**I/O Handling in Java**

**Streams**

Java programs perform I/O through streams. A *stream* 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 the **java.io** package.

**Byte Streams and Character Streams**

Java 2 defines two types of streams: byte and character. *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. The character-based streams simply provide a convenient and efficient means for handling characters.

**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 examples are : BufferedInputStream, BufferedOutputStream, FileInputStream, FileOutputStream, etc.To use the stream classes, you must import java.io. The abstract classes InputStream and OutputStream define several key methods that the other stream classes implement. Two of the most important are **read( )** 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, **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The examples are: BufferedReader, BufferedWriter, FileReader, FileWriter, etc. The abstract classes Reader and Writer define several key methods that the other stream classes implement. Two of the most important methods are **read( )** and **write( ),** which read and write characters of data, respectively. These methods are overridden by derived stream classes.

**Reading Console Input**

*Java does not have a generalized console input method that parallels the standard C function* ***scanf( )*** *or C++ input operators* ***cin****.*In Java, console input 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. **BuffereredReader** supports a buffered input stream. Its most commonly used constructor is shown here:

BufferedReader(Reader *inputReader*)

Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters. To obtain an **InputStreamReader** object that is linked to **System.in**, use the following constructor:

InputStreamReader(InputStream *inputStream*)

Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream*. Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

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

After this statement executes, **br** is a character-based stream that is linked to the

console through **System.in**.

**Reading Characters**

To read a character from a **BufferedReader**, use **read( )**. The version of **read( )** that we will be using is

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.\*;

class BRRead {

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

{

char c;

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

This output may look a little different from what you expected, because **System.in** is line buffered, by default. This means that no input is actually passed to the program until you press ENTER. As you can guess, this does not make **read( )** particularly valuable for interactive, console input.

**Reading Strings**

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

As you can see, it returns a **String** object. The following program demonstrates **BufferedReader** and the **readLine( )** method;the program reads and displays lines of text until you enter the word “stop”:

// Read a string from console using a BufferedReader.

import java.io.\*;

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 'stop' to quit.");

do {

str = br.readLine();

System.out.println(str);

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

}

}

**File Handling in Java**

**File**

Although most of the classes defined by java.io operate on streams, the File class does not. It deals directly with files and the file system. That is, the File class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself. A File object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies. Files are a primary source and destination for data within many programs.

File(String directoryPath)

File(String directoryPath, String filename)

File(File dirObj, String filename)

File(URI uriObj)

Here, directoryPath is the path name of the file, filename is the name of the file, dirObj is a File object that specifies a directory, and uriObj is a URI object that describes a file.

The following example creates three files: f1, f2, and f3. The first File object is constructed with a directory path as the only argument. The second includes two arguments—the path and the filename. The third includes the file path assigned to f1 and a filename; f3 refers to the same file as f2.

File f1 = new File("/");

File f2 = new File("/","autoexec.bat");

File f3 = new File(f1,"autoexec.bat");

Note: Java does the right thing with path separators between UNIX and Windows conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly. Remember, if you are using the Windows convention of a backslash character (\), you will need to use its escape sequence (\\) within a string. The Java convention is to use the UNIX- and URL-style forward slash for

path separators.

File defines many methods that obtain the standard properties of a File object. For example, getName( ) returns the name of the file, getParent( ) returns the name of the parent directory, and exists( ) returns true if the file exists, false if it does not. The File class, however, is not symmetrical. By this, we mean that there are many methods that allow you to examine the properties of a simple file object, but no corresponding function exists to change those attributes. The following example demonstrates several of the File methods:

// Demonstrate File.

import java.io.File;

class FileDemo {

static void p(String s) {

System.out.println(s);

}

public static void main(String args[]) {

File f1 = new File("/java/COPYRIGHT");

p("File Name: " + f1.getName());

p("Path: " + f1.getPath());

p("Abs Path: " + f1.getAbsolutePath());

p("Parent: " + f1.getParent());

p(f1.exists() ? "exists" : "does not exist");

p(f1.canWrite() ? "is writeable" : "is not writeable");

p(f1.canRead() ? "is readable" : "is not readable");

p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));

p(f1.isFile() ? "is normal file" : "might be a named pipe");

p(f1.isAbsolute() ? "is absolute" : "is not absolute");

p("File last modified: " + f1.lastModified());

p("File size: " + f1.length() + " Bytes");

}

}

When you run this program, you will see something similar to the following:

File Name: COPYRIGHT

Path: /java/COPYRIGHT

Abs Path: /java/COPYRIGHT

Parent: /java

exists

is writeable

is readable

is not a directory

is normal file

is absolute

File last modified: 812465204000

File size: 695 Bytes

Most of the File methods are self-explanatory. isFile( ) and isAbsolute( ) are not. isFile( ) returns true if called on a file and false if called on a directory. Also, isFile( ) returns false for some special files, such as device drivers and named pipes, so this method can be used to make sure the file will behave as a file. The isAbsolute( ) method returns true if the file has an absolute path and false if its path is relative. File also includes two useful utility methods. The first is renameTo( ), shown here:

boolean renameTo(File newName)

Here, the filename specified by newName becomes the new name of the invoking File object. It will return true upon success and false if the file cannot be renamed.

The second utility method is delete( ), which deletes the disk file represented by the path of the invoking File object. It is shown here:

boolean delete( )

You can also use delete( ) to delete a directory if the directory is empty. delete( ) returns true if it deletes the file and false if the file cannot be removed.

Here are some other File methods that you will find helpful. (They were added by Java 2.)

|  |  |
| --- | --- |
| **Method** | **Description** |
| void deleteOnExit( ) | Removes the file associated with the invoking object when the Java Virtual Machine terminates. |
| boolean isHidden( ) | Returns **true** if the invoking file is hidden. Returns **false** otherwise. |
| boolean setLastModified(long *millisec*) | Sets the time stamp on the invoking file to that specified by *millisec*, which is the number of milliseconds from January 1, 1970, Coordinated Universal Time (UTC). |
| boolean setReadOnly( ) | Sets the invoking file to read-only. |

**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. In this case, you can call list( ) on that object to extract the list of other files and directories inside. It has two forms. The first is shown here:

String[ ] list( )

The list of files is returned in an array of String objects. The program shown here illustrates how to use list( ) to examine the contents of a directory:

import java.io.File;

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");

}

}

}

Here is sample output from the program. (Of course, the output you see will be different, based on what is in the directory.)

Directory of /java

bin is a directory

lib is a directory

demo is a directory

COPYRIGHT is a file

README is a file

index.html is a file

include is a directory

src.zip is a file

.hotjava is a directory

src is a directory

**Creating 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.
* To create a directory for which no path exists, use the **mkdirs( )** method. It creates both a directory and all the parents of the directory.

**Reading and Writing Files using Byte Stream classes**

**FileInputStream**

The FileInputStream class creates an InputStream that you can use to read bytes from a file. Its two most common constructors are shown here:

FileInputStream(String filepath)

FileInputStream(File fileObj)

Either can throw a FileNotFoundException. Here, filepath is the full path name of a file, and fileObj is a File object that describes the file.

The following example creates two FileInputStreams that use the same disk file and each of the two constructors:

FileInputStream f0 = new FileInputStream("/autoexec.bat")

File f = new File("/autoexec.bat");

FileInputStream f1 = new FileInputStream(f);

Although the first constructor is probably more commonly used, the second allows us to closely examine the file using the File methods, before we attach it to an input stream. When a FileInputStream is created, it is also opened for reading.

FileInputStream overrides six of the methods in the abstract class InputStream. We are discussing the following methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| void close( ) | Closes the input source. Further read attempts will generate an **IOException**. |
| int read( ) | Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered. |
| int read(byte *buffer*[ ]) | Attempts to read up to *buffer*.*length* bytes into *buffer* and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered. |
| int read(byte *buffer*[ ], int *offset*,  int *numBytes*) | Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered. |

**Program to read a File using read( ) method with single byte :**

import java.io.\*;

class FileInputStreamDemo {

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

{

InputStream fin = new FileInputStream(args[0]);

int i;

do {

i = fin.read();

if(i != -1) System.out.print((char) i);

} while(i != -1);

fin.close();

}

}

**Program to read a chunk of bytes i.e. using buffer:**

import java.io.\*;

class FileInputStreamBufferDemo {

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

{

InputStream fin = new FileInputStream(args[0]);

int noBytesRead;

byte buffer[]= new byte[4];

do {

noBytesRead = fin.read(buffer);

if(noBytesRead != -1)

{

for(int i=0;i<noBytesRead;i++)

System.out.print((char) buffer[i]);

}

} while(noBytesRead != -1);

fin.close();

}

}

**Program to read a chunk of bytes i.e. using buffer with offset:**

The code will be same as above only read() method will be as follows:

noBytesRead = fin.read(buffer,0,3);

In this case only 3 bytes will be read from file and getting stored in buffer from index 0

If we write the following method the case will be same as above program:

noBytesRead = fin.read(buffer,0,4);

**Writing bytes into a File using FileOutputStream:**

**FileOutputStream**

**FileOutputStream** creates an **OutputStream** that you can use to write bytes to a file. Its most commonly used constructors are shown here:

FileOutputStream(String *filePath*)

FileOutputStream(File *fileObj*)

FileOutputStream(String *filePath*, boolean *append*)

FileOutputStream(File *fileObj*, boolean *append*)

They can throw a **FileNotFoundException** or a **SecurityException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, the file is opened in append mode. The fourth constructor was added by Java 2, version 1.4.

Creation of a **FileOutputStream** is not dependent on the file already existing. **FileOutputStream** will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an **IOException** will be thrown.



import java.io.\*;

class FileOutputStreamDemo {

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

String source = "Now is the time for all good men\n"

+ " to come to the aid of their country\n"

+ " and pay their due taxes.";

byte buf[] = source.getBytes();

OutputStream f0 = new FileOutputStream("file1.txt");

for (int i=0; i < buf.length; i += 1) {

f0.write(buf[i]);

}

f0.close();

OutputStream f1 = new FileOutputStream("file2.txt");

f1.write(buf);

f1.close();

**Reading and Writing Files using Character Stream classes**

FileReader

The FileReader class creates a Reader that you can use to read the contents of a file. Its two most commonly used constructors are shown here:

FileReader(String filePath)

FileReader(File fileObj)



import java.io.\*;

class FileReaderCharDemo {

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

FileReader fr = new FileReader(args[0]);

int ch;

while((ch = fr.read()) != -1) {

System.out.print((char)ch);

}

fr.close();

// Reading using character buffer

System.out.println("The output of reading using buffer is as follows::\n");

fr = new FileReader(args[0]);

char buffer[] = new char[4];

int numCharsRead;

while((numCharsRead = fr.read(buffer)) != -1)

{

for(int i=0;i<numCharsRead;i++)

System.out.print(buffer[i]);

}

fr.close();

}

}

The following example shows how to read lines from a file and print these to the standard output stream. It reads its own source file, which must be in the current directory

import java.io.\*;

class FileReaderStringDemo {

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

FileReader fr = new FileReader(args[0]);

BufferedReader br = new BufferedReader(fr);

String s;

while((s = br.readLine()) != null) {

System.out.println(s);

}

fr.close();

}

}

Writing data to the file using FileWriter class

FileWriter

**FileWriter** creates a **Writer** that you can use to write to a file. Its most commonly used constructors are shown here:

FileWriter(String *filePath*)

FileWriter(String *filePath,* boolean *append)*

FileWriter(File *fileObj*)

FileWriter(File *fileObj*, boolean *append*)

They can throw an **IOException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, then output is appended to the end of the file. The fourth constructor was added by Java 2, version 1.4.

Creation of a **FileWriter** is not dependent on the file already existing. **FileWriter** will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an **IOException** will be thrown.



import java.io.\*;

class FileWriterCharStreamDemo {

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

String source = "Now is the time for all good men\n"

+ " to come to the aid of their country\n"

+ " and pay their due taxes.";

char buffer[] = source.toCharArray();

FileWriter f0 = new FileWriter("file1.txt");

for (int i=0; i < buffer.length; i += 1) {

f0.write(buffer[i]);

}

f0.close();

FileWriter f1 = new FileWriter("file2.txt");

f1.write(buffer);

f1.close();

}

}