



内容提要

第7章 I/O流与文件读写

File类

输入/输出(I/O)流

文本文件的读与写

二进制文件的读与写

对象的序列化

压缩与解压文件

随机文件的读与写



File 类

- **java.io** 包中的 **File** 类用于表示一个文件、一个目录或一个文件和目录的组合，主要用于获取文件、目录的各种信息，并对它们进行管理（创建、删除、重命名等）
- 文件的复制、文件内容的读与写操作，则要通过文件输入输出流实现



File类：构造方法

● 构造方法

File(String name)	指定与 File 对象关联的文件或目录的名称 name （可带相对或绝对路径）
File(String path, String name)	指定路径（绝对或相对） path ，并指定文件或目录名称 name
File(File dir, String name)	根据已有的表示目录的 File 对象和指定的文件或目录名来构造
File(URI uri)	通过统一资源标识符 (uniform resource identifier, URI)来定位文件或目录

绝对路径: C:\\java\\HelloWorld.java 或者 C:/java/HelloWorld.java

相对路径: \\ java \\HelloWorld.java 或者 java /HelloWorld.java

URI: new java.net.URI("file:///C:/java/HelloWorld.java")



File类：常用方法

● 常用方法—获取文件或目录信息

String getName()	返回 File 对象表示的文件或目录的名称。该名称是路径名称序列中的最后一个名称
String getParent()	返回父目录名称；若没有父目录，则返回null
String getPath()	返回路径名称，它是 File 构造方法中的路径参数path的值
String getAbsolutePath()	返回绝对路径
long length()	返回文件或目录的大小
long lastModified()	返回文件或目录最近一次被修改的时间（毫秒）



File类：常用方法

● 常用方法—文件或目录的测试操作

boolean exists()	File 对象表示的文件或目录是否存在
boolean isFile()	是否为文件
boolean isDirectory()	是否为目录
boolean isAbsolute()	路径是否为绝对路径
boolean isHidden()	是否隐藏
boolean canRead()	表示的文件是否可读
boolean canWrite()	表示的文件是否可写



File类：常用方法

● 常用方法—目录操作

boolean mkdir()	创建 File 对象表示的目录
String[] list()	返回 File 对象所表示的目录下的所有文件和子目录的名称
String[] list (FilenameFilter filter)	返回 File 对象所表示的目录下的指定类型的全部文件的名称
File[] listFiles()	返回 File 对象所表示的目录下的所有文件和子目录对应的 File 对象
File[] listFiles (FileFilter filter)	返回 File 对象所表示的目录下的指定类型的全部文件对应的 File 对象
static File[] listRoots()	返回文件系统可用的分区对应的 File 对象
long getTotalSpace()	获取 File 对象所表示分区的总空间
long getFreeSpace()	获取分区的自由空间



File类：常用方法

● 常用方法—文件操作

boolean createNewFile() throws IOException	若 File 对象表示的文件不存在，则创建它（一个空文件）
boolean renameTo(File dest)	将文件重命名为 dest 对应的文件名
boolean delete()	删除 File 对象所表示的文件或目录；如果表示的是一个目录，则该目录必须为空才能删除。



File类：常用方法

- 示例：使用**File**类对文件进行测试操作并获取文件信息

```
import java.io.*;

public class UseFile {
    public static void main(String[] args) {
        File file=new File("Doc","images/windows.jpg");
        if(file.exists()){
            System.out.println("文件名: "+file.getName());
            __.println("该文件的路径: "+file.getPath());
            __.println("绝对路径: "+file.getAbsolutePath());
            __.println("大小: "+file.length()+" 字节");
            __.println("是否可写: " + file.canWrite());
            __.println("是否隐藏: " + file.isHidden())
        }
        else __.println("指定的文件不存在");
    }
}
```




File类：常用方法

- 示例：使用File类对文件进行测试操作并获取文件信息

```
import java.io.*;

public class UseFile {
    public static void main(String[] args) {
        File file=new File("Doc","images/windows.jpg");
        if(file.exists()){
            System.out.println("文件名: "+file.getName());
            __.println("该文件的路径: "+file.getPath());
            __.println("绝对路径: "+file.getAbsolutePath());
            __.println("大小: "+file.length()+" 字节");
        }
    }
}
```

控制台

<已终止> UseFile [Java 应用程序] C:\jre1.8.0_92\bin\javaw.exe (2016年6月5日 下午2:35:35)

文件名: windows.jpg

该文件的路径: Doc\images\windows.jpg

绝对路径: D:\software\JavaSoftware\workspace\Ch7Example\Doc\images\windows.jpg

大小: 22408 字节

是否可写: true

是否隐藏: false



File类：常用方法

- 示例：使用File类对获取目录下的子目录和文件信息

```
public static void listDir(File dir){  
    String dirs="",files="";  
    File[] list=dir.listFiles();  
    for(File f:list) {  
        if(f.isFile())  
            files+="\t"+f.getName()+"\n"; // 所有文件  
        else  
            dirs+="\t"+f.getName()+"\n"; // 所有子目录  
    }  
    __.print(dir.getAbsolutePath()+"下的目录: \n"+dirs);  
    __.print(dir.getAbsolutePath()+"下的文件: \n"+files);  
}
```



File类：常用方法

- 示例：使用File类对获取目录下的子目录和文件信息

```
public static void main(String[] args) {  
    File dir=new File("Doc3"); // 改为"c:/" 试试  
    if(!dir.exists())  
        _____.println("指定的目录不存在："+dir.getAbsolutePath());  
    listDir(dir);  
}
```

控制台

<已终止> DirectorOperate [Java 应用程序] C:\jre1.8.0_92\bin\javaw.exe (2016年6月5日 下午2:59:42)

D:\software\JavaSoftware\workspace\Ch7Example\Doc3下的目录:

data

images

D:\software\JavaSoftware\workspace\Ch7Example\Doc3下的文件:

readme.txt

sn.txt



输入/输出 (I/O) 流

- 通常程序需要从外部获取/输出信息
 - 这个“外部”范围很广，包括诸如键盘、显示器、文件、磁盘、网络、另外一个程序等
 - “信息”也可以是什么类型的，例如一个对象、一串字符、一幅图像、一段声音等
- 通过使用java.io包中的输入/输出流类就可以达到输入输出信息的目的



I/O流： I/O流的概念

● I/O流的概念

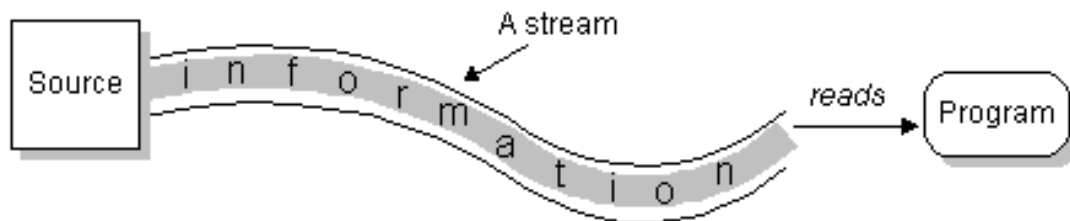
- 在Java中将信息的输入与输出过程抽象为I/O流
 - 输入是指数据流入程序
 - 输出是指数据从程序流出
- 一个流就是一个从源流向目的地的数据序列
- IO流类一旦被创建就会自动打开
- 通过调用close方法，可以显式关闭任何一个流，如果流对象不再被引用，Java的垃圾回收机制也会隐式地关闭它



I/O流： I/O流的概念

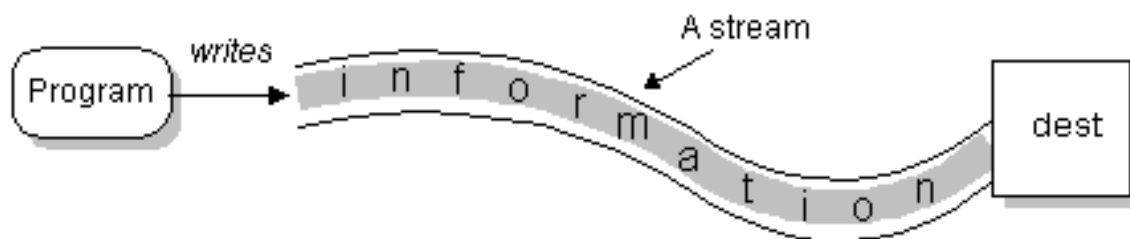
● 输入流

- 为了从信息源获取信息，程序打开一个输入流，程序可从输入流读取信息



● 输出流

- 当程序需要向目标位置写信息时，便需要打开一个输出流，程序通过输出流向这个目标位置写信息





I/O流： I/O流的概念

● 源和目标的类型

对象	能否作为源	能否作为目标
disk file	✓	✓
monitor		✓
keyboard	✓	
running program	✓	✓
Internet connection	✓	✓
mouse	✓	
image scanner	✓	

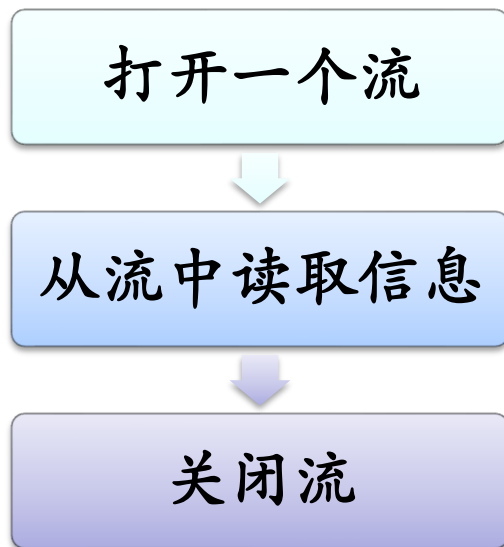


I/O流： I/O流的概念

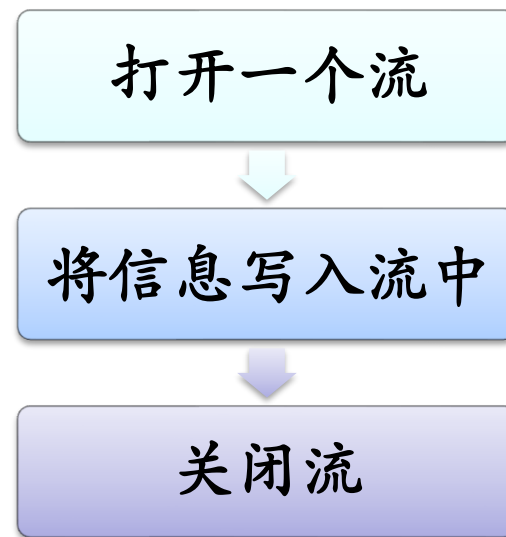
● 读写数据的方法

- 不论数据从哪来，到哪去，也不论数据本身是何类型，读写数据的方法大体上都是一样的

读数据



写数据





I/O流：预定义I/O流类概述

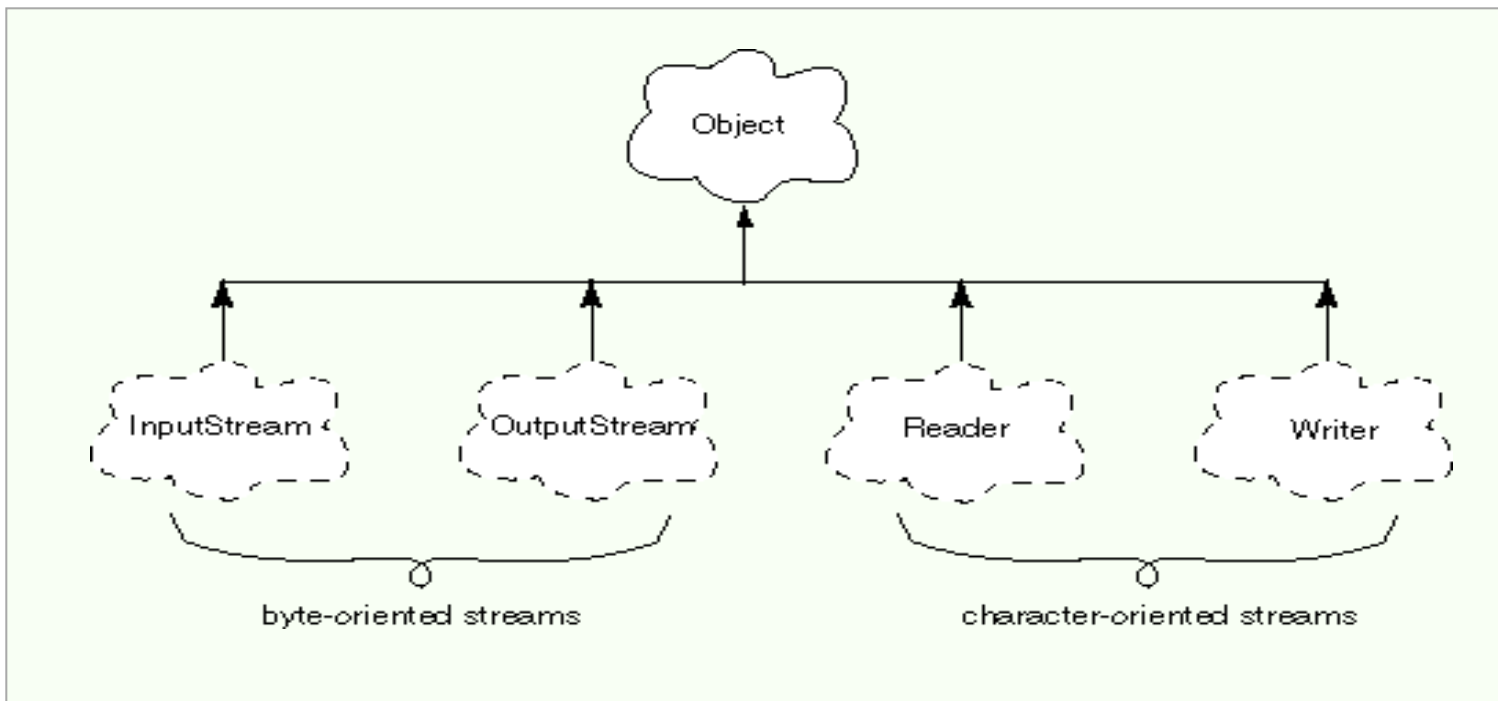
- 输入/输出流可以从以下几个方面进行分类
 - 从流的方向划分
 - 输入流
 - 输出流
 - 从流的分工划分
 - 节点流
 - 处理流
 - 从流的内容划分
 - 面向字符的流
 - 面向字节的流



I/O流：预定义I/O流类概述

● java.io包的顶级层次结构

- 面向字符的流：专门用于字符数据
- 面向字节的流：用于一般目的





I/O流：预定义I/O流类概述

● 面向字符的流

- 针对字符数据的特点进行过优化，提供一些面向字符的有用特性
- 源或目标通常是文本文件
- 实现内部格式（即字符编码）和文本文件中的外部格式（即字符编码）之间转换
 - 内部格式：**16-bit char** 数据类型，即用双字节表示一个字符的**Unicode**编码（**UTF-16**）
 - 外部格式：**ASCII**，**UTF-8**(1~3字节的**Unicode**)，**UTF-16**(双字节的**Unicode**)，**GBK**(汉字国标码，双字节编码)等



I/O流：预定义I/O流类概述

● 面向字符的流

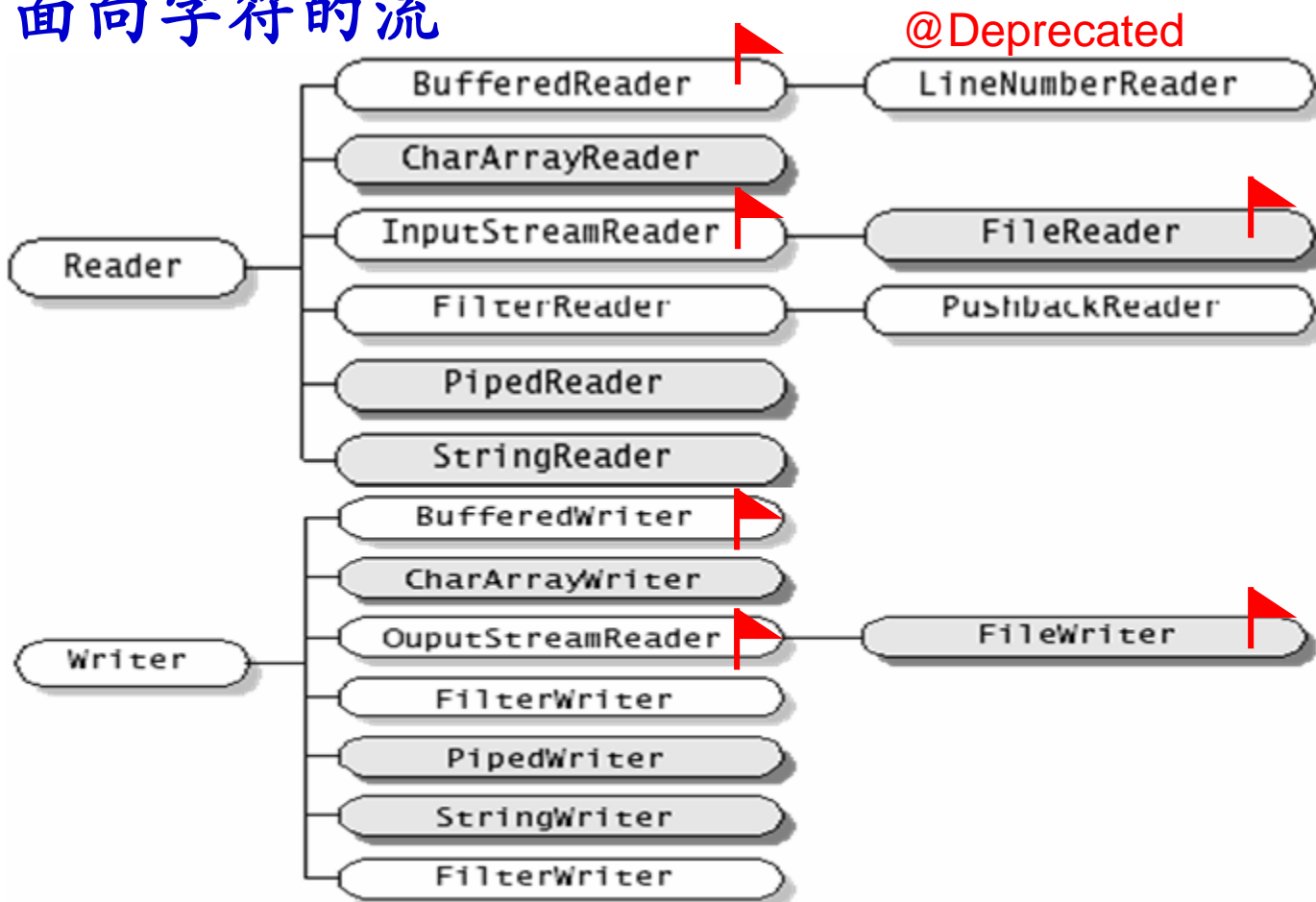
– Reader类和Writer类

- java.io包中所有字符流的抽象基类
- Reader/Writer提供了输入/输出字符的API
- 它们的子类又可分为两大类
 - 节点流：从源读入字符或往目的地写出字符
 - 处理流：对字符流执行某种处理
- 多数程序使用这两个抽象类的一系列子类来读入/写出文本信息
 - 例如FileReader/FileWriter用来读/写文本文件



I/O流：预定义I/O流类概述

● 面向字符的流



其中阴影部分为节点流



I/O流：预定义I/O流类概述

● 面向字节的流

- 数据源或目标中含有非字符数据，必须用字节流来输入/输出
- 通常被用来读写诸如图片、声音之类的二进制数据
- 绝大多数数据是被存储为二进制文件的，世界上的文本文件大约只能占到2%，通常二进制文件要比含有相同数据量的文本文件小得多



I/O流：预定义I/O流类概述

● 面向字节的流

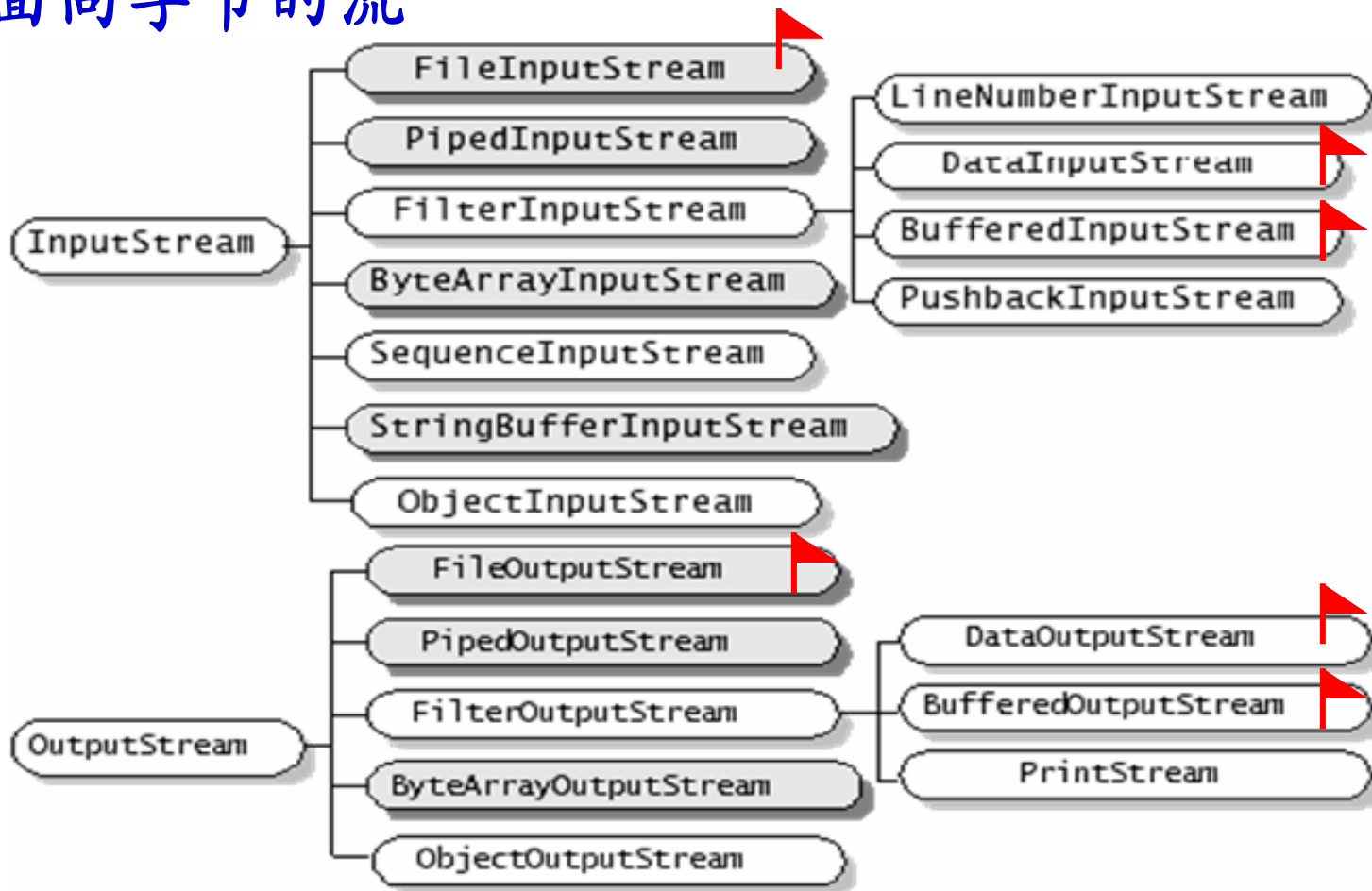
– InputStream类和OutputStream类

- 是用来处理8位字节流的抽象基类，程序使用这两个类的子类来读写8位的字节信息
- 它们的子类又可分为两大类
 - 节点流：从源读入字节或往目的地写出字节
 - 处理流：对字节流执行某种处理



I/O流：预定义I/O流类概述

● 面向字节的流



其中阴影部分为节点流



I/O流：预定义I/O流类概述

● 标准输入输出

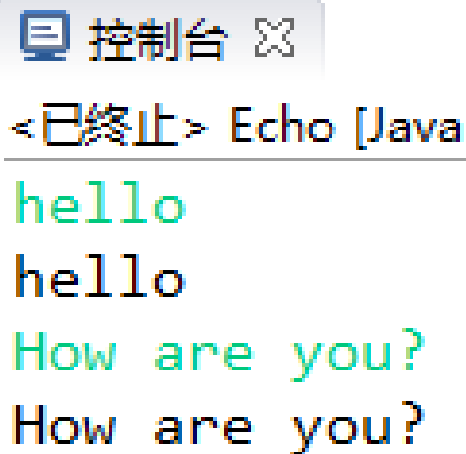
- 标准I/O流对象（System类的静态成员变量）
 - **System.in**: InputStream类型，代表标准输入流，此流已经打开，默认状态对应于键盘输入。
 - **System.out**: PrintStream类型，代表标准输出流，已经打开，默认状态对应于屏幕输出
 - **System.err**: PrintStream类型，代表标准错误信息输出流，已经打开，默认状态对应于屏幕输出
- 标准I/O重新导向
 - **setIn(InputStream)**: 设置标准输入流
 - **setOut(PrintStream)**: 设置标准输出流
 - **setErr(PrintStream)**: 设置标准错误输出流



I/O流：预定义I/O流类概述

【例】从键盘读入信息并在显示器上显示

```
import java.io.*;
public class Echo {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        String s = br.readLine(); // 从键盘输入字节流（已缓
        冲）中读取一行字符
        while(s.length() != 0) {
            System.out.println(s);
            s = br.readLine(); // 读下一行字符
        }
        br.close(); // 关闭流
    }
}
```



控制台

<已终止> Echo [Java

hello

hello

How are you?

How are you?



I/O流：预定义I/O流类概述

```
BufferedReader br = new BufferedReader(  
    new InputStreamReader(System.in));
```

- 说明：

- **System.in**

- 程序启动时由Java系统自动创建的流对象，它是原始的字节流，不能直接从中读取字符，需要对其进行进一步的处理

- **InputStreamReader(System.in)**

- 以System.in为参数创建一个InputStreamReader流对象，相当于字节流和字符流之间的一座桥梁，读取字节并将其转换为字符

- **BufferedReader br**

- 对InputStreamReader处理后的信息进行缓冲，以提高效率



I/O流：预定义I/O流类概述

● 标准输入输出

- Java 5.0终于也有了自己的printf!
 - `out.printf("%-12s is %2d long", name, l);`
 - `out.printf("value = %2.2F", value);`
 - `%n` 是平台无关的换行标志
- Java 5.0中增加了一个方便的扫描API: 把文本转化成基本类型或者String
 - `Scanner s = new Scanner(System.in);`
`int n = s.nextInt();`
 - 还有下列方法:
`nextLine(), nextByte(), nextDouble(), nextFloat,`
`nextInt(), nextLong(), nextShort()`



I/O流：预定义I/O流类概述

【例】重导向标准输入System.in和标准输出System.out

```
import java.io.*;
public class Redirecting {
    public static void main(String[] args) throws IOException {
        BufferedInputStream bis = new BufferedInputStream(
            new FileInputStream("Redirecting.java"));
        PrintStream ps = new PrintStream(new
            BufferedOutputStream(new FileOutputStream("test.out")));
        System.setIn(bis);
        System.setOut(ps);
        System.setErr(ps);
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        String s;
        while((s = br.readLine()) != null) System.out.println(s);
        ps.close(); bis.close(); br.close(); // Remember this!
    }
}
```



I/O流：预定义I/O流类概述

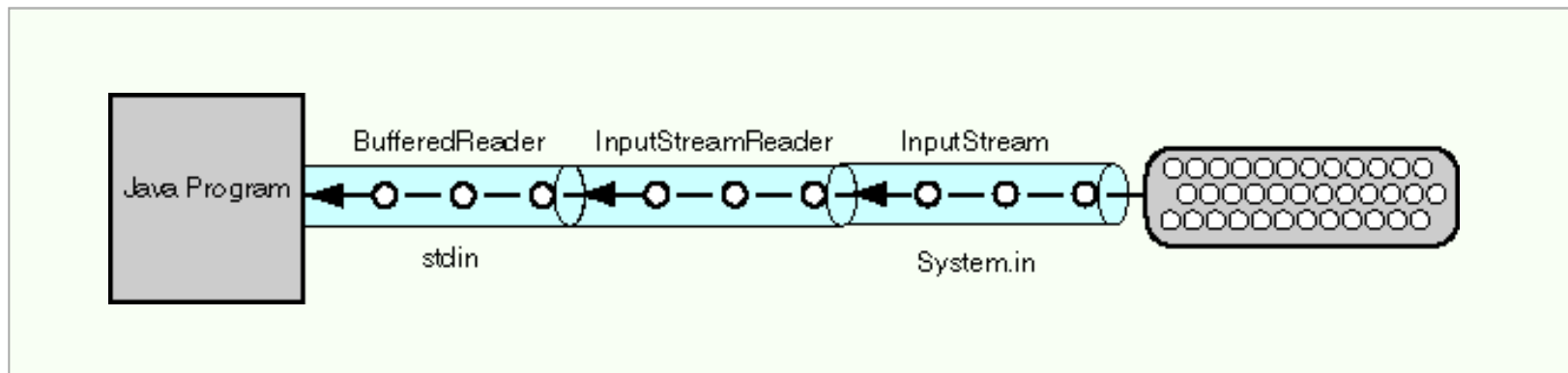
● 处理流

- 不直接与数据源或目标相连，而是基于另一个流来构造
- 从流读写数据的同时对数据进行处理
- 上例中的**InputStreamReader**和**BufferedReader**都属于处理流
 - **InputStreamReader**读取字节并转换为字符
 - **BufferedReader**对另一个流产生的数据进行缓冲



I/O流：预定义I/O流类概述

● 处理流



用一行表达式实现：

```
BufferedReader stdin = new BufferedReader(  
    new InputStreamReader(System.in));
```



I/O流：预定义I/O流类概述

● I/O异常

- 多数IO方法在遇到错误时会抛出异常，因此调用这些方法时必须
 - 在方法头声明抛出IOException异常
 - 或者在try块中执行IO，然后捕获IOException



文本文件的读与写：写入文件

● **FileWriter** 类

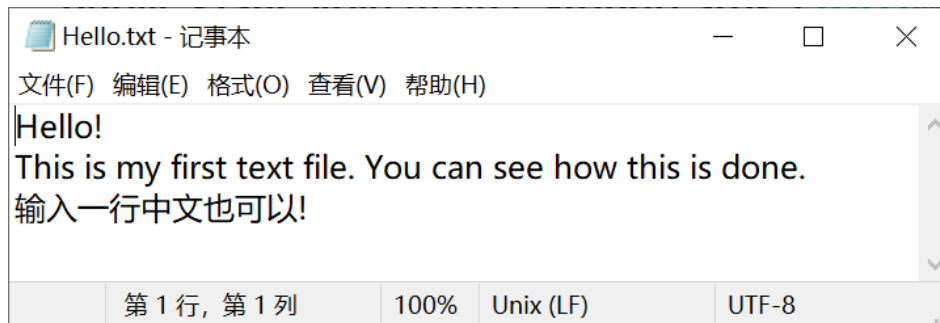
- 派生自 **OutputStreamWriter** 类（其 **write** 方法将字符转换成字节），用于将字符按默认的字符集写入文件
- 常用构造方法
 - **FileWriter(String fileName)** throws **IOException** 打开文件并清空；若文件不存在，则创建一个新文件
 - **FileWriter(String fileName, boolean append)** throws **IOException** 打开文件，并设定是否将以追加方式写入文本，而不是清空文件
- 常用方法（自己没有新增的方法）
 - **void write(String str)** throws **IOException** 写入文本
 - **void flush()** throws **IOException** 刷新该流的缓冲
 - **void close()** throws **IOException** 关闭文件



文本文件的读与写：写入文件

【例】在C盘根目录创建文本文件Hello.txt，并往里写入若干行文本

```
import java.io.*;
public class FileWriterTester {
    public static void main ( String[] args ) throws IOException {
        String fileName = "C:\\Hello.txt";
        FileWriter fw = new FileWriter( fileName );
        fw.write( "Hello!\n" );
        fw.write( "This is my first text file. " );
        fw.write( "You can see how this is done.\n" );
        fw.write("输入一行中文也可以\n");
        fw.close();
    }
}
```





文本文件的读与写：写入文件

Eclipse中文本文件编码与JVM默认字符编码的设置

第7章 I/O流与文件读写

Ch7Example 的属性

资源

路径(P): /Ch7Example
类型(Y): 项目
位置(L): D:\教学资料\JavaSoftware\workspace\Ch7Example
上次修改(M): 2021年10月14日 下午11:15:01

文本文件编码(T)

☒ 从容器继承 (UTF8) (I)
☐ 其他(O): UTF8

☐ Store the encoding of derived resources separately

新的文本文件行定界符(F)

☒ 从容器(窗口)继承(E)
☐ 其他(H): 窗口

运行配置

创建、管理和运行配置
运行 Java 应用程序

名称(N): FileWritterTester

另存为

☒ 本地文件(O)
☐ 共享文件(H): \Ch7Example

在收藏夹菜单显示(R)

☐ 调试
☒ 运行

编码

☒ 缺省值 - 继承的 (UTF8) (U)
☐ 其他(E) GBK

标准输入和输出

☒ 分配控制台 (对于输入来说是必需的) (A)
☐ 输入文件(F):

恢复(V) 应用(Y)

运行(R) 关闭

继承的编码设置来源于eclipse.ini
-Dfile.encoding=UTF8

两者可以相同, 也可以不同



文本文件的读与写：写入文件

Eclipse中文件编码与JVM默认字符编码的设置

```
FileWriter fw = new FileWriter( fileName);  
//返回此流使用的字符编码的名称。  
System.out.println(fw.getEncoding());  
// 默认字符集的名称  
System.out.println(Charset.defaultCharset().name());  
// 默认字符集的别名  
System.out.println(Charset.defaultCharset().aliases());  
// 也是获取默认字符的名称  
System.out.println(System.getProperty("file.encoding"));
```

```
UTF8  
UTF-8  
[unicode-1-1-utf-8, UTF8]  
UTF8
```



文本文件的读与写：写入文件

● **BufferedWriter** 类

- **OutputStreamWriter** 类(**FileWriter** 的父类)的 **write** 方法的每次调用都会导致在给定字符集上调用 **编码转换器**。在写入底层输出流之前，得到的这些字节将在缓冲区中累积。可以指定此缓冲区的大小，不过，默认的缓冲区对多数用途来说已足够大
- 为了获得最高效率，可考虑将 **OutputStreamWriter** 包装到 **BufferedWriter** 中，以 **避免频繁调用转换器**。
- **BufferedWriter** 类增加了一个内部（字符）缓冲区，执行写操作时不是立即将数据（字符串）写入字节流，而是将数据写入该缓冲区。当缓冲区写满或强制刷新（**flush()**）或关闭（**close()**）时，一次性地将缓冲区内的数据写入字节流中(此时才要调用 **编码转换器**)
- 如果需要写入的内容很多，就应该使用 **BufferedWriter**



文本文件的读与写：写入文件

● **BufferedWriter** 类

— 常用构造方法

- **BufferedWriter(Writer out)** 创建一个使用默认大小输出缓冲区的缓冲字符输出流
- **BufferedWriter(Writer out, int sz)** 指定缓冲区的大小。在大多数情况下，默认值就足够大了

— 常用方法

- **BufferedWriter**和**FileWriter**类都用于输出字符流，方法几乎一样，但前者多了一个**newLine()**方法用于换行
- 并非所有平台都使用新行符 (`'\n'`) 来终止各行，**newLine()** 方法使用平台自己的行分隔符概念，可以输出在当前计算机上正确的换行符
- **close** 方法关闭缓冲流时，也将关闭基础流（例如**FileWriter**）

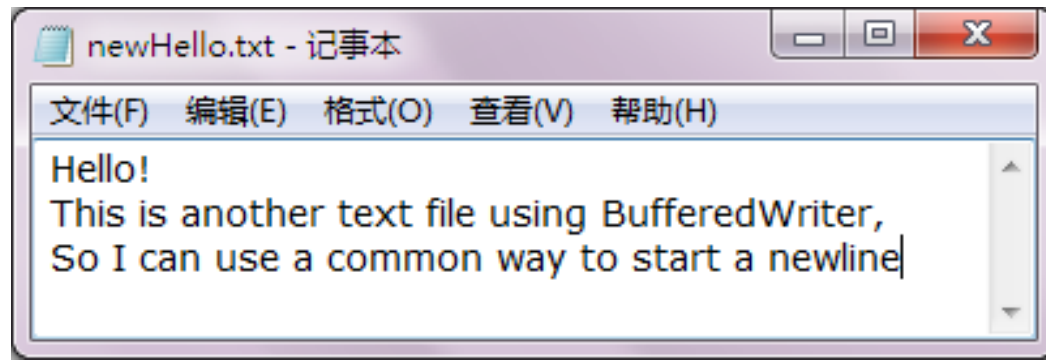


文本文件的读与写：写入文件

【例】使用BufferedWriter类实现文本的高效写入文件

```
import java.io.*;
class BufferedWriterTester {
    public static void main ( String[] args ) throws IOException
    {
        String fileName = "C:/newHello.txt" ;
        BufferedWriter bw = new BufferedWriter(
            new FileWriter( fileName ) );

        bw.write("Hello!");
        bw.newLine(); //换行
        bw.write("This is another text file using BufferedWriter,");
        bw.newLine();
        bw.write( "So I can use a common way to start a newline" );
        bw.close();
    }
}
```





文本文件的读与写：读取文件

● FileReader 类

- 派生自InputStreamReader类（其Read方法按默认的字符集将字节转换成字符），用于从文件读取字符
- 常用构造方法
 - **FileReader(String fileName)** throws FileNotFoundException 打开文件
- 常用方法（自己没有新增的方法）
 - **int read()** throws IOException 读取单个字符，如果已到达流的末尾，则返回 -1
 - **int read(char[] cbuf)** throws IOException 将字符读入数组，如果已到达流的末尾，则返回 -1
 - **long skip(long n)** throws IOException 跳过n个字符
 - **void close()** throws IOException 关闭文件



文本文件的读与写：读取文件

● **BufferedReader** 类

- 给输入字符流增加了一个内部缓冲区，实现高效读取
- 常用构造方法
 - **BufferedReader(Reader in)** 创建一个使用默认大小输入缓冲区的缓冲字符输入流
 - **BufferedReader(Reader in, int sz)** 指定缓冲区的大小。在大多数情况下，默认值就足够大了
- 常用方法
 - **BufferedReader**和**FileReader**类都用于输入字符流，包含的方法几乎完全一样，但前者多了一个**readLine()**方法，可以对换行符进行鉴别，一行一行地读取输入流中的内容



文本文件的读与写：读取文件

【例】从Hello.txt中读取文本并显示在屏幕上使用

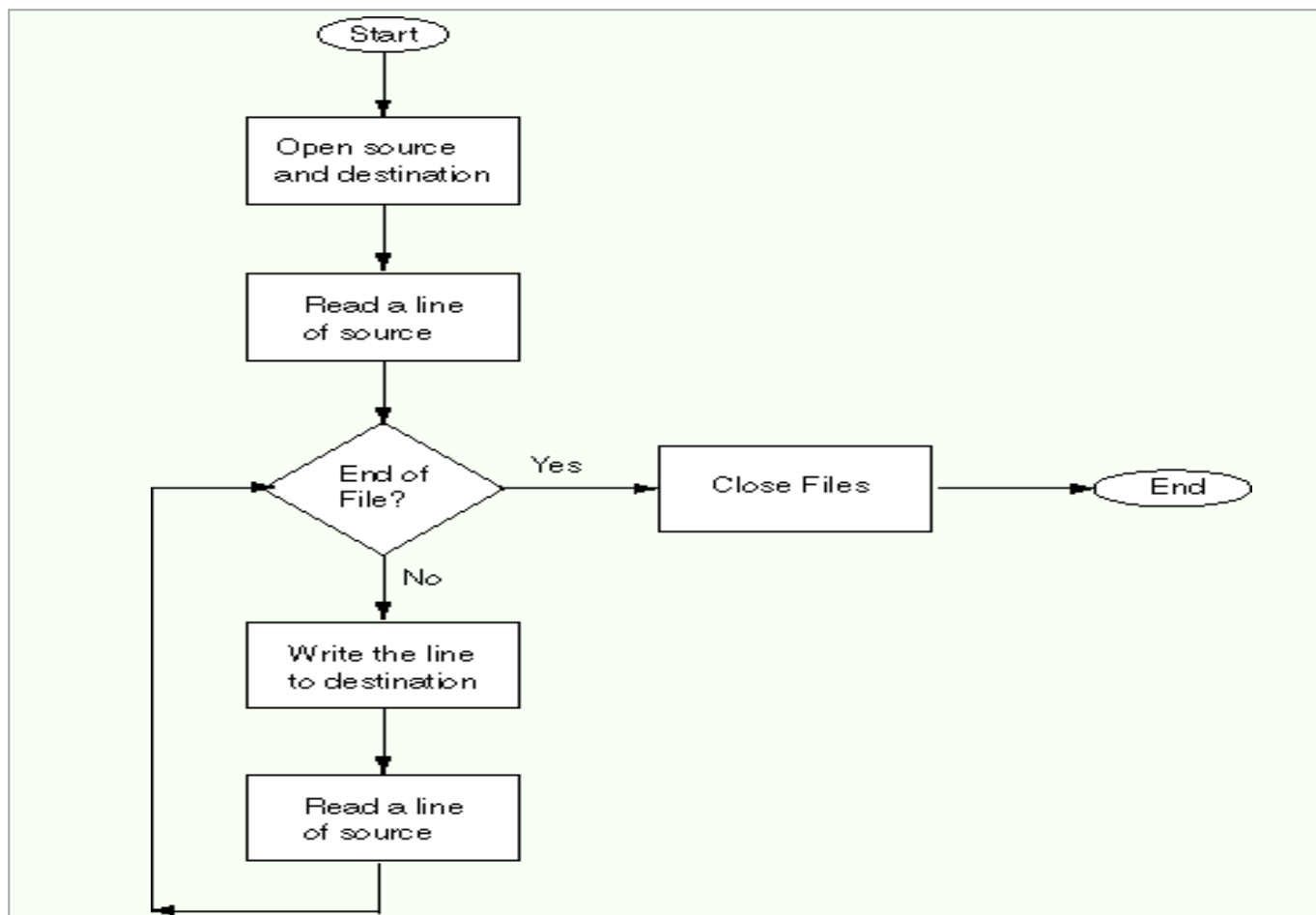
```
import java.io.*;
class BufferedReaderTester {
    public static void main ( String[] args ) {
        String fileName = "C:/Hello.txt" ;
        try { BufferedReader br = new BufferedReader(
            new FileReader( fileName ) );
            String line = br.readLine(); //读取一行内容
            while ( line != null ) { __.println( line );
                line = br.readLine(); }
            br.close();
        } catch ( IOException e ) { __.println(e.getMessage()); }
    }
}
```

```
//逐个字符读取
int ch=br.read(); // 读取一个字符,返回int
while (ch != -1) {
    __.print((char)ch); //换行符也会输出
    ch=br.read(); } // 读取一个字符,返回int
```



文本文件的读与写：读取文件

【例】文本文件读写综合：文本文件的复制





文本文件的读与写：读取文件

【例】 文本文件读写综合：文本文件的复制

```
public static void main ( String[] args ) {  
    String source = "C:/Hello.txt" ,dest= "C:/Hello_copy.txt", line;  
    try { BufferedReader br = new BufferedReader(  
                                                new FileReader( source) );  
        BufferedWriter bw = new BufferedWriter(  
                                                new FileWriter( dest) );  
        -----  
        line = br.readLine(); //从源文件读取一行内容  
        while ( line != null ) {  
            bw.write( line ); //向目标文件写入一行内容  
            bw.newLine();  
            line = br.readLine(); //从源文件读取下一行内容  
        }  
        -----  
        br.close(); bw.close(); //关闭文件  
    } catch ( IOException iox )  
        System.out.println( iox.getMessage() );  
}
```



文本文件的读与写：读取文件

【例】将文本文件的复制功能封装成一个类

TextFileCopyMarker

```
private String sourceName, destName;  
private BufferedReader source;  
private BufferedWriter dest;  
private String line;
```

```
private boolean openFile();  
private boolean copyFile();  
private boolean closeFile();  
public boolean copy(String src, String dst );
```



二进制文件的读与写：写入文件

● 本节要点

- 二进制文件
- **OutputStream**(抽象类：输出字节流)
 - **FileOutputStream**(派生类：文件输出字节流)
 - **write()** 写入字节
 - **DataOutputStream**(派生类FilterOutputStream的派生类：数据输出流，将基本类型数据转换成字节流)
 - **writeInt()**、**writeDouble()**、**writeBytes()** 等
 - **BufferedOutputStream**(派生类FilterOutputStream的派生类：对输出字节流/数据流进行缓冲)



二进制文件的读与写：写入文件

● 二进制文件

- 本质上，所有文件都是由8位的字节组成的
- 如果文件字节中的内容应被解释为字符，则文件被称为文本文件；如果被解释为其它含义，则文件被称为二进制文件
- 例如字处理软件Word产生的doc文件中，数据要被解释为字体、格式、图形和其他非字符信息。因此，这样的文件是二进制文件，不能用Reader/Writer流正确读写



二进制文件的读与写：写入文件

● **FileOutputStream** 类

- 派生自抽象类 **OutputStream**
- 该类用于向文件写入诸如图像数据之类的原始字节
- 常用构造方法
 - **FileOutputStream(String fileName)** throws **FileNotFoundException** 由指定文件名构造文件输出流，文件被清空；若文件不存在，则新建一个文件
 - **FileOutputStream(String fileName, boolean append)** throws **IOException** 打开文件，并设定是否将以追加方式写入字节，而不是清空文件
- 常用方法
 - **void write(byte[] b)** throws **IOException** 将字节数组写入此文件输出流中
 - **void write(byte[] b, int off, int len)** throws **IOException** 把将字节数组中从偏移量 **off** 开始的 **len** 个字节写入此文件输出流末尾
 - **void close()** throws **IOException** 关闭此文件输出流



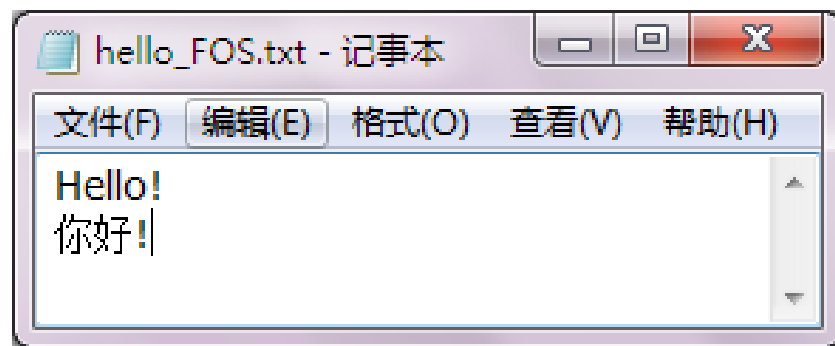
二进制文件的读与写：写入文件

【例】用FileOutputStream类实现向文件写入若干行文本

```
import java.io.*;
public class UseFOStoWriteText {
    public static void main(String[] args) throws IOException{
        String fileName="c:/hello_FOS.txt",line="";
        FileOutputStream fos=new FileOutputStream(fileName);

        line="Hello!\r\n";
        fos.write(line.getBytes()); //getBytes()使用平台的默认字
        符集将此 String 编码为 byte 序列，并将结果存储到一个新的
        byte 数组中
        line="你好!\r\n";
        fos.write(line.getBytes());

        fos.close(); //关闭文件输出流
    }
}
```

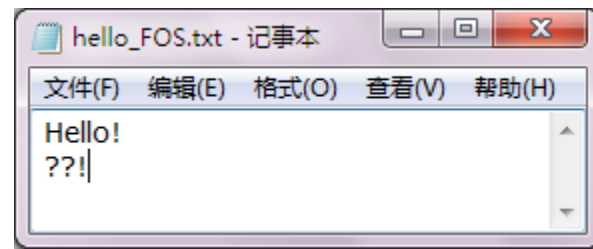




二进制文件的读与写：写入文件

【例】修改上例，写文本时指定特定的字符集而不是默认

```
import java.io.*;
import java.nio.charset.*;
public class UseFOStoWriteText 2{
    public static void main(String[] args) throws IOException{
        String fileName="c:/hello_FOS.txt",line="";
        FileOutputStream fos=new FileOutputStream(fileName);
        Charset charset=Charset.forName("ASCII"); //指定字符集
        line="Hello!\r\n";
        fos.write(line.getBytes(charset)); // 使用给定的 charset 将
        此 String 编码到 byte 序列，并将结果存储到新的 byte 数组
        line="你好!\r\n";
        fos.write(line.getBytes(charset)); //汉字映射不到ASCII码，
        因此出现乱码（用?替代）
        fos.close(); //关闭文件输出流
    }
}
```





二进制文件的读与写：写入文件

- 用 **OutputStreamWriter** 与 **FileOutputStream** 写文本文件
 - **OutputStreamWriter** 是字符流通向字节流的桥梁：它使用指定或平台默认的字符集将要写入流中的字符编码成字节。

主要构造方法

OutputStreamWriter([OutputStream](#) out)

创建使用默认字符编码的 **OutputStreamWriter**。

OutputStreamWriter([OutputStream](#) out, [Charset](#) cs)

创建使用给定字符集的 **OutputStreamWriter**。

OutputStreamWriter([OutputStream](#) out, [String](#) charsetName)

创建使用指定字符集的 **OutputStreamWriter**。

主要方法

void [close](#)()

关闭该流并释放与之关联的所有资源

void [flush](#)()

刷新该流的缓冲

[String](#) [getEncoding](#)()

返回此流使用的字符编码的名称

void [write](#)(char[] cbuf, int off, int len)

写入字符数组的某一部分

void [write](#)(char c)

写入单个字符

void [write](#)([String](#) str, int off, int len)

写入字符串的某一部分



二进制文件的读与写：写入文件

```
String fn = "upc.txt";
BufferedReader br;
try {
    bw = new BufferedWriter(
        new OutputStreamWriter(
            new FileOutputStream(fn), "UTF-8"));
    bw.write("中国石油大学");
    bw.newLine(); // 写入换行
    bw.write("1953-2023");
    bw.close();
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```



二进制文件的读与写：写入文件

● **DataOutputStream** 类

- 为抽象类 **OutputStream** 的子类 **FilterOutputStream** (过滤输出流) 的子类
- 该类对原始字节流 (包括缓冲的字节流) 进行包装, 提供向字节流 (例如文件输出流) 写入各种基本数据类型的方法, 即将基本类型数据转换成字节并写入字节流
- 构造方法
 - **DataOutputStream(OutputStream out)** 创建一个新的数据输出流, 将数据写入指定基础输出流。内部计数器 **written** 被设置为零。
- 常用方法



二进制文件的读与写：写入文件

第7章 I/O流与文件读写

protected int written	到目前为止已写入数据输出流的字节数
final int size()	返回计数器 written 的当前值，即到目前为止写入此数据输出流的字节数
void flush() throws IOException	清空此数据输出流。这迫使所有缓冲的输出字节被写出到流中。 DataOutputStream 的 flush 方法调用其基础输出流的 flush 方法。
final void writeBoolean (boolean v) throws IOException	将一个 boolean 值以 1-byte 值形式写入基础输出流。成功执行后，计数器 written 增加 1
final void writeChar (int v) throws IOException	将一个 char 值以 2-byte 值形式写入基础输出流中，先写入高字节
final void writeChars (String s) throws IOException	将字符串按字符顺序写入基础输出流。通过 writeChar 方法将每个字符写入数据输出流。成功执行后，则计数器 written 增加 s 长度的2倍。



二进制文件的读与写：写入文件

第7章 I/O流与文件读写

final void writeShort (int v) throws IOException	将一个 short 值以 2-byte 值形式写入基础输出流中，先写入高字节。成功执行后，计数器 written 增加 2
final void writeInt (int v) throws IOException	将一个 int 值以 4-byte 值形式写入基础输出流中，先写入高字节。成功执行后，计数器 written 增加4
final void writeLong (long v) throws IOException	将一个 long 值以 8-byte 值形式写入基础输出流。成功执行后，计数器 written 增加 8
final void writeFloat (float v) throws IOException	使用 Float 类中的 floatToIntBits 方法将 float 参数转换为一个 int 值，然后将该 int 值以 4-byte 值形式写入基础输出流中，先写入高字节。成功执行后，计数器 written 增加4



二进制文件的读与写：写入文件

第7章 I/O流与文件读写

final void writeDouble (double v) throws IOException	将使用 Double 类中的 doubleToLongBits 方法将 double 参数转换为一个 long 值，然后将该 long 值以 8-byte 值形式写入基础输出流中，先写入高字节。成功执行后，则计数器 written 增加 8。
final void writeByte (int v) throws IOException	将一个 int 值的低八位写入，舍去高24为位。成功执行后，计数器 written 增加 1
final void writeBytes (String s) throws IOException	字符串中每个字符舍去高八位后写入基础流中。成功执行后，则计数器 written 增加 s 长度



二进制文件的读与写：写入文件

【例】将几个int,short,String型数据写入数据文件data1.dat

```
import java.io.*;
public class DataOutputStreamTester {
    public static void main ( String[] args ) {
        try { DataOutputStream dos = new DataOutputStream(
            new FileOutputStream("c:/data1.dat" ) );
            dos.writeInt( 0 );    // 0x00000000
            dos.writeInt( 65 );   // 0x00000041
            dos.writeInt( 255 );  // 0x000000FF
            dos.writeInt( -1 );   // 0xFFFFFFFF—补码=原码除符号位外取反+1
            dos.writeShort(3);    // 将写入的字符个数, 0x0003
            dos.writeChars("A经B"); // '经' — 0x7ECF
            dos.close();
        } catch ( IOException iox ){ iox.printStackTrace(); }
    }
}
```

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	00	00	00	00	00	00	00	41	00	00	00	FF	FF	FF	FF	FF	;A...
00000010h:	00	03	00	41	7E	CF	00	42									; ...A~?B



二进制文件的读与写：写入文件

● **BufferedOutputStream** 类

- **FileOutputStream**等输出流的**write**方法，每次调用都立即将字节写入文件，这样将频繁地向操作系统请求写，导致输出效率较低。
- **BufferedOutputStream**对输出字节流进行缓冲，从而减少系统写请求次数，实现字节的高效写入
- 常用构造方法
 - **BufferedOutputStream(OutputStream out)** 创建一个使用默认大小输出缓冲区的缓冲字节输出流
 - **BufferedOutputStream(OutputStream out, int size)** 指定缓冲区的大小。在大多数情况下，默认值就足够大了



二进制文件的读与写：写入文件

● BufferedOutputStream 类

– 常用方法

- **write、close方法** 与FileOutputStream一样，不过close方法也关闭其基础流(例如FileOutputStream)
- **void flush() throws IOException** 刷新此缓冲的输出流。这迫使所有缓冲的字节被写到底层输出流中

– 例如：

```
BufferedOutputStream bos=new BufferedOutputStream (  
    new FileOutputStream(fileName) );
```

```
line="Hello!\r\n";
```

```
bos.write(line.getBytes());
```

```
line="你好!\r\n";
```

```
fos.write(line.getBytes());
```

```
fos.close(); //关闭缓冲输出流（它也会关闭文件输出流）
```



二进制文件的读与写：写入文件

【例】对输入字节流缓冲，然后写入基本类型数据

```
import java.io.*;
public class DataOutputStreamTester {
    public static void main ( String[] args ) {
        try { DataOutputStream dos =
            new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream("c:/data1.dat" ) ));

            dos.writeInt( 0 );    // 0x00000000
            dos.writeInt( 65 );   // 0x00000041
            dos.writeInt( 255 );  // 0x000000FF
            dos.writeInt( -1 );   // 0xFFFFFFFF—补码=原码除符号位外取反+1
            dos.writeShort(3);    // 将写入的字符个数， 0x0003
            dos.writeChars("A经B"); // '经' — 0x7ECF
            dos.close();
        } catch ( IOException iox ){ iox.printStackTrace(); }
    }
}
```



二进制文件的读与写：读取文件

● FileInputStream 类

- 派生自抽象类InputStream
- 该类用于从文件读取诸如图像数据之类的原始字节
- 常用构造方法
 - **FileInputStream(String fileName)** throws FileNotFoundException 由指定文件名构造文件输入流
- 常用方法
 - **int read()** throws IOException 从此输入流中读取1个字节；如果已到达文件末尾，则返回 -1
 - **int read(byte[] b)** throws IOException 从此输入流中将最多 b.length 个字节的数据读入一个 字节数组中；返回读入缓冲区b的字节总数，如果因为已经到达文件末尾而没有更多的数据，则返回 -1
 - **long skip(long n)** throws IOException 跳过n个字节；返回实际跳过的字节数
 - **void close()** throws IOException 关闭此文件输入流



二进制文件的读与写：读取文件

【例】用FileInputStream类读取文本文件c:/Hello.txt

```
import java.io.*; import java.nio.charset.Charset;
public class UseFIStoReadTextfile {
    public static void main(String[] args) {
        String fileName = "c:/hello.txt", str="";
        int m=0;
        byte[] b=new byte[1024]; // 字节缓冲区
        try{ FileInputStream fis=new FileInputStream(fileName);
            m=fis.read(b);
            while(m!=-1){
                str=new String(b, 0, m); //通过使用平台的默认字符集解码
                指定的 byte 子数组，构造一个新的 String
                //str=new String(b, 0, m,Charset.forName("ASCII")); //使
                用指定的字符集解码
                System.out.println(str);
                m=fis.read(b);
            }
            fis.close();
        }catch(FileNotFoundException e){ __.println(e.getMessage());}
        catch(IOException e){ __.println(e.getMessage());}
    }
}
```



二进制文件的读与写：读取文件

- 用 **InputStreamReader** 与 **FileInputStream** 读取文本文件
 - **InputStreamReader** 是字节流通向字符流的桥梁：它使用指定或平台默认的字符集读取字节并将其解码为字符。

主要构造方法

InputStreamReader([InputStream](#) in)

创建一个使用默认字符集的 **InputStreamReader**。

InputStreamReader([InputStream](#) in, [Charset](#) cs)

创建使用给定字符集的 **InputStreamReader**。

InputStreamReader([InputStream](#) in, [String](#) charsetName)

创建使用指定字符集的 **InputStreamReader**。

方法摘要

void close ()	关闭该流并释放与之关联的所有资源。
String getEncoding ()	返回此流使用的字符编码的名称。
int read ()	读取单个字符。
int read (char[] cbuf, int offset, int length)	将字符读入数组中的某一部分。
Boolean ready ()	判断此流是否已经准备好用于读取。



二进制文件的读与写：读取文件

```
String fn = "upc.txt";
BufferedReader br;
try {
    br = new BufferedReader(
        new InputStreamReader(
            new FileInputStream(fn), "UTF-8"));
    String line = br.readLine();
    while (line != null) {
        System.out.println(line);
        line = br.readLine();
    }
    br.close();
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```




二进制文件的读与写

【试一试】 字符串以ASCII格式写入，而以GBK格式读出

```
String fileName="d:/hello_FOS.txt", line="";  
FileOutputStream fos=new FileOutputStream(fileName);  
line="Hello!\r\n";  
fos.write(line.getBytes(Charset.forName("ASCII")));  
line="你好!\r\n";  
fos.write(line.getBytes(Charset.forName("ASCII")));
```

```
int m=0;  
byte[ ] b=new byte[1024]; // 字节缓冲区  
FileInputStream fis=new FileInputStream(fileName);  
while((m=fis.read(b))!=-1){  
    str=new String(b, 0, m);  
    //str=new String(b, 0, m,Charset.forName("ASCII"));  
    System.out.println(str);  
}
```

控制台

<已终止> UseFIStoReadTextfile

Hello!

??!



二进制文件的读与写：读取文件

● **DataInputStream** 类

- 为抽象类 **InputStream** 的子类 **FilterInputStream**（过滤输入流）的子类
- 数据输入流类对原始字节流（例如文件输入流）进行包装，从中读取字节并转换成各种基本类型数据
- 一般用于读取由数据输出流 **DataOutputStream** 写入的数据
- **DataInputStream** 写的字节是连续的，中间没有分隔符，所以应该记住写入的数据的含义、数据类型、顺序等情况，以便将来利用
- 构造方法
 - **`DataInputStream(InputStream out)`** 创建一个新的数据输入流，将从指定基础输入流读取数据
- 常用方法



二进制文件的读与写：读取文件

第7章 I/O流与文件读写

final int read (byte[] b) throws IOException	从包含的输入流中读取一定数量的字节，并将它们存储到缓冲区数组 b 中。返回实际读取的字节数
final int read (byte[] b, int off, int len) throws IOException	从包含的输入流中将最多 len 个字节读入一个 byte 数组中。尽量读取 len 个字节，但读取的字节数可能少于 len 个，也可能为零。返回实际读取的字节数。
final xxx readXxx () throws IOException	从所包含的输入流中读取此操作需要的字节， xxx 表示char,int,float等基本类型
int readUnsignedByte () throws IOException	读取此输入流的下一个字节，并将它解释为一个无符号 8 位数（int高24为补0）
final int skipBytes (int n) throws IOException	从包含的输入流中跳过n个字节，返回实际跳过的字节数
void close () throws IOException	关闭此输入流，此方法只执行基础流的 close() 方法



二进制文件的读与写：读取文件

【例】从数据文件data1.dat中读取由数据输出流写入的数据

```
import java.io.*;
public class UseDlStoReadBinaryFile {
    public static void main ( String[] args ) {
        byte[] b;
        try { DataInputStream dis = new DataInputStream(
            new FileInputStream("c:/data1.dat") );
            __.println(dis.readInt());
            __.println(dis.readInt());
            __.println(dis.readInt());
            __.println(dis.readInt());
            b=new byte[dis.readShort()*2];
            int n=dis.read(b); // 读取字符串所在字节
            __.println(new String(b,0,n));
            dis.close();
        } catch (FileNotFoundException e){
            __.println(e.getMessage()); }
        catch (IOException e){ __.println(e.getMessage());}
    }
}
```

控制台

<已终止> UseDlSto

0

65

255

-1

A经B



二进制文件的读与写：读取文件

【例】上例修改如下，运行结果是什么？

```
import java.io.*;
public class UseDlStoReadBinaryFile {
    public static void main ( String[] args ) {
        try { DataInputStream dis = new DataInputStream(
            new FileInputStream("c:/data1.dat") );

            dis.skip(4*3); //跳过12个字节
            __.println(dis.readInt());

            dis.skip(-4); //回跳4个字节
            __.println(Indis.readInt().);

            dis.close();
        } catch (FileNotFoundException e){
            __.println(e.getMessage()); }
        catch (IOException e){ __.println(e.getMessage());}
    }
}
```

控制台

<已终止> UseD

-1

-1



二进制文件的读与写：读取文件

● BufferedInputStream 类

- **FileInputStream**等输入流的**read**方法，每次调用都立即从文件读取字节，这样将频繁地向操作系统请求读，导致输取效率较低。
- **BufferedInputStream**对输入字节流进行缓冲，从而减少系统读请求次数，实现字节的高效读取
- 常用构造方法
 - **BufferedInputStream(InputStream in)** 创建一个使用默认大小输入缓冲区的缓冲字节输入流
 - **BufferedInputStream(InputStream in, int size)** 指定缓冲区的大小。在大多数情况下，默认值就足够大了



二进制文件的读与写：读取文件

● BufferedInputStream 类

– 常用方法

- **read**、**skip**、**close** 方法 与 **FileInputStream** 一样，
不过 **close** 方法也关闭其基础流(例如 **FileInputStream**)

– 例如：

```
String fileName = "c:/hello.txt", str="";  
int m=0;  
byte[] b=new byte[1024]; // 字节缓冲区  
BufferedInputStream bis=new BufferedInputStream (  
    new FileInputStream(fileName) );  
while((m=bis.read(b))!=-1){  
    str=new String(b, 0, m); //通过使用平台的默认字符集解码指  
    定的 byte 子数组，构造一个新的 String  
    System.out.println(str);  
}  
bis.close(); //关闭缓冲输入流（它也会关闭文件输入流）
```




二进制文件的读与写：读取文件

【例】读取基本类型数据之前，对输入字节流进行缓冲

```
import java.io.*;
public class UseDlStoReadBinaryFile {
    public static void main ( String[] args ) {
        try { DataInputStream dis = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("c:/data1.dat") ) );
```

```
        dis.skip(4*3); //跳过12个字节
        __.println(dis.readInt());
```

```
        dis.skip(-4); //回跳4个字节
        __.println(InDis.readInt());
```

```
        dis.close();
    } catch (FileNotFoundException e){
        __.println(e.getMessage()); }
    catch (IOException e){ __.println(e.getMessage()); }
}
```

缓冲后无法回跳

控制台

<已终止> UseD

-1

196673



二进制文件的读与写：综合应用

● 使用IO流实现任意文件的复制

- 基本步骤

- ① 打开源文件，创建目标文件
- ② 如果未到源文件末尾，则继续；否则转步骤④
- ③ 从源文件读取数据块，并写入目标文件，转步骤②
- ④ 关闭源文件和目标文件

- 不用关心文件的内容具体表示什么，所有的东西都以0、1序列即字节对待即可

- 关键技术

- 使用 **FileInputStream** 和 **FileOutputStream** 实现文件的读与写
- 为提高效率，使用 **BufferedInputStream** 和 **BufferedOutputStream** 对输入输出字节流进行缓冲

留作上机题



对象的序列化

- 对象的序列化(Object serialization)是指将对象按照一定规则转换成一个字节流；反序列化(deserialization)则是指从字节流中重新构建对象
- 主要用途
 - 序列化使对象的信息可以永久保存到磁盘文件，在需要的时候，再通过反序列化读取这个对象
 - 在分布式环境中的远程方法调用 (RMI)：使本来存在于其他机器的对象可以表现出好象就在本地机器上的行为。将消息发给远程对象时，需要通过对象序列化来传输参数和返回值
- 用于对象信息存储和读取的输入输出流类：
 - ObjectOutputStream
 - ObjectOutputStream



对象的序列化

- **ObjectInputStream和ObjectOutputStream**
 - 实现对象的读写
 - **ObjectOutputStream** 代表对象输出流，其方法 **writeObject(Object obj)** 可以实现对象的序列化，将得到的字节序列写到目标输出流中
 - **ObjectInputStream** 代表对象输入流，其 **readObject()** 方法能从源输入流中读取字节序列，将其反序列化为对象
 - **transient** 变量不序列化，静态变量序列化后由各对象共享
 - 类通过实现 **java.io.Serializable** 接口以启用其序列化功能；对象要想实现序列化，其所属的类必须实现 **Serializable** 接口



对象的序列化

- **ObjectInputStream和ObjectOutputStream**
 - 必须通过另一个流构造ObjectOutputStream, 例如

```
FileOutputStream fos = new FileOutputStream("time.dat");
ObjectOutputStream s = new ObjectOutputStream(fos);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```
 - 必须通过另一个流构造ObjectInputStream, 例如

```
FileInputStream in = new FileInputStream("time.dat");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```



对象的序列化

● Serializable接口

- 空接口即没有方法或字段，，仅用于标识可序列化的语义，使类的对象可实现序列化。
- **Serializable** 接口的定义

```
package java.io;
public interface Serializable {
    // there's nothing in here!
};
```
- 实现**Serializable**接口的语句

```
public class MyClass implements Serializable {
    ...
}
```
- 使用关键字**transient**可以阻止对象的某些成员被自动写入文件



对象的序列化

【例】创建一个书籍对象，并把它输出到一个文件book.dat中，然后再把该对象读出来，在屏幕上显示对象信息

```
class Book implements Serializable {  
    int id;  
    String name;  
    String author;  
    float price;  
    public Book(int id,String name,String author,float price)  
    {  
        this.id=id;  
        this.name=name;  
        this.author=author;  
        this.price=price;  
    }  
}
```



对象的序列化

【例】创建一个书籍对象，并把它输出到一个文件book.dat中，然后再把该对象读出来，在屏幕上显示对象信息

```
import java.io.*;
public class SerializableTester {
    public static void main(String args[]) throws
        IOException, ClassNotFoundException {
        Book book=new Book(100032,"Java Programming
                               Skills","Wang Sir",30);
        ObjectOutputStream oos=new ObjectOutputStream(
            new FileOutputStream("c:/book.dat"));
        oos.writeObject(book);
        oos.close();
    }
}
```



对象的序列化

【例】创建一个书籍对象，并把它输出到一个文件book.dat中，然后再把该对象读出来，在屏幕上显示对象信息

```
book=null;
ObjectInputStream ois=new ObjectInputStream(
    new FileInputStream("c:/book.dat"));
book=(Book)ois.readObject();
ois.close();
System.out.println("ID is:"+book.id);
System.out.println("name is:"+book.name);
System.out.println("author is:"+book.author);
System.out.println("price is:"+book.price);
}
```




对象的序列化

— 运行结果

将生成book.dat文件，并在屏幕显示：

ID is:100032

name is:Java Programming Skills

author is:Wang Sir

price is:30.0

— 说明

如果希望增加 **Book** 类的功能，使其还能够具有借书方法 **borrowBook**，并保存借书人的借书号 **borrowerID**，可对 **Book** 类添加如下内容：

```
transient int borrowerID;
```

```
public void borrowBook(int ID){
```

```
    this.borrowerID=ID;
```

```
}
```



对象的序列化

- 修改程序

- 在main方法中创建了Book类的一个对象后，紧接着调用borrowBook方法

```
book.borrowBook(2018);
```

- 从读入的对象中输出borrowerID

```
System.out.println("Borrower ID is:"+book.borrowerID);
```

- 运行结果

- 显示borrrrowID为0，因为声明为transient，所以不保存
- 如果去掉transient关键字，则可以正确读出2018。这对于保护比较重要的信息（例如密码等）是很有必要的



对象的序列化

● Externalizable接口

- 继承自 **Serializable** 接口
- 实现该接口的类由程序员来控制序列化的行为（可以决定要序列化哪些属性并逐个写入），而仅实现 **Serializable** 接口的类采用默认的序列化方式（除标识为 **transient** 之外的所有属性都被自动写入）
- API 中的说明为
`public interface Externalizable extends Serializable`
- 其中有两个方法 **writeExternal()** 和 **readExternal()**，因此实现该接口的类必须实现它们
 - **ObjectOutputStream** 的 **writeObject()** 方法只写入对象的标识，然后调用对象所属类的 **writeExternal()** --- 逐个写入要序列化的属性值
 - **ObjectInputStream** 的 **readObject()** 方法调用对象所属类的 **readExternal()** --- 逐个读取属性值



对象的序列化

【例】实现Externalizable接口，控制对象的序列化方式

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

```
public class Blip implements Externalizable {
```

```
    int i;
```

```
    String s;
```

```
    public Blip() { System.out.println("Blip Constructor"); }
```

```
    public Blip(String x, int a) {
```

```
        System.out.println("Blip(String x, int a)");
```

```
        s = x;
```

```
        i = a;
```

```
    }
```

```
    public String toString() { return s + i; }
```

实现Externalizable接口的类必须有无参构造方法



对象的序列化

```
public void writeExternal(ObjectOutput out) throws
```

```
IOException{
```

```
    System.out.println("Blip.writeExternal");
```

```
    // You must do this:
```

```
    out.writeObject(s);
```

```
    out.writeInt(i);
```

```
}
```

逐个写入要序列化的实例变量；
注意：transient变量也会被写入

```
public void readExternal(ObjectInput in)
```

```
    throws IOException, ClassNotFoundException {
```

```
    System.out.println("Blip.readExternal");
```

```
    // You must do this:
```

```
    s = (String)in.readObject();
```

```
    i = in.readInt();
```

```
}
```

逐个读取（反序列化）
实例变量；注意：应当
与写入顺序一致



对象的序列化

```
public static void main(String[] args)
throws IOException, ClassNotFoundException {
    String fileName="c:\\blip.out";
    __.println("Constructing objects:");
    Blip b = new Blip("A String ", 47);
    __.println(b);
    ObjectOutputStream oos = new ObjectOutputStream(
        new FileOutputStream(fileName));
    __.println("Saving object:"); oos.writeObject(b);
    oos.close();
    // Now get it back:
    ObjectInputStream ois = new ObjectInputStream(
        new FileInputStream(fileName));
    __.println("Recovering b:"); b = (Blip)ois.readObject();
    __.println(b);
    ois.close();
}
}
```

控制台

```
<已终止> Blip [Java 应用程序] C:\jre
Constructing objects:
Blip(String x, int a)
A String 47
Saving object:
Blip.writeObject
Recovering b:
Blip Constructor
Blip.readExternal
A String 47
```



文件的压缩与解压缩

- `java.util.zip`包中提供了一些类，使我们可以以压缩格式对流进行读写
- 它们都继承自字节流类`OutputStream`和`InputStream`
- 其中`GZIPOutputStream`和`ZipOutputStream`可分别把数据压缩成`GZIP`格式和`Zip`格式
- `GZIPInputStream`和`ZipInputStream`可以分别把压缩成`GZIP`格式或`Zip`的数据解压缩恢复原状

```
java.lang.Object
├─ java.io.OutputStream
│   └─ java.io.FilterOutputStream
│       └─ java.util.zip.DeflaterOutputStream
```

所有已实现的接口：

`Closeable`, `Flushable`

直接已知子类：

`GZIPOutputStream`, `ZipOutputStream`



文件的压缩与解压缩

● 简单的GZIP压缩格式

– GZIPOutputStream

- 父类是DeflaterOutputStream
- 可以把数据压缩成GZIP格式

– GZIPInputStream

- 父类是InflaterInputStream
- 可以把压缩成GZIP格式的数据解压缩

```
java.lang.Object
├── java.io.InputStream
│   ├── java.io.FilterInputStream
│   │   └── java.util.zip.InflaterInputStream
```

所有已实现的接口：

Closeable

直接已知子类：

GZIPInputStream, ZipInputStream



文件的压缩与解压缩

【例】将文件“c:/page.htm”压缩为文件“c:/page.gz”，然后解压该文件，显示其中内容，并另存为“c:/newPage.htm”

```
import java.io.*;    import java.util.zip.*;

public class GZIPTester {
    public static void main(String[] args) throws IOException {
        BufferedInputStream bis =new BufferedInputStream(
            new FileInputStream("c:/page.htm") );
        GZIPOutputStream GZIPos = new GZIPOutputStream(
            new FileOutputStream("c:/page.gz") );
        __.println("Writing compressing file from c:/page.txt to c:/page.gz");
        int n;
        byte[] buf=new byte[256]; //字节缓冲区
        while((n = bis.read(buf)) != -1) GZIPos.write(buf,0,n); //写压缩文件
        bis.close();  GZIPos.close();
    }
}
```



文件的压缩与解压缩

```
____.println("Reading file form c:/page.gz to monitor");  
BufferedReader br = new BufferedReader(  
    new InputStreamReader(  
        new GZIPInputStream(  
            new FileInputStream("c:/page.gz") ) ) );
```

```
String s;  
while((s = br.readLine()) != null) ____.println(s);  
br.close();
```

```
____.println("Writing decompression to c:/newPage.htm");  
GZIPInputStream GZIPis=new GZIPInputStream(  
    new FileInputStream("c:/page.gz"));  
FileOutputStream fos=new FileOutputStream("c:/newPage.htm");  
while((n=GZIPis.read(buf))!=-1) fos.write(buf,0,n);  
GZIPis.close();    fos.close();  
}  
}
```



文件的压缩与解压缩

● 运用ZIP压缩多个文件

– Zip文件

- 可能含有多个文件，所以有多个入口 (**Entry**)
- 每个入口用一个**ZipEntity**对象表示，该对象的**getName()**方法返回文件的最初名称

– ZipOutputStream

- 父类是**DeflaterOutputStream**
- 可以把数据压缩成**ZIP**格式

– ZipInputStream

- 父类是**InflaterInputStream**
- 可以把压缩成**ZIP**格式的数据解压缩



文件的压缩与解压缩

【例】 将若干文件压缩至"c:/ZIPtest.zip", 然后解压缩到一个指定的目录"c:/UnzipTest/"

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

```
import java.util.zip.*;
```

```
public class ZipOutputStreamTester {
```

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

```
        int n;
```

```
        byte[] buf=new byte[256];
```

```
        String[] fileNames={"c:/page.htm","c:/Hello.txt","c:/book.dat",  
                             "c:/windows/","c:/windows/winhl32.exe"};
```

```
        ZipOutputStream Zipos=new ZipOutputStream(  
            new BufferedOutputStream(  
                new FileOutputStream("c:/ZIPtest.zip"))));
```

//逐个文件压缩至压缩包

```
for(String fName:fileNames) {
```

```
    __.println("Writing file " + fName);
```

```
    Zipos.putNextEntry(new ZipEntry(fName.substring(3))); //注意
```

这里去掉了根目录名c:/

```
    if(!fName.endsWith("/")){ //如果不是目录，而是文件，则将其压  
缩到压缩包
```

```
        BufferedInputStream bis =new BufferedInputStream(  
            new FileInputStream(fName));
```

```
        while((n = bis.read(buf)) != -1) Zipos.write(buf,0,n);
```

```
        bis.close();
```

```
    }
```

```
}
```

```
Zipos.close();
```

//解压到指定目录

```
String destPath="c:/UnzipTest/";
```

```
File dir=new File(destPath);
```

```
if(!dir.exists()) dir.mkdirs();
```

```

ZipInputStream Zipis= new ZipInputStream(
    new BufferedInputStream(
        new FileInputStream("c:/ZIPtest.zip")));
ZipEntry en=null;
while((en=Zipis.getNextEntry())!=null){
    if(en.isDirectory()){ //如果是路径则创建路径
        dir=new File(destPath+en.getName());
        if(!dir.exists()) dir.mkdirs();
    }
    else{ // 如果是文件则解压缩
        String fName=destPath+en.getName();
        BufferedOutputStream bos=new BufferedOutputStream(
            new FileOutputStream(fName));
        while((n=Zipis.read(buf))!=-1) bos.write(buf, 0, n);
        bos.close();
    }
}
Zipis.close();
}
}

```



随机文件的读与写

- **IO流**（包括**文件IO流**）中的数据只能必须按从前完后的顺序读取。有时可能需要在文件中任意位置进行读写即进行随机存储，例如销售记录的浏览、查找、修改、删除等。
- 随机文件的应用程序必须指定文件的格式。最简单的是要求文件中的所有记录均保持相同的固定长度。利用固定长度的记录，程序可以容易地计算出任何一条记录相对于文件头的确切位置
- **Java.io**包提供了**RandomAccessFile**类用于随机文件的创建和访问



随机文件的读与写

● RandomAccessFile 类

- 与IO流类(只能读或写)不同, 该类即可以读也可以写
- 可跳转到文件的任意位置读/写数据
- 可在随机文件中插入数据, 而不破坏文件中其他数据
- 实现了 **DataInput** 和 **DataOutput** 接口, 可使用普通的读写方法读写基本类型数据
- 有个位置指示器(文件指针), 指向当前读写处的位置。刚打开文件时, 文件指示器指向文件的开头处。对文件指针显式操作的方法有:
 - **int skipBytes(int n)**: 文件指针向前移动n个字节
 - **void seek(long)**: 移动文件指针到指定的位置
 - **long getFilePointer()**: 得到当前的文件指针
- 方便随机读取等长记录格式文件, 但仅限于操作文件, 不能访问其它IO设备, 如网络、内存映像等



随机文件的读与写

● RandomAccessFile 类

- 直接从根类Object派生
- 构造方法
 - RandomAccessFile(File file, String mode) throws FileNotFoundException
 - RandomAccessFile(String name, String mode) throws FileNotFoundException
- 建立一个RandomAccessFile时，要指出你要执行的操作：仅从文件读，还是同时读写
 - new RandomAccessFile("farrago.txt", "r");
 - new RandomAccessFile("farrago.txt", "rw");



随机文件的读与写

— 常用方法（均抛出异常：throws [IOException](#)）

void setLength (long newLength)	设置文件长度，即字节数，文件将被扩展或截短
long length ()	返回文件的长度，即字节数
void seek (long pos)	移动文件指针， pos 指定从文件开头的偏离字节数。可以超过文件总字节数，但只有写操作后，才能扩展文件大小
int skipBytes (int n)	跳过n个字节，返回数为实际跳过的字节数
int read ()	从文件中读取一字节，字节的高24位为0。如遇到结尾，则返回-1
final double readDouble ()	读取8个字节的Double型数据
final String readLine ()	读取一行文本（必须以"/r/n"结束）
final void writeChar (int v)	写入一个字符，两个字节，高位先写入
final void writeChars (String s)	写入一个字符串，2倍字符个数的字节
final void writeInt (int v)	写入四个字节的int型数字
void close ()	关闭文件



随机文件的读与写

【例】创建一个雇员类，包括姓名、年龄。姓名不超过8个字符，年龄是int类型。每条记录固定为20个字节。编写一个雇员管理类，使用RandomAccessFile向文件添加、修改、读取雇员信息记录。

Employee

```
public final static short MAX_LENGTH_of_NAME=8;  
public final static long LENGTH_RECORD=MAX_LENGTH_of_NAME*2+4;  
private char name[]=new char[MAX_LENGTH_of_NAME];  
private int age;
```

```
public Employee(String name,int age);  
public int getAge();  
public void setAge(int age);  
public String getName();  
public void setName(String name);
```

char 类型
为2个字节



随机文件的读与写

【例】创建一个雇员类，包括姓名、年龄。姓名不超过8个字符，年龄是int类型。每条记录固定为20个字节。编写一个雇员管理类，使用RandomAccessFile向文件添加、修改、读取雇员信息记录。

EmployeeManager

```
private String fileName;  
private long recordsCount;  
private RandomAccessFile raf;  
  
public EmployeeManager(String fileName)throws Exception  
private void open() throws Exception;  
public void close() throws Exception;  
public void append(Employee em) throws Exception;  
public void modify(Employee em,long ID)throws Exception;  
public Employee read(long ID)throws Exception
```

//Employee.java

```
public class Employee {  
    public final static short MAX_LENGTH_of_NAME=8; //名字最多8字符,  
        不足部分为空字符 ('\u0000')  
    public final static long LENGTH_RECORD=  
        MAX_LENGTH_of_NAME*2+4; //记录长度(字节)  
    private char name[]=new char[MAX_LENGTH_of_NAME];  
    private int age;  
  
    public Employee(String name,int age){  
        this.setName(name);  
        this.age=age;  
    }  
    public int getAge(){return age;}  
    public void setAge(int age){this.age=age;}  
    public String getName(){ return new String(name);}  
    public void setName(String name){  
        System.arraycopy(name.toCharArray(), 0, this.name, 0,  
            Math.min(MAX_LENGTH_of_NAME, name.length()));  
    }  
}
```

//EmployeeManager.java

```
import java.io.*;

public class EmployeeManager {
    private String fileName;
    private long recordsCount=0; //记录总数
    private RandomAccessFile raf=null;
    public EmployeeManager(String fileName)throws Exception{
        this.fileName=fileName;
        File file=new File(fileName);
        long fileLength=file.length();//文件长度
        if(fileLength>0) {
            recordsCount=fileLength/Employee.LENGTH_RECORD;
        }
        this.open(); // 打开文件
    }
    private void open() throws Exception{ //打开文件
        raf=new RandomAccessFile(fileName, "rw");
    }
    public void close() throws Exception{ //关闭文件
        if(raf!=null) raf.close();
    }
}
```

//EmployeeManager.java

//向文件末尾追加新记录

```
public void append(Employee em) throws Exception{  
    //将文件指针移到文件末尾  
    raf.seek(recordsCount*Employee.LENGTH_RECORD);
```

// 写入记录,注意顺序

```
    raf.writeChars(em.getName());  
    raf.writeInt(em.getAge());
```

```
    recordsCount++; //记录计数器更新
```

```
}
```

//修改编号为ID记录

```
public void modify(Employee em,long ID)throws Exception{  
    if(this.recordsCount<ID || ID<1)  
        throw new Exception("记录编号越界! ");  
    raf.seek(Employee.LENGTH_RECORD*(ID-1)); //定位记录
```

// 写入记录,注意顺序

```
    raf.writeChars(em.getName());  
    raf.writeInt(em.getAge());
```

```
}
```

//EmployeeManager.java

//读取编号为ID记录

```
public Employee read(long ID)throws Exception{  
    if(this.recordsCount<ID || ID<1)  
        throw new Exception("记录编号越界! ");
```

```
    char[] buf=new char[Employee.MAX_LENGTH_of_NAME];  
    int age;
```

```
    raf.seek(Employee.LENGTH_RECORD*(ID-1)); //定位记录
```

//读取记录, ,注意写入时各数据项的顺序

```
    for(int i=0;i<Employee.MAX_LENGTH_of_NAME;i++)  
        buf[i]=raf.readChar();  
    age=raf.readInt();
```

```
    return new Employee(new String(buf),age);
```

```
}
```

```
}
```


//EmployeeManagerTester.java

```
public class EmployeeManagerTester {  
    public static void main(String[] args) {  
        String fileName="C:/employee.dat";  
        Employee em=null;  
        try{ EmployeeManager emm=new EmployeeManager(fileName);  
            emm.append(new Employee("张三",25));  
            emm.append(new Employee("马骐(biāo)",36));  
            emm.append(new Employee("阿迪力.买买提吐热",21));  
  
            em=emm.read(3);  
            System.out.println(em.getName()+"\t"+em.getAge());  
            em=emm.read(1);  
            System.out.println(em.getName()+"\t"+em.getAge());  
  
            emm.modify(new Employee("张三丰",27), 1);  
            em=emm.read(1);  
            System.out.println(em.getName()+"\t"+em.getAge());  
  
            emm.close();  
        }catch(Exception e){e.printStackTrace();}  
    }  
}
```

控制台

<已终止> EmployeeManagerTester

阿迪力.买买提吐	21
张三	25
张三丰	27



小 结

1. File类用于获取文件和目录的信息及其管理
2. IO流主要分为字节流和字符流
 - a) 字符流（抽象基类Reader/Writer）：
 - **InputStreamReader**类用于从外部（设备）读取字符，派生类**FileReader**专门用于从文本文件读取字符
 - **OutputStreamWriter**类用于向外部（设备）写入字符，派生类**FileWriter**专门用于向文本文件写入字符
 - **BufferedReader/BufferedWriter**类对上述字符流进行缓冲，以提高字符IO的效率
 - b) 字节流（抽象基类InputStream/OutputStream）：
 - **FileInputStream/FileOutputStream**用于从文件读取字节/将字节写入文件，为原始字节流之一
 - **BufferedInputStream/BufferedOutputStream**对原始字节流进行缓冲，以提高字节IO的效率
 - **DataInputStream/DataOutputStream**对对原始字节流或缓冲字节流进行包装，实现基本数据类型的IO
3. RandomAccessFile类用于随机存储具有固定格式的文件



习 题

1. P180习题7.1~73大题

2. 上机:

a) 使用FileReader和BufferedReader读取文本文件; 从键盘输入文本, 并FileWriter和BufferedWriter将文本写入文件

【提示】: 从键盘输入

- 可以采用InputStreamReader 和BufferedReader实现
`BufferedReader br=new BufferedReader (new
InputStreamReader(System.in));`
`br.readLine().trim()`用于读取一行键盘输入的文本
- 也可以采用Scanner来实现
`Scanner scan = new Scanner(System.in);`
`scan.nextLine().trim()`用于读取一行键盘输入的文本



习 题

- b) 使用IO流实现任意文件（例如各种图像/音频/视频/文本/office文档文件、压缩文件）的复制
- 编写一个类，提供静态方法Copy(String, String)完成文件复制；在编写一个测试类
 - 提示用户从键盘输入源文件名和目标文件名，出现异常则循环提示（允许用户从循环中退出）
 - 复制操作之前能检测文件是否存在，目标文件如果存在则要询问是否覆盖
- 【提示】：用File类检测文件是否存在
- c) 给例中雇员记录管理类employeeManager添加两个方法
- ① Employ[] readAll(): 用于读取所有记录
 - ② delete(long ID): 用于删除记录



谢谢大家!