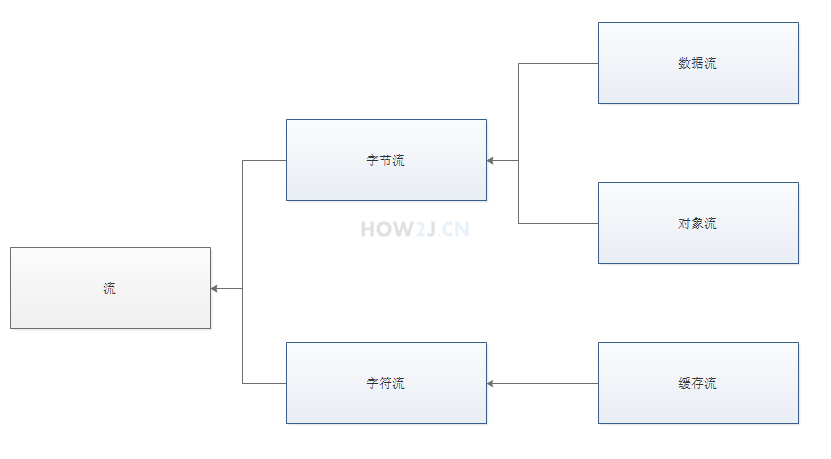
假设你的项目目录是 e:/project，遍历这个目录下所有的java文件（包括子文件夹），找出文件内容包括 ****Magic****的那些文件，并打印出来。



代码：

**package** IO;  
  
**import** java.io.File;  
**import** java.io.FileReader;  
**import** java.io.IOException;  
  
**public class** SearchFileContent {  
 */\*\*  
 \** ***@param file*** *查找的目录  
 \** ***@param search*** *查找的字符串  
 \*/* **public static void** search(File file, String search){  
 **if**(file.isFile()){  
 **if**(file.getName().toLowerCase().endsWith(**".txt"**)){  
 String fileContent = *readFileContent*(file);  
 **if**(fileContent.contains(search)){  
 System.***out***.printf(**"找到子目标字符串%s,在文件:%s%n"**,search,file);  
 }  
 }  
 }  
 **if**(file.isDirectory()){  
 File fs[] = file.listFiles();  
 **for**(File f : fs){  
 *search*(f, search);  
 }  
 }  
 }  
  
 **public static** String readFileContent(File file){  
 **try**(FileReader fr = **new** FileReader(file)){  
 **char** all[] = **new char**[(**int**)file.length()];  
 fr.read(all);  
 **return new** String(all);  
 }**catch** (IOException e){  
 e.printStackTrace();  
 **return null**;  
 }  
 }  
  
 **public static void** main(String args[]){  
 File folder = **new** File(**"d:/JAVAIOTest"**);  
 *search*(folder, **"test"**);  
 }  
}

1. lO流关系图



1. ArrayList存放对象 （相当于C++ vector 如果不指定泛型什么类型都可以放）

**public class** TestCollection {  
 **public static void** main(String args[]){  
 ArrayList heros = **new** ArrayList();  
 heros.add(**new** Hero(**"garen"**, 626));  
 System.***out***.println(heros.size());  
 heros.add(**new** Hero(**"Temmo"**, 123));  
 System.***out***.println(heros.size());  
 Hero h = (Hero)heros.get(1);  
 System.***out***.println(h.**name**);  
 }  
}

1. add 增加
2. contains 通过方法contains 判断一个对象是否在容器中  
   判断标准： 是否是同一个对象，而不是name是否相同
3. get 获取指定位置的对象
4. indexOf 获取对象所处的位置 （没有返回-1）
5. remove 删除

remove用于把对象从ArrayList中删除  
remove可以根据下标删除ArrayList的元素

heros.remove(2);

也可以根据对象删除

heros.remove(specialHero);

1. set set用于替换指定位置的元素

System.out.println("把下标是5的元素，替换为\"hero 5\"");

heros.set(5, **new** Hero("hero 5"));

1. size 用于获取ArrayList的大小
2. toArray 转换为数组
3. addAll 把另一个容器所有对象都加进来
4. clear 清空

ArrayList实现了接口List

常见的写法会把引用声明为接口List类型

List heros = **new** ArrayList();

1. 遍历
2. for

**for** (**int** i = 0; i < heros.size(); i++) {  
 Hero h = heros.get(i);  
 System.***out***.println(h);  
}

1. iterator 迭代器遍历

*//第一种写法*Iterator<Hero> it = heros.iterator();  
**while**(it.hasNext()){  
 Hero h = it.next();  
 System.***out***.println(h.**name**);  
}  
*//第二种 迭代器for写法***for**(Iterator<Hero> it2 = heros.iterator(); it2.hasNext();){  
 Hero hero = it2.next();  
 System.***out***.println(hero.**name**);  
}

1. for: 用增强型for循环

**for**(Hero h : heros){  
 System.***out***.println(h.**name**);  
}

1. 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

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

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

1. LinkedList
2. 实现了List接口，有add,remove,contains等等方法
3. 实现了**双向链表结构**deque

声明：LinkedList<Hero> ll = new LinkedList<>();

在最后插入：ll.addLast(new Hero(“name”))

在前面插入：ll.addFirst(new Hero(“name”))

获取最后：ll.getLast()

获取最前：ll.getFrist()

删除最前：ll.removeFirst()

删除最后：ll.removeLast()

1. 实现了Queue接口（队列）

offer 在最后添加元素

poll 取出第一个元素

peek 查看第一个元素

1. 二叉排序

**package** Collection;  
  
**public class** Node {  
 **public** Node **leftNode**;  
 **public** Node **rightNode**;  
 **public** Object **value**;  
  
 **public void** add(Object v){  
 **if**(**value** == **null**) **value** = v;  
 **else**{  
 **if**((Integer) v - ((Integer)**value**) <= 0){  
 **if**(**leftNode** == **null**) **leftNode** = **new** Node();  
 **leftNode**.add(v);  
 }**else**{  
 **if**(**rightNode** == **null**) **rightNode** = **new** Node();  
 **rightNode**.add(v);  
 }  
 }  
 }  
  
 **public static void** Shownum(Node now){  
 **if**(now.**leftNode** != **null**) *Shownum*(now.**leftNode**);  
 System.***out***.println((Integer) now.**value**);  
 **if**(now.**rightNode** != **null**) *Shownum*(now.**rightNode**);  
 }  
   
 **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** num : randoms){  
 roots.add(num);  
 }  
 *Shownum*(roots);  
 }  
}

中序遍历的另一种方式 返回一个排好序的List

 // 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;

    }

1. HashMap

HashMap存储数据的方式是------键值对

声明：HashMap<String,String> distionary = new HashMap<>();

添加：dictionary.put(“a”,”b”);

获取键对应的值：dictionary.get(“a”);

清空：dictionary.clear()

练习：反转key和value

keySet()可以获取所有的key，values()可以获取所有的value

**package** Collection;  
  
**import** java.util.HashMap;  
**import** java.util.Set;  
  
**public class** TestCollection {  
 **public static void** main(String[] args) {  
 HashMap<String,String> map = **new** HashMap<>();  
 map.put(**"adc"**,**"物理英雄"**);  
 map.put(**"apc"**,**"魔法英雄"**);  
 map.put(**"tank"**,**"坦克"**);  
  
 Set<String> keys = map.keySet();  
 HashMap<String,String> tmp = **new** HashMap<>();  
 **for**(String k : keys){  
 String value = map.get(k);  
 tmp.put(value, k);  
 }  
 map.clear();  
 map.putAll(tmp);  
 }  
}

1. HashSet （作为HashMapd的key存在）

元素不能重复

set中的元素，没有顺序

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

names.add("gareen");

遍历：无get()方法，所以遍历需要用到迭代器，或者增强型for循环

HashSet<Integer> st = **new** HashSet<>();  
**for**(**int** i = 0; i < 10; ++ i) st.add(i);  
1.  
**for**(Iterator<Integer> it = st.iterator(); it.hasNext();){  
 Integer i = it.next();  
 System.***out***.print(i);  
}

1. **for**(Integer num : st){  
    System.***out***.print(num);  
   }

其它的Set

HashSet： 无序  
LinkedHashSet： 按照插入顺序  
TreeSet： 从小到大排序

1. Collections 工具类

Collections是一个类，容器的工具类.

List<Integer> num = **new** ArrayList<>();

1. 反转 reverse 1，2，3，4，5->5，4，3，2，1

Collections.*reverse*(num);

1. 混淆 shuffle 1，2，3，4，5->随机

Collections.*shuffle*(num);

1. 排序 sort 随机->1，2，3，4，5

Collections.*sort*(num);

1. 交换 swap

Collections.*swap*(num, 0, 5);

交换两个元素的位置

1. 滚动 rotate 1，2，3，4，5->4，5，1，2，3

Collections.*rotate*(num, 2);

1. 比较器 Comparator

假设Hero有 三个属性 name,hp,damage

当调用Collections.sort(heros)的时候因无发判断按照哪个顺序进行排序出错

这时就需要提供一个Comparator给定如何进行两个对象之间的大小比较

**public static void** main(String args[]){  
 Random r = **new** Random();  
 List<Hero> heros = **new** ArrayList<>();  
 **for**(**int** i = 0; i < 10; ++ i){  
 heros.add(**new** Hero(**"hero "** + i, r.nextInt(100), r.nextInt(100)));  
 }  
 System.***out***.println(heros);  
 *//引入Comparator，指定比较的算法* Comparator<Hero> c = **new** Comparator<Hero>() {  
 @Override  
 **public int** compare(Hero o1, Hero o2) {  
 **if**(o1.**hp** >= o2.**hp**){  
 **return** 1;  
 }**else return** -1;  
 }  
 };  
 Collections.*sort*(heros, c);  
 System.***out***.println(heros);  
}

或者：

Collections.*sort*(heros, **new** Comparator<Hero>() {  
 @Override  
 **public int** compare(Hero o1, Hero o2) {  
 **return** (**int**)(o1.**hp** - o2.**hp**);  
 }  
});

1. 接口Comparable

使Hero类实现Comparable接口

在类里面提供比较算法

Collections.sort就有足够的信息进行排序了，也无需额外提供比较器Comparator

如果返回-1, 就表示当前的更小，否则就是更大

**public class** Hero **implements** Comparable<Hero> {  
 **public** String **name**;  
 **public float hp**;  
 **public int damage**;  
 **public** Hero() {}  
 **public** Hero(String name) {  
 **this**.**name** = name;  
 }  
 **public** String toString() {  
 **return "hero [name="** + **name** + **", hp="** + **hp** + **", damage="** + **damage** + **"]\r\n"**;  
 }  
 **public** Hero(String name, **int** hp, **int** damage) {  
 **this**.**name** = name;  
 **this**.**hp** = hp;  
 **this**.**damage** = damage;  
 }  
 @Override  
 **public int** compareTo(Hero o) {  
 **if**(**damage** < o.**damage**)  
 **return** 1;  
 **else return** -1;  
 }  
}

Collections.*sort*(heros);

练习：自定义顺序的TreeSet

默认情况下，TreeSet中的数据是从小到大排序的，不过TreeSet的构造方法支持传入一个Comparator

**public class** TestCollection{  
 **public static void** main(String args[]){  
 Comparator<Integer> c = **new** Comparator<Integer>() {  
 @Override  
 **public int** compare(Integer o1, Integer o2) {  
 **return** o2 - o1;  
 }  
 };  
 Set<Integer> treeSet = **new** TreeSet<>(c);  
 **for**(**int** i = 0; i < 10; ++ i) treeSet.add(i);  
 System.***out***.println(treeSet);  
 }  
}

输出：[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

24.泛型

以Stack栈为例子， 如果不使用泛型

当需要一个只能放Hero的栈的时候，就需要设计一个HeroStack  
 当需要一个只能放Item的栈的时候，就需要一个ItemStack

若支持泛型的话，设计一个支持泛型的栈MyStack

设计这个类的时候，在类的声明上，加一个<T>,表示该类支持泛型。

T是type的缩写，也可以使用任何其他的合法的变量，比如A,B,X都可以，但是一般约定成俗使用T,代表类型。

**public class** MyStack <T>{  
 LinkedList<T> **stk** = **new** LinkedList<>();  
 **public void** push(T t){  
 **stk**.addLast(t);  
 }  
 **public** T pull(){  
 **return stk**.removeLast();  
 }  
 **public** T peek(){  
 **return stk**.getLast();  
 }  
 **public static void** main(String args[]){  
 MyStack<Hero> heroStack = **new** MyStack<>();  
 heroStack.push(**new** Hero());  
 MyStack<Item> itemStack = **new** MyStack<>();  
 itemStack.push(**new** Item());  
 }  
}

练习：支持泛型的二叉树

把二叉树中的Node类，改造成支持泛型

**public class** Node<T> {  
 **public** Node<T> **left**;  
 **public** Node<T> **right**;  
 **public** T **value**;  
  
 **public void** add(T t){  
 **if**(**null** == **value**)  
 **value** = t;  
 **else**{  
 **if**((Integer)t - ((Integer)**value**) <= 0){  
 **if**(**left** == **null**)  
 **left** = **new** Node<T>();  
 **left**.add(t);  
 }**else**{  
 **if**(**right** == **null**)  
 **right** = **new** Node<T>();  
 **right**.add(t);  
 }  
 }  
 }  
 *//中序遍历所有节点* **public** List<T> values(){  
 List<T> values = **new** ArrayList<>();  
 **if**(**left** != **null**) values.addAll(**left**.values());  
 values.add(**value**);  
 **if**(**right** != **null**) values.addAll(**right**.values());  
 **return** values;  
 }  
  
 **public static void** main(String args[]){  
 **int** random[] = **new int**[] { 67, 7, 30, 73, 10, 0, 78, 81, 10, 74 };  
 Node<Integer> root = **new** Node<>();  
 **for**(**int** i : random){  
 root.add(i);  
 }  
 System.***out***.println(root.values());  
 }  
}

25.通配符

1. ? extends 只能取数据，不能插入

ArrayList heroList<? extends Hero> 表示这是一个Hero泛型或者其子类泛型  
heroList 的泛型可能是Hero  
heroList 的泛型可能是APHero  
heroList 的泛型可能是ADHero  
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的****  
  
但是，不能往里面放东西，因为  
放APHero就不满足<ADHero>  
放ADHero又不满足<APHero>

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

        ArrayList<APHero> apHeroList = **new** ArrayList<APHero>();

        apHeroList.add(**new** APHero());

        ArrayList<? **extends** Hero> heroList = apHeroList;

        //? extends Hero 表示这是一个Hero泛型的子类泛型

        //heroList 的泛型可以是Hero

        //heroList 的泛型可以使APHero

        //heroList 的泛型可以使ADHero

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

        Hero h= heroList.get(0);

        //但是，不能往里面放东西

        heroList.add(**new** ADHero()); //编译错误，因为heroList的泛型 有可能是APHero

    }

1. ? super 只能插入，不能取出

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

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

        ArrayList<? **super** Hero> heroList = **new** ArrayList<Object>();

        //? super Hero 表示 heroList的泛型是Hero或者其父类泛型

        //heroList 的泛型可以是Hero

        //heroList 的泛型可以是Object

        //所以就可以插入Hero

        heroList.add(**new** Hero());

        //也可以插入Hero的子类

        heroList.add(**new** APHero());

        heroList.add(**new** ADHero());

        //但是，不能从里面取数据出来,因为其泛型可能是Object,而Object是强转Hero会失败

        Hero h= heroList.get(0);

    }

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？