Chapter 11 : Java I/O System

There was a significant change in the I/O library after Java 1.0, when the original **byte**-oriented library was supplemented with **char**-oriented, Unicode-based I/O classes.

11.1 The File class

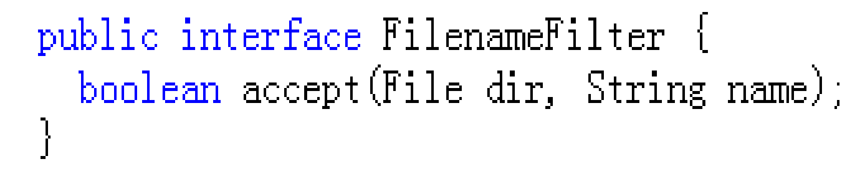
The File class can represent either the *name* of a particular file or the *names* of a set of files in a directory.

11.1.1 A directory listener

1. Call **list( )** with no arguments ---- the full list

2. Use a “directory filter” ---- a restricted list

Sample : DirList.java



```
public interface FilenameFilter {
  boolean accept(File dir, String name);
}
```

This interface provides the **accept( )** method to the **list( )** method so that **list( )** can “call back” **accept( )** to determine which file names should be included in the list.

Sample : DirList3.java

11.1.2 Checking for and creating directories

You can use a **File** object to create a new directory or an entire directory path. You can also look at the characteristics of files, see whether a **File** object represents a file or a directory, and delete a file.

Sample : MakeDirectories.java

11.2 Input and output

I/O libraries often use the abstraction of a *stream*, which represents any data source or sink as an object capable of producing or receiving pieces of data.

Everything derived from the **InputStream** or **Reader** classes have basic methods called **read( )** .

Everything derived from **OutputStream** or **Writer** classes have basic methods called **write( )**.

Other classes provide a more useful interface (based on these classes). Thus, you will layer multiple objects together to provide your desired I/O functionality.

11.2.1 **Types of InputStream**

Data sources can be:

1. An array of bytes.
2. A String object.
3. A file.

4. A “pipe”.

5. A sequence of other streams.

6. Other sources (Internet connection).

Each of these has an associated subclass of **InputStream**. See Table 11-1 on Page 582.

11.2.2 **Types of OutputStream**

See Table 11-2 on Page 583.

11.3 **Adding attributes and useful interfaces**

The use of layered objects to dynamically and transparently add responsibilities to individual objects is referred to as the *Decorator* pattern.

The decorator pattern specifies that all objects that wrap around your initial object have the same interface. Thus you send the same message to an object whether it's been decorated or not.

The Java I/O library requires many different combinations of features, which is why the decorator pattern is used.

The classes that provide the decorator interface to control a particular **InputStream** or **OutputStream** are the **FilterInputStream** and **FilterOutputStream**.

11.3.1 Reading from an InputStream with FilterInputStream

1. **DataInputStream** allows you to read different types of primitive data as well as **String** objects. This, along with its companion **DataOutputStream**, allows you to move primitive data from one place to another via a stream.

2. The remaining classes modify the way an **InputStream** behaves internally.

See Table 11-3 on Page 586.

11.3.2 Writing to an OutputStream with FilterOutputStream

The original intent of **PrintStream** was to print all of the primitive data types and **String** objects in a viewable format. The two important methods in **PrintStream** are **print( )** and **println( )**.

BufferedOutputStream is a modifier and tells the stream to use buffering.

See Table 11-4 on Page 588.

11.4 Readers & Writers

Java 1.1 made some significant modifications to the fundamental I/O stream library.

The **InputStream** and **OutputStream** classes still provide valuable functionality in the form of **byte-**oriented I/O, while the **Reader** and **Writer** classes provide Unicode-compliant, character-based I/O.

Java provides "bridge" classes: **InputStreamReader** converts an **InputStream** to a **Reader** and **OutputStreamWriter** converts an **OutputStream** to a **Writer**.

11.4.1 Sources and sinks of data

Almost all of the original Java I/O stream classes have corresponding **Reader** and **Writer** classes.

The most sensible approach to take is to *try* to use the **Reader** and **Writer** classes whenever you can. If you find your code won't compile, you have to use the **byte**-oriented libraries.

See the table on Page 590.

11.4.2 Modifying stream behavior (new decorators)

See the table on Page 591.

Whenever you want to use **readLine( )**, you should use a **BufferedReader**.

For storing and retrieving data in a transportable format you should use the **InputStream** and **OutputStream** hierarchies.(**DataInputStream & DataOutputStream**).

11.5 RandomAccessFile

RandomAccessFile is used for files containing records of known size so that you can move from one record to another using **seek( )**.

Note : It's a completely separate class.

A **RandomAccessFile** works like a **DataInputStream** pasted together with a **DataOutputStream**, along with the methods getFilePointer( ) ,seek( ) and length( ).

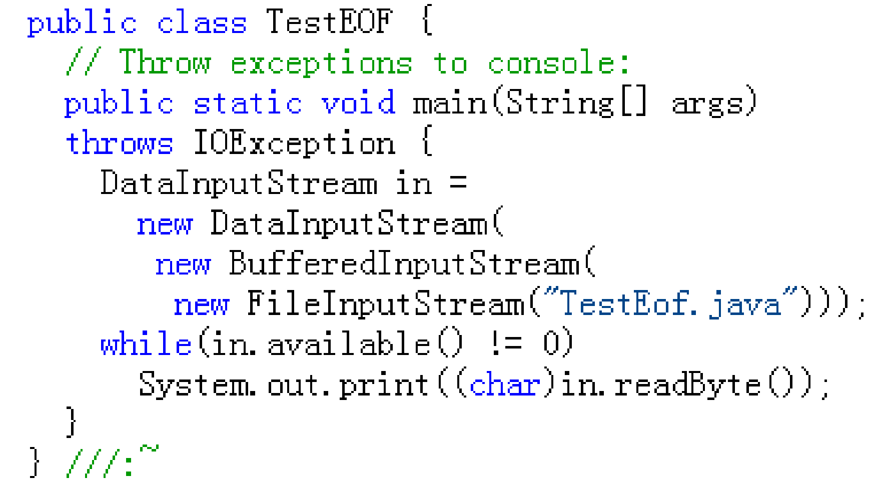
The seeking methods are available only in **RandomAccessFile**, which works for files only.

11.6 Typical uses of I/O streams

Sample : IOStreamDemo.java *

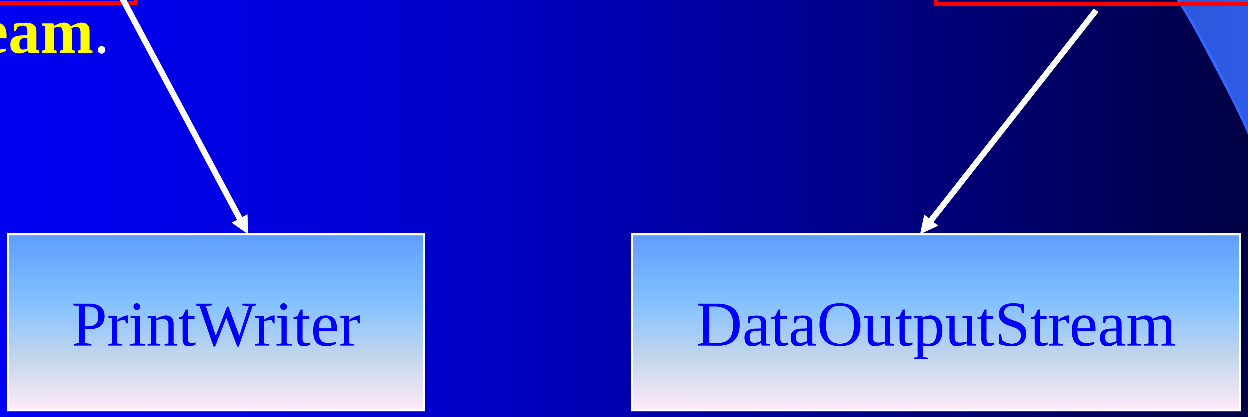
To read "formatted" data, you can use a **DataInputStream.**

If you read the characters from a **DataInputStream** using **readByte( )**, you can use **available( )** to find out how many more characters are available.



```java
public class TestEOF {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException {
    DataInputStream in =
      new DataInputStream(
       new BufferedInputStream(
        new FileInputStream(“TestEof.java”)));
    while(in.available() != 0)
      System.out.print((char)in.readByte());
  }
} ///:~
```

There are two primary kinds of output streams : one writes data for human consumption, and the other writes data to be reacquired by a **DataInputStream**.



11.7 Standard I/O

All the program’s input can come from *standard input*, all its output can go to *standard output*, and all of its error messages can be sent to *standard error*.

Programs can easily be chained together through standard I/O.

11.7.1 Reading from standard input

System.in is a raw **InputStream**, with no wrapping. It must be wrapped before you can read from it.

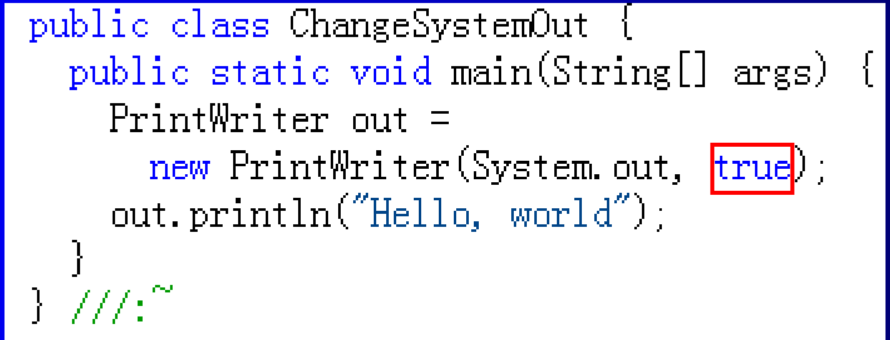
Typically, you’ll want to read input using **readLine( )**, so you’ll wrap **System.in** in a **BufferedReader**. ---- Use InputStreamReader to convert System.in to a Reader.

Sample : Echo.java *

readLine( ) can throw an **IOException**.

System.in should usually be buffered.

11.7.2 Changing System.out to a PrintWriter



```
public class ChangeSystemOut {
  public static void main(String[] args) {
    PrintWriter out =
      new PrintWriter(System.out, true);
    out.println("Hello, world");
  }
} ///:~
```

11.7.3 Redirecting standard I/O

setIn(InputStream)
setOut(PrintStream)
setErr(PrintStream)

I/O redirection manipulates streams of bytes, not streams of characters.

Sample : Redirecting.java

11.8 Java ARchives (JARs)

A JAR file consists of a single file containing a collection of zipped files along with a "manifest" that describes them.

The **jar** utility that comes with Sun’s JDK automatically compresses the files of your choice.

```
jar [options] destination [manifest] inputfile(s)
```

See the table on Page 611.

```
jar cf myJarFile.jar *.class
```

This creates a JAR file called **myJarFile.jar** that contains all of the class files in the current directory, along with an automatically generated manifest file.

A JAR file created on one platform will be transparently readable by the **jar** tool on any other platform.

11.9 Object serialization

Java's *object serialization* allows you to take any object that implements the **Serializable** interface and turn it into a sequence of bytes that can later be fully restored to regenerate the original object. ---- ***lightweight persistence***

The serialization mechanism automatically compensates for differences in operating systems.

To serialize an object, you create some sort of **OutputStream** object and then wrap it inside an **ObjectOutputStream** object. At this point you need only call **writeObject( )** and your object is serialized and sent to the **OutputStream**.

Object serialization not only saves an image of your object but it also follows all the references contained in your object and saves *those* objects. ---- web of objects

Sample : Worm.java *

Note : No constructor, not even the default constructor, is called in the process of deserializing a **Serializable** object. The entire object is restored by recovering data from the **InputStream**.

11.9.1 Controlling serialization

You can control the process of serialization by implementing the **Externalizable** interface. ---- **writeExternal( )** and **readExternal( )**

Note : When a **Serializable** object is recovered, it is constructed entirely from its stored bits, with no constructor calls. However, With an **Externalizable** object, all the normal default construction behavior occurs, and *then* **readExternal( )** is called.

Sample : Blip3.java *

If you are inheriting from an **Externalizable** object, you'll typically call the base-class versions of **writeExternal( )** and **readExternal( )** to provide proper storage and retrieval of the base-class components.

11.9.2 The transient keyword

One way to prevent sensitive parts of your object from being serialized is to implement your class as **Externalizable**.

If you're working with a **Serializable** object, you can turn off serialization on a field-by-field basis using the **transient** keyword.

Sample : Logon.java

11.9.3 Serialization of static fields

serializeStaticState( ) and **deserializeStaticState( )**

See Sample : CADState.java