**Managing Input / Output files in Java**

**Introduction**

The approach to store the data inside a program using variables and arrays poses the fallowing problems.

* The data is lost either when a variable goes out of scope or when the program is terminated. That is, the storage is temporary,
* It is difficult to handle large volumes of data using variables and arrays.

We can overcome these problems by storing data on secondary storage devices such as floppy disks or hard disks. The data is stored in these devices using the concept of files. Data stored in files is often called persistent data.

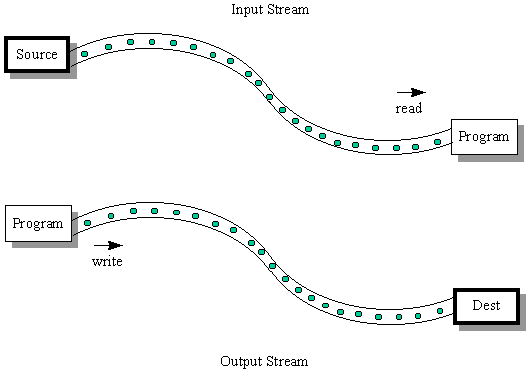
A file is a collection of related records placed in a particular area on the disk. A record is composed of several fields and a field is a group of characters. Characters in Java are Unicode characters composed of two bytes, each byte containing eight binary digits, 1 or 0.

**Concept of Streams**

In file processing input refers to the flow of data into a program and output means the flow of data out of a program. Input to a program may come from the keyboard, the mouse, the memory, the disk, a network, or another program. Similarly, output from a program may go to the screen , the printer, the memory, the disk, a network, or another program. These input and output devices share certain common characteristics such as unidirectional movement of data, treating; data as a sequence of bytes or characters and support to the sequential access to the data.

Java uses the concept of **streams** to represent the ordered sequence of data A stream presents a uniform, easy-to-use , object oriented interface between the program and the input, output devices.

A stream in Java is a path along which data flows (like a river or a pipe along which water flows), It has a **source** (of data) and a **destination** (for that data) .Both the source and the destination may be physical devices or programs or other streams in the same program.



Using input and out streams

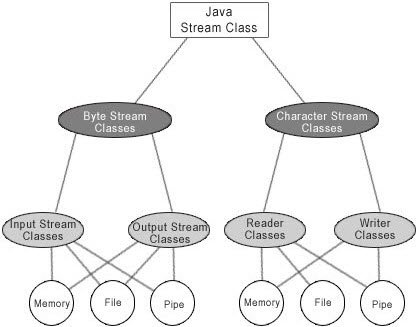
Java streams are classified into two basic types, namely, input stream and output stream. An input stream extracts (i.e., reads) data from the source (file) and sends it to the program. Similarly, the output stream takes data from the program and sends (i.e., writes) it to the destination (file).

**Stream Classes**

The **java.io** package contains a large number of stream classes that provide capabilities for processing all types of data. These classes may be categorized into two groups based on the data type on which they operate.

1. Byte stream classes that provide support for handling I/O operations on bytes.
2. Character stream classes that provide support for managing I/O operations on characters.

These two groups may further be classified based on their purpose. Byte stream and character stream classes contain specialized classes to deal with input and output operations independently on various types of devices. We can also cross-group the streams based on the type of source or destination they read from or write to. The source (or destination) may be memory, a file or a pipe.



Classification of java stream classes

**Byte Stream Classes**

Byte stream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Since the streams are unidirectional they can transmit bytes in only one direction and therefore, Java provides two kinds of byte stream classes:

**input stream classes** and **output stream classes**.

**Input Stream Classes**

Input stream classes that are used to read 8-bit bytes, include a super class known as **InputStream**  and a number of subclasses for supporting various input-related functions**.**

InputStream

**Hierarchy of Input Stream Classes**

The super class **InputStream** is an abstract class, and, therefore, we cannot create instances of this class. Rather, we must use the subclasses that inherit from this class. The InputStream class defines methods for performing input functions such as

* Reading bytes
* Closing streams
* Marking positions in streams
* Skipping ahead in a stream
* Finding the number of bytes in a stream

Methods provided by InputStream class are…

|  |  |
| --- | --- |
| **Method** | **Description** |
| read( ) | Reads a byte from the input stream |
| read( byte b[ ] ) | Reads an array of bytes into b |
| read( byte b[ ],int m, int n) | Reads n bytes into b starting from m th byte |
| available( ) | Gives number of bytes available in the input |
| skip(n) | Skips over n bytes from the input stream |
| reset( ) | Goes back to the beginning of the stream |
| close( ) | Closes the input stream |

Note that the class **DatalnputStream** extends **FilterInputStream** and implements the interface **DataInput**. Therefore, the DatalnputStream class implements the methods described in DataInput in addition to using the methods of InputStream class. The DataInput interface contains the following methods;

* readShort ( )
* readDouble( )
* readInt( )
* readLine( )
* readLong( )
* readChar( )
* readFloat( )
* readBollean( )
* readUTF( )

**Output Stream Classes**

Output stream classes are derived from the base class OutputStream. Like lnputStream, the OutputStream is an abstract class and therefore we cannot instantiate it. The several subclasses of the OutputStream can be used for performing the output operations

The OutputStream includes methods that are designed to perform the following tasks:

* Writing bytes
* Closing streams
* Flushing streams

OutputStream

**Hierarchy of Output Stream Classes**

Methods defined by OutputStream class are…

|  |  |
| --- | --- |
| **Method** | **Description** |
| write( ) | writes a byte from the output stream |
| write( byte b[ ] ) | writes all bytes in the array b to the output stream |
| write( byte b[ ],int m, int n) | Writes n bytes from array b starting from m th byte |
| close( ) | Closes the output stream |
| flush( ) | Flushes the output stream |

The DataOutputStream, a counterpart of DataInputStream, implements the interface DataOutput and therefore implements the following methods contained in DataOutput interface.

* writeShort ( )
* writeDouble( )
* writeInt( )
* writeLong( )
* writeBytes( )
* writeChar( )
* writeFloat( )
* writeBollean( )
* writeUTF( )

**Character Stream Classes**

Character streams can be used to read and write 16-bit Unicode characters. Like byte streams, there are two kinds of character stream classes namely **Reader Stream** classes and **Writer Stream** classes.

**Reader Stream Classes**

Reader Stream classes are designed to read character from the files. **Reader** class is the base class for all other classes of this group. These classes are functionally very similar to the input stream classes, except input streams use bytes as their fundamental unit of information, while reader streams use characters.

The **Reader** class contains methods that are identical to those available in the **InputStream** class, except Reader is designed to handle characters. Therefore, reader classes can perform all the functions implemented by the input stream classes.

Reader

FileReader

PushbackReader

**Hierarchy of Reader Stream Classes**

**Writer Stream Classes**

Like Output Stream classes, the writer stream classes are designed to perform all output operations on files. Only difference is that while output stream classes are designed to write bytes, the writer stream classes are designed to write characters.

The **Writer** class is an abstract class which acts as a base class for all the other writer stream classes. This base class provides support for all output operations by defining methods that are identical to those in **OutputStream** class.

Writer

OutputStreamWriter

FileWriter

**Hierarchy of writer stream classes**

**Using Streams**

Both the Character stream group and the Byte stream group contain parallel pairs of classes that perform the same kind of operation but for the different data type. The following table gives a list of some tasks and the character streams and byte streams that are available to implement them.

|  |  |  |
| --- | --- | --- |
| **Task** | **Character Stream Class** | **Byte Stream Class** |
| Performing input operations | Reader | InputStream |
| Buffering input | BufferedReader | BufferedInputStream |
| Keeping track of line numbers | LineNumberReader | LineNumberInputStream |
| Reading from an array | CharArrayReader | ByteArrayInputStream |
| Reading from files | FileReader | FileInputStream |
| Filtering the input | FilterReader | FilterInputStream |
| Reading from a string | StringReader | StringBufferInputStream |
| Reading primitive types |  | DataInputStream |
| Writing to a file | FileWriter | FileOutputStream |
| Printing values and objects | PrintWriter | PrintStream |

**Reading and writing files**

The java.io package includes a class known as File class that provides support for creating files and directories. The class includes several constructors for instantiating the File objects. This class contains several methods for supporting the operations such as creating a file, opening a file, deleting a file, renaming the file etc.,

**Example for writing and reading files :**

import java.io.\*;

public class FileTest

{

public static void main(String args[])

{

try

{

String str="It is my First file program..";

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

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

{

os.write( str.charAt(x)); // writes the bytes

}

os.close();

InputStream is = new FileInputStream("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 will create a file **test.txt** and will write given string in that file.. Same would be the output on the screen.

**Working with *File* class**

The **File** class is Java’s representation of a file or directory path name. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them. The File class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

It is an abstract representation of file and directory pathnames.

A pathname, whether abstract or in string form can be either absolute or relative. The parent of an abstract pathname may be obtained by invoking the getParent () method of this class.

First of all, we should create the File class object by passing the filename or directory name to it.

A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.

Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

**Creating a File Object**

A File object is created by passing in a String that represents the name of a file, or a String or another File object. For example,

File a = new File("d:\abc\Test.txt");

Defines an abstract file name for the geeks file in directory /usr/local/bin. This is an absolute abstract file name.

**Constructors**

**File(File parent, String child)** : Creates a new File instance from a parent abstract pathname and a child pathname string.

**File(String pathname) :** Creates a new File instance by converting the given pathname string into an abstract pathname.

**File(String parent, String child) :** Creates a new File instance from a parent pathname string and a child pathname string.

**File(URI uri) :** Creates a new File instance by converting the given file: URI into an abstract pathname.

**URL:** “**Uniform Resource Identifier”** is a sequence of characters used for identification of a particular resource. It enables for the interaction of the representation of the resource over the network using specific protocols.

**Methods**

**boolean canExecute():** Tests whether the application can execute the file denoted by this abstract pathname.

**boolean canRead() :** Tests whether the application can read the file denoted by this abstract pathname.

**boolean canWrite() :** Tests whether the application can modify the file denoted by this abstract pathname.

**int compareTo(File pathname)** : Compares two abstract pathnames lexicographically.

**boolean createNewFile() :** Atomically creates a new, empty file named by this abstract pathname .

**static File createTempFile(String prefix, String suffix)** : Creates an empty file in the default temporary-file directory.

**boolean delete() :** Deletes the file or directory denoted by this abstract pathname.

**boolean equals(Object obj) :** Tests this abstract pathname for equality with the given object.

**boolean exists() :** Tests whether the file or directory denoted by this abstract pathname exists.

**String getAbsolutePath() :** Returns the absolute pathname string of this abstract pathname.

**long getFreeSpace() :** Returns the number of unallocated bytes in the partition .

**String getName() :** Returns the name of the file or directory denoted by this abstract pathname.

**String getParent() :** Returns the pathname string of this abstract pathname’s parent.

**File getParentFile() :** Returns the abstract pathname of this abstract pathname’s parent.

**String getPath() :** Converts this abstract pathname into a pathname string.

**boolean isDirectory() :** Tests whether the file denoted by this pathname is a directory.

**boolean isFile() :** Tests whether the file denoted by this abstract pathname is a normal file.

**boolean isHidden()** : Tests whether the file named by this abstract pathname is a hidden file.

**long length()** : Returns the length of the file denoted by this abstract pathname.

**String[] list()** : Returns an array of strings naming the files and directories in the directory .

**File[] listFiles()** : Returns an array of abstract pathnames denoting the files in the directory.

**boolean mkdir()** : Creates the directory named by this abstract pathname.

**boolean renameTo(File dest)** : Renames the file denoted by this abstract pathname.

**boolean setExecutable(boolean executable)** : A convenience method to set the owner’s execute permission.

**boolean setReadable(boolean readable) :** A convenience method to set the owner’s read permission.

**boolean setReadable(boolean readable, boolean ownerOnly) :** Sets the owner’s or everybody’s read permission.

**boolean setReadOnly()** : Marks the file or directory named so that only read operations are allowed.

**boolean setWritable(boolean writable)** : A convenience method to set the owner’s write permission.

Example programs

1. check the file or directory exists or not ,if exists info about the file /directory

import java.io.\*;

import java.lang.\*;

public class FileExists {

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

{

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

System.out.println("enter a file name");

String fname=br.readLine();

//pass the filename or directory name to File object

File f = new File(fname);

//apply File class methods on File object

System.out.println("File name :"+f.getName());

System.out.println("Path: "+f.getPath());

System.out.println("Absolute path:" +f.getAbsolutePath());

System.out.println("Parent:"+f.getParent());

System.out.println("Exists :"+f.exists());

if(f.exists())

{

System.out.println("Is writeable:"+f.canWrite());

System.out.println("Is readable"+f.canRead());

System.out.println("Is a directory:"+f.isDirectory());

System.out.println("File Size in bytes "+f.length());

}

}

}

**# Program 2**

**//program to search for a directory and print the info about the directory**

import java.io.BufferedReader;

import java.io.File;

import java.io.IOException;

import java.io.InputStreamReader;

class Contents

{

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

//enter the path and dirname from keyboard

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

System.out.println("Enter dirpath:");

String dirpath=br.readLine();

System.out.println("Enter the dirname");

String dname=br.readLine();

//create File object with dirpath and dname

File f = new File(dirpath, dname);

//if directory exists,then

if(f.exists())

{

//get the contents into arr[]

//now arr[i] represent either a File or Directory

String arr[]=f.list();

//find no. of entries in the directory

int n=arr.length;

//displaying the entries

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

System.out.println(arr[i]);

//create File object with the entry and test

//if it is a file or directory

File f1=new File(arr[i]);

if(f1.isFile())

System.out.println(": is a file");

if(f1.isDirectory())

System.out.println(": is a directory");

}

System.out.println("No of entries in this directory "+n);

}

else

System.out.println("Directory not found");

}

}

//program to copy a source file into Another file

import java.io.\*;

public class FileCopy {

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

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

System.out.println("enter source file name to copy");

String fname=br.readLine();

FileInputStream fis = new FileInputStream(fname);

System.out.println("enter target file name ");

String fname1=br.readLine();

FileOutputStream fos = new FileOutputStream(fname1);

int b;

while ((b=fis.read()) != -1)

fos.write(b);

fis.close();

fos.close();

}

}

//PROGRAM TO DELETE A FILE

// Java program to delete a file

import java.io.\*;

import java.util.\*;

import java.nio.file.\*;

public class FileDelete

{

public static void main(String[] args)

{

Scanner s=new Scanner(System.in);

System.out.println("enter file name to delete : ");

String i=s.next();

try

{

Files.deleteIfExists(Paths.get(i));

}

catch(NoSuchFileException e)

{

System.out.println("No such file/directory exists");

}

catch(DirectoryNotEmptyException e)

{

System.out.println("Directory is not empty.");

}

catch(IOException e)

{

System.out.println("Invalid permissions.");

}

System.out.println("File deleted successfully");

}

}