1、课程名称：Java IO

2、知识点

2.1、本节预计讲解的知识点

1、 讲解 Java IO 包中的各个字节操作类
2、 **File** 类的使用及注意
3、 字节操作流：**OutputStream**、**InputStream**
4、 字符操作流：**Reader**、**Writer**
5、 对象序列化：**Serializable**

3、具体内容

无论是哪种编程语言，输入跟输出都是重要的一部分，Java 也不例外，而且 Java 将输入 / 输出的功能和使用范畴做了很大的扩充。它采用了流的 机制来实现输入 / 输出，所谓流，就是数据的有序排列，而流可以是从某个源（称为流源或 Source of Stream）出来，到某个目的地（称为流汇或 Sink of Stream）去的。由流的方向，可以分成输入流和输出流，一个程序从输入流读取数据向输出流写数据。
所有的 io 操作都在 java.io 包中定义。

3.1、File 类（重点）

3.1.1、File 类的基本概念

java.io.File 类的定义如下：

```
public class File
extends Object
implements Serializable, Comparable<File>
```

此类直接是 Object 的子类，而且实现了 Comparable 接口，证明，此类的对象数组可以排序。

在整个 IO 包中，最重要的类就是一个 File 类，而且也只有 File 类是唯一与文件本身有关的操作类，与文件本身有关指的是，文件的创建、删除、重命名、得到路径、创建时间等等。在 File 类中提供了以下的操作方法和常量：

No.	方法或常量名称	类型	描述
1	public static final String separator	常量	表示路径分隔符“\”
2	public static final String pathSeparator	常量	表示路径分隔，表示“;”
3	public File(String pathname)	构造	构造 File 类实例，要传入路径
4	public boolean createNewFile() throws IOException	普通	创建新文件

5	public boolean delete()	普通	删除文件
6	public String getParent()	普通	得到文件的上一级路径
7	public boolean isDirectory()	普通	判断给定的路径是否是文件夹
8	public boolean isFile()	普通	判断给定的路径是否是文件
9	public String[] list()	普通	列出文件夹中的文件
10	public File[] listFiles()	普通	列出文件夹中的所有文件
11	public boolean mkdir()	普通	创建新的文件夹
12	public boolean renameTo(File dest)	普通	为文件重命名
13	public long length()	普通	返回文件大小

以上的两个常量并不符合命名规范，因为很早就出现了。

3.1.2、File 类的相关操作

下面通过一些操作代码来了解以上的各个方法的使用。

3.1.2.1、操作实例一：创建文件

如果要想创建文件，则使用 createNewFile()方法完成。

```
package org.iodemo.filedemo;
import java.io.File;
import java.io.IOException;
public class FileDemo01 {
    public static void main(String[] args) {
        File file = new File("d:\\test.txt");
        try {
            file.createNewFile(); // 创建文件
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

以上确实已经将一个文件创建成功了，但是这样的操作中，路径是存在问题的，因为在不同的操作系统中路径的分隔符是不一样的，例如：

- windows 下。分隔符是“\”
- linux 下。分隔符是“/”

Java 本身是具备高的可移植性的，那么这样一来就要求程序可以适应各个操作系统平台的要求，则以上的写法就不合适了，因为固定了操作平台。

那么此时，在编写 IO 操作的时候一定要注意，所有的分隔符需要使用 separator 进行指定。

```
package org.iodemo.filedemo;
import java.io.File;
import java.io.IOException;
public class FileDemo02 {
    public static void main(String[] args) {
        File file = new File("d:" + File.separator + "test.txt");
```

```
        try {
            file.createNewFile(); // 创建文件
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3.1.2.2、操作实例二：删除文件

使用 delete()方法进行文件的删除操作。

```
package org.iodemo.filedemo;
import java.io.File;
public class FileDemo03 {
    public static void main(String[] args) {
        File file = new File("d:" + File.separator + "test.txt");
        file.delete();// 删除文件
    }
}
```

以上确实可以删除文件，但是此时文件的执行速度较慢，会造成延迟，这一点在开发的时候一定要特别注意，往往会因为文件有延迟而可能造成开发中出现的问题。

但是，以上的操作也存在些问题。以上是不管文件是否存在，就直接删除了，至少在删除之前要判断一下文件是否存在之后再删除，所以使用 exists()方法判断文件是否存在。

```
package org.iodemo.filedemo;
import java.io.File;
public class FileDemo04 {
    public static void main(String[] args) {
        File file = new File("d:" + File.separator + "test.txt");
        if (file.exists()) { // 文件是否存在
            file.delete();// 删除文件
        }
    }
}
```

那么能不能继续扩充一下呢？判断文件是否存在，如果存在则删除，不存在则创建。

```
package org.iodemo.filedemo;
import java.io.File;
import java.io.IOException;
public class FileDemo05 {
    public static void main(String[] args) {
        File file = new File("d:" + File.separator + "test.txt");
        if (file.exists()) { // 文件是否存在
            file.delete();// 删除文件
        } else {
            try {
                file.createNewFile();
```

```
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

3.1.2.3、操作实例三：判断类型

在操作开发中经常会牵扯到设置一个路径之后进行操作，但是文件和文件夹的操作应该有所区别吧，所以这个时候在进行操作之前最好先判断给定的路径是文件还是文件夹。

```
package org.iodemo.filedemo;
import java.io.File;
public class FileDemo06 {
    public static void main(String[] args) {
        File file1 = new File("d:" + File.separator + "test.txt"); // 文件路径
        File file2 = new File("d:"); // 文件夹路径
        System.out.println(file1.isFile());
        System.out.println(file2.isDirectory());
        System.out.println("文件大小：" + file1.length());
        System.out.println("文件路径：" + file1.getPath());
        System.out.println("文件路径：" + file1);
    }
}
```

3.1.2.4、操作实例四：列出目录的内容

如果现在给定了一个文件夹，要求可以将这个文件夹中的内容全部列出，此时，可以使用以下的两种方法：

- 第一种方法：public String[] list()
- 第二种方法：public File[] listFiles()

范例一：使用 list()方法

```
package org.iodemo.filedemo;
import java.io.File;
public class FileDemo07 {
    public static void main(String[] args) {
        File file = new File("d:" + File.separator); // 文件夹路径
        String str[] = file.list(); // 列出目录内容
        for (int x = 0; x < str.length; x++) {
            System.out.println(str[x]);
        }
    }
}
```

此方法列出的只是各个目录或文件的名称。

范例二：使用 listFiles()方法

```
package org.vince.iodemo.filedemo;
import java.io.File;
public class FileDemo08 {
    public static void main(String[] args) {
        File file = new File("d:" + File.separator); // 文件夹路径
        File files[] = file.listFiles(); // 列出
        for (int x = 0; x < files.length; x++) {
            System.out.println(files[x]);
        }
    }
}
```

第二种方法列出的是全部的文件的完整路径，这样肯定更加适合操作，所以使用第二种方法比较合理。

3.1.3、思考

要求完成以下的功能：任意给定一个目录，要求可以将此目录中的全部文件和子文件夹中的所有文件列出。

因为每一个目录中有可能有其他的子目录或子文件，那么此时就需要使用递归的方式循环列出。

```
package org.iodemo.filedemo;
import java.io.File;
public class ListDemo {
    public static void main(String[] args) {
        File file = new File("D:" + File.separator);
        fun(file);
    }
    public static void fun(File file) {
        if (file.isDirectory()) { // 如果是目录，继续列出
            File f[] = file.listFiles(); // 列出全部目录内容
            if (f != null) { // 判断是否列出文件
                for (int x = 0; x < f.length; x++) {
                    fun(f[x]);
                }
            }
        } else {
            System.out.println(file);
        }
    }
}
-----------------------------------------------------
package iodemo.filedemo;
import java.io.File;
public class FileDemo {
    public static void main(String[] args) {
        File file=new File("d:"+File.separator);
        fun(file);
    }
```

```
    public static void fun(File file){
        if(file.isFile()){
                    System.out.println(file);
        }else{
            File files[]=file.listFiles();
            if(files!=null){//为空表示该文件夹中没有内容或者隐藏不可见，如果不判断，可能导致空指向异常
                for(int i=0;i<files.length;i++){
                        fun(files[i]);
                }
            }
        }
    }
}
```

3.2、输入和输出流（重点）

举例：

WU 同学喝水的过程属于输入流，排出过程属于输出流。

但是要想喝水之前，必须把嘴打开，可以操作资源，要想输出，XX 打开。。。。

在整个 IO 操作中，输入和输出流是一个最重要的概念，在 Java 的 IO 中，输入和输出分为两种类型，一种称为字节流，另外一种称为字符流。

- 字节流：主要是操作字节数据（byte），分为 OutputStream，字节输出流、InputStream，字节输入流
- 字符流：主要是操作字符数据（char），分为 Writer，字符输出流，Reader，字符输入流

但是，不管使用那种操作，字节流和字符流的操作都是采用如下的步骤完成：

1、 找到一个要操作的资源，可能是文件，可能是其他的位置

2、 根据字节流或字符流的子类，决定输入及输出的位置

3、 进行读或写的操作

4、 关闭

3.2.1、字节输出流：OutputStream

观察 OutputStream 类的定义：

```
public abstract class OutputStream
extends Object
implements Closeable, Flushable
```

此类是一个抽象类实现了 Closeable 和 Fluashable 接口。此类的常用方法如下所示：

No.	方法名称	类型	描述
1	public void close() throws IOException	普通	关闭
2	public void flush() throws IOException	普通	刷新操作
3	public void write(byte[] b) throws IOException	普通	将一组字节写入到输出流之中
4	public void write(byte[] b,int off,int len) throws IOException	普通	将指定范围的字节数组进行输出

5	public abstract void write(int b) throws IOException	普通	每次写入一个字节，byte → int

但是，以上的类只是一个抽象类，抽象类必须通过子类完成，现在要向文件中输出，使用 FileOutputStream 类。

此类的构造方法如下：

No.	方法名称	类型	描述
1	public FileOutputStream(File file) throws FileNotFoundException	构造	接收 File 类的实例，表示要操作的文件位置。
2	public FileOutputStream(File file,boolean append) throws FileNotFoundException	构造	接收 File 类实例，并指定是否可以追加

操作的时候肯定以父类为操作的标准，所以现在只关心 OutputStream 类。

范例：向文件中输出“Hello World!!!”

```
package org.iodemo.fileoutputstreamdemo;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
public class OutputStreamDemo01 {
    public static void main(String[] args) {
        File file = new File("D:" + File.separator + "test.txt"); // 指定要操作的文件
        OutputStream out = null; // 定义字节输出流对象
        try {
            out = new FileOutputStream(file); // 实例化操作的父类对象
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        String info = "Hello World";// 要打印的信息
        byte b[] = info.getBytes(); // 将字符串变为字节数组
        try {
            out.write(b);// 输出内容
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            out.close(); // 关闭
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

此时，程序已经将字符串的内容进行输出了，但是如果再次执行呢。

发现，每次执行的时候所有的内容都被覆盖了，那么如果现在希望完成的功能是追加的话，该怎么做呢？

```
package org.iodemo.fileoutputstreamdemo;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
```

```
import java.io.IOException;
import java.io.OutputStream;
public class OutputStreamDemo02 {
    public static void main(String[] args) {
        File file = new File("D:" + File.separator + "test.txt"); // 指定要操作的文件
        OutputStream out = null; // 定义字节输出流对象
        try {
            out = new FileOutputStream(file, true); // 实例化操作的父类对象，可以追加内容
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        String info = "Hello World!!!";// 要打印的信息
        byte b[] = info.getBytes(); // 将字符串变为字节数组
        try {
            out.write(b);// 输出内容
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            out.close(); // 关闭
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上面确实追加了，但是发现没次追加的时候都是在最后追加，如果现在希望可以换行呢？如果要想追加换行，使用“\r\n”完成。

```
String info = "Hello World!!!\r\n";// 要打印的信息
```

进一步深入：

在使用字节输出流输出的时候有一点也是必须注意的，因为 OutputStream 中存在以下的一个方法，可以一次写入一个，所以，之前的操作也可以采用循环的方式完成：

```
package org.iodemo.fileoutputstreamdemo;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
public class OutputStreamDemo04 {
    public static void main(String[] args) {
        File file = new File("D:" + File.separator + "test.txt"); // 指定要操作的文件
        OutputStream out = null; // 定义字节输出流对象
        try {
            out = new FileOutputStream(file, true); // 实例化操作的父类对象，可以追加内容
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
```

```
        String info = "Hello World!!!\r\n";// 要打印的信息
        byte b[] = info.getBytes(); // 将字符串变为字节数组
        try {
            for (int x = 0; x < b.length; x++) {
                out.write(b[x]);// 输出内容
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            out.close(); // 关闭
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3.2.2、字节输入流：InputStream

字节输入流使用 InputStream 类完成，此类的定义如下：

```
public abstract class InputStream
extends Object
implements Closeable
```

本类也是一个抽象类，本类中的常用操作方法如下：

No.	方法名称	类型	描述
1	public void close() throws IOException	普通	关闭
2	public abstract int read() throws IOException	普通	读取每一个字节
3	public int read(byte[] b) throws IOException	普通	向字节数组中读取，同时返回读取的个数
4	public int read(byte[] b,int off,int len) throws IOException	普通	指定读取的范围

InputStream 类本身属于抽象类，肯定需要子类支持，子类从文件中读取肯定是 FileInputStream。

No.	方法名称	类型	描述
1	public FileInputStream(File file) throws FileNotFoundException	构造	通过 File 类实例，创建文件输入流

范例：从文件之中读取出内容

```
package org.iodemo.fileinputstreamdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
public class InputStreamDemo01 {
    public static void main(String[] args) {
        File file = new File("D:" + File.separator + "test.txt");// 要读取的文件路径
```

```
        InputStream input = null; // 字节输入流
        try {
            input = new FileInputStream(file);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        byte[] b = new byte[1024];// 开辟byte数组空间，读取内容
        int len = 0;
        try {
            len = input.read(b); // 读取
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            input.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println(new String(b, 0, len));
    }
}
```

此时，文件已经读取成功了。

进一步深入：

也可以通过 read()方法采用循环读的方式。

```
package org.iodemo.fileinputstreamdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
public class InputStreamDemo02 {
    public static void main(String[] args) {
        File file = new File("D:" + File.separator + "test.txt");// 要读取的文件路径
        InputStream input = null; // 字节输入流
        try {
            input = new FileInputStream(file);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        byte b[] = new byte[1024];// 开辟byte数组空间，读取内容
        int len = 0;
        try {
            int temp = 0; // 接收每次读取的内容
            while ((temp = input.read()) != -1) {// 如果不为-1表示没有读到底
                b[len] = (byte) temp; // int --> byte
                len++;
```

```
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            input.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println(new String(b, 0, len));
    }
}
```

但是，现在有些人说了，老师，你现在开辟的空间太大了。没必要开辟这么大，能不能根据文件大小来决定开辟多少空间呢？

可以，在 File 类中定义了 length()方法，但是此方法的返回值类型是 long。

```
package org.iodemo.fileinputstreamdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
public class InputStreamDemo03 {
    public static void main(String[] args) {
        File file = new File("D:" + File.separator + "test.txt");// 要读取的文件路径
        InputStream input = null; // 字节输入流
        try {
            input = new FileInputStream(file);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        byte b[] = new byte[(int) file.length()];// 根据文件大小，开辟byte数组空间
        try {
            input.read(b); // 读取
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            input.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println(new String(b));
    }
}
```

以上的操作只能在文件的输入中使，如果现在输入流来自其他地方，则无法这样使用。

3.2.3、字符输出流：Writer

字符流的操作形式实际上与字节的操作形式非常的类似，此类的定义如下：

```
public abstract class Writer
extends Object
implements Appendable, Closeable, Flushable
```

此类也是个抽象类，此类的常用方法如下：

No.	方法名称	类型	描述
1	public void write(String str) throws IOException	普通	直接将字符串写入输出
2	public void write(char[] cbuf) throws IOException	普通	输出字符数组
3	public abstract void close() throws IOException	普通	关闭
4	public abstract void flush() throws IOException	普通	刷新

与 OutputStream 一样，使用 FileWriter 类完成操作，此类的构造方法如下：

No.	方法名称	类型	描述
1	public FileWriter(File file) throws IOException	构造	根据 File 类构造 FileWriter 实例
2	public FileWriter(File file,boolean append) throws IOException	构造	根据 File 类构造 FileWriter 实例，可以追加内容

范例：使用 Writer 完成文件内容的输出

```
package org.iodemo.filewriterdemo;
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;
public class WriterDemo01 {
    public static void main(String[] args) throws Exception { // 所有异常抛出
        File file = new File("D:" + File.separator + "test.txt"); // 指定要操作的文件
        Writer out = null; // 定义字节输出流对象
        out = new FileWriter(file); // 实例化操作的父类对象
        String info = "Hello World!!!";// 要打印的信息
        out.write(info);// 输出内容
        out.close(); // 关闭
    }
}
```

使用以上的操作可以直接将字符串的内容进行输出。

3.2.4、字符输入流：Reader

使用 Reader 完成字符的输入功能，此类的定义如下：

```
public abstract class Reader
extends Object
implements Readable, Closeable
```

此类还是抽象类，此类的方法如下：

No.	方法名称	类型	描述
1	public int read() throws IOException	普通	读取一个内容
2	public int read(char[] cbuf) throws IOException	普通	读取一组内容，返回读入的大小

3	public abstract void close() throws IOException	普通	关闭

此类，依然需要使用 FileReader 类进行实例化操作，FileReader 类中的构造方法定义如下：

No.	方法名称	类型	描述
1	public FileReader(File file) throws FileNotFoundException	构造	接收 File 类的实例

范例：使用 Reader 进行文件的读取操作

```
package org.iodemo.filereaderdemo;
import java.io.File;
import java.io.FileReader;
import java.io.Reader;
public class ReaderDemo01 {
    public static void main(String[] args) throws Exception {
        File file = new File("D:" + File.separator + "test.txt");// 要读取的文件路径
        Reader input = null; // 字节输入流
        input = new FileReader(file);
        char b[] = new char[1024];// 开辟char数组空间，读取内容
        int len = input.read(b); // 读取
        input.close();
        System.out.println(new String(b, 0, len));
    }
}
```

可以发现，不管是字节流也好，还是字符流，其基本的操作形式都是一样的，即：一切以父类为操作的标准。

3.2.5、字节流和字符流的区别

之前的代码中可以发现，两者的操作形式一样，功能也一样，那么这两者到底有那些区别呢？

区别：字节流没有使用到缓冲区，而是直接操作输出的，而字符流使用到了缓冲区，是通过缓冲区操作输出的。

```
package org.iodemo.fileoutputstreamdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
public class OutputStreamDemo05 {
    public static void main(String[] args) throws Exception {
        File file = new File("D:" + File.separator + "test.txt"); // 指定要操作的文件
        OutputStream out = null; // 定义字节输出流对象
        out = new FileOutputStream(file); // 实例化操作的父类对象
        String info = "Hello World!!!";// 要打印的信息
        byte b[] = info.getBytes(); // 将字符串变为字节数组
        out.write(b);// 输出内容
        // 本程序没有进行关闭
    }
}
```

此时，虽然没有关闭输出流，但是内容依然可以输出，证明，字节操作流是直接与输出本身有关的，那么如果现在同样的操作使用字符流呢？

```
package org.iodemo.filewriterdemo;
```

```
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;
public class WriterDemo02 {
    public static void main(String[] args) throws Exception { // 所有异常抛出
        File file = new File("D:" + File.separator + "test.txt"); // 指定要操作的文件
        Writer out = null; // 定义字节输出流对象
        out = new FileWriter(file); // 实例化操作的父类对象
        String info = "Hello World!!!";// 要打印的信息
        out.write(info);// 输出内容
        // 此处没有关闭
    }
}
```

此时已经完成了程序，但是程序执行之后发现并没有任何的内容，因为所有的内容都放在缓冲区之中，只有在关闭的时候才会清空缓冲区，进行输出，如果此时希望可以把内容输出的话，则必须手工调用 flush()方法完成。

```
package org.iodemo.filewriterdemo;
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;
public class WriterDemo03 {
    public static void main(String[] args) throws Exception { // 所有异常抛出
        File file = new File("D:" + File.separator + "test.txt"); // 指定要操作的文件
        Writer out = null; // 定义字节输出流对象
        out = new FileWriter(file); // 实例化操作的父类对象
        String info = "Hello World!!!";// 要打印的信息
        out.write(info);// 输出内容
        out.flush();// 强制清空缓冲区
        // 此处没有关闭
    }
}
```

从以上的两个操作比较，是使用字节流还是使用字符流好呢？

- 明显使用字节流操作会方便一些，例如：图片、电影都是字节保存的。
- 所以后面讲解的重点也放在字节流的操作之中。

3.2.6、习题

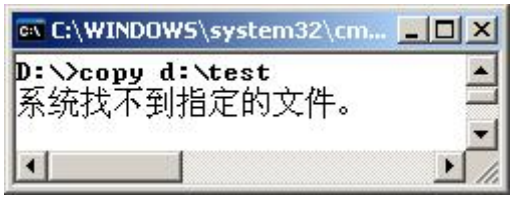
使用 Java 模仿 DOS 系统中的 COPY 命令，完成两个文件的拷贝操作。

- 执行效果：java Copy 源文件 目标文件

在这样的一个程序中需要考虑那几点问题：

1、 输入的参数个数必须是两个
2、 要拷贝的源文件必须存在
3、 此程序只能使用字节流
4、 在进行拷贝的时候要采用边读边写的方式完成

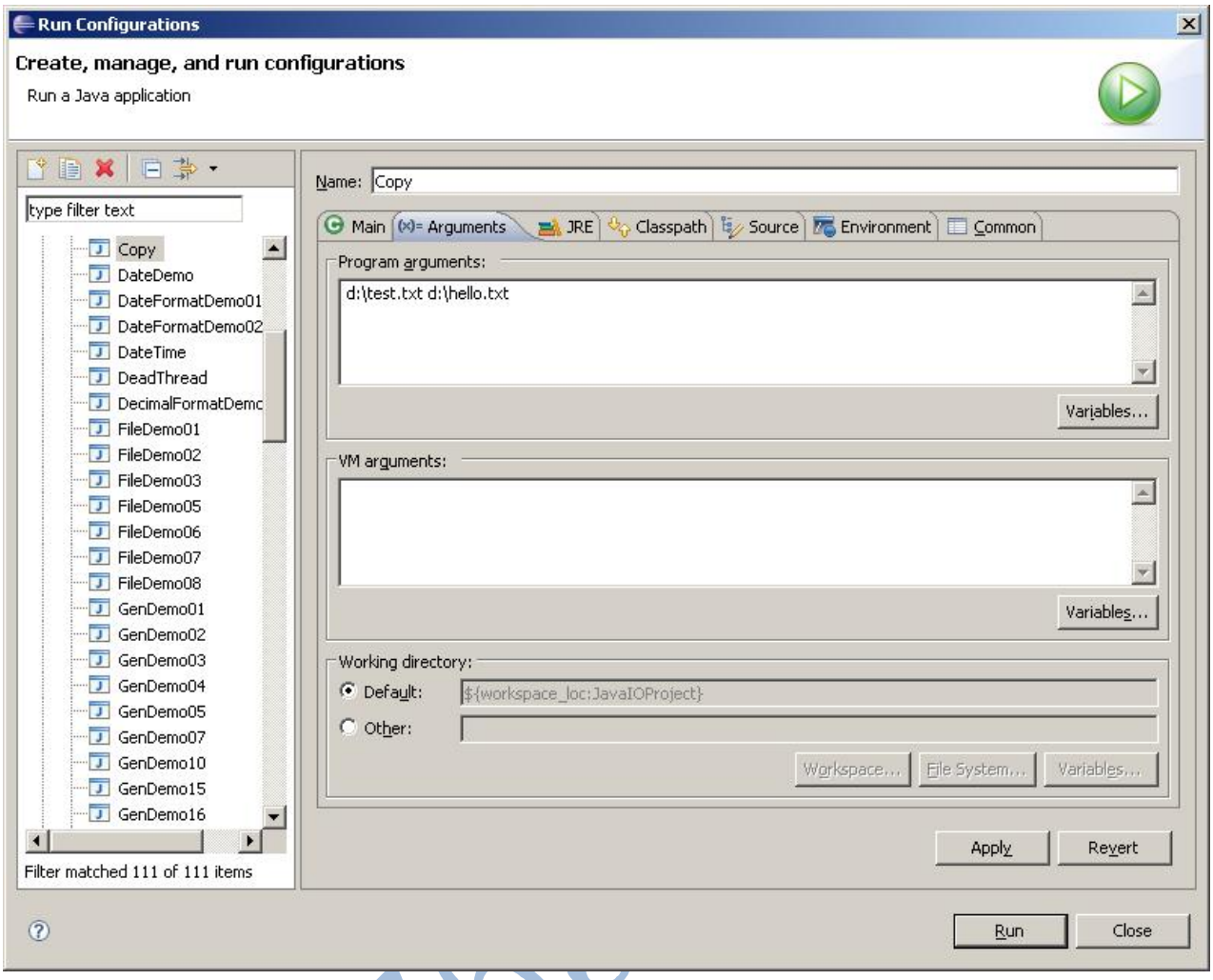
就好象 Copy 命令的执行一样。



范例：代码实现

```
package org.iodemo.copydemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
public class Copy {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("命令语法出错，应该给出文件的路径。");
            System.out.println("例如：java Copy 源文件路径 目标文件路径");
            System.exit(1);// 系统退出
        }
        File srcFile = new File(args[0]); // 源文件的File对象
        if (srcFile.exists()) { // 如果源文件存在
            File objFile = new File(args[1]);// 拷贝的目标文件
            try {
                InputStream input = new FileInputStream(srcFile);
                OutputStream output = new FileOutputStream(objFile);
                int temp = 0;
                while ((temp = input.read()) != -1) {
                    output.write(temp); // 写入
                }
                input.close();
                output.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("源文件不存在！");
        }
    }
}
```

如果是在 Eclipse 中运行，则必须配置运行时的参数。



3.3、字节-字符转换流（理解）

在整个 IO 包中，除了字节和字符流之外，还包含了一种称为转换流，可以将一个字节流变为字符流，也可以将一个字符流变为字节流，主要使用以下的两个类完成：

- OutputStreamWriter：可以将输出的字符流变为字节流的输出形式
- InputStreamReader：将输入的字节流变为字符流输入形式

OutputStreamWriter 是 Writer 类的子类，此类的构造方法如下：

No.	方法名称	类型	描述
1	public OutputStreamWriter(OutputStream out)	构造	将字符流变字节流

现在，使用 OutputStreamWriter 完成一个字符-字节的转换操作

```
package org.iodemo.outputinputdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;
public class OutputStreamWriterDemo {
    public static void main(String[] args) throws Exception {
        File file = new File("D:" + File.separator + "temp.txt");
```

```
        // 通过转换类，将字符输出流变为字节输出流
        Writer out = new OutputStreamWriter(new FileOutputStream(file));
        out.write("Hello World!!!");
        out.close();
    }
}
```

InputStreamReader 是 Reader 的子类，可以将字节的输入流变为字符输入流。

```
package org.iodemo.outputinputdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.io.Reader;
public class InputStreamReaderDemo {
    public static void main(String[] args) throws Exception {
        File file = new File("D:" + File.separator + "temp.txt");
        Reader input = new InputStreamReader(new FileInputStream(file));
        char c[] = new char[100];
        int len = input.read(c);// 读取数据
        System.out.println(new String(c, 0, len));
        input.close();
    }
}
```

3.4、打印流：PrintStream（重点）

打印流的主要功能是用于输出，在整个 IO 包中打印流分为两种类型：

- 字节打印流：PrintStream
- 字符打印流：PrintWriter

在本章，使用字节打印流完成操作。

到底什么叫打印流呢？

OutputStream 本身已经提供了很好的输出功能，但是使用 OutputStream 输出其他数据，例如：boolean、char、float 等就比较麻烦了，所以在 Java 中为了方便输出，提供了专门的打印流。

观察 PrintStream 类的定义格式：

```
public class PrintStream
extends FilterOutputStream
implements Appendable, Closeable
```

PrintStream 本身也属于 OutputStream 的子类。但是继续观察 PrintStream 类中的一些方法。

No.	方法名称	类型	描述
1	public PrintStream(OutputStream out)	构造	接收 OutputStream 类的实例。
2	public PrintStream(File file) throws FileNotFoundException	构造	接收 File 类实例，向文件中输出
3	public PrintStream format(String format, Object... args)	普通	表示格式化输出
4	public void print(数据类型 f)	普通	打印输出，不换行
5	public PrintStream printf(String format,Object... args)	普通	格式化输出
6	public void println(数据类型 f)	普通	打印输出，换行

从以上类的定义中可以发现，此类可以完成输出的功能，但是这个类的输出功能比直接使用 OutputStream 输出强，因为可以输出任意的数据类型。也就是说此类实际上是将 OutputStream 加强了。

在此类的构造方法中，需要接收 OutputStream 类的实例，那么实际上此时就意味着，哪个子类为 PrintStream 实例化，PrintStream 就具备了向指定位置的输出能力。

那么这样的设计在设计模式上讲称为**装饰设计模式。**

范例：使用 PrintStream 完成输出

```
package org.iodemo.printstreamdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.io.PrintStream;
public class PrintStreamDemo01 {
    public static void main(String[] args) throws Exception {
        File file = new File("D:" + File.separator + "temp.txt");
        OutputStream output = new FileOutputStream(file);
        PrintStream out = new PrintStream(output);
        out.print("hello");
        out.print(" world ");
        out.println("!!!");
        out.print("1 X 1 = " + (1 * 1));
        out.println("\n输出完毕");
        out.close();
    }
}
```

所以，以后在开发中输出数据的时候不要再直接使用 OutputStream，而都使用 PrintStream 类。因为输出方便。

PrintStream 在 JDK 1.5 之后实际上又有所增加，增加了格式化的输出操作，提供了 printf()方法。

如果要想进行格式化的操作，则必须指定格式化的操作模板，模板的指定如下：

No.	模板标记	描述
1	%s	表示字符串
2	%d	表示整数
3	%n.mf	表示小数，一共的数字长度是 n，其中小数是 m 位
4	%c	表示字符

进行格式化输出：

```
package org.iodemo.printstreamdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.PrintStream;
public class PrintStreamDemo02 {
    public static void main(String[] args) throws Exception {
        String name = "张三";
        float salary = 800.897f;
        int age = 10;
        PrintStream out = new PrintStream(new FileOutputStream(new File("d:"
                + File.separator + "temp.txt")));
        out.printf("姓名：%s，年龄：%d，工资：%7.2f。", name, age, salary);
```

```
        out.close();
    }
}
```

以上的操作基本上全部的模板都可以使用“%s”代替。

```
out.printf("姓名：%s，年龄：%s，工资：%s。", name, age, salary);
```

此操作就是为了照顾大部分的之前的 C 语言的开发人员。

3.5、对象序列化（重点）

3.5.1、对象序列化的基本概念

对象序列化就是指将一个对象转化成二进制的 byte 流。以将对象保存在文件上为例。

- 将对象保存在文件上的操作称为对象的序列化操作
- 将对象从文件之中恢复称为反序列化的操作

如果要想实现对象的序列化，则需要使用 ObjectOutputStream 类完成，要想实现反序列，则需要使用 ObjectInputStream 类完成。

3.5.2、Serializable 接口

而且在使用对象序列化的时候一定要注意一点，要被序列化对象所在的类必须实现 java.io.Serializable 接口。

观察此接口的定义：

```
public interface Serializable{
}
```

此接口中没有任何的方法定义。与 Cloneable 接口一样，此接口也属于标识接口，表示可以被序列化。

虽然此接口中没有定义任何的方法，但是在操作的时候也不要让所有的类都实现此接口。只能在需要的地方进行实现，因为现在接口中没有方法，并不表示以后的接口中同样没有方法。

范例：定义 Person 类，此类可以被序列化

```
package org.iodemo.serdemo;
import java.io.Serializable;
public class Person implements Serializable {
    /**
     * 此常量表示的是一个序列化的版本编号，为的是可以在不同的JDK 版本间进行移植
     */
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;
    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
```

```
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String toString() {
        return "姓名: " + this.name + "; 年龄: " + this.age;
    }
}
```

3.5.3、进行序列化操作：ObjectOutputStream

ObjectOutputStream 专门用于对象的输出操作，此类的常用方法如下：

No.	方法名称	类型	描述
1	public ObjectOutputStream(OutputStream out) throws IOException	构造	接收 OutputStream 实例，进行实例化操作
2	public final void writeObject(**Object obj**) throws IOException	普通	输出一个对象
3	public void close() throws IOException	普通	关闭

发现 ObjectOutputStream 类的使用与 PrintStream 非常类似，根据实例化其子类的不同，输出的位置也不同。

在使用 writeObject()方法的时候，发现使用 Object 进行参数的接收，那么证明所有的对象都可以使用此方法输出，都会自动发生向上转型的关系。

范例：将 Person 的对象保存在文件之中

```
package org.iodemo.serdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
public class ObjectOutputStreamDemo {
    public static void main(String[] args) throws Exception { // 所有异常抛出
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(
                new File("D:" + File.separator + "person.ser")));
        Person per = new Person("张三", 30);
        oos.writeObject(per);// 输出
        oos.close();// 关闭
    }
}
```

此时，对象已经被成功的序列化到了文件之中。

但是必须提醒的是，对象不一定只能序列化到文件之中，任何地方都有可能序列化，根据实例化 ObjectOutputStream 类的对象不同，输出的位置也不同。

3.5.4、进行反序列化操作：ObjectInputStream

使用 ObjectInputStream 可以完成对象的反序列化操作。此类的常用方法如下：

No.	方法名称	类型	描述
1	public ObjectInputStream(InputStream in) throws IOException	构造	根据输入流的不同，实例哈 ObjectInputStream 类的对象
2	public final Object readObject() throws IOException,ClassNotFoundException	普通	读取对象

写入对象的时候所有的对象都向 Object 进行转型操作，所以读取的时候也使用 Object 进行读取的操作。

范例：进行对象的反序列化操作

```
package org.iodemo.serdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
public class ObjectInputStreamDemo {
    public static void main(String[] args) throws Exception { // 所有异常抛出
        ObjectInputStream oos = new ObjectInputStream(new FileInputStream(
                new File("D:" + File.separator + "person.ser")));
        Person p = (Person)oos.readObject() ;
        System.out.println(p);
    }
}
```

此时已经将被序列化的内容读取进来了，实际上序列化序列的只是每个类中的属性，因为各个对象只能靠属性区分。

3.5.5、序列化一组对象

在序列化操作中，每次只能序列化一个对象，如果现在要想序列化一组对象该如何操作呢？

此时，可以采用**对象数组**的形式，因为对象数组可以向 Object 进行转型操作。

```
package org.iodemo.serdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
public class SerArrayObjectDemo {
    public static void main(String[] args) throws Exception {
        Person per[] = { new Person("张三", 30), new Person("李四", 35),
                new Person("王五", 50) };
        ser(per); // 序列化一组对象
        Person p[] = (Person[]) dser(); // 反序列化
        for (int x = 0; x < p.length; x++) {
            System.out.println(p[x]) ;
        }
    }
```

```
    public static void ser(Object obj) throws Exception { // 所有异常抛出
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(
                new File("D:" + File.separator + "person.ser")));
        oos.writeObject(obj);// 输出
        oos.close();// 关闭
    }
    public static Object dser() throws Exception { // 所有异常抛出
        ObjectInputStream oos = new ObjectInputStream(new FileInputStream(
                new File("D:" + File.separator + "person.ser")));
        Object obj = oos.readObject();
        return obj;
    }
}
```

3.5.6、transient 关键字

之前的所有操作中，可以发现，只要对象一进行序列化之后，所有的内容都被保存了，但是，如果现在对象中的某个属性不希望被序列化，则此时就可以使用 transient 关键字进行声明。

```
    private transient String name;
    private transient int age;
```

则以后在进行对象序列化的时候，name 和 age 属性将无法被序列化下来。则在进行反序列化操作的时候，所有内容的返回值都是默认值。

3.6、内存操作流

之前已经学习过了文件的操作流，输入输出都是从文件中进行的，那么如果现在希望产生一些临时的文件，但这些文件又不想在硬盘中直接生成，那么此时就可以使用内存输入、输出流完成。

内存输入流使用 ByteArrayInputStream，是 InputStream 的直接子类，此类的常用方法如下：

No.	方法名称	类型	描述
1	public ByteArrayInputStream(byte[] buf)	构造	将内容输入到内存之中

现在的操作位置是，所有的内容都向内存中输出，但是对于内存而言属于数据的输入。所以使用 ByteArrayInputStream 完成内存数据的输入功能。

内存输出流使用 ByteArrayOutputStream，是 OutputStream 的子类，此类的常用方法如下：

No.	方法名称	类型	描述
1	public String toString()	普通	取出全部的内容，由内存中取出

实际上对于 ByteArrayOutputStream 来讲，基本的操作方法还是以 OutputStream 为主的，还是写的操作，只是这个时候的写是指由内存向程序中输出。

范例：完成一个字母大写变为小写的功能，使用内存流完成

```
package org.bytearraydemo;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
public class CharChangeDemo {
```

```
    public static void main(String[] args) throws Exception {
        String info = "HELLOWORLD";// 定义字符串由大写字母组成
        // 所有的内容向内存中输入
        InputStream input = new ByteArrayInputStream(info.getBytes());
        // 所有的内存中的内容需要通过ByteArrayOutputStream输出
        OutputStream out = new ByteArrayOutputStream();
        int temp = 0;
        while ((temp = input.read()) != -1) {// 读取内存中的内容
            // 将大写字母变为小写字母
            char c = Character.toLowerCase((char) temp);
            out.write(c); // 从内存中输出，所有的内容保存在ByteArrayOutputStream中
        }
        input.close();
        out.close();
        String change = out.toString(); // 取出输出的内容
        System.out.println(change);
    }
}
```

一定要清楚的明白，内存操作流是以内存为操作中断的，InputStream 表示向内存中输入，OutputStream 表示从内存中输出。

3.7、缓冲区读取（重点）

在使用 System.in 进行键盘输入的时候会造成输入的乱码，所以最好将输入的内容一次性读取进来，要想实现这样的功能就可以使用 BufferedReader 类完成。

No.	方法名称	类型	描述
1	public BufferedReader(Reader in)	构造	接收 Reader 类的实例
2	public String readLine() throws IOException	普通	读取一行数据

实际上现在就可以将 System.in 放入到缓冲区之中，但是在缓冲区中需要的构造方法的参数类型是 Reader 的子类，而 System.in 是 InputStream 的子类，则此时如果要想使用，则必须将字节输入流变为字符输入流，使用 InputStreamReader 类。

```
package org.buffereddemo;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class BufferedReaderDemo01 {
    public static void main(String[] args) {
        BufferedReader buf = new BufferedReader(
                new InputStreamReader(System.in));
        System.out.print("请输入内容：");
        String str = null;
        try {
            str = buf.readLine();// 输入数据
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
        System.out.println("输入的内容是：" + str);
    }
}
```

此时，不管输入任何的内容都不会出现问题，而且没有长度限制了，所以以上的操作是键盘输入数据的标准格式。

3.8、习题

根据显示要求，完成以下的一个菜单：

```
====== Xxx 系统 ========
    [1]、增加数据
    [2]、查看数据
    [3]、修改数据
    [4]、删除数据
    [0]、退出系统
```

在之前菜单程序的基础之上，加入以下的功能：

- 完成一个单人的信息管理程序，使用 Person 类即可
- 可以增加数据、修改数据、删除数据、查看数据
- 可以将信息保存在文件之中

如果要想完成以上的功能，则可以将程序分为两个部分：

- 第一个部分进行菜单的显示功能操作
- 第二部分加入具体的功能操作

所以在编写程序的时候一定要注意一个原则，类的设计原则：

- 一个类只完成一个具体的功能，例如，程序中的菜单就是完成菜单功能，文件操作就是完成对象的保存及读取操作，菜单中增加菜单操作类，这样以后扩充的时候可以不用修改菜单。
- 类设计原则，完成具体的独立的各个功能，之后某一个局部的修改不影响其他位置的程序执行。

3.9、字符编码问题（理解）

在程序中如果没有处理好字符的编码，则就有可能出现乱码问题。但是如果要进行编码的话，则首先应该了解一下常用的字符编码是什么：

编码：

在计算机世界里，任何的文字都是以指定的编码方式存在的，在 JAVA 程序的开发中最常见的是以下的几种编码：ISO8859-1、GBK/GB2312、unicode、UTF。

Iso8859-1：编码属于单字节编码，最多只能表示 0——255 的字符范围，主要在英文上应用

GBK/GB2312：中文的国际编码，专门用来表示汉字，是双字节编码

unicode：java 中就是使用此编码方式，也是最标准的一种编码，是使用 16 进制表示的编码。但此编码不兼容 iso8859-1 编码。

UTF：由于 unicode 不支持 iso8859-1 编码，而且容易占用更多的空间，而且对于英文母也需要使用两个字节编码，这样使用 unicode 不便于传输和储存，因此产生了 utf 编码，utf 编码兼容了 iso8859-1 编码，也可以用来表示所有语言字符，不过 utf 是不定长编码，每个字符的长度从 1——6 个字节不等，一般在中文网页中使用此编码，因为这样可以节省空间。

以后在开发中比较常见的编码就是 GBK、ISO 8859-1、UTF-8 编码。

那么如果在本机环境中不想造成乱码出现的话，则必须了解本机的编码是什么，此时就可以使用以下的代码完成：

```
package org.chardemo;
public class CharDemo01 {
    public static void main(String[] args) {
        System.out.println(System.getProperties().getProperty("file.encoding"));
    }
}
```

因为现在本机属于中文环境，所以编码使用的是 GBK，那么问，如果现在程序中使用了 ISO8859-1 编码，肯定会出现乱码。

```
package org.chardemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
public class CharDemo02 {
    public static void main(String[] args) throws Exception {
        OutputStream out = new FileOutputStream(new File("D:" + File.separator
                + "temp.txt"));
        out.write("中国，你好！".getBytes("ISO8859-1"));// 转变编码
        out.close();
    }
}
```

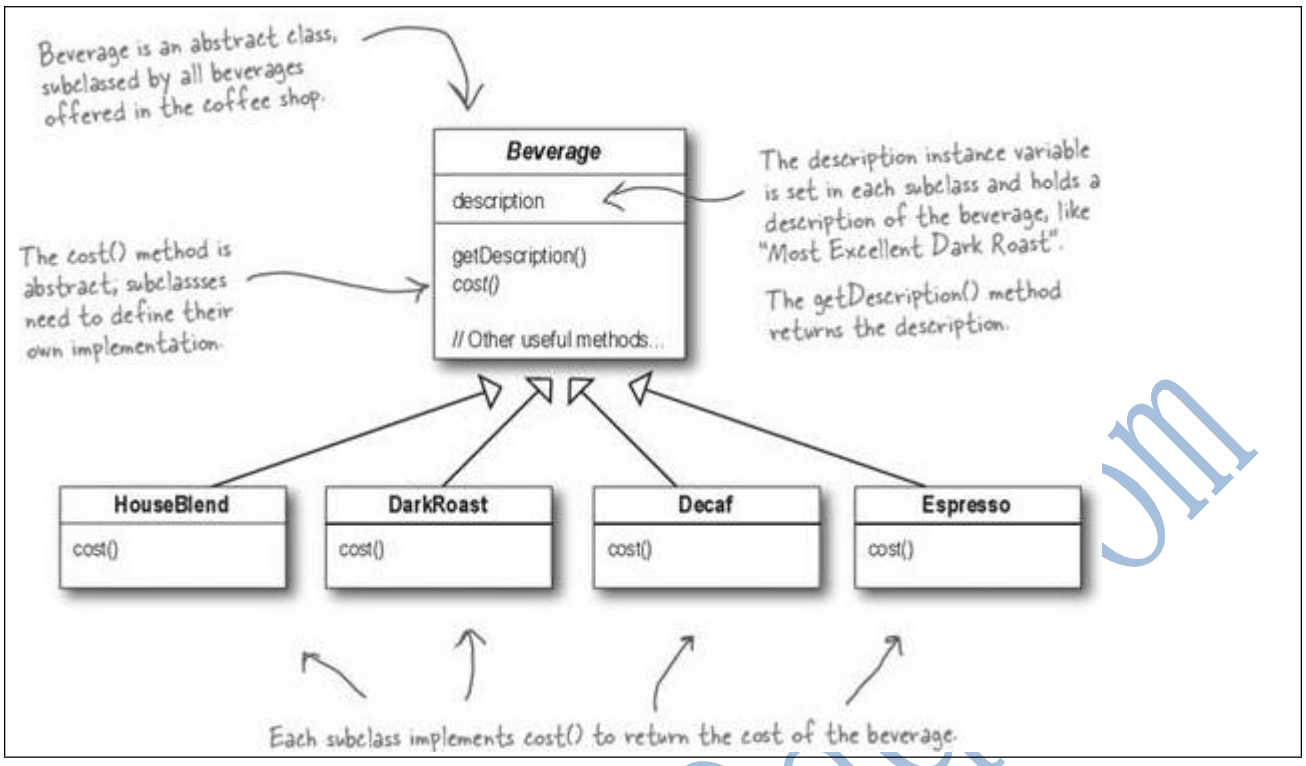
程序中的乱码造成的根本原因：程序使用的编码与本机的编码不统一造成的，在网络通讯中，发送方和接收方的编码不统一也会造成乱码。

3.10、装饰者模式 Decorator（重点）

引出问题：

Central Perk 的名字因为《老友记》而享誉全球，他们的分店几乎开遍世界各地。他们发展的实在是太快了，所以他们此时正在急于实现一套由计算机管理的自动化记账系统。

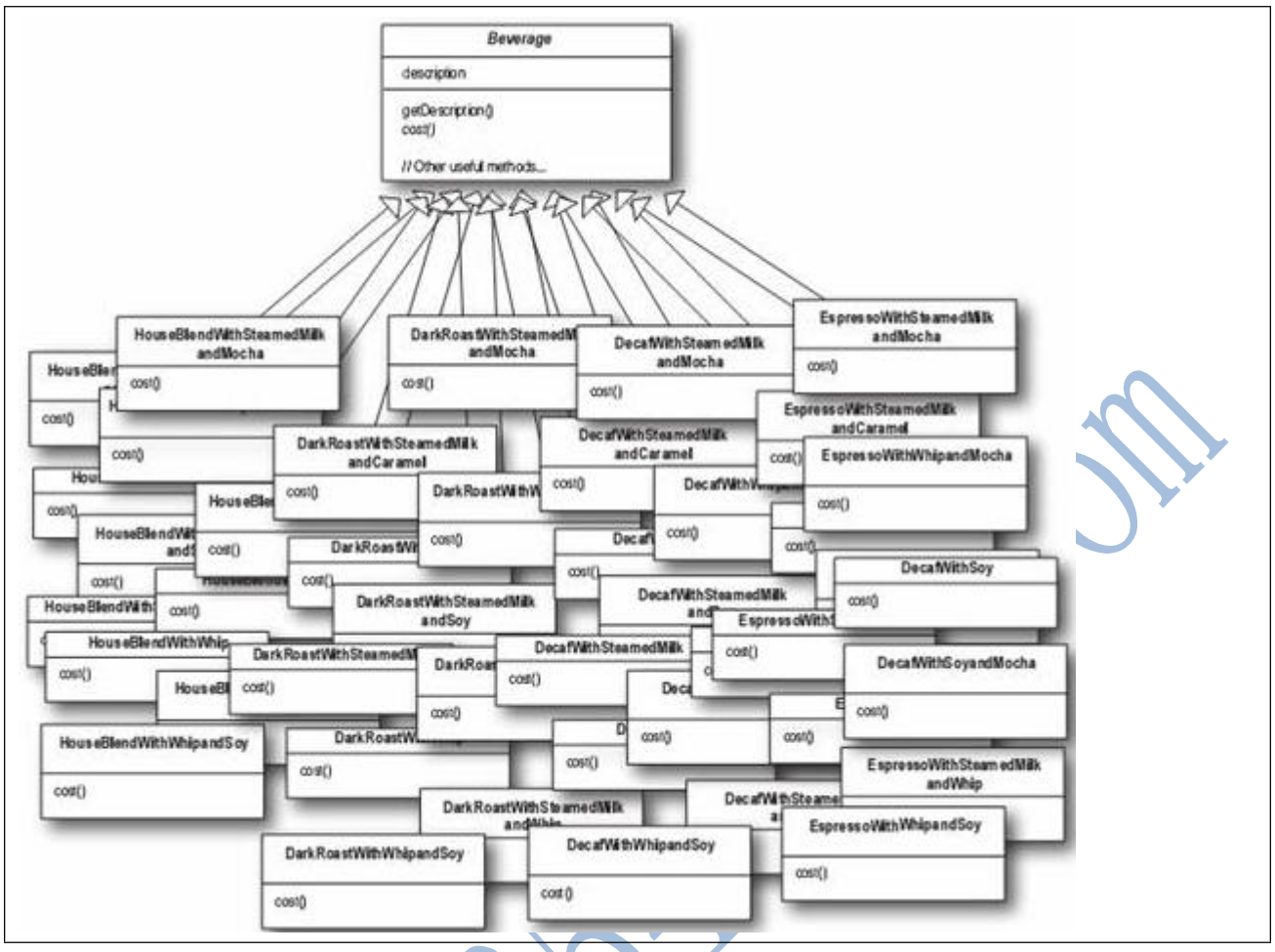
经过研究了他们的需求以后，开发者设计了如下图的类结构：



Beverage 是所有饮料的基类；cost()是抽象方法，所有子类都需要定义它们自己的 cost()实现来返回特定饮料的价钱；description 变量也是在子类里赋值的，表示特定饮料的描述信息，getDescription()方法可以返回这个描述；

除了咖啡以外，Central Perk 还提供丰富的调味品，比如：炼乳、巧克力、砂糖、牛奶等，而且这些调味品也是要单独按份收费的，所以调味品也是订单系统中重要的一部分。

于是，考虑到调味品的管理，开发者又有了下面这样的类结构：



所以下面我们将拜访一下今天的主角—装饰者模式，看看她能给我们带来什么惊喜吧！

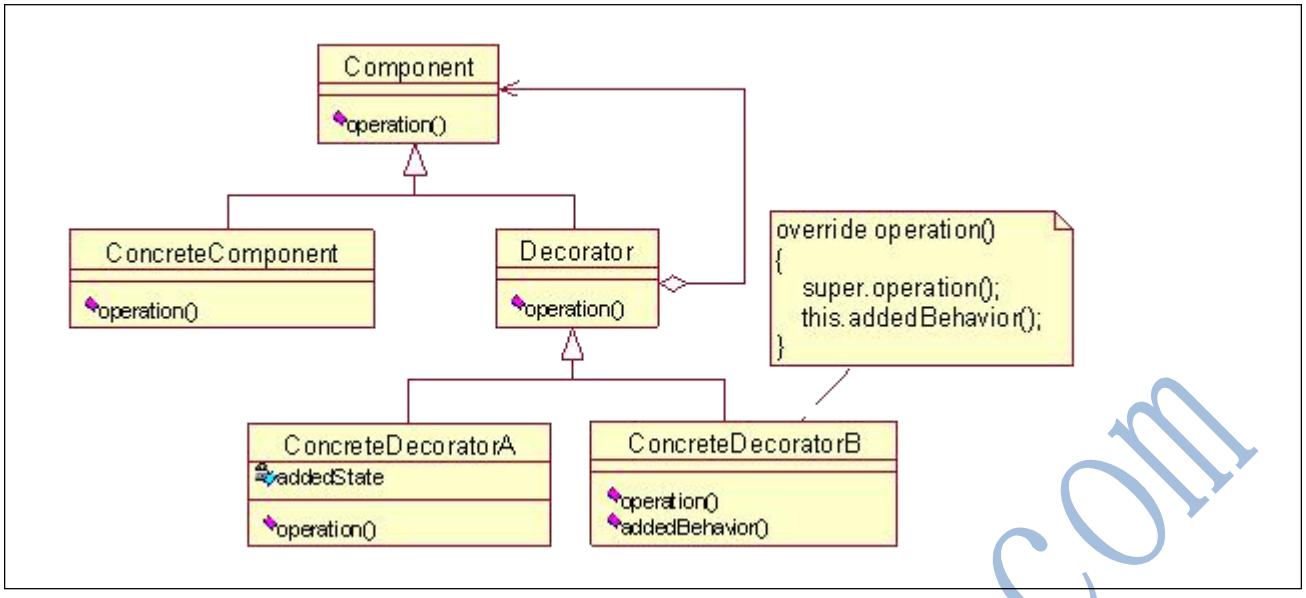
意图：

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。该模式以对客户端透明的方式扩展对象的功能。

适用环境

- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
- 处理那些可以撤消的职责。
- 当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

装饰者模式的类图



Component（被装饰对象基类）

定义对象的接口，可以给这些对象动态增加职责；

ConcreteComponent（具体被装饰对象）

定义具体的对象，Decorator 可以给它增加额外的职责；

Decorator（装饰者抽象类）

维护一个指向 Component 实例的引用，并且定义了与 Component 一致的接口；

ConcreteDecorator（具体装饰者）

具体的装饰对象，给内部持有的具体被装饰对象增加具体的职责；

涉及角色

抽象构件角色：定义一个抽象接口，来规范准备附加功能的类。

具体构件角色：将要被附加功能的类，实现抽象构件角色接口。

抽象装饰者角色：持有对具体构件角色的引用并定义与抽象构件角色一致的接口。

具体装饰角色：实现抽象装饰者角色，负责为具体构件添加额外功能。

实现：

OO 原则：动态地将责任附加到对象上。想要扩展功能，装饰者提供有别于继承的另一种选择。

要点：

1、继承属于扩展形式之一，但不见得是达到弹性设计的最佳方案。

2、在我们的设计中，应该允许行为可以被扩展，而不须修改现有的代码。

3、组合和委托可用于在运行时动态地加上新的行为。

4、除了继承，装饰者模式也可以让我们扩展行为。

5、装饰者模式意味着一群装饰者类，这些类用来包装具体组件。

6、装饰者类反映出被装饰的组件类型（实际上，他们具有相同的类型，都经过接口或继承实现）。

7、装饰者可以在被装饰者的行为前面与/或后面加上自己的行为，甚至将被装饰者的行为整个取代掉，而达到特定的目的。

8、你可以有无所数个装饰者包装一个组件。

9、装饰者一般对组建的客户是透明的，除非客户程序依赖于组件的具体类型。

4、总结

1、 使用 File 类可以进行文件的本身操作，创建、删除。
2、 文件的内容可以使用字节流或字符流完成操作
   - 字节流：OutputStream、InputStream
   - 字符流：Writer、Reader
   - 如果要想操作的是文件的话，则使用 FileOutputStream、FileInputStream、FileReader、FileWriter
   - 字符流在操作的时候使用了缓冲区
3、 字节-字符的转换流：可以将输出的字符流变为字节流，也可以将输入的字节流变为字符流。
4、 打印流提供了非常方便的文件打印功能，使用装饰设计模式完成
5、 对象序列化：
   - 序列化：ObjectOutputStream
   - 反序列化：ObjectInputStream
   - 对象所在的类必须实现 Serializable 接口，表示此对象可以被序列化
   - 如果类中的某个属性不希望被序列化，则可以使用 transient 关键字声明
   - 如果要想序列化一组对象，则使用对象数组完成
6、 内存操作流
7、 了解字符乱码的产生
8、 只要输出操作就使用 PrintStream，只要是输入操作，可以使用 BufferedReader 或 Scanner
9、 通过代码反复的熟悉类的设计
10、 理解装饰者设计模式的原理及具体实现，应用

5、作业

1、 实现一个文件分割器，把一个大文件分割成若干个小文件（可根据情况自行设计），分割后的文件扩展名为 dat，文件名为：data1.dat、data2.dat、data3.dat……。
2、 把分割后的文件再合并（文件还原）成完整文件，与源文件一致。