**The Character Streams**

While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters. Since one of the main purposes of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters. In this section, several of the character I/O classes are discussed. As explained earlier, at the top of the character stream hierarchies are the **Reader** and **Writer** abstract classes.

**Java Reader**

The Java Reader class, java.io.Reader, is the base class for all Reader subclasses in the Java IO API. A Java Reader is like a [**Java InputStream**](https://jenkov.com/tutorials/java-io/inputstream.html) except that it is character based rather than byte based. In other words, a Java Reader is intended for reading text (characters), whereas an InputStream is intended for reading raw bytes. A Reader is typically connected to some source of data like a file, char array, network socket etc.

You will normally use a Reader subclass rather than a Reader directly. Java IO contains a lot of Reader subclasses. Here is a list of the Java Reader subclasses:

* **InputStreamReader**
* **CharArrayReader**
* **FileReader**
* **PipedReader**
* **BufferedReader**
* **FilterReader**
* **PushbackReader**
* **PrintWriter**

Methods of Reader.

|  |  |
| --- | --- |
| **Method** | **Description** |
| abstract void close( ) | Closes the input source. Further read attempts will generate an **IOException**. |
| void mark(int *numChars*) | Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read. |
| boolean markSupported( ) | Returns **true** if **mark( )**/**reset( )** are supported on this stream. |
| static Reader nullReader( ) | Returns an open, but null reader, which is a reader that contains no data. Thus, the reader is always at the end of the stream and no input can be obtained. The reader can, however, be closed. |
| int read( ) | Returns an integer representation of the next available character from the invoking input stream. –1 is returned when an attempt is made to read at the end of the stream. |
| int read(char[ ] *buffer*) | Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when an attempt is made to read at the end of the stream. |
| int read(CharBuffer *buffer*) | Attempts to read characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when an attempt is made to read at the end of the stream. |
| abstract int read(char[ ] *buffer*, int *offset*, int *numChars*) | Attempts to read up to *numChars* characters into *buffer* starting at *buffer*[*offset*], returning the number of characters successfully read. –1 is returned when an attempt is made to read at the end of the stream. |
| boolean ready( ) | Returns **true** if the next input request will not wait. Otherwise, it returns **false**. |
| void reset( ) | Resets the input pointer to the previously set mark. |
| long skip(long *numChars*) | Skips over *numChars* characters of input, returning the number of characters actually skipped. |
| long transferTo(Writer *writer*) | Copies the contents of the invoking reader to *writer*, returning the number of characters copied. |

The **read**() method of a Java Reader returns an int that contains the char value of the next character read. If the read() method returns -1, there is no more data to read in the Reader, and it can be closed. That is, -1 as int value, not -1 as byte or char value. There is a difference here!

Here is an example of reading all characters from a Java Reader:

Reader reader = new FileReader("/path/to/file/thefile.txt");

int theCharNum = reader.read();

while(theCharNum != -1) {

char theChar = (char) theCharNum;

System.out.print(theChar);

theCharNum = reader.read();

}

Notice how the code example first reads a single character from the Java Reader and checks if the char numerical value is equal to -1. If not, it processes that char and continues reading until -1 is returned from the Reader read() method.

The Java Reader class also has a read() method that takes a char array as parameter, as well as a start offset and length. The char array is where the read() method will read the characters into. The offset parameter is where in the char array the read() method should start reading into. The length parameter is how many characters the read() method should read into the char array from the offset and forward. Here is an example of reading an array of characters into a char array with a Java Reader:

Reader reader = new FileReader("/path/to/file/thefile.txt");

char[] theChars = new char[128];

int charsRead = reader.read(theChars, 0, theChars.length);

while(charsRead != -1) {

System.out.println(new String(theChars, 0, charsRead));

charsRead = reader.read(theChars, 0, theChars.length);

}

The **read(char[], offset, length)** method returns the number of characters read into the char array, or -1 if there are no more characters to read in the Reader, for instance, if the end of the file the Reader is connected to has been reached.

Reading an array of characters at a time is faster than reading a single character at a time from a Java Reader. The difference can easily be a factor 10 or more in performance increase, by reading an array of characters rather than reading a single character at a time.

The Java Reader class has a method named skip() which can be used to skip over a number of characters in the input that you do not want to read. You pass the number of characters to skip as parameter to the skip() method. Here is an example of skipping characters from a Java Reader :

long charsSkipped = reader.skip(24);

This example tells the Java Reader to skip over the next 24 characters in the Reader. The skip() method returns the actual number of characters skipped.

When you are finished reading characters from a Reader you should remember to close it. Closing an Reader is done by calling its close() method. Here is how closing an Reader looks: ***reader.close();’***

**Java Writer**

The Java Writer class (java.io.Writer) is the base class for all Writer subclasses in the Java IO API. A Writer is like an OutputStream except that it is character based rather than byte based. In other words, a Writer is intended for writing text, whereas an OutputStream is intended for writing raw bytes.

Writer Subclasses

You will normally use a Writer subclass rather than a Writer directly. Subclasses of Writer include OutputStreamWriter, CharArrayWriter, FileWriter, plus many others. Here is a list of the Java Writer subclasses:

* [**FileWriter**](https://jenkov.com/tutorials/java-io/filewriter.html)
* [**OutputStreamWriter**](https://jenkov.com/tutorials/java-io/outputstreamwriter.html)
* [**BufferedWriter**](https://jenkov.com/tutorials/java-io/bufferedwriter.html)
* [**PipedWriter**](https://jenkov.com/tutorials/java-io/pipedwriter.html)
* [**CharArrayWriter**](https://jenkov.com/tutorials/java-io/chararraywriter.html)
* [**FilterWriter**](https://jenkov.com/tutorials/java-io/filterwriter.html)
* [**StringWriter**](https://jenkov.com/tutorials/java-io/stringwriter.html)
* [**PrintWriter**](https://jenkov.com/tutorials/java-io/printwriter.html)

Methods of Writer.

|  |  |
| --- | --- |
| **Method** | **Description** |
| Writer append(char *ch*) | Appends *ch* to the end of the invoking output stream. Returns a reference to the invoking stream. |
| Writer append(CharSequence *chars*) | Appends *chars* to the end of the invoking output stream. Returns a reference to the invoking stream. |
| Writer append(CharSequence *chars*, int *begin*, int *end*) | Appends the subrange of *chars* specified by *begin* and *end*–1 to the end of the invoking output stream. Returns a reference to the invoking stream. |
| abstract void close( ) | Closes the output stream. Further write attempts will generate an **IOException**. |
| abstract void flush( ) | Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers. |
| static Writer nullWriter( ) | Returns an open, but null writer, which is a writer to which no output is actually written. Thus, its output methods can be called but don’t actually produce output. The writer can, however, be closed. |
| void write(int *ch*) | Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write** with an expression without having to cast it back to **char**. However, only the low-order 16 bits are written. |
| void write(char[ ] *buffer*) | Writes a complete array of characters to the invoking output stream. |
| abstract void write(char[ ] *buffer*, int *offset*, int *numChars*) | Writes a subrange of *numChars* characters from the array *buffer*, beginning at *buffer*[*offset*] to the invoking output stream. |
| void write(String *str*) | Writes *str* to the invoking output stream. |
| void write(String *str*, int *offset*, int *numChars*) | Writes a subrange of *numChars* characters from the string *str*, beginning at the specified *offset*. |

**Write( int ):** The Java Writer write(int) method writes the lower 16 bit of the int to the destination the Writer is connected to, as a single character. Here is an example of writing a single character to a Java Writer:

Writer writer = new FileWriter("data/output.txt");

writer.write('A');

**write(char[] ):** The Java Writer also has a write(char[]) method which can write an array of characters to the destination the Writer is connected to. The write(char[]) method returns the number of characters actually written to the Writer. Here is an example of writing an array of chars to a Java Writer:

Writer writer = new FileWriter("data/output.txt");

char[] chars = new char[]{'A','B','C','D','E'};

writer.write(chars);}

**Java FileReader:**

The Java FileReader class, java.io.FileReader makes it possible to read the contents of a file as a stream of characters. It works much like the [**FileInputStream**](https://jenkov.com/tutorials/java-io/fileinputstream.html) except the FileInputStream reads bytes, whereas the FileReader reads characters. The FileReader is intended to read text, in other words. One character may correspond to one or more bytes depending on the character encoding scheme.

The Java FileReader is a subclass of the [**Java Reader**](https://jenkov.com/tutorials/java-io/reader.html) class, so it has many of the same methods.

Reader fileReader = new FileReader("c:\\data\\input-text.txt");

int data = fileReader.read();

while(data != -1) {

//do something with data...

doSomethingWithData(data);

data = fileReader.read();

}

fileReader.close();



Methods of FileReader

* **read()**

The **read**() method of the Java FileReader returns an int which contains the char value of the character read. If the **read**() method returns -1, there is no more data to read in the FileReader, and it can be closed. That is, -1 as int value, not -1 as byte value. There is a difference here! Here is an example of reading characters from a Java FileReader until there are no more characters to read:

FileReader fileReader = new FileReader("c:\\data\\input-text.txt");  
 int **data** = fileReader.read();

while(**data**!=-1)  
 {  
 **data** = fileReader.read();  
 }

* **read(Char[])**

The Java FileReader also has a method that can read an array of characters instead of reading one character at a time. Here is an example of reading multiple characters into a char array:

FileReader fileReader = **new** FileReader("c:\\data\\input-text.txt");  
  
char[] destination = **new** char[1024];  
  
int charsRead = fileReader.read(destination, 0, destination.length);

This **read**(char[]) method takes the char array to read the characters into as first parameter, the offset into the array from which the characters should be written, and the maximum length of characters to write. The **read**(char[]) method returns the number of characters that was actually read into the char array. If there are less characters in the file than the maximum number of characters you specified, then there will be less than maximum number of characters read. Otherwise the **read**(char[]) method will attempt to read maximum number of characters into the array.

Closing a FileReader. fileReader.close();

// try-with-resources.

try(FileReader fileReader =  
 **new** FileReader("c:\\data\\text.txt")){  
  
 **int** data = fileReader.read();// int value is the no of characters copied.  
 while(data != -1) {  
 System.out.print((**char**) data));  
 data = fileReader.read();  
 }  
}

**Java FileWriter**

The Java FileWriter class, java.io.FileWriter, makes it possible to write characters to a file. In that respect the Java FileWriter works much like the [**FileOutputStream**](https://jenkov.com/tutorials/java-io/fileoutputstream.html) except that a FileOutputStream is byte based, whereas a FileWriter is character based. The FileWriter is intended to write text, in other words. One character may correspond to one or more bytes, depending on the character encoding scheme in use. The Java FileWriter class is a subclass of the [**Java Writer**](https://jenkov.com/tutorials/java-io/writer.html) class, by the way.

Example:

Writer fileWriter = new FileWriter("data\\filewriter.txt");  
fileWriter.write("data 1");  
fileWriter.write("data 2");  
fileWriter.write("data 3");  
fileWriter.close();

Methods of FileWriter

* ***write(int):*** The Java FileWriter **write**(int) method writes the lower 16 bit of the int to the destination the FileWriter is connected to, as a single character. Here is an example of writing a single character to a Java FileWriter:

FileWriter fileWriter = new FileWriter("data/output.txt");

fileWriter.write('A');

* ***write(char[]):*** The Java FileWriter also has a write(char[]) method which can write an array of characters to the destination the FileWriter is connected to. The **write**(char[]) method returns the number of characters actually written to the FileWriter. Here is an example of writing an array of chars to a Java FileWriter:

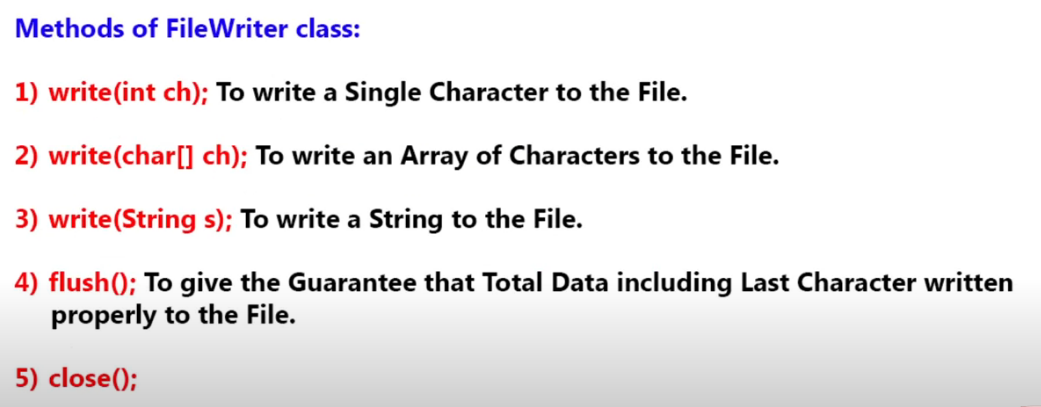
FileWriter fileWriter = new FileWriter("data/output.txt");

char[] chars = new char[]{'A','B','C','D','E'};

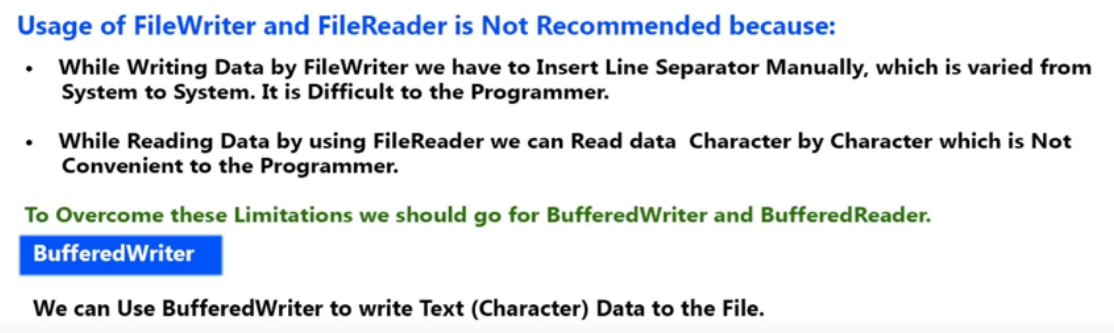
fileWriter.write(chars);

* ***flush():*** The Java FileWriter's flush() method flushes all data written to the FileWriter to the underlying file. The data might be buffered in OS memory somewhere, even if your Java code has written it to the FileWriter. By calling flush() you can assure that any buffered data will be flushed (written) to disk. Here is an example of flushing data written to a Java FileWriter by calling its **flush**() method:

fileWriter.flush();



**Limitations of FileReader and FileWriter**



**Buffered Writer**

The Java BufferedWriter class, java.io.BufferedWriter, provides buffering to Writer instances. Buffering can speed up IO quite a bit. Rather than writing one character at a time to the network or disk, the BufferedWriter writes a larger block at a time. This is typically much faster, especially for disk access and larger data amounts.

Constructors:

BufferedWriter bw = new BufferedWriter( Writer writerObj )

BufferedWriter bw = new BufferedWriter( Writer writerObj , int bufferSize)

Methods of Buffered Writer:

* ***write(int):*** The Java BufferedWriter write(int) method writes the lower 16 bit of the int to its internal buffer, as a single character. Here is an example of writing a single character to a Java BufferedWriter:

bufferedWriter.write('A'); // bufferedWriter.write(97);

* ***write(char[]):*** The Java BufferedWriter also has a write(char[]) method which can write an array of characters to its internal buffer. The write(char[]) method returns the number of characters actually written to the Writer. Here is an example of writing an array of chars to a Java Writer:

char[] chars = new char[]{'A','B','C','D','E'};

bufferedWriter.write(chars);

* ***write(String):*** Is used to write string to the internal buffer.
* ***flush():*** The Java BufferedWriter's flush() method flushes all data written to the BufferedWriter to the underlying data destination. By calling **flush**() you can assure that any buffered data will be flushed (written) to disk (or network, or whatever else the destination of your BufferedWriter has). Here is an example of flushing data written to a Java BufferedWriter by calling its **flush**() method:

bufferedWriter.flush();

* ***close():***When you are finished writing characters to the Java BufferedWriter you should remember to close it. Closing a BufferedWriter will also close the Writer instance to which the BufferedWriter is writing.

bufferedWriter.flush();

* ***newline():*** To insert a line separator.

Buffered Write Example:

package com.skg.file;

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

public class BufferWriterDemo {

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

BufferedWriter bufferedWriterObj = new BufferedWriter(new FileWriter("abc.txt"));

bufferedWriterObj.write(100);

bufferedWriterObj.newLine();// Adding a new Line

char[] ch = { 'd', 'r', 'y' };

bufferedWriterObj.write(ch);// Writing array of characters

bufferedWriterObj.newLine();// Adding a new Line

bufferedWriterObj.write("My name is khan !!"); // writing a string

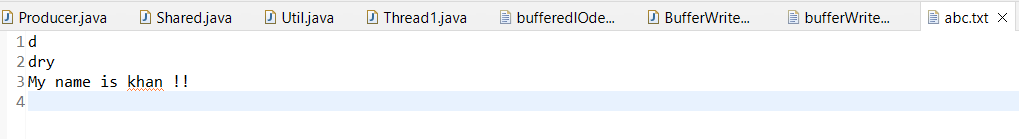
bufferedWriterObj.newLine();// Adding a new Line

bufferedWriterObj.flush(); // To make sure everything is written to file

bufferedWriterObj.close();

}

}



**Buffered Reader:**

The Java BufferedReader class, java.io.BufferedReader, provides buffering for your Java Reader instances. Buffering can speed up IO quite a bit. Rather than read one character at a time from the underlying Reader, the Java BufferedReader reads a larger block (array) at a time. This is typically much faster, especially for disk access and larger data amounts.

The Java BufferedReader is similar to the [**BufferedInputStream**](https://jenkov.com/tutorials/java-io/bufferedinputstream.html) but they are not exactly the same. The main difference between BufferedReader and BufferedInputStream is that BufferedReader reads characters (text), whereas the BufferedInputStream reads raw bytes.

The Java BufferedReader class is a subclass of the [**Java Reader**](https://jenkov.com/tutorials/java-io/reader.html) class, so you can use a BufferedReader anywhere a Reader is required.

BufferedReader bufferedReader = new BufferedReader(new FileReader("c:\\data\\input-file.txt"));

BufferedReader Buffer size:

int bufferSize = 8 \* 1024;

BufferedReader bufferedReader = new BufferedReader(

new FileReader("c:\\data\\input-file.txt"),

bufferSize

);

Except for adding buffering to Reader instances, a Java BufferedReader behaves pretty much like a Reader. The BufferedReader has one extra method though, the **readLine**() method. This method can be handy if you need to read input one line at a time. Here is a BufferedReader **readLine**() example:

String line = bufferedReader.readLine();

Reading character from a BufferedReader

Reader reader =

new BufferedReader(

new FileReader("/path/to/file/thefile.txt"));

int theCharNum = reader.read();

while(theCharNum != -1) {

char theChar = (char) theCharNum;

System.out.print(theChar);

theCharNum = reader.read();

}

As mentioned earlier, the BufferedReader will actually read an **array of characters from the underlying Reader and return these characters one by one**, rather than forwarding every **read**() call to the underlying Reader. When all characters in the internal buffer have been read, the BufferedReader attempts to refill the buffer again, until no more characters can be read from the underlying Reader.

Reading Array of character from a BufferedReader

Reader reader =

new BufferedReader(

new FileReader("/path/to/file/thefile.txt"));

char[] theChars = new char[128];

int charsRead = reader.read(theChars, 0, theChars.length);

while(charsRead != -1) {

System.out.println(new String(theChars, 0, charsRead));

charsRead = reader.read(theChars, 0, theChars.length);

}

The read(char[], offset, length) method returns the number of characters read into the char array, or -1 if there are no more characters to read in the BufferedReader, for instance if the end of the file the BufferedReader is connected to has been reached.

Reading a line from a buffered Reader

BufferedReader bufferedReader =

new BufferedReader(

new FileReader("/path/to/file/thefile.txt"));

String line = bufferedReader.readLine();

while(line != null) {

System.out.println(line);

line = bufferedReader.readLine();

}

The readLine() method returns a String. If there are no more lines to read from the BufferedReader, the readLine() method returns null.

**PrintWriter**

The Java PrintWriter class (java.io.PrintWriter) enables you to write formatted data to an underlying Writer. For instance, writing int, long and other primitive data formatted as text, rather than as their byte values.

The Java PrintWriter is useful if you are generating reports (or similar) where you have to mix text and numbers. The PrintWriter class has all the same methods as the **PrintStream** except for the methods to write raw bytes. Being a Writer subclass the PrintWriter is intended to write text.

It is the Most Enhanced Writer to Write Character Data to the File. The Main Advantage of PrintWriter is we can write any Type of Primitive Type Data Directly to the File.

Constructors:

1)PrintWriter pw=new PrintWriter(String filename);

2)PrintWriter pw=new PrintWriter(Filef);

3)PrintWriter pw=new PrintWriter(Writerw);

**PrintWriter** has several constructors. The following have been supplied by **PrintWriter** from the start:

PrintWriter(OutputStream outputStream)

PrintWriter(OutputStream outputStream, boolean autoFlushingOn)

PrintWriter(Writer outputStream)

PrintWriter(Writer outputStream, boolean autoFlushingOn)

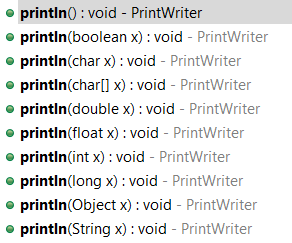
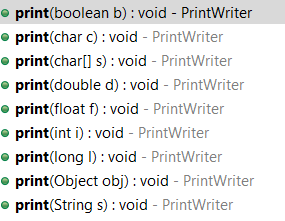
The next set of constructors gives you an easy way to construct a **PrintWriter** that writes its output to a file.

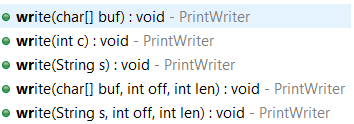
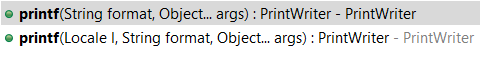
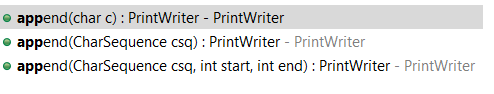
PrintWriter(File outputFile) throws FileNotFoundException

PrintWriter(File outputFile, String charSet) throws FileNotFoundException, UnsupportedEncodingException

PrintWriter(String outputFileName) throws FileNotFoundException

PrintWriter(String outputFileName, String charSet) throws FileNotFoundException, UnsupportedEncodingException

Example:

import java.io.IOException;

import java.io.PrintWriter;

public class PrintWriterDemo {

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

PrintWriter pw = new PrintWriter("printWriterDemo1.txt");

pw.write(100);

pw.println(100);

pw.println(true);

pw.println('c');

pw.println("durga");

pw.append((char) 101);

pw.flush();

pw.close();

}

}

Conclusion1:

* The Most Enhanced Writer to Write Character Data to the File is PrintWriter whereas
* The Most Enhanced Reader to Read Character Data from the File is Buffered Reader.

Conclusion2:

In General we can Use Readers and Writers to Handle Character Data(Text Data).Where as we can Use Streams to Handle Binary Data.

We can Use FileInputStream to Read Binary Data from the File and we can, use FileOutputStream to Write Binary Data to the File(Like Images,Video Files,Audio

Files,Etc ...).