Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Input Output in Java : Introduction/5b7666a04227cf4f24a3053f

TUTORIAL

Input Output in Java : Introduction

Topics

Storage of data in variables and arrays is temporary—the data is lost when a local variable "goes out of scope" or when the program terminates. Programs use files for long-term retention of large amounts of data, even after programs that create the data terminate. We refer to data maintained in files as persistent data, because the data exists beyond the duration of program execution. Computers store files on secondary storage devices such as magnetic disks, optical disks and magnetic tapes. File processing is one of the most important capabilities a language must have to support commercial applications that typically process massive amounts of persistent data. File processing is a subset of Java's stream-processing capabilities that enable a program to read and write bytes in memory, in files and over network connections. JDK has two sets of I/O packages:

- the Standard I/O (in package java.io), introduced since JDK 1.0 for stream-based I/O, and
- the New I/O (in packages java.nio), introduced in JDK 1.4, for more efficient buffer-based I/O.

JDK 1.5 introduces the formatted text-I/O via new classes java.util.Scanner and Formatter, and C-like printf() and format() methods for formatted output using format specifiers.

JDK 1.7 enhances supports for file I/O via the so-called NIO.2 (non-blocking I/O) in new package java.nio.file and its auxiliary packages. It also introduces a new try-with-resources syntax to simplify the coding of close() method.

Streams

Different kinds of I/O hardware, such as keyboards, disks, and magnetic stripe readers, behave in different ways and even similar types of hardware behave differently from one manufacturer to another. Most programming languages standardize their view of the way I/O works using the concept of a stream. A stream is a sequence of information—either bytes for binary information or characters for text information—for input and output. A stream is connected to a source for input or a destination for output. The stream handles the details of the

different types of hardware. Connecting a stream to a source or destination is called opening the stream, and disconnecting it, which is done when the source or destination is no longer being used, is called closing the stream.

The act of obtaining information from an input stream is called reading and the act of appending information to an output stream is called writing. On input, the information is read, starting with the first byte (or character), and each successive read performed by the program reads the next bytes or characters. Similarly, on output the stream starts as empty and each write appends to the end of the stream.

On reading, since the amount of information contained in a stream is finite, there will be a situation when there are no more (or not enough) bytes or characters remaining in the stream. This situation is called reaching end-of-file (EOF) since, traditionally, files have been the usual source for a stream. This leads to the stream I/O programming pattern as below,

```
open stream
        statements involving I/O to/from the stream
close stream
```

In which a stream is first opened, the information in it is processed (or generated), and finally the stream is closed.

Java views each file as a sequential stream of bytes. Each file ends either with an end-of-file marker or at a specific byte number recorded in a system-maintained administrative data structure. Java abstracts this concept from the programmer. A Java program processing a stream of bytes simply receives an indication from the system when the program reaches the end of the stream—the program does not need to know how the underlying platform represents files or streams. A Java program opens a file by creating an object and associating a stream of bytes with the object. Java also can associates streams of bytes associated with devices. In fact, Java creates three stream objects that are associated with devices when a Java program begins executing—System.in, System.out and System.err. The streams associated with these objects provide communication channels between a program and a particular device. For example, object System.in (the standard input stream object) normally enables a program to input bytes from the keyboard, object System.out (the standard output stream object) normally enables a program to output data to the screen and object System.err (the standard error stream object) normally enables a program to output error messages to the screen. Each of these streams can be redirected. For System.in, this enables the program to read bytes from a different source. For System.out and System.err, this enables the output to be sent to a different location, such as a file on disk. Class System provides methods setIn, setOut and setErr to redirect the standard input, output and error streams.

Java programs perform file processing by using classes from package java.io. This package includes definitions for the stream classes, such as FileInputStream (for

byte-based input from a file), FileOutputStream (for byte-based output to a file), FileReader (for character-based input from a file) and FileWriter (for character based output to a file). Files are opened by creating objects of these stream classes that inherit from classes InputStream, OutputStream, Reader and Writer, respectively. Thus, the methods of these stream classes can all be applied to file streams as well. To perform input and output of data types, objects of class ObjectInputStream, DataInputStream, ObjectOutputStream and DataOutputStream will be used together with the byte-based file stream classes FileInputStream and File-OutputStream.

Formatted Data-Streams: DataInputStream & DataOutputStream

The `DataInputStream` and `DataOutputStream` can be stacked on top of any `InputStream` and `OutputStream` to parse the raw bytes so as to perform I/O operations in the desired data format, such as `int` and `double`.

To use `DataInputStream` for formatted input, you can chain up the input streams as follows:

```
DataInputStream in = new DataInputStream(
                        new BufferedInputStream(
                           new FileInputStream("in.dat")));
```

`DataInputStream` implements `DataInput` interface, which provides methods to read formatted primitive data and `String`, such as:

```
// 8 Primitives
public final int readInt() throws IOExcpetion;       // Read 4 bytes and
convert into int
public final double readDouble() throws IOExcpetion;  // Read 8 bytes and
convert into double
public final byte readByte() throws IOExcpetion;
public final char readChar() throws IOExcpetion;
public final short readShort() throws IOExcpetion;
public final long readLong() throws IOExcpetion;
public final boolean readBoolean() throws IOExcpetion;    // Read 1 byte.
Convert to false if zero
public final float readFloat() throws IOExcpetion;

public final int readUnsignedByte() throws IOExcpetion;   // Read 1 byte in [0,
255] upcast to int
public final int readUnsignedShort() throws IOExcpetion;  // Read 2 bytes in
[0, 65535], same as char, upcast to int
public final void readFully(byte[] b, int off, int len) throws IOException;
public final void readFully(byte[] b) throws IOException;

// Strings
public final String readLine() throws IOException;
     // Read a line (until newline), convert each byte into a char - no unicode
support.
public final String readUTF() throws IOException;
    // read a UTF-encoded string with first two bytes indicating its UTF bytes
length
```

```
public final int skipBytes(int n)  // Skip a number of bytes
```

Similarly, you can stack the `DataOutputStream` as follows:

```
DataOutputStream out = new DataOutputStream(
                          new BufferedOutputStream(
                             new FileOutputStream("out.dat")));
```

`DataOutputStream` implements `DataOutput` interface, which provides methods to write formatted primitive data and `String`. For examples,

```
// 8 primitive types
public final void writeInt(int i) throws IOExcpetion;      // Write the int as
4 bytes
public final void writeFloat(float f) throws IOExcpetion;
public final void writeDouble(double d) throws IOExcpetion; // Write the double
as 8 bytes
public final void writeByte(int b) throws IOExcpetion;     // least-significant
byte
public final void writeShort(int s) throws IOExcpetion;    // two lower bytes
public final void writeLong(long l) throws IOExcpetion;
public final void writeBoolean(boolean b) throws IOExcpetion;
public final void writeChar(int i) throws IOExcpetion;

// String
public final void writeBytes(String str) throws IOExcpetion;
     // least-significant byte of each char
public final void writeChars(String str) throws IOExcpetion;
     // Write String as UCS-2 16-bit char, Big-endian (big byte first)
public final void writeUTF(String str) throws IOException;
     // Write String as UTF, with first two bytes indicating UTF bytes length

public final void write(byte[] b, int off, int len) throws IOException
public final void write(byte[] b) throws IOException
public final void write(int b) throws IOException     // Write the least-
significant byte
```

Example: The following program writes some primitives to a disk file. It then reads the raw bytes to check how the primitives were stored. Finally, it reads the data as primitives.

```java
import java.io.*;
class Main
{
  public static void main(String[] args)
  {
    String filename = "datafile.dat";
    String message = "Hi,CodeQuotient!";

    // Write primitives to an output file
    try (DataOutputStream out = new DataOutputStream( new
BufferedOutputStream( new FileOutputStream(filename))))
```

```
    {
      out.writeByte(107);
      out.writeShort(0xFFF0);  // -1
      out.writeInt(0x10B7);
      out.writeFloat(55.87f);
      out.writeDouble(55.56);
      out.writeBoolean(true);
      out.writeBoolean(false);
      for (int i = 0; i < message.length(); ++i)
        out.writeChar(message.charAt(i));
      out.writeChars(message);
      out.writeBytes(message);
      out.flush();
    } catch (IOException ex) {
      ex.printStackTrace();
    }

    // Read primitives
    try (DataInputStream in = new DataInputStream( new BufferedInputStream( new FileInputStream(filename))))
    {
      System.out.println("byte:    " + in.readByte());
      System.out.println("short:   " + in.readShort());
      System.out.println("int:     " + in.readInt());
      System.out.println("float:   " + in.readFloat());
      System.out.println("double:  " + in.readDouble());
      System.out.println("boolean: " + in.readBoolean());
      System.out.println("boolean: " + in.readBoolean());

      System.out.print("char:    ");
      for (int i = 0; i < message.length(); ++i) {
        System.out.print(in.readChar());
        // Can also be done by System.out.print((char)in.readByte());
      }
      System.out.println();
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```