# Java - Streams, Files and I/O

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters etc.

A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Java does provide strong, flexible support for I/O as it relates to files and networks but this tutorial covers very basic functionality related to streams and I/O. We would see most commonly used example one by one:

Streams

Java programs perform I/O through streams. Astream is an abstraction that either produces

or consumes information. A stream is linked to a physical device by the Java I/O system.

All streams behave in the same manner, even if the actual physical devices to which they are

linked differ. Thus, the same I/O classes and methods can be applied to any type of device.

This means that an input stream can abstract many different kinds of input: from a disk file,

a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk

file, or a network connection. Streams are a clean way to deal with input/output without

having every part of your code understand the difference between a keyboard and a network,

for example. Java implements streams within class hierarchies defined in thejava.io package.

Byte Streams and Character Streams

Java defines two types of streams: byte and character. Byte streamsprovide a convenient

means for handling input and output of bytes. Byte streams are used, for example, when

reading or writing binary data. Character streamsprovide a convenient means for handling

input and output of characters. They use Unicode and, therefore, can be internationalized.

Also, in some cases, character streams are more efficient than byte streams.

The original version of Java (Java 1.0) did not include character streams and, thus, all

I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented

classes and methods were deprecated. This is why older code that doesn’t use character streams

should be updated to take advantage of them, where appropriate.

One other point: at the lowest level, all I/O is still byte-oriented. The character-based

streams simply provide a convenient and efficient means for handling characters.

An overview of both byte-oriented streams and character-oriented streams is presented

in the following sections.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes:

InputStream and OutputStream. Each of these abstract classes has several concrete subclasses

that handle the differences between various devices, such as disk files, network connections,

and even memory buffers. The byte stream classes are shown in Table 13-1. A few of these

classes are discussed later in this section. Others are described in Part II. Remember, to use

the stream classes, you must import java.io.

The abstract classes InputStream and OutputStreamdefine several key methods that

the other stream classes implement. Two of the most important areread( ) and write( ),

which, respectively, read and write bytes of data. Both methods are declared as abstract

inside InputStream and OutputStream. They are overridden by derived stream classes.

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract

classes, Readerand Writer. These abstract classes handle Unicode character streams. Java

has several concrete subclasses of each of these. The character stream classes are shown in

Table 13-2.

The abstract classes Readerand Writer define several key methods that the other stream

classes implement. Two of the most important methods areread( ) and write( ), which read

and write characters of data, respectively. These methods are overridden by derived stream

classes.

Stream Class Meaning

BufferedInputStream Buffered input stream

BufferedOutputStream Buffered output stream

ByteArrayInputStream Input stream that reads from a byte array

ByteArrayOutputStream Output stream that writes to a byte array

DataInputStream An input stream that contains methods for reading the Java standard

data types

DataOutputStream An output stream that contains methods for writing the Java standard

data types

FileInputStream Input stream that reads from a file

FileOutputStream Output stream that writes to a file

FilterInputStream Implements InputStream

FilterOutputStream Implements OutputStream

InputStream Abstract class that describes stream input

ObjectInputStream Input stream for objects

ObjectOutputStream Output stream for objects

OutputStream Abstract class that describes stream output

PipedInputStream Input pipe

PipedOutputStream Output pipe

PrintStream Output stream that containsprint( )and println( )

PushbackInputStream Input stream that supports one-byte “unget,” which returns a byte to

the input stream

RandomAccessFile Supports random access file I/O

SequenceInputStream Input stream that is a combination of two or more input streams that

will be read sequentially, one after the other

TABLE 13-1 The Byte Stream Classes

Stream Class Meaning

BufferedReader Buffered input character stream

BufferedWriter Buffered output character stream

CharArrayReader Input stream that reads from a character array

CharArrayWriter Output stream that writes to a character array

FileReader Input stream that reads from a file

FileWriter Output stream that writes to a file

FilterReader Filtered reader

FilterWriter Filtered writer

Stream Class Meaning

InputStreamReader Input stream that translates bytes to characters

LineNumberReader Input stream that counts lines

OutputStreamWriter Output stream that translates characters to bytes

PipedReader Input pipe

PipedWriter Output pipe

PrintWriter Output stream that containsprint( )and println( )

PushbackReader Input stream that allows characters to be returned to the input stream

Reader Abstract class that describes character stream input

StringReader Input stream that reads from a string

StringWriter Output stream that writes to a string

Writer Abstract class that describes character stream output

TABLE 13-2 The Character Stream I/O Classes(continued)

TABLE 13-2 The Character Stream I/O Classes

The Predefined Streams

As you know, all Java programs automatically import thejava.lang package. This package

defines a class called System, which encapsulates several aspects of the run-time environment.

For example, using some of its methods, you can obtain the current time and the settings of

various properties associated with the system. Systemalso contains three predefined stream

variables: in , out, and err. These fields are declared as public , static, and final within

System. This means that they can be used by any other part of your program and without

reference to a specific Systemobject.

System.out refers to the standard output stream. By default, this is the console. System.in

refers to standard input, which is the keyboard by default. System.err refers to the standard

error stream, which also is the console by default. However, these streams may be redirected

to any compatible I/O device.

System.in is an object of type InputStream ; System.outand System.err are objects of

type PrintStream. These are byte streams, even though they typically are used to read and

write characters from and to the console. As you will see, you can wrap these within character-based streams, if desired.

The preceding chapters have been usingSystem.outin their examples. You can use

System.err in much the same way. As explained in the next section, use of System.in is a little more complicated.

**Reading Console Input:**

Java input console is accomplished by reading from **System.in**. To obtain a character-based stream that is attached to the console, you wrap **System.in** in a **BufferedReader** object, to create a character stream. Here is most common syntax to obtain BufferedReader:

BufferedReader br = new BufferedReader(new

InputStreamReader(System.in));

Once BufferedReader is obtained, we can use read( ) method to reach a character or readLine( ) method to read a string from the console.

**Reading Characters from Console:**

To read a character from a BufferedReader, we would read( ) method whose sytax is as follows:

int read( ) throws IOException

Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns -1 when the end of the stream is encountered. As you can see, it can throw an IOException.

The following program demonstrates read( ) by reading characters from the console until the user types a "q":

// Use a BufferedReader to read characters from the console.

import java.io.\*;

public class BRRead {

public static void main(String args[]) throws IOException

{

char c;

// Create a BufferedReader using System.in

BufferedReader br = new BufferedReader(new

InputStreamReader(System.in));

System.out.println("Enter characters, 'q' to quit.");

// read characters

do {

c = (char) br.read();

System.out.println(c);

} while(c != 'q');

}

}

Here is a sample run:

Enter characters, 'q' to quit.

123abcq

1

2

3

a

b

c

q

**Reading Strings from Console:**

To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is shown here:

String readLine( ) throws IOException

The following program demonstrates BufferedReader and the readLine( ) method. The program reads and displays lines of text until you enter the word "end":

// Read a string from console using a BufferedReader.

import java.io.\*;

public class BRReadLines {

public static void main(String args[]) throws IOException

{

// Create a BufferedReader using System.in

BufferedReader br = new BufferedReader(new

InputStreamReader(System.in));

String str;

System.out.println("Enter lines of text.");

System.out.println("Enter 'end' to quit.");

do {

str = br.readLine();

System.out.println(str);

} while(!str.equals("end"));

}

}

Here is a sample run:

Enter lines of text.

Enter 'end' to quit.

This is line one

This is line one

This is line two

This is line two

end

end

**Writing Console Output:**

Console output is most easily accomplished with **print( )** and **println( )**, described earlier. These methods are defined by the class **PrintStream** which is the type of the object referenced by **System.out**. Even though System.out is a byte stream, using it for simple program output is still acceptable.

Because PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ). Thus, write( ) can be used to write to the console. The simplest form of write( ) defined by PrintStream is shown here:

void write(int byteval)

This method writes to the stream the byte specified by byteval. Although byteval is declared as an integer, only the low-order eight bits are written.

**Example:**

Here is a short example that uses write( ) to output the character "A" followed by a newline to the screen:

import java.io.\*;

// Demonstrate System.out.write().

public class WriteDemo {

public static void main(String args[]) {

int b;

b = 'A';

System.out.write(b);

System.out.write('\n');

}

}

This would produce simply 'A' character on the output screen.

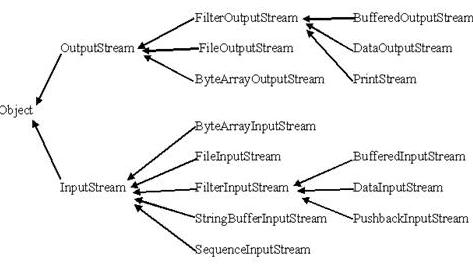
A

**Note:** You will not often use write( ) to perform console output because print( ) and println( ) are substantially easier to use.

**Reading and Writing Files:**

As described earlier, A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are FileInputStream and FileOutputStream which would be discussed in this tutorial:

**FileInputStream:**

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file.:

InputStream f = new FileInputStream("C:/java/hello");

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows:

File f = new File("C:/java/hello");

InputStream f = new FileInputStream(f);

Once you have *InputStream* object in hand then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException. |
| 2 | **protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException. |
| 3 | **public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file. |
| 4 | **public int read(byte[] r) throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned. |
| 5 | **public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int. |

There are other important input streams available, for more detail you can refer to the following links:

* ByteArrayInputStream
* DataInputStream

**FileOutputStream:**

FileOutputStream is used to create a file and write data into it.The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file.:

OutputStream f = new FileOutputStream("C:/java/hello")

Following constructor takes a file object to create an output stream object to write the file. First we create a file object using File() method as follows:

File f = new File("C:/java/hello");

OutputStream f = new FileOutputStream(f);

Once you have *OutputStream* object in hand then there is a list of helper methods which can be used to write to stream or to do other operations on the stream.

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException. |
| 2 | **protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException. |
| 3 | **public void write(int w)throws IOException{}** This methods writes the specified byte to the output stream. |
| 4 | **public void write(byte[] w)** Writes w.length bytes from the mentioned byte array to the OutputStream. |

There are other important output streams available, for more detail you can refer to the following links:

* ByteArrayOutputStream
* DataOutputStream

**Example:**

Following is the example to demonstrate InputStream and OutputStream:

import java.io.\*;

public class fileStreamTest{

public static void main(String args[]){

try{

byte bWrite [] = {11,21,3,40,5};

OutputStream os = new FileOutputStream("C:/test.txt");

for(int x=0; x < bWrite.length ; x++){

os.write( bWrite[x] ); // writes the bytes

}

os.close();

InputStream is = new FileInputStream("C:/test.txt");

int size = is.available();

for(int i=0; i< size; i++){

System.out.print((char)is.read() + " ");

}

is.close();

}catch(IOException e){

System.out.print("Exception");

}

}

}

The above code would create file test.txt and would write given numbers in binary format. Same would be output on the stdout screen.

**File Navigation and I/O:**

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

* File Class
* FileReader Class
* FileWriter Class

**Directories in Java:**

**Creating Directories:**

There are two useful **File** utility methods which can be used to create directories:

* The **mkdir( )** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
* The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory:

import java.io.File;

public class CreateDir {

public static void main(String args[]) {

String dirname = "/tmp/user/java/bin";

File d = new File(dirname);

// Create directory now.

d.mkdirs();

}

}

Compile and execute above code to create "/tmp/user/java/bin".

**Note:** Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

**Reading Directories:**

A directory is a File that contains a list of other files and directories. When you create a File object and it is a directory, the isDirectory( ) method will return true.

You can call list( ) on that object to extract the list of other files and directories inside. The program shown here illustrates how to use list( ) to examine the contents of a directory:

import java.io.File;

public class DirList {

public static void main(String args[]) {

String dirname = "/java";

File f1 = new File(dirname);

if (f1.isDirectory()) {

System.out.println( "Directory of " + dirname);

String s[] = f1.list();

for (int i=0; i < s.length; i++) {

File f = new File(dirname + "/" + s[i]);

if (f.isDirectory()) {

System.out.println(s[i] + " is a directory");

} else {

System.out.println(s[i] + " is a file");

}

}

} else {

System.out.println(dirname + " is not a directory");

}

}

}

This would produce following result:

Directory of /mysql

bin is a directory

lib is a directory

demo is a directory

test.txt is a file

README is a file

index.html is a file

include is a director