Java程序设计

2019春季

彭启民

pengqm@aliyun.com

- 迭代器
  - 可以遍历并选择序列中的对象，而开发人员不需要了解该序列的底层结构。
  - 通常被称为“轻量级”对象，创建代价小。

```
public interface Iterator {
        boolean hasNext();      // cf: hasMoreElements()
        Object next();          // cf: nextElement()
        void remove();          // Optional
    }
```

- 使用Iterator接口的最大好处在于将容器的实现与容器的接口分开，这就意味着你可以使用相同的方法访问不同的容器类，不用关心容器是由什么样的数据结构实现的，即使容器类的具体实现发生了变化，也无需更改迭代器代码。

迭代器

- Iterator
  - 调用容器对象的iterator()方法返回一个Iterator
  - next()获得序列中的下一元素，位置自动向前走一步。
    - 第一次调用Iterator的next()方法时返回序列的第一个元素。
    - 查看与位置的变化紧密的结合在一起
  - hasNext()检查序列中是否还有元素。
  - remove()将迭代器新返回的元素删除。
- Iterator是Java迭代器最简单实现，只能单向移动

- 例：

```
Vector c;
……
Iterator it=c.iterator();
while(it.hasNext())
{
   Object obj=it.next();
   //do something with obj
}
```

- 在迭代过程中不能用非remove的方法增减元素。
- Enumeration（枚举器）
  - 在Java发展中，Enumeration逐渐被Iterator取代，新开发的代码一般推荐使用后者。
  - boolean hasMoreElements()
  - Object nextElement()

- 错误用法：//为什么出错?

```
it.remove();
it.remove();
```

- 正确用法：

```
it.remove();
it.next();
it.remove();
```

- Foreach类型的循环语句（1.5版本）

遍历数组，以前的写法：

```
for ( Iterator iterator = list.iterator(); iterator.hasNext(); )
{
Integer n = (Integer)iterator.next();
...
}//for
```

在1.5版本中可以写为：

```
for ( Integer n : list )
{
...
}//for
```

- 使用for-each循环访问集合元素

```
static void printAll(Collection c)
  {
     for (Object e:c )
          System.out.println(e);
  }
```

- ListIterator
  - 为List设计
  - 可以从两个方向遍历List
    - 增加了Object previous()和boolean hasPrevious()方法
  - 可以从List中插入和删除元素。

```
public interface ListIterator extends Iterator {
 // from Iterator (forward iteration)
     boolean hasNext();
     Object next();
// backward iteration:
     boolean hasPrevious();
     Object previous();

     int  nextIndex();  // == cursor position == pos of next() object
     int  previousIndex();  //  == nextIndex() – 1 = pos of previous() object
                                //  ; == -1 if cursor = 0;
     void remove();          // Optional
     void set(Object o);     // Optional
     void add(Object o);     // Optional
}
```

- Map接口基本方法
  - Object get(Object key)
  - Object put(Object key,Object balue)
  - Set keySet()
  - Set entrySet()

- The standard idiom for iterating over the keys in a Map:

```
for (Iterator i = m.keySet().iterator();  i.hasNext(); ) {
    System.out.println(i.next());
    if(no-good(…)) i.remove() ;  }   // support removal from the back Map
```

- Iterating over key-value pairs

```
for (Iterator i=m.entrySet().iterator(); i.hasNext(); ) {
    Map.Entry e = (Map.Entry) i.next();
    System.out.println(e.getKey() + ": " + e.getValue());       }
```

- 容器Vs.数组
  - 数组是JAVA语言内置的数据类型，是一个线性的序列，可以快速访问其元素。
  - 一个数组创建后容量就固定了，而且在其生命周期里是不能改变的。
  - JAVA对数组会做边界检查的，越界访问时，会抛出RuntimeException，以牺牲效率为代价。

- 集合对象会自动扩展，以容纳添加到其中的所有对象。
- 集合中只能容纳对象。

- 与容器类相比，数组会在编译的时候作类型检查，从而防止你插入错误类型的对象，或者在提取对象的时候把对象的类型给搞错了，JAVA在编译和运行的时候都能阻止你将一个不恰当的消息传给对象。
- 数组效率高于容器类
  - 有些容器类是基于数组实现的，比如ArrayList。
- 数组可以持有primitives。

- 容器类存放于java.util包中。
- 容器类存放的都是对象的引用，而非对象本身，出于表达上的便利，我们称集合中的对象就是指集合中对象的引用（reference)。

- JAVA里面提供的容器处理对象的时候就好像这些这些对象都没有自己的类型一样，容器将它所含的元素都看成是JAVA中所有类的根类Object类型的，这样我们只需创建一种容器，就能把所有的类型的对象全部放进去。

Java的8种包装类

- 为解决8种基本数据类型不能当成Object类型使用的问题，Java提供了包装类
- 可用包装类的xxxValue()方法包装的基本类型变量

byte ----------> Byte

short ----------> Short

int ----------> Integer

long ----------> Long

char ----------> Character

float ----------> Float

double ----------> Double

boolean ----------> Boolean

Integer a = new Integer(“9"); int c=a.intValue();

- 自动包装和解包(1.5版本)

```
Integer i=6;Object j=9;
int a=i; int b = (Integer)j;//int b=j;编译会报错，原因？
```

向一个ArrayList中加入一个整数，以前：

```
List list = new ArrayList();
list.add( new Integer( 10 ) );
```

新：

```
list.add( 10 );
```

- 提示：
  - 要根据不同问题的需要选择合适的容器，以此来达到功能的要求和效率的最优尽量返回接口而非实际的类型
  - 单线程环境中，或者访问仅仅在一个线程中进行，考虑非同步类，其效率较高
  - 多个线程可能同时操作一个类，应使用同步类
  - 对哈希表的操作，作为key的对象要正确复写equals和hashCode方法

异常及处理

```
public class Exception1
{
    public static void main(String []args){
        int a[]={1,2,3,4,5},sum=0;
        for (int i=0;i<=5;i++) sum=sum+a[i];
        System.out.println("sum"+sum);
        System.out.println("Successfuly!");
    }
}
```

异常

- Java异常是一个描述在代码段中发生的异常（也就是出错）情况的对象。
- 当异常情况发生时，一个代表该异常的对象被创建并且在导致该错误的方法中被抛出（throw）。
- 该方法可以选择自己处理异常或传递该异常。这两种情况下，该异常都将被捕获（catch）并被处理。

两种异常

- 运时异常/非检查异常（unckecked exception）
  - Error 和 RuntimeException 及其子类
  - Java编译器不检查，当程序中可能出现这类异常时，即使没有用try...catch语句捕获它，也没有用throws字句声明抛出它，还是会编译通过。
  - 导致这种异常的原因通常是由于执行了错误的操作。一旦出现错误，建议让程序终止。修正代码，而不是异常处理器处理
    - 除0错误ArithmeticException
    - 错误的强制类型转换错误ClassCastException
    - 数组索引越界ArrayIndexOutOfBoundsException
    - 使用了空对象NullPointerException

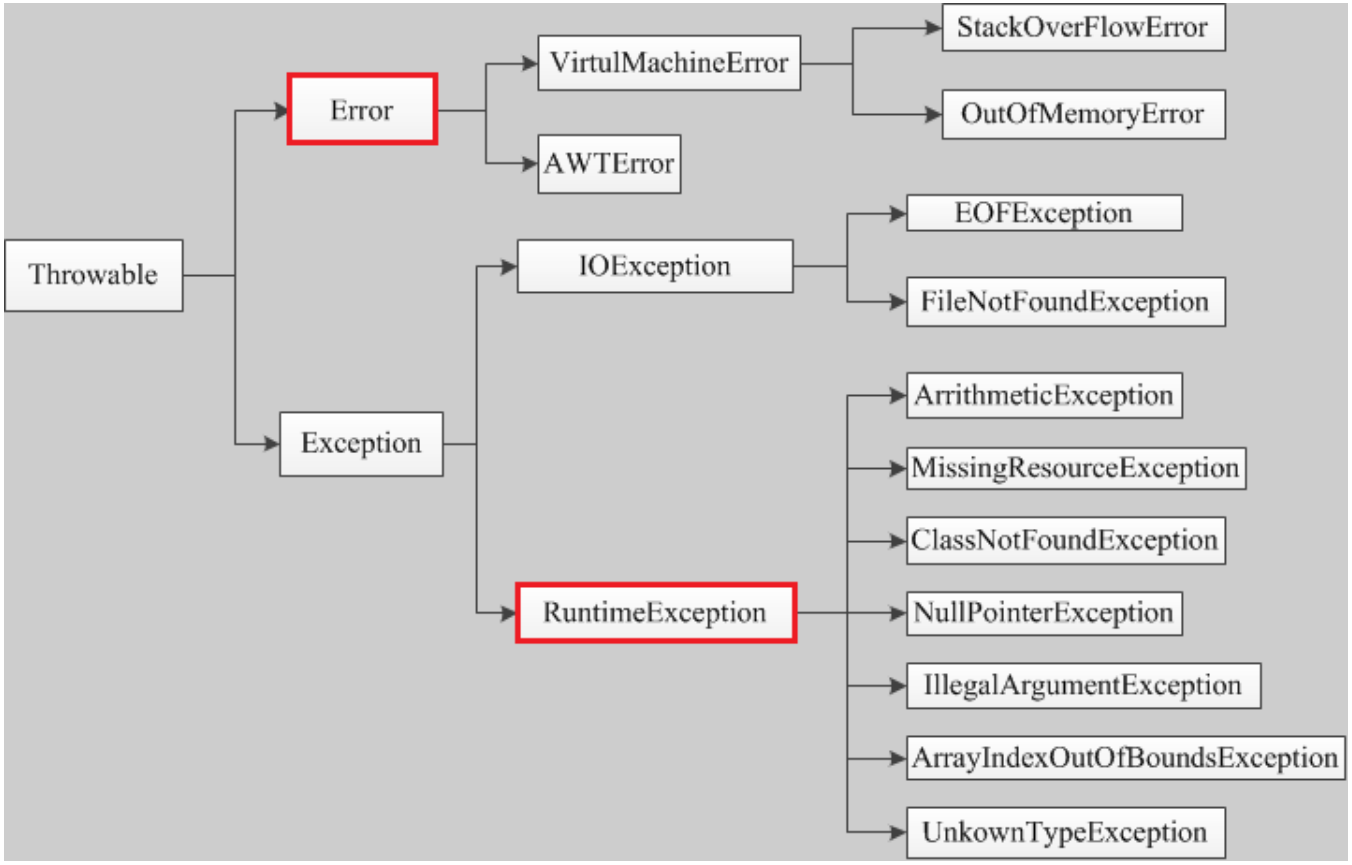
两种异常

- 受检查异常（checked exception）
  - 除了Error 和 RuntimeException外，其他异常都属于受检查异常，一般是由程序的运行环境导致
    - SQLException
    - IOException,
    - ClassNotFoundException
  - 必须用try...catch捕获处理，或者用throws语句声明抛出，否则编译不会通过。
  - 对于受检查异常可以由程序处理，如果抛出异常的方法本身不处理或者不能处理它，那么方法的调用者就必须去处理该异常，否则调用会出错，编译也无法通过。

说明：两种异常都可通过程序来捕获处理。但对于运行异常不建议用try...catch...捕获处理，应通过调试尽量避免。

Throwable类

- Java异常指一个继承Throwable类的实例
  - 两个子类Exception和Error在java.lang包中。



- Error类
  - 被认为是不能恢复的严重错误，如系统内部错误和资源耗尽错误等。不应该抛出这种类型的错误，而是让程序中断。
- Exception类
  - 定义可能遇到的轻微错误，分为派生自RuntimeException类的异常和非派生自RuntimeException类的异常。这时，可以写代码来处理异常并继续程序执行，而不是让程序中断。

- 区分RuntimeException和非RuntimeException
  - 编程错误导致RuntimeException，如被零除、数组越界访问、空指针访问等
  - 其他异常则是由于意外情况而发生的，如试图读取文件结尾以后的数据、试图打开错误的URL、试图根据并不代表已存在类的字符串来查找Class对象等。

空指针异常（NullPointerException)

```
String str=null;
System.out.println(str.length());
```

数组下标越界(ArrayIndexOutOfBoundsException)

```
int[] ary={1,2,3}
for{int i=0;i<=3;i++){
  System.out.println(ary[i]);
}
```

算数异常（ArithmeticException）

```
int one=12;
int two=0;
System.out.println(one/two);
```

类型转换异常（ClassCastException)

```
class Animal{
}
class Dog extends Animal{
}
class Cat extends Animal{
}
public class Test{
  public static void main(String[] args){
    Animal a1=new Dog();
    Animal a2=new Cat();
    Dog d1=(Dog)a1;
    Dog d2=(Dog)a2;
  }
}
```

异常处理

- 系统自动处理
- 使用try~catch~finally语句
- 使用throw语句直接抛出异常（抛出异常）或使用throws语句间接抛出异常（声明异常）。
  - finally为了完成执行的代码而设计的，主要是为了程序的健壮性和完整性，无论有没有异常发生都执行代码。
  - finally块通常用来做资源释放操作：关闭文件，关闭数据库连接等等。良好的编程习惯：在try块中打开资源，在finally块中清理释放这些资源。

语法结构

- try~catch~finally语句

```
 try
{        可能产生异常的代码段;                    }
catch（异常类名1 对象名1）
    {    处理语句组1;                  }
catch（异常类名2 对象名2）
    {    处理语句组2;   }
  ……
finally
{
    最终处理语句;
}
```

异常冒泡与链化

Java终结式异常处理模式（termination model of exception handling）：执行流恢复到处理了异常的catch块后接着执行

```
public class Exception1
{
    public static void main(String []args){
        int a[]={1,2,3,4,5},sum=0;
        try{
                for (int i=0;i<=5;i++) sum=sum+a[i];
        }catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("数组越界");
        }
        System.out.println("sum"+sum);
        System.out.println("Successfuly!");
    }
}
```

- throw语句

  用来明确地抛出一个异常，

  throw 异常;

  然后在包含它的所有try块中从内向外寻找与其匹配的catch语句块。

```
public class ThrowDemo{
   public static void main(String[] args){
      try{
          double d = 100 / 0.0;
          System.out.println(“浮点数除零: ”+d);
          if(String.valueOf(d).equals(“Infinity”)) throw new
    ArithmeticException(“除零异常”);
      }
      catch(Arithmetic e){
         System.out.println(e);
     }
  }
}
```

Output like this:

浮点数除零: Infinity

Java.lang.ArithmeticException:除零异常

- throws语句
  - 如果一个方法可能导致异常但不处理它，此时可在方法声明中包含 throws 子句，发生了异常，由调用者处理。
  - throws列举方法可能引发的所有异常。

  例：

  private static void throwsTest(参数表) throws 逗号间隔的异常表 { 方法体 }

public void throwsTest() throws AException,Bexception,CException{ ……}

```
public class MyException {
    public static void main(String[] args){
        MyException e = new MyException();
        SimpleException se = new SimpleException();
        try {
            e.a();
        } catch (SimpleException se) {
            se.printStackTrace();
        }
    }

    public void a() throws SimpleException{
        throw new  SimpleException();
    }
}

class SimpleException extends Exception { }
```

```
SimpleException
    at MyException.a(MyException.java:15)
    at MyException.main(MyException.java:8)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.ja
va:43)
    at java.lang.reflect.Method.invoke(Method.java:606)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)

Process finished with exit code 0
```

try,catch,finally的各种组合用法

- try+catch
  运行try块，有异常抛出则转到catch块去处理，然后执行catch块后面的语句
- try+catch+finally
  运行try块，有异常抛出则转到catch块执行完毕后，执行finally块的代码，再执行finally块后面的代码。
  如果没有异常抛出，执行完try块后，转去执行finally块的代码。然后执行finally块后面的语句
- try+finally
  运行try块，如果有异常抛出，转向执行finally块的代码，而finally块后面的代码不会被执行！（因为没有处理异常，所以遇到异常后，执行完finally后，方法就已抛出异常的方式退出）。
  注意：这种方式由于没有捕获异常，所以要在方法后面声明抛出异常。

创建自己的异常

- 内置异常不可能始终足以捕获所有错误，因此需要用户自定义的异常类
- 用户自定义的异常类应为 Exception 类（或者Exception 类的子类）的子类
- 创建的任何用户自定义的异常类都可以获得 Throwable类定义的方法

断言

断言Assertion

- 预期程序中应该处于何种状态（JDK1.4）
  - 例：在某个时间点上判定某个变量必然为某值
- 两种结果
  - 预期结果与实际执行结果相同，断然成立，否则断言不成立

- 语法

  assert boolean_expression;

  表达式为真时不做什么，为假时会发生 java.lang.AssertionError

  assert boolean_expression: detail_expression;

  表达式为真时不做什么，为假时会显示 detail_expression的值

- 对内部不变量的判断
- 对控制流程不变量的判断

```
Public class AssertionDemo{
   public static void main(String[] args){
      if(argc.length > 0){
         System.out.println(argc[0]);
      }
      else {
         assert argc.length == 0;
        // assert argc.length == 0: “没有参数”;
        System.out.println(“请输入参数”);
     }
  }
}
```

- java –ea AssertionDemo
- java –enableassertions AssertionDemo

- 异常对应非预期的错误，断言是正常预期
- 不能当作if之类的判断式使用
- 不应被当作程序执行流程的一部分

输入输出

- 在Java中，应用程序所需要读入的数据和写出的数据是通过I/O操作实现的
- 读写数据的源或目的包括文件、内存、网络连接等，其中，最常用的是文件。

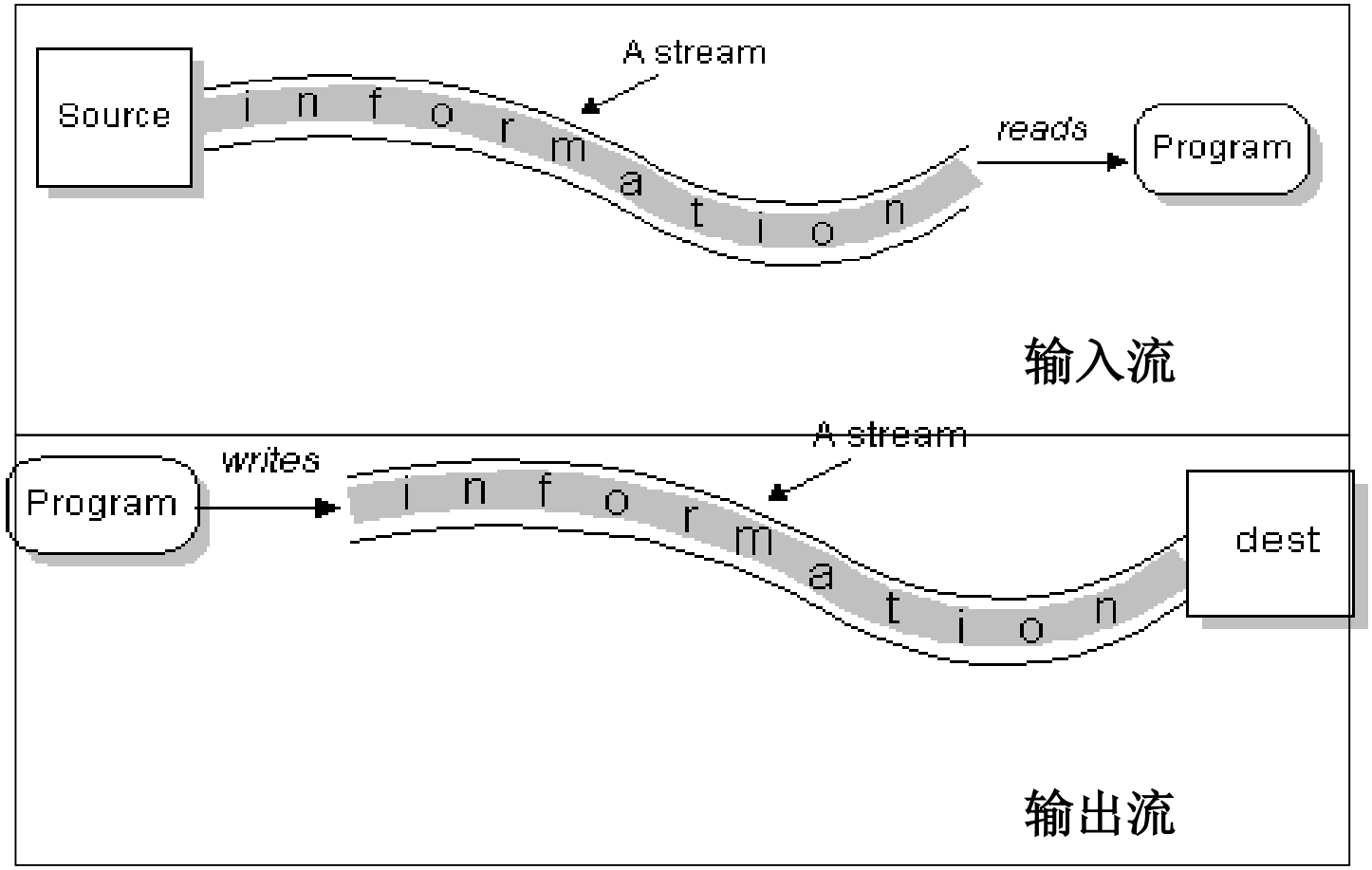
流

- Java用流的观念来管理输入输出
- Java语言的输入输出功能强大而灵活
- 需要包装许多不同的对象
  - 标准输入输出
  - 文件的操作
  - 网络数据流
  - 字符串流
  - 对象流
  - zip文件流

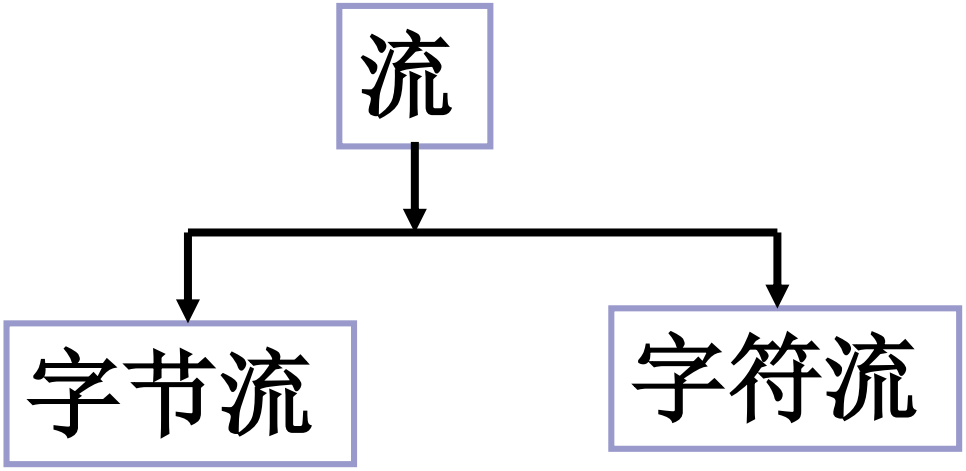
- Node Stream ：从特定源如磁盘文件或内存某区域进行读或写入
- Filter Steam：使用其它的流作为输入源或输出目的地。

- 数据流
  - 可被顺序访问的无限长的字符序列
    - 是从源到目的的字节的有序序列，先进先出
  - 在java中有关流的操作使用io包

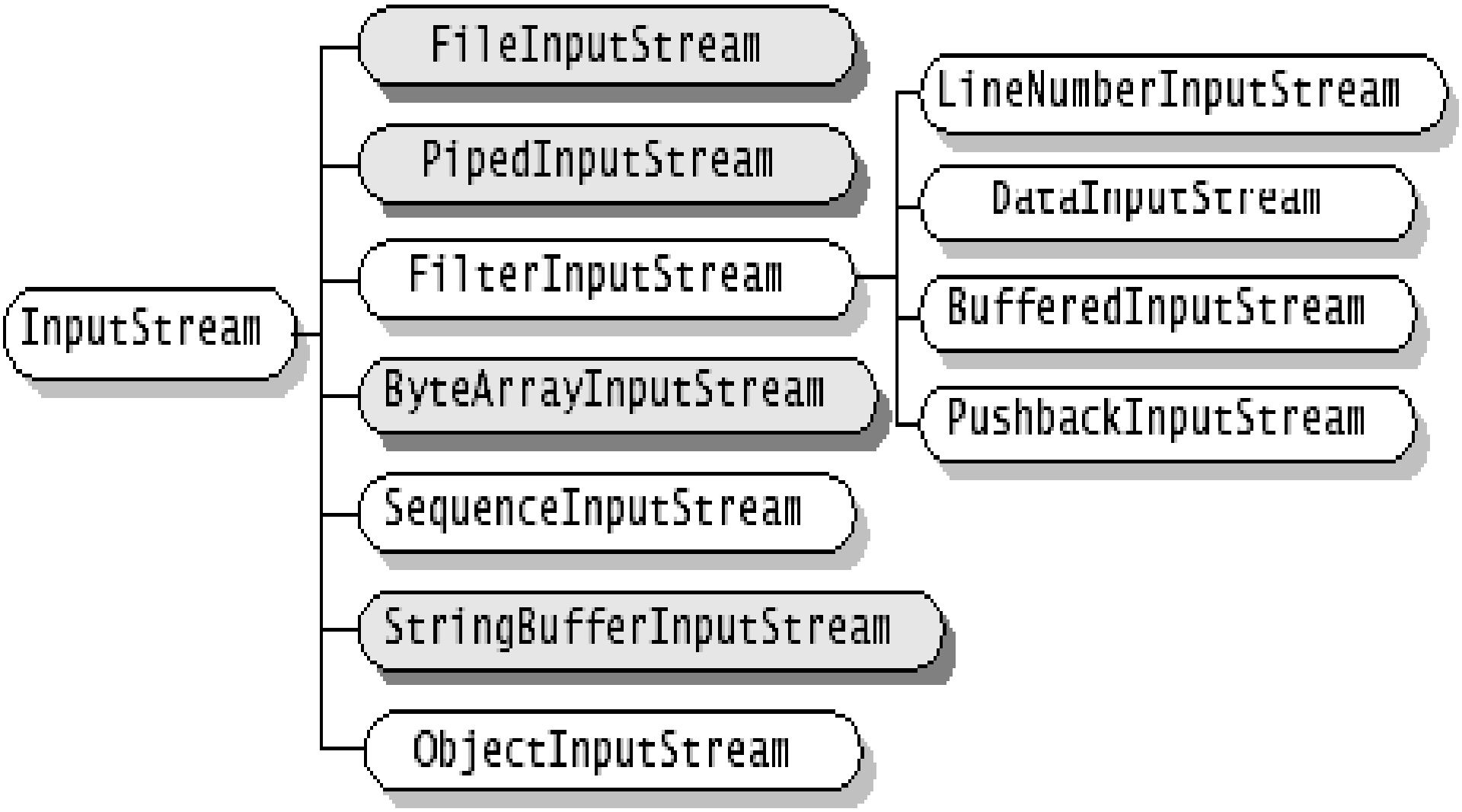
```
import java.io.*;
```


Source
A stream
i n f o r m a t i o n
reads
Program
输入流
Program
writes
A stream
i n f o r m a t i o n
dest
输出流

- 流式I/O类根据操作的数据类型（16位字符或字节）分成两个层次体系



- 字节流
  - InputStream
  - OutputStream
- 字符流
  - Reader
  - Writer


InputStream
FileInputStream
PipedInputStream
FilterInputStream
ByteArrayInputStream
SequenceInputStream
StringBufferInputStream
ObjectInputStream
LineNumberInputStream
DataInputStream
BufferedInputStream
PushbackInputStream

- InputStream
  - 三个基本read()方法

    int read()　　//读一个字节返回

    int read(byte[ ]) // 将数据读入byte[], 返回读的字节数

    int read(byte[], int offset, int length)
  - 其它方法

    void close( ) //关闭流。自顶向下关闭Filter stream

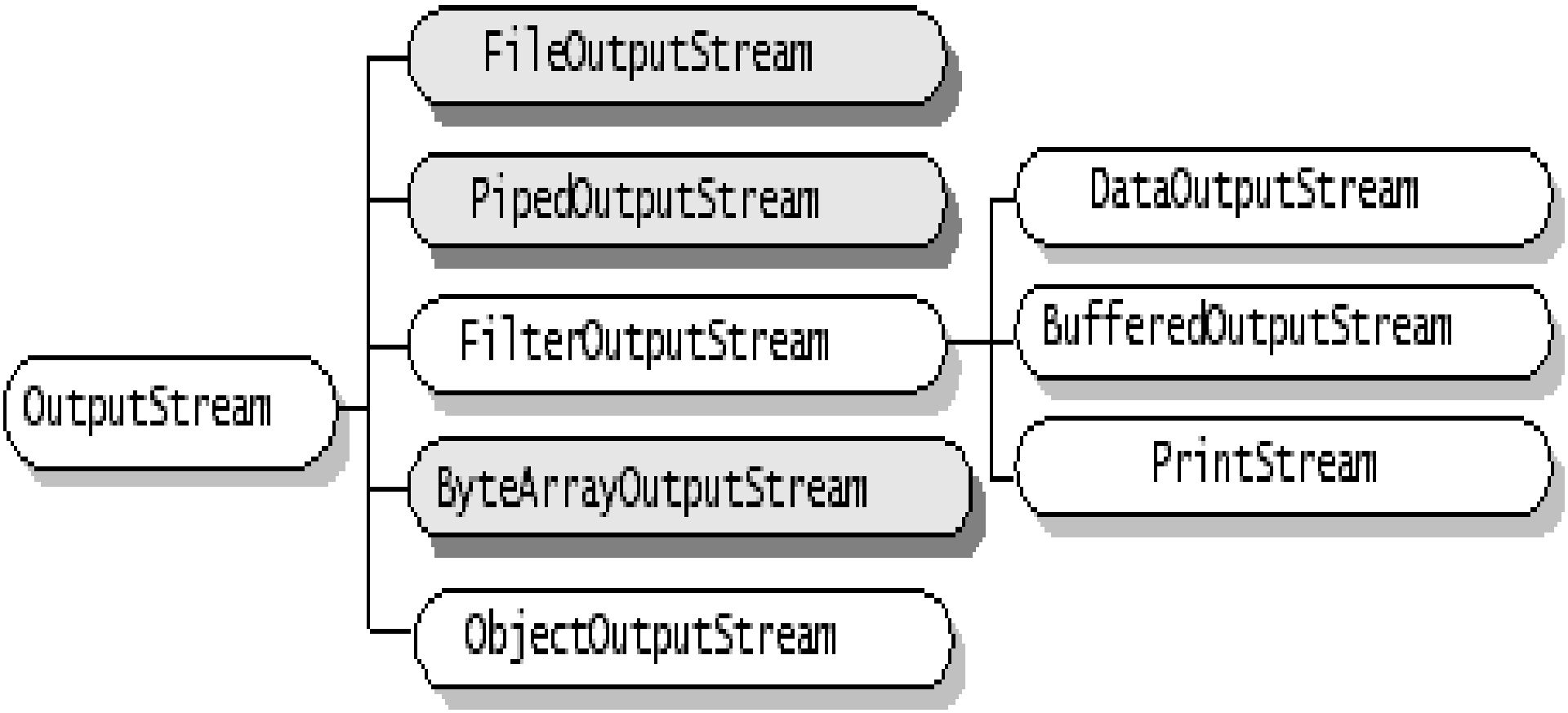
    int available() //返回未读的字节数

    long skip(long n)　　// 跳过n个字节

    boolean markSupported( ) //测试打开的流是否支持书签

    void mark(int) //标记当前流，并建立int大小缓冲区

    void reset( ) // 返回标签出


OutputStream
FileOutputStream
PipedOutputStream
FilterOutputStream
ByteArrayOutputStream
ObjectOutputStream
DataOutputStream
BufferedOutputStream
PrintStream

- OutputStream
  - 三个基本的write( )方法

    void write(int) // 写一个字节

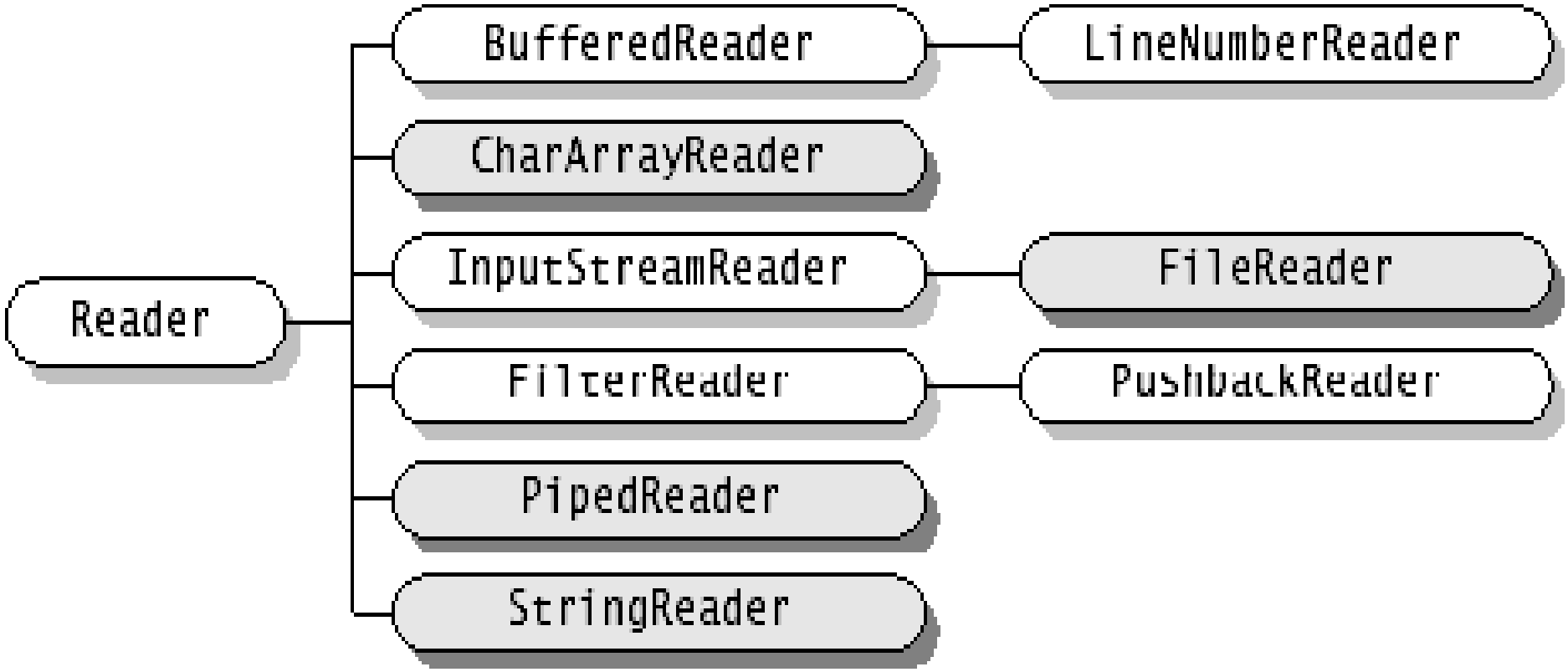
    void write(byte[ ]) // 写一个字节数组

    void write(byte[ ], int offset, int length)
  - 其它方法

    void close( )

    void flush( ) // 刷新缓冲区强行写

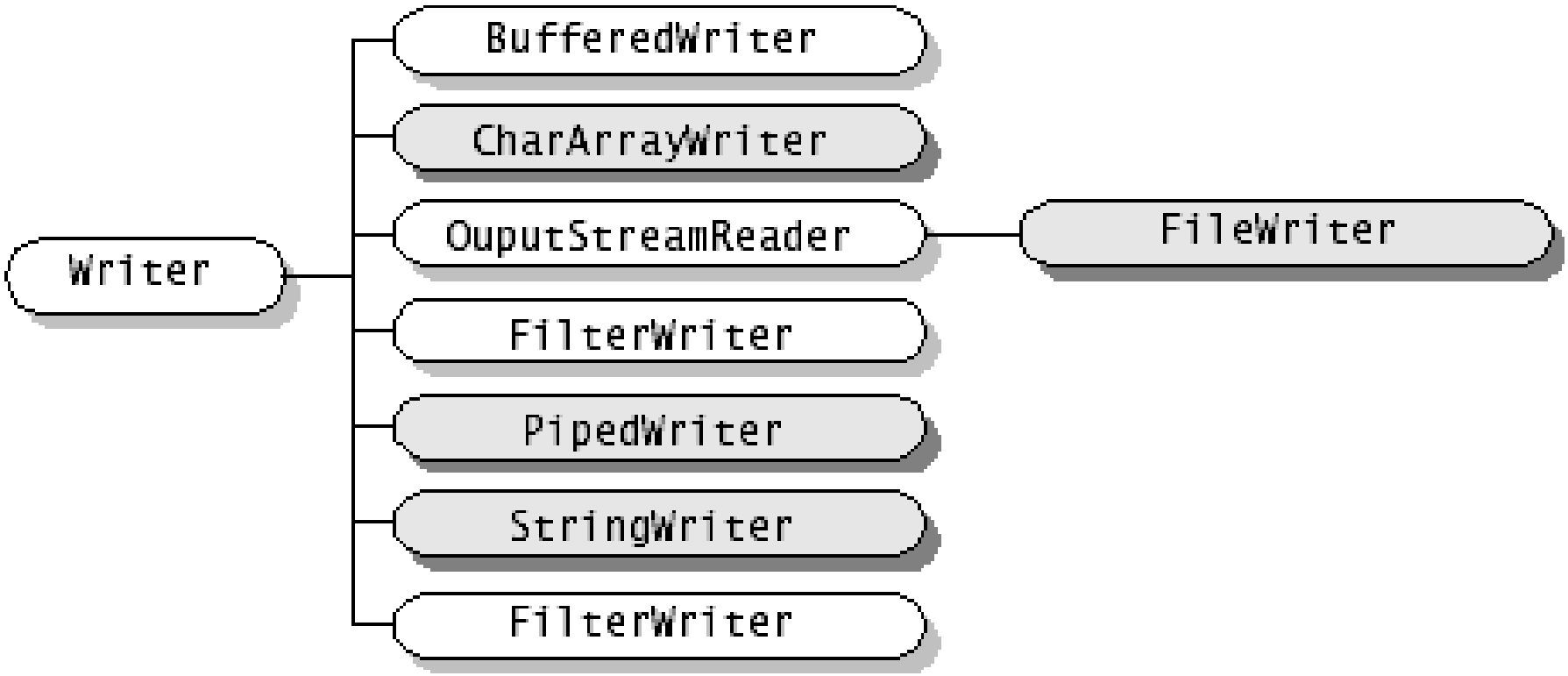
- 字符流
  - Reader和Writer是字符流的两个**抽象超类**。
  - Reader和Writer 类实现字节和字符间的自动转换。
  - 每一个核心输入、输出流，都有相应的Reader和Writer版本。


Reader
BufferedReader
LineNumberReader
CharArrayReader
InputStreamReader
FileReader
FilterReader
PushbackReader
PipedReader
StringReader

- Reader的基本方法

  int read()；//读单个字符
  int read(char cbuf[])；//读字符放入数组中
  int read(char cbuf[], int offset, int length)；
  //读字符放入数组的指定位置

  void close( ) //关闭流。
  long skip(long n)　// 跳过n个字符
  boolean markSupported( ) //测试打开的流是否支持书签
  void mark(int) //标记当前流，并建立int大小缓冲区
  void reset( ) // 返回标签出
  boolean ready() //测试当前流是否准备好进行读


Writer
BufferedWriter
CharArrayWriter
OuputStreamReader
FilterWriter
PipedWriter
StringWriter
FilterWriter
FileWriter

- Writer的基本方法

```
int write(int c) ; // 写单个字符
int write(char cbuf[]) ;// 写字符数组
int write(char cbuf[], int offset, int length) ;
int write(String str) ;
int write(String str, int offset, int length) ;

void close( )
void flush( ) // 强行写
```

字节流Vs.字符流

- 所有的流——InputStream、 OutputStream、Reader、 Writer 在创建时自动打开
- 程序中可以调用close方法关闭流，否则Java运行环境的垃圾收集器将隐含将流关闭。

- 字节流
  - 适用于各类文件
  - 每次读写8位字节
  - 效率较低
- 字符流
  - 适用于16位的字符文件
  - 每次读写16位字符
  - 效率较高

- InputStream与Reader操作相似，数据类型不同
  - InputStreamReader是InputStream和Reader之间的桥梁
- OutputStream与 Writer操作相似，数据类型不同

基本文件操作

```
import java.io.*;

public class FileDemo{
        public static void main(String[] args){
                try{
                        FileWriter out = new FileWriter("D:\\test.txt");
                        //FileReader in = new FileReader("D:\\test.txt");
                        out.write(“基本文件操作演示");
                        out.close();
                        //char[] s = new char[5];
                        //in.read(s);
                        //System.out.println(s);
                        //in.close();
                }catch(Exception e){
                        System.out.println("error");
                }
        }
}
```

用Buffer改进文件操作

- BufferedReader
  - 缓存字符以高效读取字符串，数组和文本行
- 与BufferedWriter
  - 缓存字符以高效写入字符串，数组和文本行

```
import java.io.*;
public class FileDemo1{
    public static void main(String[] args){
            try{
                BufferedWriter out = new BufferedWriter(new
FileWriter("D:\\test.txt"));
                BufferedReader in = new BufferedReader(new
FileReader("D:\\test.txt"));
                out.write("Hello,World");
                out.close();
                String s;
                s = in.readLine();
                System.out.println(s);
                in.close();
            }catch(Exception e){
                System.out.println("error");
            }
   }
}
```

- 过滤流
  - 对其他输入/输出流进行特殊处理，它们在读/写数据的同时可以对数据进行特殊处理。
    - 另外还提供了同步机制，使得某一时刻只有一个线程可以访问一个输入/输出流
  - 两个**抽象类**，构造方法是保护方法
    - FilterInputStream
    - FilterOutputStream

- 类FilterInputStream和FilterOutputStream分别重写了父类InputStream和OutputStream的所有方法，同时，它们的子类也应该重写它们的方法以满足特定的需要

- 要使用过滤流，首先必须把它连接到某个输入/输出流上，通常在构造方法的参数中指定所要连接的流：

  FilterInputStream(InputStream in);

  FilterOutputStream(OutputStream out);

- 缓冲过滤流
  - 类BufferedInputStream和BufferedOutputStream实现了带缓冲的过滤流，可以提高读写效率
  - 在初始化时，除了要指定所连接的I/O流之外，还可以指定缓冲区的大小，一般为内存页或磁盘块等地整数倍，如8912字节或更小。

BufferedInputStream(InputStream in[, int size])

BufferedOutputStream(OutputStream out[, int size])

- 标准I/O
  - 标准输入文件是键盘
  - 标准输出文件是你的终端屏幕
  - 标准错误输出文件也指向屏幕，也可以指向另一个文件以便和正常输出区分

- System.in
  - Java的System类中包含的一个InputStream类型的静态对象
  - 以字节为单位进行读取
    - read()：从输入中读一个字节
    - skip(long n)：在输入中跳过n个字节

- JavaSE6中可以使用java.util.Scanner类取得用户输入
  - delimiter()　返回此 Scanner 当前正在用于匹配分隔符的 Pattern。
  - hasNext()　判断扫描器中当前扫描位置后是否还存在下一段。
  - hasNextLine()　如果在此扫描器的输入中存在另一行，则返回 true。
  - next()　查找并返回来自此扫描器的下一个完整标记。
  - nextLine()　此扫描器执行当前行，并返回跳过的输入信息。

```
import java.util.Scanner;
public class ScannerDemo{
    public static void main(String[] argc){
      Scanner scner = new Scanner(System.in);
      System.out.print(“请输入您的名字：”);
      System.out.printf(“您好，%s！\n”,scner.next());
   }
}
```

- System.out
  - Java的System类中包含的一个PrintStream类型的静态对象
    - print()
    - println()
  - 支持Java的任意基本类型作为参数

- System.err
- 重新定义标准输入输出流
  - java.lang.System的静态方法
    - static void setIn（InputStream in）
    - static void setOut（PrintStream out）

```
import java.io.*;
public class Redirecting {

      public static void main（String[] args
）  throws IOException {
            PrintStream console = System.out;
            BufferedInputStream in = new BufferedInputStream
 （ new FileInputStream（ "Redirecting.java"））；
            PrintStream out = new PrintStream
 （ new BufferedOutputStream（ new FileOutputStream（
"test.out"）））；
            System.setIn（in）；
            System.setOut（out）；
  BufferedReader br = new BufferedReader
 （ new InputStreamReader（System.in））；
            String s;
            while（（s = br.readLine（）） != null）
                  System.out.println（s）；
            out.close（）；
            System.setOut（console）；
     }
}
```

文件处理

- **File类**
- 字符文件
  - **FileReader类**
  - **FileWriter类**
- 字节文件
  - **FileInput类**
  - **FileOutput类**

- File类
  - 所有对文件的操作都要使用
  - 对象创建方法

    File(File parent, String child) //根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例。

    File(String pathname) //通过将给定路径名字符串转换成抽象路径名来创建一个新 File 实例。

    File(String parent, String child) //根据 parent 路径名字符串和 child 路径名字符串创建一个新 File 实例。

    File(URI uri) //通过将给定的 file: URI 转换成一个抽象路径名来创建一个新的 File 实例。

- 文件名的处理

  String getName( )； //得到一个文件名（不包括路径）

  String getPath( )； //得到一个文件的路径名

  String getAbsolutePath( )；//得到一个文件绝对路径名

  String getParent( )； //得到一个文件的上一级目录名

  String renameTo(File newName); //将当前文件名更名为给定文件的完整路径

- 文件属性测试

  boolean exists( )； //测试当前File对象所指示的文件是否存在

  boolean canWrite( )；//测试当前文件是否可写

  boolean canRead( )；//测试当前文件是否可读

  boolean isFile( )； //测试当前文件是否是文件（不是目录）

  boolean isDirectory( )； //测试当前文件是否是目录

- 普通文件信息和工具

  long lastModified( );//得到文件最近一次修改的时间

  long length( ); //得到文件的长度，以字节为单位

  boolean delete( ); //删除当前文件

- 目录操作

  boolean mkdir( ); //根据当前对象生成一个由该对象指定的路径

  String list( ); //列出当前目录下的文件

- File类与Java的文件管理
  - 文件和目录都是用File对象表示
  - 对于文件或目录的其他操作，如重命名、删除、列表显示等，需要使用Java的文件管理File类
  - 先创建一个File对象，并指定文件名或目录名，若指定文件名或目录名不存在，则File对象的新建并不会创建一个文件或目录；
  - createNewFile方法创建文件
  - mkdir方法创建目录。
  - 通过isFile方法和isDirectory方法来判断区分File对象代表的是文件还是目录

```
// .txt files only.
import java.io.*;
class DirListOnly {
    public static void main(String args[]) {
        String dirname = “//java";
        File f1 = new File(dirname);
        FilenameFilter only = new OnlyExt(“txt");
        String s[] = f1.list(only);
        for (int i=0; i < s.length; i++) {
           System.out.println(s[i]);
        }
    }
}
```

- listFiles( )方法
  - Java 2增加，list( )方法的一个变化形式
  - 以File对象数组的形式返回文件列表，而不是用字符串形式返回

File[ ] listFiles( )

File[ ] listFiles(FilenameFilter FFObj)

File[ ] listFiles(FileFilter FObj)

- 字符文件
  - FileReader 读取文件
  - FileWriter 写入文件
  - BufferedReader 输入到缓冲区
  - BufferedWriter 输出到缓冲区

- 字符流的操作方法
  - 从输入流中按行读取字符

    String readLine()；
  - 向输出流写入多个字符

    write(String s, int off, int len)；//将指定的字符串s从偏移量 off 开始的 len 个字符写入文件输出流

```
try{
    File f=new File(“test.txt”); // File f=new
   File(“e:\\txt”，  “test.txt");
    FileReader fr=new FileReader(f);
  BufferedReader buffin=new BufferedReader(fr);
     while((s=buffin.readLine())!=null)
      txt.append(s+'\n');
   }
   catch(IOException e){ ......  }
```

```
try{
     w_file = new FileWriter("b.txt");
     buf_writer = new BufferedWriter(w_file);
     String str …….;
      str = ……
     buf_writer.write(str,0,str.length());
     buf_writer.flush();
 }
catch(IOException e)
   { System.out.println(e); }
```

- 字节文件
  - File fp=new File(“file1.dat”); //建立文件对象
  - FileInputStream类用来打开一个输入文件
  - FileOutputStream类用来打开一个输出文件

- 字节流的操作方法
  - FileInputStream
    - read():从流中读入数据
    - close():关闭流
  - FileOutputStream
    - write(byte b[ ], int off, int len):在数组b中，从off开始，写入len个字节的数据。

```
File file=new File("d:/jtest/test.dat");
fileInput = new FileInputStream(file);
byte buffer[ ] = new byte[2056];
int bytes = fileInput.read( buffer, 0, 2056 );
str = new String( buffer, 0, bytes );
```

打入jar包内的文件不能用File操作！
jar包是一个单独的文件而非文件夹，不能用绝对或相对路径的文件URL来定位.

方法：
　类.class.getResource(“jar包内资源名")
　类.class.getResourceAsStream(“jar包内资源名")
例：

```
InputStream is =
this.getClass().getResourceAsStream("/resource/readme.txt");
```

- 随机存取文件
  - 能在任意位置读写文件
  - 打开随机存取文件有两种方法
    - 用文件名

myRAFile=new RandomAccessFile(String name,String mode);

    - 用文件对象

myRAFile=new RandomAccessFile(File file,String mode);

    - mode参数决定了访问文件的权限，如只读'r'或读写'wr'等。

- 随机存取文件RandomAccessFile
  - 创建一个随机文件

    new RandomAccessFile(“file1.txt”, “r”);

    new RandomAccessFile(“file2.txt”, “rw”);
  - zip文件需要用随机方法处理
  - 文件目录给出个文件的入口,可以随机读取.
  - 随机文件可以同时完成读和写操作

- RandomAccessFile的操作方法
  - readXXX()或writeXXX()
  - skipBytes();将指针向下移动若干字节
  - seek():将指针调到所需位置
  - getFilePointer():返回指针当前位置
  - length():返回文件长度
  - 利用seek(long pos)方法查找随机文件中的信息

```
import java.io.*;
class r
{
    public static void main(String args[]){
        try{
            RandomAccessFile file=new
                                RandomAccessFile(“A.dat","rw");
             long filePoint=0;                  //定义指针
             long fileLength=file.length();
             while(filePoint<fileLength){
                String s=file.readLine();       //读取文件中的数据
                 System.out.println(s);
                 filePoint=file.getFilePointer();
             }
             file.close();
         }
       catch(Exception e){}
    }
}
```

```
import java.io.*
class raTest{
        public static void main(String args[]) throws IOException{
           RandomAccessFile RAFile;
           String s=“\nInformation to Append";
            RAFile=new RandomAccessFile("/tmp/java.log","rw");
            //move to the end of the file
           RAFile.seek(myRAFile.length());
            //Start appending!
           RAFile.writeBytes(s);
           RAFile.close();
        }
}
```