Module: Core Java

Session 22: Java IO

* I/O Basics
* Streams
* Predefined Stream Objects
* Data types and Encoding
* Classes Hierarchy of java.io package
* Basic input and output classes
* The File Class
* The FilenameFilter Interface
* The RandomAccessFile Class

**Objective**

At the end of this chapter you will be able to:

* Know about the I/O Basics
* Understand Streams and the Predefined Stream Objects
* Get some knowledge about Data types and Encoding
* Go deep into the Classes hierarchy of java.io package
* Basic input and output classes
* Get detail information about the File class
* Know about the FilenameFilter Interface
* Deal with the RandomAccessFile class

**I/O Basics**

Any programmer should know the basic idea of how to read from files and write to files while dealing with any new language. This is because he / she will always have the need of saving and loading data in most of the software developed. Hence, Java I/O has come up with simple facilities that include standardized API for reading and writing **character** and **byte** data from various data sources. This chapter will reflect the inside out of java.io package, scrutinizing I/O classes, methods, and various techniques for handling I/O in your Java code. But the java.io package, as the term suggests deals with input and output operations like reading a plain comma, a XML data file and delimited text file or something very different like a network stream.

#### Streams

Java I/O is based on the concept of streams. A stream is defined as a flowing sequence of characters. Most of the programs work with external data stored either in local files or they come from other computers on the network. Java’s concept allows it to work with the streams of data. When the physical data storage is mapped to a logical stream of data, a Java program reads data serially from this stream. The term serially here means byte after byte and character after character. Some of the types of streams are byte streams (InputStream, OutputStream) and character streams (Reader and Writer). There are different types of data, and hence different types of streams. You can see the flow of data in Fig. 1.

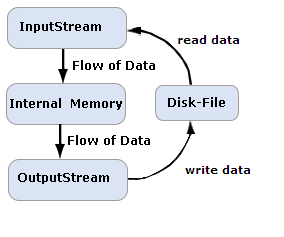


Fig. 1: Flow of Data

You need to know the following steps to work with a stream:

1. Open a stream pointing a specific data source: a file, a socket, URL, etc.
2. Then read or write data from/to this stream.
3. Close the stream.

There are lots of sources to provide input to Java program and output can reach a lot of places where it is destined. Thus the main work of stream classes between the sources and destinations remains abstracted. The image Fig.1 shows how we can read data from file with an InputStream and how we can write data to a file with an OutputStream.

Input stream functions like a like a siphon that sucks up water and output stream functions like a hose that discharges water. So, you can imagine a siphon-hose relationship where a siphon and a hose are connected to flow water from one place to another. A Siphon gets limited water source if it draws water from a small or finite vessel whereas if done from a river it gets unlimited water. Similarly, input stream may read from a file that is a finite source and from user prompt that is an unlimited source.

When sending a stream of data it is said that we are writing a stream. When receiving a stream of data we are said to be reading a stream. If an error occurs when reading or writing a stream, an exception (usually IOException) is thrown. This kind of exception can be handled by surrounding your stream statements with a try - catch block.

### Predefined Stream Objects

Three streams are made available to the programmer by the System class, i.e.

System.out - standard output stream

System.in - standard input stream

System.err - standard error stream

The first source of input most programmer’s encounter is System.in. You can find this with stdin in C. The java.lang.System class give console output through the static output field, that is, System.out. This system is similar to stdout in C language that may follow the similar way. Again, stderr runs as System.err. This is meant for printing error messages and debugging inside the catch clauses. Standard error stream deals with error related text messages that are revealed to the command-line application user. The standard output is different which often may be redirected to a file or another application and is not seen by the user.

The following example accepts a character from the user and prints it back in the prompt.

import java.io.\*;

public class InOutDemo

{

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

{

System.out.println((char)System.in.read());

}

}

In the above example you can see that the data read by System.in.read is casted into char, because read() method returns a byte. The input and output stream read and write bytes respectively. This means readers can read and writers can write characters. Hence, to understand how i/o operations are carried on with stream of bytes, you should know how Java deals with data types like bytes, integers, characters, and other primitive data types. You will also learn when and why one of these data types is converted into another.

#### Data types and Encoding

Numbers are only part of the data that a typical Java program needs to read and write. Many programs are composed of characters that need to handle text. As the whole work in computer is understood by numbers, character are thus encoded by matching each character in a given script to a particular number. Let us take an example of a common ASCII code, where we map the following characters: character A to the number 65, character B to the number 66; character C to the number 67 and so on.

### ASCII

ASCII stands for American Standard Code for Information Interchange. When computer understands numbers, ASCII code serves the purpose, as it is the numerical representation of a character. It is a seven-bit character set consisting of 27 or 128 different characters whose numeric values range from 0 to 127. These characters are sufficient to handle American English and European languages to a great extent. If you want to read a byte value between 0 and 127 from a stream, then convert it to a char, which is the corresponding ASCII character.

### ISO Latin-1

ISO 8859-1.was referred as ISO/IEC 8859-1. It is first part of ISO/IEC 8859, which is a standard character encoding of the Latin alphabet. It is sometimes called as Latin-1. This character encoding is used in places like America, Western Europe, Oceania, and much of [Africa](http://en.wikipedia.org/wiki/Africa). It is also commonly used in most standard romanizations of East-Asian languages. Every character is encoded as a single eight-bit code value. These code values can be used in almost any data interchange system to communicate in the following European languages. To communicate in the following European languages, these code values can be used in any data interchange system.

### Unicode

Unicode provides a unique number for every character, irrespective of platform, program and language. The Unicode Standard has been adopted by industry leaders such as Apple, IBM, HP, Microsoft, Oracle, JustSystem, Sun, SAP and Sybase etc.Each Unicode character is defined by 16 bits.

The first 256 characters of Unicode i.e., the characters whose high-order byte is zero are identical to the characters of the ISO Latin-1 character set. Thus, 65 is ASCII A and Unicode A; 66 is ASCII B and Unicode B and so on. Streams generally read a byte at a time, but each Unicode character occupies two bytes. Thus, to read a Unicode character, you multiply the first byte read by 256, add it to the second byte read, and cast the result to a char. For example:

int b1 = System.in.read();

int b2 = System.in.read();

char c = (char) (b1\*256 + b2);

### UTF-8

Unicode is a relatively inefficient encoding when most of your text consists of ASCII characters. Every character requires the same number of bytes (two) even though some characters are used much more frequently than others. A more efficient encoding would use fewer bits for the more common characters. This is what UTF-8 does. In UTF-8 the ASCII alphabet is encoded using a single byte, just as in ASCII.

**Classes Hierarchy of java.io package**

See Fig. 2 for java.io hierarchy of classes.

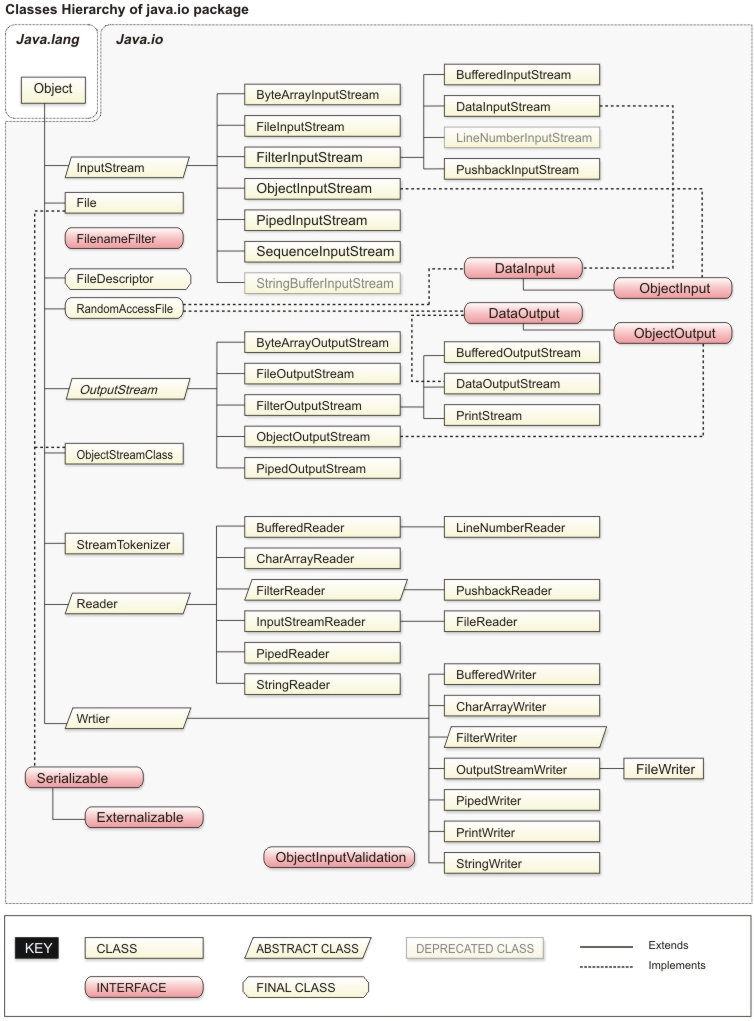


Fig. 2: java.io hierarchy of classes

#### Basic input and output classes

The java.io package includes a large number of classes dealing with all input and output operations. These classes consist of:

* Byte streams (subclasses of **InputStream** or **OutputStream).**
* Character streams (subclasses of **Reader** and **Writer).**

### Byte Streams

A program can use any one of the subclass of the InputStream or OutputStream, if there is necessity of reading or writing program respectively. InputStream and OutputStream define the lowest level interface for all byte streams because they are abstract classes. These streams contain methods that help in reading or writing an unstructured flow of byte-level data. But being abstract classes, generic input or output stream cannot be generated. Their subclasses are implemented by Java for activities such as reading from and writing to files and communicating with sockets. However, various kinds of byte streams can be used interchangeably as all byte streams inherit the structure of InputStream or OutputStream.To handle tasks like buffering, filtering, or handling higher-level data types specialized types of streams can also be layered.

### Character Streams

Streams, handling only character data are actually introduced by java.io Reader and Writer character stream classes. While using these classes any programmer can think it only in terms of characters and string data that allows the underlying implementation to handle the conversion of bytes to a specific character encoding. The character stream’s main advantage is that they make it easy to write programs that do not depend upon a specific character encoding. Hence, they are easy to internationalize. Unicode is an international standard character encoding that has the capability to represent most of the world's written languages. Java stores strings in this Unicode. The second main advantage of character streams is their efficiency. They are much more efficient than byte streams. Many of Java's original byte streams implementations are oriented around byte-at-a-time read and write operations. On the other hand, the character-stream classes are inclined towards buffer-at-a-time read and write operations.

### Streams

**InputStream Class**

The java.io.InputStream class fall under abstract superclass for all input streams. When there is need to read bytes of data from a stream, it declares three basic methods. Moreover, it has methods to close and flush streams, to check the number of bytes of data available to be read, to skip over input, to mark a position in a stream and reset back to that position and also to determine whether marking and resetting back are supported. The following list provides the details of some important methods in the InputStream class.

public abstract int read() throws IOException

public int read(byte[] data) throws IOException

public int read(byte[] data, int offset, int length) throws IOException

public long skip(long n) throws IOException

public int available() throws IOException

public void close() throws IOException

In the InputStream class, bytes can be read from three different sources:

* An array of bytes
* A file
* A pipe

**The read( ) Method**

InputStream class has read() method which is the fundamental method. This can read a single data with unsigned byte and returns the integer value of the unsigned byte.

Signature of the method is:

public abstract int read() throws IOException

The following code reads 10 bytes from the System.in input stream and stores them in the int array data:

int[] data = new int[10];

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

{

data[i] = System.in.read();

}

##### The available() Method

You should always know how many bytes are available to be read before you try to read them as per the convenience. The available() method of the InputStream class tells you how many bytes can be read without blocking. No data availabile for reading purpose means, it returns.

**The skip() Method**

In Java we have skip() method that can jump over a certain number of bytes in the input.

public long skip(long bytesToSkip) throws IOException

The skip() method argument means the number of bytes to skip. As a return value, it gives the number of bytes actually skipped, which may be less than bytesToSkip. Thus it returns -1 if the end of stream is encountered. The argument as well as return value are long which means it allows skip() to handle extremely long input streams.

**The close() Method**

When the work of the stream is over, you should close it. This allows the operating system to free any resources associated with the stream. To understand about these resources, you should know that it varies with the type of the stream and depends on your platform. It is not necessary that you should close all streams. For example System.in generally does not need to be closed. To close a stream, you invoke its close() method:

public void close() throws IOException

**InputStream marking**

Some, but not all, InputStreams support marking. Marking allows you to go back to a marked place in the stream like a bookmark for future reference. Remember, not all InputStreams support marking. To test if the stream supports the mark() and reset() methods, use the boolean markSupported() method. The mark() method, which takes an integer read\_limit, marks the current position in the input stream so that reset() can return to that position as long as no more than the specified number of bytes have been read between the mark() and reset().

### Output Streams

**OutputStream Class**

There are three basic methods declared by java.io.OutputStream class to write bytes of data onto a stream. It also has methods for closing and flushing streams. OutputStream is an abstract class. Their subclass provides the abstract write(int b) method implementations. It may be possible that four nonabstract methods are overridden. The following example of the FileOutputStream class overrides all five methods with original methods knowing how to write bytes into files on the host platform.

public abstract void write(int b) throws IOException

public void write(byte[] data) throws IOException

public void write(byte[] data, int offset, int length) throws IOException

public void flush() throws IOException

public void close() throws IOException

Bytes can be written to three different types of sinks:

* An array of bytes
* A file
* A pipe

**The write() Method :**

The write(int b) method writes a single unsigned byte of data whose value should be between 0 and 255. If you pass a number larger than 255 or smaller than zero, it's reduced modulo 256 before being written. The following demo generates the ASCII chart and uses the write() method for it.

Example:

import java.io.\*;

public class AsciiChart {

public static void main(String[] args) {

for (int i = 32; i < 127; i++)

{

System.out.write(i);

// break line after every eight characters.

if (i % 8 == 7) System.out.write('\n');

else System.out.write('\t');

}

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

}

}

The write() Method to work with byte array

Writing larger chunks of data can be faster than writing them byte by byte. There are two overloaded variants of the write() method doing this operation and they are:

public void write(byte[] data) throws IOException

public void write(byte[] data, int offset, int length) throws IOException

The entire byte array data is written by the first variant. Only the sub-array of data that starts at offset and continues for length bytes is written by second array.

The following example constructs a byte array filled with an ASCII chart, then sends it onto the console in one call to write().

Example:

import java.io.\*;

public class AsciiChart

{

public static void main(String[] args) {

byte[] b = new byte[(127-31)\*2];

int index = 0;

for (int i = 32; i < 127; i++) {

b[index++] = (byte) i;

// Break line after every eight characters.

if (i % 8 == 7) b[index++] = (byte) '\n';

else b[index++] = (byte) '\t';

}

b[index++] = (byte) '\n';

try {

System.out.write(b);

}

catch (IOException e) { System.err.println(e); }

}

}

The flush() and close() Method

There are many output streams buffer that writes to improve performance. It does this by the accumulated bytes in a memory buffer with size ranging from several bytes to several thousands bytes rather than sending each byte the way it is written to its destination. As soon as the buffer fills up, all the data is sent at the same time. Whether the buffer is full or not, the flush() method prepares the data to be written. If you use a stream for a short time, then there is no need to flush it explicitly but can be done once the stream is closed.This is possible when the program exits or when you explicitly invoke the close() method. System.out, System.err, and some (but not all) other print streams automatically flush after each call to println() and after each time a new line character ('\n') appears in the string is being written. The auto-flushing can be enabled in the PrintStream constructor.

#### The File Class

When we construct a File object, it represents that a physical file or directory on the disk. When we call its methods, we manipulate the underlying disk file.

The methods for File objects are:

* Constructors
* Test methods
* Action methods
* List methods

### Constructors

There are several constructors in the File class ther allow Java code to specify the initial values of an object. Constructors for the File class are:

File(String filename)

File(String pathname, String filename)

File(File directory, String filename)

### Test Methods

Public methods in the File class perform tests on the specified file. For example:

* The exists() method asks if the file actually exists.
* The canRead() method asks if the file is readable.
* The canWrite() method asks if the file can be written to.
* The isFile() method asks if it is a file (as opposed to a directory).
* The isDirectory() method asks if it is a directory.
* The isHidden() method asks if a file or directory object is Hidden

These methods are all of the boolean type and that is why they return a true or false.

### Action methods

Public instance methods in the File class perform actions on the specified file. Let's take a look at them:

* The renameTo() method renames a file or directory.
* The delete() method deletes a file or directory.
* The mkdir() method creates a directory specified by a File object.
* The mkdirs() method creates a list of directory in nested format.

The return type of all above methods is boolean to indicate whether the action was successful.

### List methods

The names of all the entries that are not rejected by an optional FilenameFilter are returned in a directory by the list() method. The list() method returns null if the File is a normal file, or returns the names in the directory. The list() method can take a FilenameFilter filter and return names in a directory satisfying the filter. More on FilenameFilter is discussed in the next section of the chapter. The listFiles() method works the same way as list() except that it returns as File [] (File array) format where as list() returns in String [](String array) format.

The following example demonstrates several methods of file class :

Example of File .class:

import java.io.\*;

public class FileDemo1

{

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

{

File f=new File("C:/SqlStar.txt");

if(f.exists())

{

System.out.println("SqlStar.txt exists");

}else

{

System.out.println("The new file getting created");

f.createNewFile();

//f.mkdir() can be used to create a directory

}

System.out.println("Is it a File ? : " + f.isFile());

System.out.println("Is it a Directory ? : " + f.isDirectory());

System.out.println("Is it Hidden ? : " + f.isHidden());

System.out.println("Is it Readable ? : " + f.canRead());

System.out.println("Its length is ? : " + f.length());

if(f.length()==0)

{

System.out.println("Deleting the file now");

f.delete();

}}}

The following example demonstrates the use of listFiles() method along with recursion principle to extract the list of files and directory names under the C:/WINDOWS directory.

import java.io.\*;

public class ListFileDemo

{

static int filecount=0;

static int foldercount=0;

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

{

File f=new File("C:/WINDOWS");

recursion(f);

System.out.println("Total files is : " + filecount);

System.out.println("Total directory is : " + foldercount);

}

static void recursion(File ff)

{

File [] arr=ff.listFiles();

String name;

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

{

File f1=arr[x];

name=f1.getName();

if(f1.isFile())

{

System.out.println(name);

filecount++;

}

else

{

System.out.println("[ " + name + " ] ");

foldercount++;

recursion(f1); // recursion call

}

}}}

**The FilenameFilter Interface**

The FilenameFilter interface helps in filtering filenames. You simply create a class that implements the FilenameFilter. Java language does not implement any standard FilenameFilter classes. In the File class, objects that implement this interface, are used by the FileDialog class and the list().The implemented accept() method determines if the filename in a directory should be included in a file list. You can pass with the directory and a file name. If the name is included in the list, the method returns true.

The following example shows the use of FilenameFilter:

import java.io.\*;

import java.util.\*;

public class FileFilterDemo

{

public static void main(String args[])

{

File a\_directory = new File("C:/WINDOWS");

if (a\_directory.isDirectory())

{

String names[] = a\_directory.list();

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

{

System.out.println("Filename is " + names[i]);

}}

System.out.println("Parent is " + a\_directory.getParent());

if (a\_directory.isDirectory())

{

String names[] = a\_directory.list(new MyFilter());

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

{

System.out.println("Filename is " + names[i]);

}}}}

//List all files whose name start from 'A' or 'a'

class MyFilter implements FilenameFilter

{

public boolean accept(File directory, String name)

{

if (name.charAt(0) == 'A' || name.charAt(0) == 'a')

{

return true;

}

else

{

return false;

}}}

#### The RandomAccessFile Class

The java.io.RandomAccessFile class is another way to read or modify files. This class has incompatibility in a model of files that it contains with the stream/reader/writer model described later in this chapter. The stream/reader/writer model was developed for general I/O purpose. But the RandomAccessFile class utilizes a particular behavior of files that is not found in general I/O devices. You should know that RandomAccessFile is not in the hierarchy of the InputStream or OutputStream. However, you can’t find any association with those hierarchies. But, it implements the DataInput and DataOutput interfaces. In a random-access file, a desired position can be seeked within a file and then it can be read or written with a desired amount of data. The RandomAccessFile class includes methods that support seeking, reading, and writing. The constructors for the class are as follows:

RandomAccessFile(String file, String mode)

RandomAccessFile(File file, String mode)

The mode string should be either “r” or “rw”. You can take “r” to open the file, only for reading and take rw” to open the file for both reading and writing. In addition to that, you can see that seek( ) is used to move about in the file and change one of the values. Methods like getFilePointer( ) to find out where you are in the file, mark( ) to mark a position (whose value is held in a single internal variable) and reset( ) to reset that position, and length( ) to determine the maximum size of the file provides more flexibility in file handling.

The following example prints its own source code on the prompt:

Example:

import java.io.\*;

public class RandomAccessDemo

{

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

{

RandomAccessFile f=new RandomAccessFile("C:/RandomAccessDemo.java","r");

String str="";

while(str!=null)

{

try

{

System.out.println(str=f.readLine());

}

catch(Exception e)

{

System.exit(0);

}}}}

**Summary**

In this session you have learned to work with streams. We used character streams to handle text and byte streams for any other kind of data. The java.io package consists of all the I/O related classes. Streams are very powerful ways to extend the functionality of your java programs because they offer a connection to any kind of data you want to work with.

After going through this chapter you will know that:

* File I/O is the capability of storing, retrieving & manipulating (both local & remote) files.
* Java provides the java.io package which consists of platform independent classes such as File, Random Access File,Input Stream , Output Stream.
* A stream is a continuous flow of bytes. It can be input stream or an output stream depending on whether it is used as a resource or destination for the information.
* The File class encapsulates useful attributes & methods pertaining to files & directories that can be referred to using an absolute or relative path.