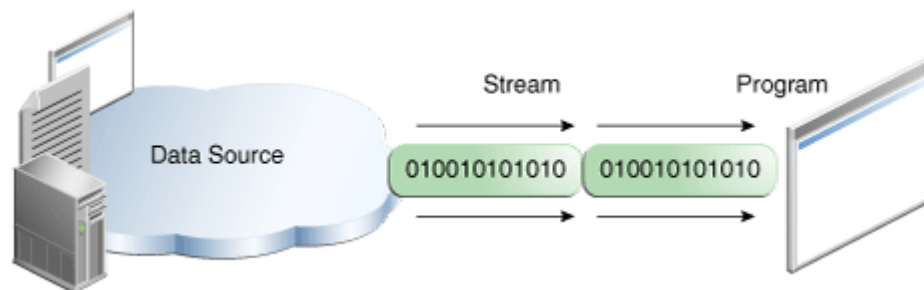


# JAVA程序设计

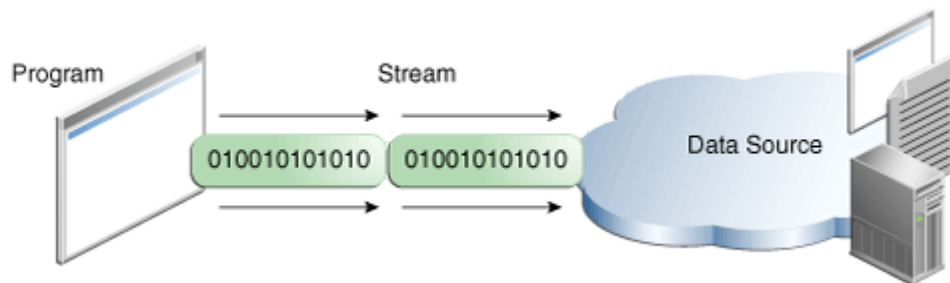
潘微科

感谢：教材《Java大学实用教程》的作者和其他老师提供PowerPoint讲义等资料！  
说明：本课程所使用的所有讲义，都是在以上资料上修改的。

# 引言



**Reading** information **into** a program.



**Writing** information **from** a program.

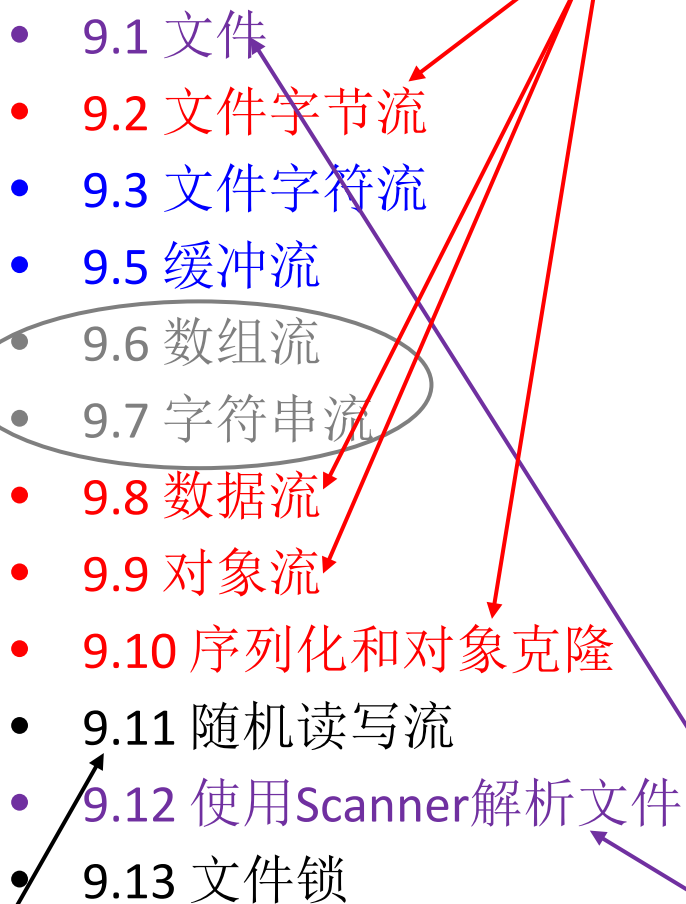
# 引言

- 读写文件时可以使用输入/输出流，简称I/O（input/output）流
- **输入流**（input stream or input **object**）的指向称作“**源**”
- 程序从**输入流**中读取“**源**”中的数据
- **输出流**（output stream or output **object**）的指向称作“**目的地**”
- 程序通过向**输出流**中写入数据，把信息传递到“**目的地**”
- 程序的“源”和“目的地”可以是**文件**、键盘、鼠标、内存或显示器窗口
- 显式地关闭任何一个打开的流是一个好的编程习惯

# 引言

- java.io中有4个重要的**abstract class**
  - InputStream（**字节**输入流）
  - OutputStream（**字节**输出流）
  - Reader（**字符**输入流）
  - Writer（**字符**输出流）

# Outline

- 9.1 文件
  - 9.2 文件字节流
  - 9.3 文件字符流
  - 9.5 缓冲流
  - 9.6 数组流
  - 9.7 字符串流
  - 9.8 数据流
  - 9.9 对象流
  - 9.10 序列化和对象克隆
  - 9.11 随机读写流
  - 9.12 使用Scanner解析文件
  - 9.13 文件锁
- 

# Outline

- 9.1 文件
- 9.12 使用Scanner解析文件
- 9.3 文件字符流
- 9.5 缓冲流
- 9.2 文件字节流
- 9.8 数据流
- 9.9 对象流
- 9.10 序列化和对象克隆
- 9.11 随机读写流
- 9.13 文件锁
- 9.6 数组流
- 9.7 字符串流

注：为了便于讲解，顺序做了适当调整

## 9.1 文件

- File类
  - Java使用File类创建的对象来获取文件本身的一些信息，如文件所在的目录、文件的长度、文件的读写权限等，文件对象并不涉及对文件的读写操作。
- 构造方法
  - File(String filename);
  - File(String directoryPath, String filename);
  - ...

## 9.1 文件

- 1.文件的属性
  - `public String getName()`: 获取文件的`名字`
  - `public boolean canRead()`: 判断文件`是否可读`
  - `public boolean canWrite()`: 判断文件`是否可被写入`
  - `public boolean exists()`: 判断文件`是否存在`
  - `public long length()`: 获取文件的`长度`（单位是`字节`）
  - `public String getAbsolutePath()`: 获取文件的`绝对路径`
  - `public String getParent()`: 获取文件的`父目录`
  - `public boolean isFile()`: 判断文件`是否是一个正常的文件`
  - `public boolean isDirectory()`: 判断文件`是否是一个目录`
  - `public boolean isHidden()`: 判断文件`是否是隐藏文件`
  - `public long lastModified()`: 获取文件`最后修改的时间`



# 9.1 文件

- 2.目录
- (1)创建目录
  - File类的对象可以调用`public boolean mkdir()`: 创建一个目录
- (2)列出目录中的文件（如果File对象是一个目录）
  - `public String[] list()`: 用字符串数组的形式返回目录下的全部文件
  - `public String[] list(FilenameFilter ff)`: 用字符串数组的形式返回目录下指定类型的全部文件
  - `public File[] listFiles()`: 用File对象数组的形式返回目录下的全部文件
  - `public File[] listFiles(FilenameFilter ff)`: 用File对象数组的形式返回目录下指定类型的全部文件

## 9.1 文件

- 3.文件的创建与删除

- 当使用File类创建一个文件对象后，例如：

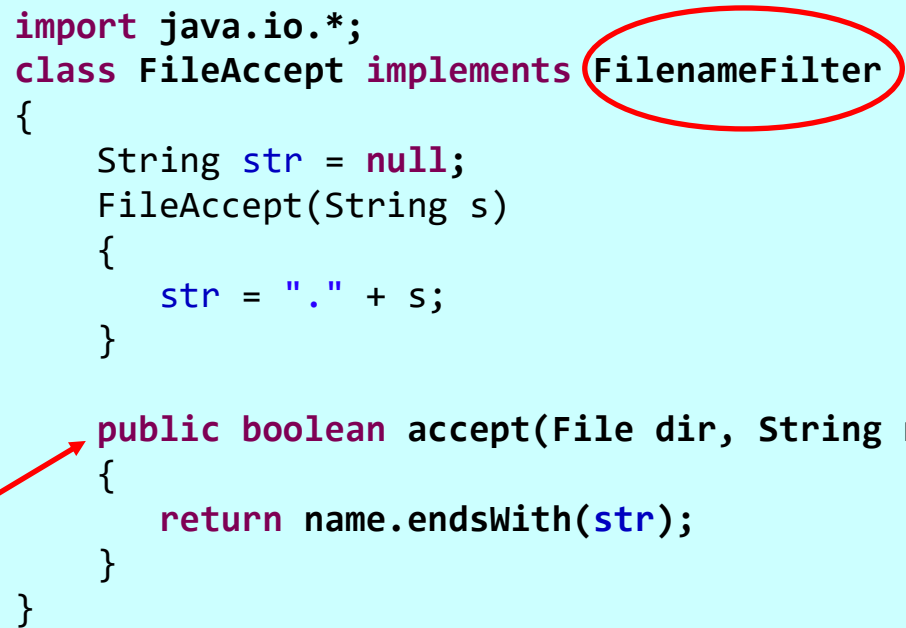
```
File file = new File("c:\\myletter","letter.txt");
```

- 如果c:\myletter目录中**没有**名字为letter.txt的文件，文件对象file需要调用public boolean createNewFile()，即 `file.createNewFile();`，从而在c:\myletter目录中建立一个名字为letter.txt的文件。
  - 如果c:\myletter目录中**已有**名字为letter.txt的文件，则打开这个文件。
- 文件对象调用方法public boolean delete()可以删除当前文件，例如：  
`file.delete();`

## 9.1 文件

```
import java.io.*;
class FileAccept implements FilenameFilter
{
    String str = null;
    FileAccept(String s)
    {
        str = "." + s;
    }

    public boolean accept(File dir, String name)
    {
        return name.endsWith(str);
    }
}
```



```
public class Example9_1
{
    public static void main(String args[])
    {
        //File dir = new File("C:/ch8"); // 推荐使用
        File dir = new File("C:\\ch8");
        // File dir = new File("C/ch8"); // illegal
        // File dir = new File("C:\\ch8"); // illegal

        FileAccept fileAccept = new FileAccept("java");
        File[] files = dir.listFiles(fileAccept);
        for(int i=0; i<files.length; i++)
        {
            System.out.println(files[i].getName() + ": " + files[i].length());
        }

        boolean flag = false;
        if(files.length>0) // 如果至少有一个文件
        {
            flag = files[0].delete();
        }
        if(flag)
        {
            System.out.println(files[0].getName() + " has been deleted.");
        }
    }
}
```

```
test.java: 0
test.java has been deleted.
```

## 9.1 文件

- 4.运行可执行文件
- 使用Runtime类声明一个对象
- 使用静态方法getRuntime()创建这个对象

```
Runtime rt = Runtime.getRuntime();
```

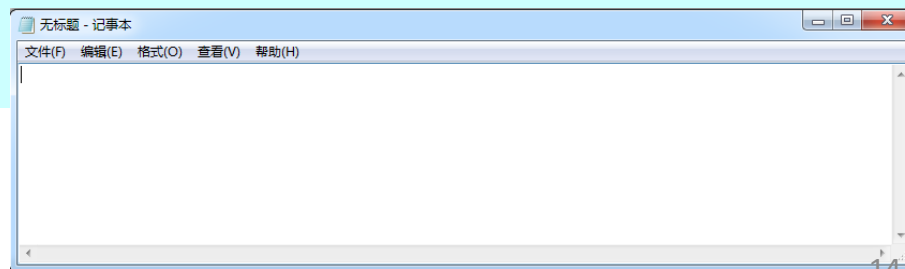
- rt可以调用exec(String command)方法打开本地机器的可执行文件或执行一个操作。

## 9.1 文件

- 【例子2】

```
import java.io.*;

public class Example9_2
{
    public static void main(String args[])
    {
        try
        {
            Runtime rt = Runtime.getRuntime();
            File file = new File("C:\\windows", "Notepad.exe");
            rt.exec(file.getAbsolutePath());
        }
        catch(Exception e){}
    }
}
```



# Outline

- 9.1 文件
- 9.12 使用Scanner解析文件
- 9.3 文件字符流
- 9.5 缓冲流
- 9.2 文件字节流
- 9.8 数据流
- 9.9 对象流
- 9.10 序列化和对象克隆
- 9.11 随机读写流
- 9.13 文件锁
- 9.6 数组流
- 9.7 字符串流

注：为了便于讲解，顺序做了适当调整

## 9.12 使用Scanner解析文件

- 应用程序可能需要解析文件中的特殊数据，此时，应用程序可以先把文件的内容全部读入内存后，再使用第6章的有关知识解析所需要的内容，其优点是处理速度快，但如果读入的内容较多，将消耗较多的内存，这就是所谓的“以空间（内存）换时间”。
- 本节介绍怎样借助Scanner类和正则表达式来解析文件，比如，要解析出文件中的特殊单词、数字等信息。使用Scanner类和正则表达式来解析文件的特点是“以时间换空间（内存）”，解析的速度相对较慢，但可以节省内存。



## 9.12 使用Scanner解析文件

- 1.使用默认分隔符标记解析文件
- 创建Scanner对象，并指向要解析的文件，例如：

```
File file = new File("hello.java");  
Scanner scanner = new Scanner(file);
```

- 那么scanner将**空格**作为分隔标记、调用next()方法依次返回file中的单词，如果file最后一个单词已被next()方法返回，scanner调用hasNext()将返回false，否则返回true。
- 对于数字型的单词，比如108、167.92等可以用nextInt()或nextDouble()方法来代替next()方法。但需要特别注意的是，如果单词不是数字型单词，调用nextInt()或nextDouble()方法将发生**InputMismatchException**异常。在处理异常时可以调用next()方法返回该非数字化单词（见后面的例子）。

## 9.12 使用Scanner解析文件

- D:/chp09/cost.txt中的内容如下：  
TV cost 876 dollar, Computer cost 2398 dollar. The milk cost 98 dollar. The apple cost 198 dollar.
- 使用Scanner对象解析文件cost.txt中的全部消费并计算出总消费

```

import java.io.*;
import java.util.*;
public class Demo
{
    public static void main(String args[])
    {
        File file = new File("D:\\chp09\\cost.txt");
        Scanner scanner = null;
        int sum=0;
        try{
            scanner = new Scanner(file);
            while(scanner.hasNext()){
                try{
                    int price = scanner.nextInt();
                    sum = sum + price;
                    System.out.println(price);
                }
                catch(InputMismatchException exp){
                    String t = scanner.next();
                }
            }
            System.out.println("Total Cost:"+sum+" dollar");
        }
        catch(Exception exp){ System.out.println(exp); }
    }
}

```

```


876
2398
98
198
Total Cost:3570 dollar

```

## 9.12 使用Scanner解析文件

- 2.使用正则表达式作为分隔标记解析文件
- 创建Scanner对象，指向要解析的文件，并使用**useDelimiter()**方法指定正则表达式作为分隔标记，例如：

```
File file = new File("hello.java");  
Scanner scanner = new Scanner(file);  
scanner.useDelimiter(正则表达式);
```




- 那么，scanner将正则表达式作为**分隔标记**。

## 9.12 使用Scanner解析文件

- 使用正则表达式（匹配所有**非数字字符串**）：
- `String regex="[^0123456789.]+"` 作为分隔标记解析communicate.txt文件中的通信费用。
- communicate.txt的内容如下：  
市话费:**176.89**元,长途费:**187.98**元,网络费:**928.66**元

```
import java.io.*;
import java.util.*;
public class Demo{
    public static void main(String args[]){
        File file = new File("D:\\chp09\\communicate.txt");
        Scanner scanner = null;
        double sum = 0;
        try {
            double fare=0;
            scanner = new Scanner(file);
            scanner.useDelimiter("[^0123456789.]+");
            while(scanner.hasNextDouble()){
                fare = scanner.nextDouble();
                sum = sum+fare;
                System.out.println(fare);
            }
            System.out.println("Total: " + sum);
        }
        catch(Exception exp){
            System.out.println(exp);
        }
    }
}
```



```
176.89
187.98
928.66
Total: 1293.53
```

## 9.12 使用Scanner解析文件

- 3.单词记忆训练
- 基于文本文件的英文单词训练程序，具体内容如下：
  - 文本文件D:/chp09/word.txt中的内容由英文单词所构成，单词之间用空格分隔，例如：first boy girl hello well。
  - 使用Scanner解析word.txt中的单词，并显示在屏幕上，然后要求用户输入该单词。
  - 当用户输入单词时，程序将从屏幕上隐藏掉刚刚显示的单词，以便考核用户是否清晰地记住了这个单词。
  - 程序读取了word.txt的全部内容后，将统计出用户背单词的正确率。

# Outline

- 9.1 文件
- 9.12 使用Scanner解析文件
- 9.3 文件字符流
- 9.5 缓冲流
- 9.2 文件字节流
- 9.8 数据流
- 9.9 对象流
- 9.10 序列化和对象克隆
- 9.11 随机读写流
- 9.13 文件锁
- 9.6 数组流
- 9.7 字符串流

注：为了便于讲解，顺序做了适当调整



## 9.3 文件字符流


- 1.FileReader类
- 构造方法
  - FileReader(String name)
  - FileReader(File file)
- 常用方法
  - int read(): 读取一个字符（即2个字节），返回0~65535之间的一个整数（Unicode字符值），如果未读出字符就返回-1。
  - int read(char b[]): 读取b.length个字符到字符数组b中，返回实际读取的字符数目；如果到达文件的末尾，则返回-1。
  - int read(char b[], int off, int len): 读取len个字符并存放到字符数组b中，返回实际读取的字符数目；如果到达文件的末尾，则返回-1。其中，off参数指定read方法在字符数组b中的什么地方存放数据。

## 9.3 文件字符流

- 2.FileWriter类
- 构造方法
  - FileWriter(String name)
  - FileWriter(File file)
- 常用方法
  - void write(char b[]): 将b.length个字符写到[输出流](#)
  - void write(char b[], int off, int len): 从给定[字符数组](#)中起始于偏移量off处开始写len个字符到[输出流](#)，参数b是存放了数据的字符数组
  - void write(String str): 把[字符串](#)中的全部字符写到[输出流](#)
  - void write(String str, int off, int len): 从[字符串](#)str中起始于偏移量off处开始写len个字符到[输出流](#)

```
import java.io.*;
public class Example9_4
{
    public static void main(String args[])
    {
        File file = new File("hello.txt");
        char b[] = "深圳大学".toCharArray();
        try{
            FileWriter output = new FileWriter(file);
            output.write(b); // 字符数组
            output.write("脚踏实地! "); // 字符串
            output.close();

            FileReader input = new FileReader(file);
            int n=0;
            while((n=input.read(b,0,2))!=-1){ // 最多读2个字符
                String str = new String(b,0,n); // 转换为字符串
                System.out.println(str);
            }
            input.close();
        }
        catch(IOException e){
            System.out.println(e);
        }
    }
}
```



深圳  
大学  
脚踏  
实地  
!

# Outline

- 9.1 文件
- 9.12 使用Scanner解析文件
- 9.3 文件字符流
- 9.5 缓冲流
- 9.2 文件字节流
- 9.8 数据流
- 9.9 对象流
- 9.10 序列化和对象克隆
- 9.11 随机读写流
- 9.13 文件锁
- 9.6 数组流
- 9.7 字符串流

注：为了便于讲解，顺序做了适当调整

## 9.5 缓冲流

- 1.BufferedReader类
- BufferedReader的构造方法：
  - BufferedReader(Reader in)
  - BufferedReader流能够读取文本行，方法是readLine()

我个人在科研数据的读取和分析中用得比较多的是  
FileReader和BufferedReader，实现按行读取的目的

## 9.5 缓冲流

- 通过向BufferedReader传递一个Reader对象（如FileReader对象），来创建一个BufferedReader对象，如：

```
FileReader fr = new FileReader("Student.txt");  
BufferedReader input = new BufferedReader(fr);
```

- 然后，input调用**readLine()**顺序读取文件Student.txt中的一行。

## 9.5 缓冲流

- 2.BufferedWriter类
- 类似地，可以将BufferedWriter流和FileWriter流连接在一起，然后使用BufferedWriter流将数据写到目的地，例如：

```
FileWriter fw = new FileWriter("hello.txt");  
BufferedWriter output = new BufferedWriter(fw);
```

- BufferedWriter流调用如下方法，把字符串s或s的一部分写入到目的地
  - write(String s)
  - write(String s, int off, int len)



```
import java.io.*;
public class Example9_5{
    public static void main(String args[]){
        try{
            FileReader fr = new FileReader("input.txt");
            BufferedReader input = new BufferedReader(fr);
            FileWriter fw = new FileWriter("output.txt");
            BufferedWriter output = new BufferedWriter(fw);

            String s=null;
            int i=0;
            while((s = input.readLine())!=null){
                i++;
                output.write(i + ": " + s);
                output.newLine();
            }
            output.flush(); output.close(); fw.close();
            input.close(); fr.close();
        }
        catch(IOException e){
            System.out.println(e);
        }
    }
}
```

| input.txt                        | output.txt                          |
|----------------------------------|-------------------------------------|
| 1 Welcome to Shenzhen University | 1 1: Welcome to Shenzhen University |
| 2 Welcome to Java Programming    | 2 2: Welcome to Java Programming    |

| input.txt                        | output.txt                          |
|----------------------------------|-------------------------------------|
| 1 Welcome to Shenzhen University | 1 1: Welcome to Shenzhen University |
| 2 Welcome to Java Programming    | 2 2: Welcome to Java Programming 32 |



# Outline

- 9.1 文件
- 9.12 使用Scanner解析文件
- 9.3 文件字符流
- 9.5 缓冲流
- 9.2 文件字节流
- 9.8 数据流
- 9.9 对象流
- 9.10 序列化和对象克隆
- 9.11 随机读写流
- 9.13 文件锁
- 9.6 数组流
- 9.7 字符串流

注：为了便于讲解，顺序做了适当调整

## 9.2 文件字节流

- 1. `FileInputStream`类
- 为了创建 `FileInputStream` 类的对象，可以使用下列构造方法：
  - `FileInputStream(String name)`
  - `FileInputStream(File file)`

## 9.2 文件字节流

- 输入流的唯一目的是提供通往数据的通道，程序可以通过这个通道读取数据，`read()`方法给程序提供一个从输入流中读取数据的基本方法。
- `read()`方法从输入流中**顺序**读取**单个字节**的数据。该方法返回字节值（0~255之间的一个整数），读取位置到达文件末尾，则返回-1。
- `read()`方法还有其他一些形式。这些形式能使程序把多个字节读到一个字节数组中：
  - `int read(byte b[]);`
  - `int read(byte b[], int off, int len);` 其中，`off`参数指定`read()`方法把数据存放在字节数组**b**中的什么地方，`len`参数指定该方法将要读取的最大字节数。上面所示的这两个`read()`方法都返回实际读取的字节数，如果它们到达输入流的末尾，则返回-1。

## 9.2 文件字节流

- 2. `FileOutputStream`类
- 构造方法
  - `FileOutputStream(String name)`
  - `FileOutputStream(File file)`
- 输出流通过使用`write()`方法把数据写入输出流到达目的地
  - `public void write(byte b[])`: 写`b.length`个字节到输出流
  - `public void write(byte b[], int off, int len)`: 从给定字节数组中起始于偏移量`off`处开始写`len`个字节到输出流, 参数`b`是存放了数据的字节数组

```
import java.io.*;
public class Example9_3
{
    public static void main(String args[])
    {
        File file = new File("hello.txt");
        byte b[] = "深圳大学".getBytes();
        try{
            FileOutputStream output = new FileOutputStream(file);
            output.write(b); // 字节数组
            output.close();

            FileInputStream input = new FileInputStream(file);
            int n=0;
            while( (n=input.read(b,0,3))!=-1 ) // 最多读3个字节
            {
                String str = new String(b,0,n); // 转换为字符串
                System.out.println(str);
            }
        }
        catch(IOException e){
            System.out.println(e);
        }
    }
}
```

# Outline

- 9.1 文件
- 9.12 使用Scanner解析文件
- 9.3 文件字符流
- 9.5 缓冲流
- 9.2 文件字节流
- 9.8 数据流
- 9.9 对象流
- 9.10 序列化和对象克隆
- 9.11 随机读写流
- 9.13 文件锁
- 9.6 数组流
- 9.7 字符串流

注：为了便于讲解，顺序做了适当调整

## 9.8 数据流

- 1.DataInputStream类和DataOutputStream类
- DataInputStream类创建的对象称为数据输入流
- DataOutputStream类创建的对象称为数据输出流

## 9.8 数据流

- 2.DataInputStream类和DataOutputStream类的构造方法
- DataInputStream(**InputStream** is)
- DataOutputStream(**OutputStream** os)



## 9.8 数据流

- 表9.1（见书182页）给出了DataInputStream类和DataOutputStream类的常用方法。

```
import java.io.*;
public class Example9_8
{
    public static void main(String args[])
    {
        try{
            FileOutputStream fos = new FileOutputStream("jerry.dat");
            DataOutputStream output = new DataOutputStream(fos);
            output.writeInt(100);
            output.writeChars("I am ok");
        }
        catch(IOException e){}

        try{
            FileInputStream fis = new FileInputStream("jerry.dat");
            DataInputStream input = new DataInputStream(fis);

            System.out.println(input.readInt());
            char c;
            while((c=input.readChar())!='\0') //' \0'表示空字符
                System.out.print(c);
        }
        catch(IOException e){}
    }
}
```



```
100
I am ok
```

# Outline

- 9.1 文件
- 9.12 使用Scanner解析文件
- 9.3 文件字符流
- 9.5 缓冲流
- 9.2 文件字节流
- 9.8 数据流
- 9.9 对象流
- 9.10 序列化和对象克隆
- 9.11 随机读写流
- 9.13 文件锁
- 9.6 数组流
- 9.7 字符串流

注：为了便于讲解，顺序做了适当调整


## 9.9 对象流

- 1.ObjectInputStream类和ObjectOutputStream类
- ObjectInputStream类创建的对象被称为对象输入流
- ObjectOutputStream类创建的对象被称为对象输出流
- 对象输出流使用writeObject(Object obj)方法将一个对象obj写入输出流
- 对象输入流使用readObject()从源中读取一个对象到程序中
- 构造方法
  - ObjectInputStream(InputStream in)
  - ObjectOutputStream(OutputStream out)

## 9.9 对象流

- Java提供给我们的绝大多数对象都是序列化的，比如组件等。
- 一个类如果实现了 **Serializable接口**，那么这个类创建的对象就是所谓的序列化的对象（a serializable object）。
- **Serializable接口中的方法对程序是不可见的**，因此实现该接口的类不需要实现额外的方法，当把一个序列化的对象写入到对象输出流时，JVM就会实现Serializable接口中的方法，进而按一定格式的文本将对象写入到目的地。

## 9.9 对象流



```
import java.io.*;
class Goods implements Serializable
{
    String name = null;
    double unitPrice;
    Goods(String name, double unitPrice){
        this.name=name;
        this.unitPrice=unitPrice;
    }
    public void setUnitPrice(double unitPrice){
        this.unitPrice=unitPrice;
    }
    public double getUnitPrice(){
        return unitPrice;
    }
    public void setName(String name){
        this.name=name;
    }
    public String getName(){
        return name;
    }
}
```

## 9.9 对象流

```
public class Example9_9
{
    public static void main(String args[])
    {
        Goods TV1 = new Goods("HaierTV",3468);
        try{
            FileOutputStream fileOut = new FileOutputStream("a.txt");
            ObjectOutputStream objectOut = new ObjectOutputStream(fileOut);
            → objectOut.writeObject(TV1);

            FileInputStream fileIn = new FileInputStream("a.txt");
            ObjectInputStream objectIn = new ObjectInputStream(fileIn);
            → Goods TV2 = (Goods)objectIn.readObject();

            TV2.setUnitPrice(8888);
            TV2.setName("GreatWall");
            System.out.printf("\nTv1:%s,%f",TV1.getName(),TV1.getUnitPrice());
            System.out.printf("\nTv2:%s,%f",TV2.getName(),TV2.getUnitPrice());
        }
        catch(Exception event){
            System.out.println(event);
        }
    }
}
```

Tv1:HaierTV,3468.000000  
Tv2:GreatWall,8888.000000

# Outline

- 9.1 文件
- 9.12 使用Scanner解析文件
- 9.3 文件字符流
- 9.5 缓冲流
- 9.2 文件字节流
- 9.8 数据流
- 9.9 对象流
- 9.10 序列化和对象克隆
- 9.11 随机读写流
- 9.13 文件锁
- 9.6 数组流
- 9.7 字符串流

注：为了便于讲解，顺序做了适当调整



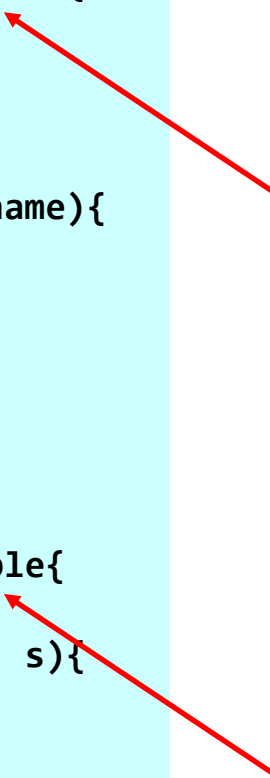
## 9.10 序列化和对象克隆

- 使用对象流很容易获取一个序列化对象的**深度克隆**（原对象有引用型变量的时候）。
- 我们只需将该对象写入到**对象输出流**，然后用**对象输入流**读回的对象就是原对象的一个**深度克隆**。

## 9.10 序列化和对象克隆

```
import java.io.*;
class Goods implements Serializable{
    String name=null;
    Goods(String name){
        this.name=name;
    }
    public void setName(String name){
        this.name=name;
    }
    public String getName(){
        return name;
    }
}

class Shop implements Serializable{
    Goods goods[];
    public void setGoods(Goods[] s){
        goods=s;
    }
    public Goods[] getGoods(){
        return goods;
    }
}
```



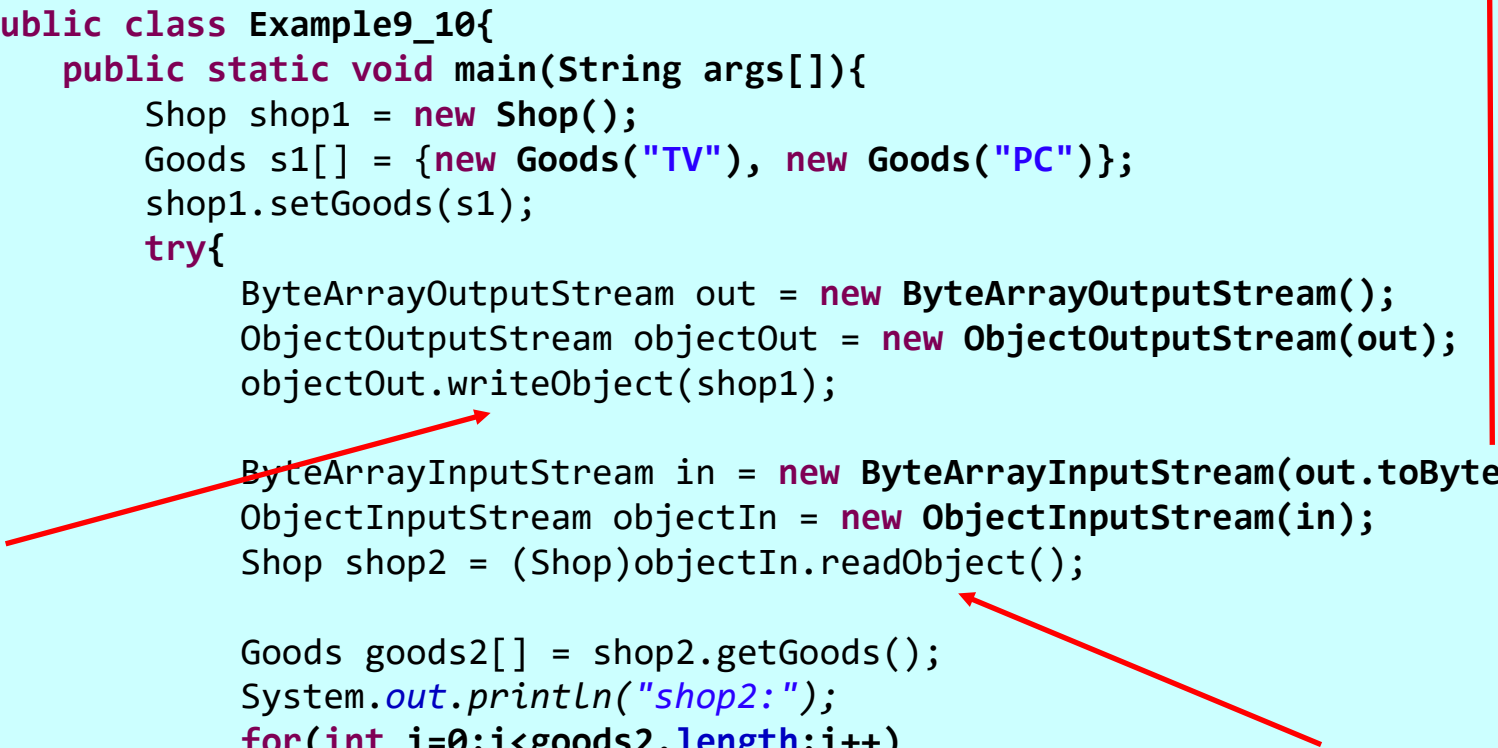
## 9.10 序列化和对象克隆

返回输出流写入到缓冲区的全部字节

```
public class Example9_10{
    public static void main(String args[]){
        Shop shop1 = new Shop();
        Goods s1[] = {new Goods("TV"), new Goods("PC")};
        shop1.setGoods(s1);
        try{
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            ObjectOutputStream objectOut = new ObjectOutputStream(out);
            objectOut.writeObject(shop1);

            ByteArrayInputStream in = new ByteArrayInputStream(out.toByteArray());
            ObjectInputStream objectIn = new ObjectInputStream(in);
            Shop shop2 = (Shop)objectIn.readObject();

            Goods goods2[] = shop2.getGoods();
            System.out.println("shop2:");
            for(int i=0;i<goods2.length;i++)
                System.out.println(goods2[i].getName());
        }
        catch(Exception event){
            System.out.println(event);
        }
    }
}
```



**shop2:**  
TV  
PC

# Outline

- 9.1 文件
- 9.12 使用Scanner解析文件
- 9.3 文件字符流
- 9.5 缓冲流
- 9.2 文件字节流
- 9.8 数据流
- 9.9 对象流
- 9.10 序列化和对象克隆
- 9.11 随机读写流
- 9.13 文件锁
- 9.6 数组流
- 9.7 字符串流

注：为了便于讲解，顺序做了适当调整

## 9.11 随机读写流

- RandomAccessFile类的构造方法
  - RandomAccessFile(String name, String mode): 参数name用来确定一个文件名，给出创建的流的**源**，也是流的**目的地**。参数mode取**r**（只读）或**rw**（可读写），决定创建的流对文件的访问权限。
  - RandomAccessFile(File file, String mode): 参数file是一个File对象，给出创建的流的**源**，也是流的**目的地**。参数mode取**r**（只读）或**rw**（可读写），决定创建的流对文件的访问权利。

If the file is **not intended to be modified**, open it with the **“r” mode**.  
This prevents unintentional modification of the file.

## 9.11 随机读写流

- A random-access file consists of **a sequence of bytes**.
- A special marker called a ***file pointer*** is positioned at one of these **bytes**.
- A read or write operation takes place **at the location of the file pointer**.
- When a file is opened, the file pointer is set at the beginning of the file.
- When you read or write data to the file, the file pointer **moves forward to the next data item**.

## 9.11 随机读写流

- 表9.2（见书189页）给出了RandomAccessFile的常用方法。
- 例子11中我们把几个int型整数写入到一个名字为tom.dat的文件中，然后按相反顺序读出这些数据。
- 例子12中RandomAccessFile流使用readLine()读取一个文件。

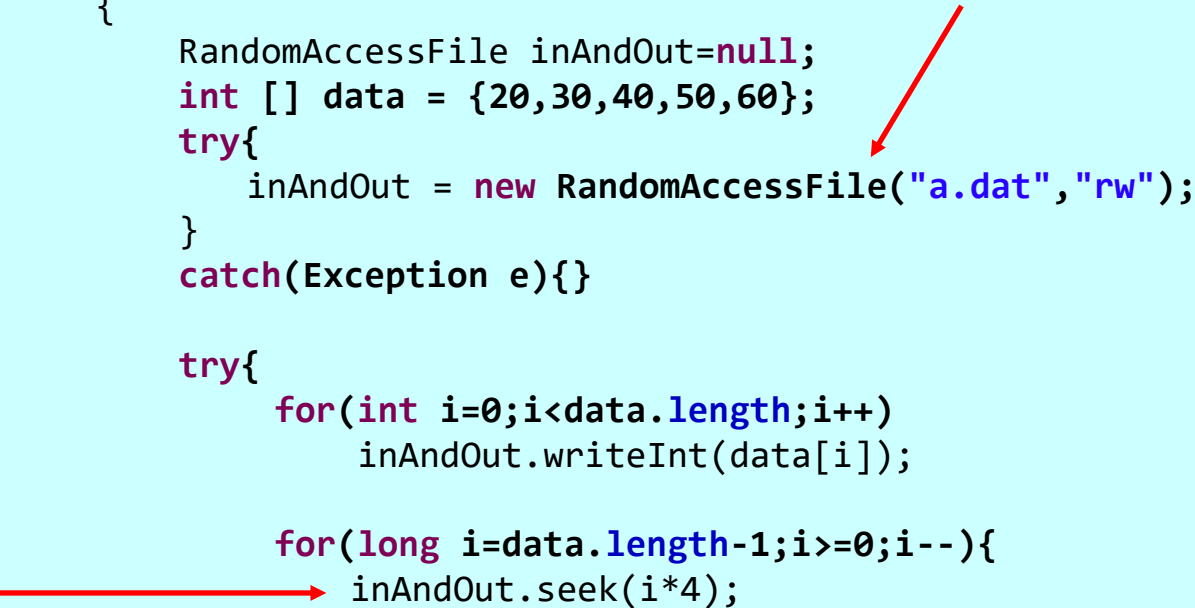
## 9.11 随机读写流

- 【例子11】{

```
import java.io.*;
public class Example9_11
{
    public static void main(String args[])
    {
        RandomAccessFile inAndOut=null;
        int [] data = {20,30,40,50,60};
        try{
            inAndOut = new RandomAccessFile("a.dat","rw");
        }
        catch(Exception e){}

        try{
            for(int i=0;i<data.length;i++)
                inAndOut.writeInt(data[i]);

            for(long i=data.length-1;i>=0;i--){
                inAndOut.seek(i*4);
                System.out.println(inAndOut.readInt());
            }
            inAndOut.close();
        }
        catch(IOException e){}
    }
}
```



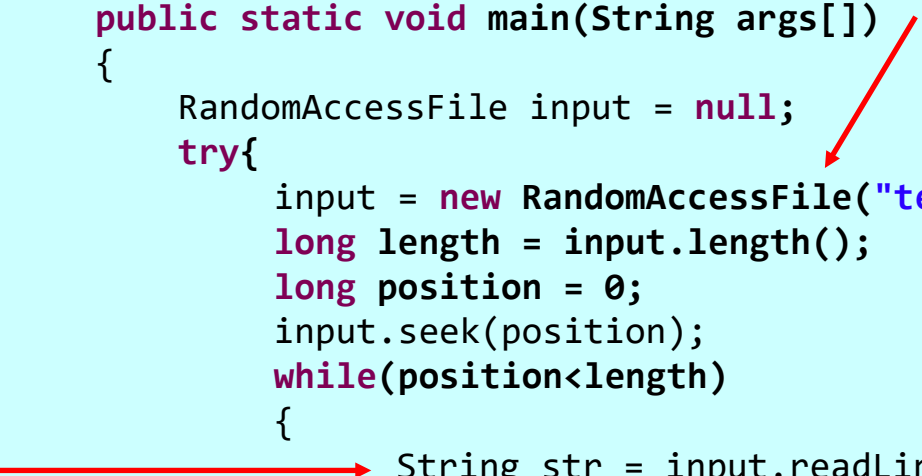
60  
50  
40  
30  
20



## 9.11 随机读写流

- 【例子12】

```
import java.io.*;
public class Example9_12
{
    public static void main(String args[])
    {
        RandomAccessFile input = null;
        try{
            input = new RandomAccessFile("test.txt","rw");
            long length = input.length();
            long position = 0;
            input.seek(position);
            while(position<length)
            {
                String str = input.readLine();
                position = input.getFilePointer();
                System.out.println(str);
            }
        }
        catch(IOException e){}
    }
}
```



Welcome to Shenzhen University  
Welcome to Java Programming

# Outline

- 9.1 文件
- 9.12 使用Scanner解析文件
- 9.3 文件字符流
- 9.5 缓冲流
- 9.2 文件字节流
- 9.8 数据流
- 9.9 对象流
- 9.10 序列化和对象克隆
- 9.11 随机读写流
- 9.13 文件锁
- 9.6 数组流
- 9.7 字符串流

注：为了便于讲解，顺序做了适当调整

## 9.13 文件锁

- JDK1.4增加了一个FileLock类，该类的对象称作文件锁。
- RandomAccessFile创建的流在读写文件时可以使用文件锁，那么只要不解除该锁，其他线程无法操作被锁定的文件。

## 9.13 文件锁

- 使用文件锁的步骤如下：
- Step 1: 使用RandomAccessFile流建立指向文件的**流对象**，该对象的读写属性必须是"rw"

```
RandomAccessFile input = new RandomAccessFile("Example.java","rw");
```

- Step 2: 流对象input调用getChannel()方法获得一个连接到底层文件的**FileChannel 对象**（信道）

```
FileChannel channel = input.getChannel();
```

- Step 3: 信道调用tryLock()或lock()方法获得一个**FileLock（文件锁）对象**，这一过程也称作**对文件加锁**

```
FileLock lock = channel.tryLock();
```

## 9.13 文件锁

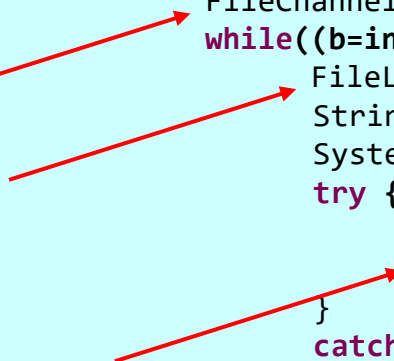
- 另外，`FileInputStream`以及`FileOutputStream`在读/写文件时都可以获得文件锁。
- 在下面的例子13中Java程序在读取文件`test.txt`时，使用了文件锁，这时你无法用其他程序来操作文件`test.txt`，比如在Java程序结束前，你用Windows下的"记事本"（`Notepad.exe`）也无法修改、保存`test.txt`。

## 9.13 文件锁

- 【例子13】

```
import java.io.*;
import java.nio.channels.*;

public class Example9_13
{
    public static void main(String args[]){
        int b;
        byte tom[] = new byte[12];
        try{
            RandomAccessFile input = new RandomAccessFile("test.txt","rw");
            FileChannel channel = input.getChannel();
            while((b=input.read(tom,0,10))!=-1){
                FileLock lock = channel.tryLock();
                String s = new String (tom, 0, b);
                System.out.print(s);
                try {
                    Thread.sleep(1000);
                    lock.release();
                }
                catch(Exception eee){
                    System.out.println(eee);
                }
            }
            input.close();
        }
        catch(Exception ee){
            System.out.println(ee);
        }
    }
}
```



# Outline

- 9.1 文件
- 9.12 使用Scanner解析文件
- 9.3 文件字符流
- 9.5 缓冲流
- 9.2 文件字节流
- 9.8 数据流
- 9.9 对象流
- 9.10 序列化和对象克隆
- 9.11 随机读写流
- 9.13 文件锁
- 9.6 数组流
- 9.7 字符串流

注：为了便于讲解，顺序做了适当调整

## 9.6 数组流

使用字节数组作为流的源和目的地

- 字节输入流： `ByteArrayInputStream`
- 字节输出流： `ByteArrayOutputStream`
- 分别使用**字节数组**作为流的源和目的地
- 构造方法
  - `ByteArrayInputStream(byte[] buf)`
  - `ByteArrayInputStream(byte[] buf, int offset, int length)`
- 第一个构造方法构造的数组字节流的**源**是**参数buf指定的数组的全部字节单元**。
- 第二个构造方法构造的数组字节流的**源**是参数buf指定的数组从offset处开始取的length个字节单元。



## 9.6 数组流

- 构造方法
  - `ByteArrayOutputStream()`
  - `ByteArrayOutputStream(int size)`
- 第一个构造方法构造的数组字节输出流指向一个默认大小为32个字节的缓冲区，如果输出流向缓冲区写入的字节个数大于缓冲区时，缓冲区的容量会自动增加。
- 第二个构造方法构造的数组字节输出流指向的缓冲区的初始大小由参数size指定，如果输出流向缓冲区写入的字节个数大于缓冲区时，缓冲区的容量会自动增加。

## 9.6 数组流

- `public byte[] toByteArray():`
  - 在程序Example9\_10中有用到（返回输出流写入到缓冲区的全部字节）
- 数组字节流读写操作不会发生IOException异常。
- 在下面的例子6中，我们向内存（输出流的缓冲区）写入ASCII表，然后再读出这些字节和字节对应的字符。

## 9.6 数组流

- 【例子6】

返回输出流写入到缓冲区的全部字节

```
import java.io.*;
public class Example9_6
{
    public static void main(String args[])
    {
        int n=-1;
        ByteArrayOutputStream output = new ByteArrayOutputStream();
        for(int i=0;i<5;i++)
        {
            output.write('A'+i);
        }

        ByteArrayInputStream input = new ByteArrayInputStream(output.toByteArray());
        while((n=input.read())!=-1)
        {
            System.out.println(n + ":" + (char)n);
        }
    }
}
```

65:A  
66:B  
67:C  
68:D  
69:E

## 9.6 数组流

- 与数组**字节**流对应的是数组**字符**流
  - CharArrayReader
  - CharArrayWriter
- 与数组字节流不同的是，数组字符流的读操作可能发生IOException异常，因此需要有try-catch语句。
- 在下面的例子7中，我们将Unicode表中的一些字符写入**内存**，然后再读出。

## 9.6 数组流

```
import java.io.*;
public class Example9_7
{
    public static void main(String args[])
    {
        int n=-1;
        CharArrayWriter output = new CharArrayWriter();
        for(int i=65;i<=69;i++)
        {
            output.write(i);
        }

        CharArrayReader input = new CharArrayReader(output.toCharArray());
        try
        {
            while((n=input.read())!=-1)
            {
                System.out.println(n + ":" + (char)n);
            }
        }
        catch(IOException e){}
    }
}
```

返回输出流写入到缓冲区的全部字符

→ try

|    |   |   |
|----|---|---|
| 65 | : | A |
| 66 | : | B |
| 67 | : | C |
| 68 | : | D |
| 69 | : | E |

# Outline

- 9.1 文件
- 9.12 使用Scanner解析文件
- 9.3 文件字符流
- 9.5 缓冲流
- 9.2 文件字节流
- 9.8 数据流
- 9.9 对象流
- 9.10 序列化和对象克隆
- 9.11 随机读写流
- 9.13 文件锁
- 9.6 数组流
- 9.7 字符串流

注：为了便于讲解，顺序做了适当调整

## 9.7 字符串流

- **StringReader**使用字符串作为流的源。
- 构造方法： `public StringReader(String s)`
- 该构造方法构造的**输入流**指向**参数s指定的字符串**
- `public int read()`: 顺序读出源中的一个字符，并返回字符在Unicode表中的位置。
- `public int read(char[] buf, int off, int len)`: 顺序地从源中读出参数len指定的字符个数，并将读出的字符存放到参数buf指定的数组中，参数off指定数组buf存放读出字符的起始位置，该方法返回实际读出的字符个数。

## 9.7 字符串流

- **StringWriter**将内存作为流的目的地。
- 构造方法: `StringWriter();` `StringWriter(int size);`
- 字符串输出流调用下列方法可以向**缓冲区**写入字符
  - `public void write(int b)`
  - `public void write(char[] buf, int off, int len)`
  - `public void write(String str)`
  - `public void write(String str, int off, int len)`
- 字符串输出流调用`public String toString()`方法, 可以返回输出流写入到缓冲区的全部字符; 调用`public void flush()`方法可以刷新缓冲区。



# 小结


- 9.1 文件 File
- 9.12 使用Scanner解析文件 Scanner
- 9.3 文件字符流 FileReader, FileWriter
- 9.5 缓冲流 BufferedReader, BufferedWriter
- 9.2 文件字节流 FileInputStream, FileOutputStream
- 9.8 数据流 DataInputStream, DataOutputStream
- 9.9 对象流 ObjectInputStream, ObjectOutputStream
- 9.10 序列化和对象克隆 Serializable
- 9.11 随机读写流 RandomAccessFile
- 9.13 文件锁 getChannel(), tryLock(), lock()
- 9.6 数组流 ByteArrayInputStream, ByteArrayOutputStream  
CharArrayReader, CharArrayWriter
- 9.7 字符串流 StringReader, StringWriter

注：为了便于讲解，顺序做了适当调整

Google image search: java.io, figure



# 补充

- Data stored in **a text file** are represented in human-readable form.
  - Data stored in **a binary file** are represented in binary form.
  - The advantage of binary files is that they are **more efficient** to process than text files (because binary I/O does not require encoding and decoding).
  - Java offers many classes for performing file input and output
    - Text I/O classes
    - Binary I/O classes
  - In general, we should use **text input** to **read a file created by a text editor or a text output program**, and use **binary input** to **read a file created by a Java binary output program**.
- 

# 补充

- **File:** obtain file properties and manipulate files, NOT create a file, NOT read/write from/to a file

# 补充

- Both text and binary I/O classes
    - FileReader, BufferedReader: **read** ...
    - FileWriter, BufferedWriter: **write** ...
- 

- Reader
  - InputStreamReader
    - **FileReader**
  - **BufferedReader**
  - CharArrayReader
  - StringReader
- Writer
  - OutputStreamWriter
    - **FileWriter**
  - **BufferedWriter**
  - CharArrayWriter
  - StringWriter
  - PrintWriter

java.io

# 补充

- **Text I/O classes**
  - Scanner (java.util.scanner): **read** string and primitive data values from a **text** file
  - PrintWriter: create a file and **write** data to a **text** file

# 补充

- **Binary I/O classes**

- **Input**Stream

- **File**InputStream

- **Filter**InputStream

- **Data**InputStream

- **Buffered**InputStream

- **Object**InputStream

- **Output**Stream

- **File**OutputStream

- **Filter**OutputStream

- **Data**InputStream

- **Buffered**InputStream

- **Object**OutputStream

Read/write **bytes** from/to files.

Reads **bytes** from the stream and converts them into appropriate **primitive values or strings**.

Converts **primitive type values or strings** into **bytes** and outputs the bytes to the stream.

Adds a **buffer** in the stream for storing bytes for **efficient** processing. We should **always use buffered I/O** to speed up input and output.

Performs I/O for **objects, primitive values and strings**. 可完全替换DataInputStream. 但是效率会低.

# 补充

- **Wrap** DataInputStream on FileInputStream

```
DataInputStream input = new DataInputStream(new  
FileInputStream("input.dat"));
```

- **Wrap** BufferedInputStream on FileInputStream

```
BufferedInputStream input = new  
BufferedInputStream(new FileInputStream("input.dat"));
```

- **Wrap** ObjectInputStream on FileInputStream

```
ObjectInputStream input = new ObjectInputStream(new  
FileInputStream("input.dat"));
```

- **Wrap** ObjectInputStream on BufferedInputStream

```
ObjectInputStream input = new ObjectInputStream( new  
BufferedInputStream(new FileInputStream("input.dat")) );
```



# 补充

- **Serializable**
  - Not every object can be written to an output stream. Objects that can be so written are said to be **serializable**. A serializable object is an instance of **java.io.Serializable** interface.
- **RandomAccessFile**
  - All of the above streams are known as **read-only** or **write-only** streams.
  - To allow a file to be read from and written to **at random locations**.
  - 很多方法与DataInputStream和DataOutputStream是一样的，因为实现了DataInput和DataOutput接口