Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Input Output in Java : Buffered Read Write/5b7668b34227cf4f24a305dd

TUTORIAL

Input Output in Java : Buffered Read Write

Topics

1.2 Implementations of InputStream/OutputStream

Input-Output using bytes

Streams which read/write bytes are called Byte streams and streams which read/write characters are called Char streams. To read/write bytes we use **InputStream** and **OutputStream** base classes and further sub-classes for the same: Various sub-classes under InputStream are FileInputStream, FilterInputStream, ByteArrayInputStream etc. and similary counterparts under OutputStream class e.g. FileOutputStream, FilterOutputStream etc. Out of these, we generally use BufferedInputStream and BufferedOutputStream (further sub-classes of FilterInputStream and FilterOutputStream) to read and write data from devices like keyboard, monitor, hard disk etc.

Reading from an InputStream

The abstract superclass InputStream declares an abstract method read() to read one data-byte from the input source:

```
public abstract int read() throws IOException
```

The `read()` method:

- returns the input byte read as an `int` in the range of 0 to 255, or
- returns -1 if "end of stream" condition is detected, or

- throws an `IOException` if it encounters an I/O error.

The `read()` method returns an `int` instead of a `byte`, because it uses -1 to indicate end-of-stream. It *blocks* until a byte is available, an I/O error occurs, or the "end-of-stream" is detected. The term "*block*" means that the method (and the program) will be suspended. The program will resume only when the method returns. Two variations of `read()` methods are implemented in the `InputStream` for reading a block of bytes into a byte-array. It returns the number of bytes read, or -1 if "end-of-stream" encounters.

```
public int read(byte[] bytes, int offset, int length) throws
IOException      // Read "length" number of bytes, store in bytes
array starting from offset of index.
public int read(byte[] bytes) throws IOException
            // Same as read(bytes, 0, bytes.length)
```

Writing to an OutputStream

Similar to the input counterpart, the abstract superclass OutputStream declares an abstract method write() to write a data-byte to the output sink. write() takes an int. The least-significant byte of the int argument is written out; the upper 3 bytes are discarded. It throws an IOException if I/O error occurs (e.g., output stream has been closed).

```
public void abstract void write(int unsignedByte) throws
IOException
```

Similar to the read(), two variations of the write() method to write a block of bytes from a byte-array are implemented:

```
public void write(byte[] bytes, int offset, int length) throws
IOException    // Write "length" number of bytes, from the bytes
array starting from offset of index.
public void write(byte[] bytes) throws IOException
            // Same as write(bytes, 0, bytes.length)
```

Opening & Closing I/O Streams

You open an I/O stream by constructing an instance of the stream. Both the InputStream and the OutputStream provides a close() method to close the stream, which performs the necessary clean-up operations to free up the system resources.

```
public void close() throws IOException  // close this Stream
```

It is a good practice to explicitly close the I/O stream, by running close() in the finally clause of try-catch-finally to free up the system resources immediately when the stream is no longer needed. This could prevent serious resource leaks. Unfortunately, the close() method also throws a IOException, and needs to be enclosed in a nested try-catch statement, as follows. This makes the codes somehow ugly.

```
FileInputStream in = null;
......
try {
   in = new FileInputStream(...);  // Open stream
   ......
   ......
} catch (IOException ex) {
   ex.printStackTrace();
} finally {  // always close the I/O streams
   try {
      if (in != null) in.close();
   } catch (IOException ex) {
      ex.printStackTrace();
   }
}
```

JDK 1.7 introduces a new try-with-resources syntax, which automatically closes all the opened resources after try or catch, as follows. This produces much neater codes.

```
try (FileInputStream in = new FileInputStream(...)) {
   ......
   ......
} catch (IOException ex) {
   ex.printStackTrace();
}  // Automatically closes all opened resource in try (...).
```

Flushing the OutputStream

In addition, the OutputStream provides a flush() method to flush the remaining bytes from the output buffer.

```
public void flush() throws IOException  // Flush the output
```

Implementations of InputStream/OutputStream

InputStream and OutputStream are abstract classes that cannot be instantiated. You need to choose an appropriate concrete subclass to establish a connection to a physical device. For example, you can instantiate a FileInputStream or FileOutputStream to establish a stream to a physical disk file.

Buffered I/O Byte-Streams - BufferedInputStream & BufferedOutputStream

The read()/write() method in InputStream/OutputStream are designed to read/write a single byte of data on each call. This is grossly inefficient, as each call is handled by the underlying operating system (which may trigger a disk access, or other expensive operations). Buffering, which reads/writes a block of bytes from the external device into/from a memory buffer in a single I/O operation, is commonly applied to speed up the I/O.

FileInputStream/FileOutputStream is not buffered. It is often chained to a BufferedInputStream or BufferedOutputStream, which provides the buffering. To chain the streams together, simply pass an instance of one stream into the constructor of another stream. For example, the following codes chain a FileInputStream to a BufferedInputStream, and finally, a DataInputStream:

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

// or we can do it in a single line as below
```

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

For example, the below program will copy each byte of a file to another file: -

```java
import java.io.*;
class Main {
  public static void main(String[] args)   {
    String inFileStr = "ByteBased.png";
    String outFileStr = "ByteBased2.png";
    FileInputStream in = null;
    FileOutputStream out = null;
    long startTime, elapsedTime;  // for speed benchmarking

    // Print file length
    File fileIn = new File(inFileStr);
    System.out.println("File size is " + fileIn.length() + " bytes");

    try {
      in = new FileInputStream(inFileStr);
      out = new FileOutputStream(outFileStr);

      startTime = System.nanoTime();
      int byteRead;
      // Read a raw byte, returns an int of 0 to 255.
      while ((byteRead = in.read()) != -1) {
        // Write the least-significant byte of int, drop the upper 3 bytes
        out.write(byteRead);
      }
      elapsedTime = System.nanoTime() - startTime;
      System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
    } catch (IOException ex) {
      ex.printStackTrace();
    } finally {  // always close the I/O streams
      try {
        if (in != null) in.close();
        if (out != null) out.close();
      } catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

We can also use **BufferedInputStream** and **BufferedOutputStream** for buffering between input and output provided with Java: -

Java

```java
import java.io.*;
class Main {
   public static void main(String[] args)    {
      String inFileStr = "ByteBased.png";
      String outFileStr = "ByteBased2.png";
      BufferedInputStream in = null;
      BufferedOutputStream out = null;
      long startTime, elapsedTime;

      // Check file length
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");

      try {
         in  = new BufferedInputStream(new FileInputStream(inFileStr));
         out = new BufferedOutputStream(new FileOutputStream(outFileStr));
         startTime = System.nanoTime();
         int byteRead;
         while ((byteRead = in.read()) != -1) {  // Read byte-by-byte from buffer
            out.write(byteRead);
         }
         elapsedTime = System.nanoTime() - startTime;
         System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
      } catch (IOException ex) {
         ex.printStackTrace();
      } finally {            // always close the streams
         try {
            if (in != null) in.close();
            if (out != null) out.close();
         } catch (IOException ex) { ex.printStackTrace(); }
      }
   }
}
```