**Overview**

Stream is a general mechanism of I/O in java. Stream provides sequential access of data.

An input stream can be used by an application to read data. An output stream to write data.

A file, network connection, arrays can act as stream.

Streams are categorized as:

1. Byte Stream means Binary data
2. Character Stream means Textual data

Whenever we have to make program of I/O then we use java.io package or java.nio package.

In Java IO, there are some classes having words like Reader/Writer and Streams, so remember this,

“East or West, Reader and Writer for text.”

General rule is that, Reader and Writer are used for character based data (textual data) and streams are used for binary data.

In this tutorial We’ll learn how to use BufferedWriter, PrintWriter, FileOutputStream, DataOutputStream, RandomAccessFile, FileChannel and the Java 7 Files utility class.

1. **How to write data into file using BufferedWriter:**

Let’s start simple – and use BufferedWriter to write a String to a new file:

**import** java.io.BufferedWriter;

**import** java.io.File;

**import** java.io.FileWriter;

**import** java.io.IOException;

**public** **class** BufferedWriterDemo {

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

// Step #1. Create a file object.

File file = **new** File("D:\\demo\\abc.txt");

// Step #2. Create a file writer object with above file.

FileWriter fileWriter = **new** FileWriter(file);

// Step #3. Create a file object with above file writer.

BufferedWriter writer = **new** BufferedWriter(fileWriter);

// Step #4. Prepare data to be stored in above file.

String message = "Hello, this content will write on file";

// Step #5. Perform write operation.

writer.write(message);

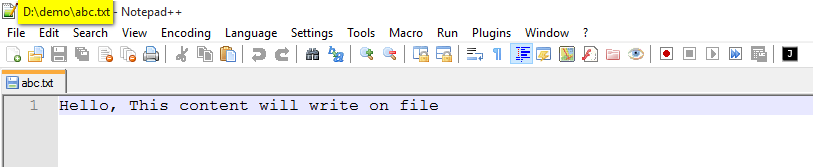
// Step #6. free the resources.

writer.close();

}

}

The output in the file will be:



Difference between FileWriter and BufferedWriter:

The FileWriter writes the characters one by one and the BufferedWriter first buffers it to the memory and writes it once.

In above program, if file is already existing, then whole content of a file will have removed and fresh content will be write on that file, means we lost existing data. So what if we want to keep old data and write some more data on same file? Here is the solution:

We can then append a String to the existing file.

**import** java.io.BufferedWriter;

**import** java.io.File;

**import** java.io.FileWriter;

**import** java.io.IOException;

**public** **class** BufferedWriterDemo2 {

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

// Step #1. Create a file object.

File file = **new** File("D:\\demo\\abc.txt");

// Step #2. Create a file writer object with above file.

FileWriter fileWriter = **new** FileWriter(file,**true**);

// Step #3. Create a file object with above file writer.

BufferedWriter writer = **new** BufferedWriter(fileWriter);

// Step #4. Prepare data to be stored in above file.

String message = "This is new Content";

// Step #5. Perform write operation.

writer.append(" ");

writer.append(message);

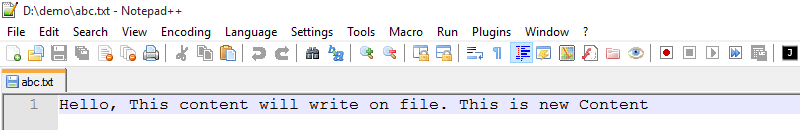
// Step #6. free the resources.

writer.close();

}

}

The file will then be:



Note: FileWriter fileWriter = **new** FileWriter(file,**true**);

We pass an extra boolean parameter, it tells to the FileWriter for append mode. True means append mode is on.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **How to write data into file using PrintWriter**

Next – let’s see how we can use a PrintWriter to write formatted text to a file:

**import** java.io.File;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.io.PrintWriter;

**public** **class** PrintWriterDemo {

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

// Step #1. Create a file object.

File file = **new** File("D:\\demo\\abc.txt");

// Step #2. Create a file writer object with above file.

FileWriter fileWriter = **new** FileWriter(file, **true**);

// Step #3. Create a file object with above file writer.

PrintWriter writer = **new** PrintWriter(fileWriter);

// Step #4. Perform write operation.

writer.println("This is JustJava");

writer.printf("My Name is %s and my age is %d yeras.", "Shadab", 30);

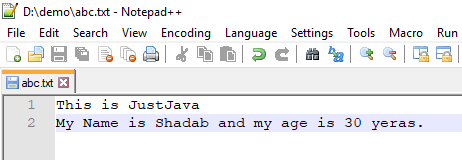
// Step #5. free the resources.

writer.close();

}

}

The resulting file will contain:



Difference between FileWriter and PrintWriter:

1. The main difference is that PrintWriter offers some additional methods for formatting like println and printf.
2. PrintWriter automatically invokes flush after every byte of data is written. In case of FileWriter, caller must take care of invoking flush.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **How to write data into file using FileOutputStream**

We can use FileOutputStream to write binary data to a file. The following code converts a String int bytes and writes the bytes to file using a FileOutputStream:

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**public** **class** FileOutputStreamDemo {

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

String str = "This is demo app";

FileOutputStream outputStream = **new** FileOutputStream("D:\\demo\\abc.txt");

**byte**[] strToBytes = str.getBytes();

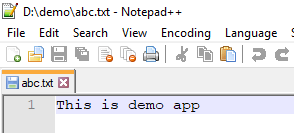
outputStream.write(strToBytes);

outputStream.close();

}

}

The output in the file will of course be:



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **How to write data into file using DataOutputStream:**

Java DataOutputStream class allows an application to write primitive Java data types to the output stream in a machine-independent way.

Next – let’s take a look at how we can use a DataOutputStream to write a String to file:

**import** java.io.DataOutputStream;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**public** **class** DataOutputStreamDemo {

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

FileOutputStream file = **new** FileOutputStream("D:\\demo\\abc.txt");

DataOutputStream data = **new** DataOutputStream(file);

data.writeInt(65);

data.flush();

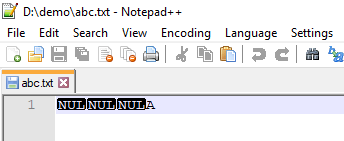
data.close();

System.***out***.println("Succcess...");

}

}

Output: Ooops this binary file not understandable by human,lol!!



DataOutputStream performs generally better because its much simpler. It can only write primitive types and Strings.

ObjectOutputStream can write any object type was well as primitives. It is less efficient but much easier to use if you want to send complex data.

1. **How to write data into file using ObjectOutputStream:**

ObjectOutputStream in Java can be used to convert an object to OutputStream. The process of converting object to stream is called [serialization in java](https://www.journaldev.com/2452/serialization-in-java).

Once an object is converted to Output Stream, it can be saved to file or database, send over the network or used in socket connections. So, we can use FileOutputStream to write Object to file.

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectOutputStream;

**import** java.io.Serializable;

**public** **class** ObjectOutputStreamDemo {

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

FileOutputStream file = **new** FileOutputStream("D:\\demo\\cars.dat");

ObjectOutputStream data = **new** ObjectOutputStream(file);

Car car1 = **new** Car();

car1.setName("BMW");

car1.setPrice(1000);

data.writeObject(car1);

data.flush();

data.close();

System.***out***.println("Succcess...");

}

}

**class** Car **implements** Serializable {

**private** String name;

**private** **float** price;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **float** getPrice() {

**return** price;

}

**public** **void** setPrice(**float** price) {

**this**.price = price;

}

/\*\*

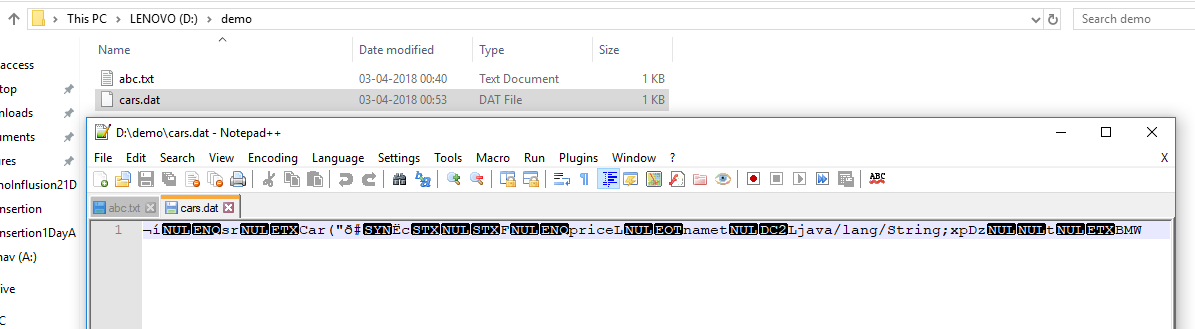
\*

\*/

**private** **static** **final** **long** ***serialVersionUID*** = 2892137946005556067L;

}

Output:



1. **How to write data into file using RandomAccessFile.**

Java also allows you to access the contents of a file in random order i.e. data items can be read and written in any order. Simply put – we need random access.

RandomAccessFile enable us to write at a specific position in the file given the offset – from the beginning of the file – in bytes. The following code writes an integer value with offset given from the beginning of the file:

**import** java.io.IOException;

**import** java.io.RandomAccessFile;

**public** **class** RandomAccessFileDemo {

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

RandomAccessFile writer = **new** RandomAccessFile("D:\\demo\\random.txt", "rw");

writer.seek(5);

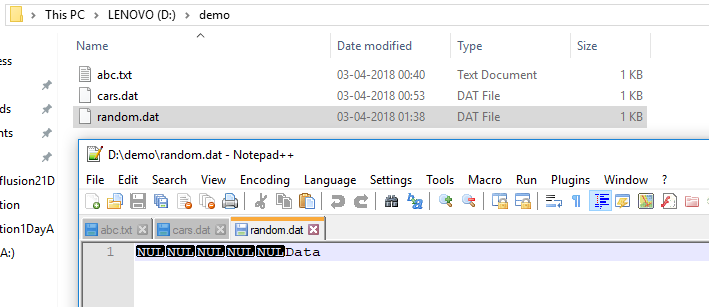
writer.write("Data".getBytes());

writer.close();

}

}

Output:



1. **How to write data into file using FileChannel.**

Channel provides an alternate way to read data from a file, it provides better performance than Input Stream or OutputStream. It can also be opened in blocking and non-blocking mode.  
  
If you are dealing with large files, FileChannel can be faster than standard IO. The following code write String to a file using FileChannel:

//ToDo: Example51’/o\

8. Write to file using Java 7

Java 7 introduces a new way of working with the filesystem, along with a new utility class – Files. Using the Files class, we can create, move, copy, delete files and directories as well; it also can be used to read and write to a file:

@Test

public void givenUsingJava7\_whenWritingToFile\_thenCorrect()

throws IOException {

String str = "Hello";

Path path = Paths.get(fileName);

byte[] strToBytes = str.getBytes();

Files.write(path, strToBytes);

String read = Files.readAllLines(path).get(0);

assertEquals(str, read);

}

9. Write to temporary file

Now, let’s try to write to temporary file. The following code creates a temporary file and writes a String to it:

@Test

public void whenWriteToTmpFile\_thenCorrect() throws IOException {

String toWrite = "Hello";

File tmpFile = File.createTempFile("test", ".tmp");

FileWriter writer = new FileWriter(tmpFile);

writer.write(toWrite);

writer.close();

BufferedReader reader = new BufferedReader(new FileReader(tmpFile));

assertEquals(toWrite, reader.readLine());

reader.close();

}

So, as you can see – it’s just the creation of the temporary file that is interesting and different – after that point, writing to the file is the same.

10. Lock File Before Writing

Finally, when writing to a file, you sometimes need to make extra sure that no one else is writing to that file at the same time. Basically – you need to be able to lock that file while writing.

Let’s make use of the FileChannel to try locking the file before writing to it:

@Test

public void whenTryToLockFile\_thenItShouldBeLocked()

throws IOException {

RandomAccessFile stream = new RandomAccessFile(fileName, "rw");

FileChannel channel = stream.getChannel();

FileLock lock = null;

try {

lock = channel.tryLock();

} catch (final OverlappingFileLockException e) {

stream.close();

channel.close();

}

stream.writeChars("test lock");

lock.release();

stream.close();

channel.close();

}

Note that if the file is already locked when we try to acquire the lock, an OverlappingFileLockException will be thrown.

11. Notes

After exploring so many methods of writing to a file, let’s discuss some important notes:

If we try to read from a file that doesn’t exist, a FileNotFoundException will be thrown

If we try to write to a file that doesn’t exist, the file will be created first and no exception will be thrown

It is very important to close the stream after using it, as it is not closed implicitly, to release any resources associated with it

In output stream, the close() method calls flush() before releasing the resources which forces any buffered bytes to be written to the stream

Looking at the common usage practices, we can see – for example – that PrintWriter is used to write formatted text; FileOutputStream to write binary data; DataOutputStream to write primitive data types; RandomAccessFile to write to a specific position; FileChannel to write faster in larger files. Some of the APIs of these classes do allow more, but this is a good place to start.

12. Conclusion

This article illustrates the many options of writing data to a File using Java.