CHAPTER 10

Character streams

In this and the following two chapters, we will discuss the character streams, which are streams that parallel the byte streams, but are oriented around the transmission of character-based textual data.

10.1 Overview

With the basic byte-oriented streams, support for the communication of textual data was fairly limited. Other than some crude support from the data streams, all text communications were assumed to be in 8-bit ASCII (ISO Latin 1). This was clearly incommensurable with Java's cross-platform nature and the 16-bit Unicode character set supported by the language.

To address this deficiency, a parallel set of character-oriented stream classes was introduced that allows transport of true 16-bit Unicode character data (figure 10.1). These streams address not only the issue of character transport but also the issues of character-encoding conversion and efficiency.

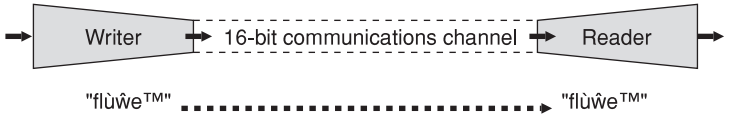

Figure 10.1 The character streams

The standard character streams mirror many of the byte-oriented stream classes, including filters, buffers, and file streams, all derived from the superclasses `Reader` and `Writer`. In addition, two bridge classes are defined, `OutputStreamWriter` and `InputStreamReader`, that bridge between byte streams and character streams. These two classes incorporate function to convert from characters to bytes and vice versa according to a specified encoding. This allows an ASCII data source to be easily converted to a Unicode character source, and similarly allows Unicode data to be easily written to a file according to the local character encoding, whether it be 8-bit dingbats, UTF-8, or 16-bit little-endian Unicode (figure 10.2).

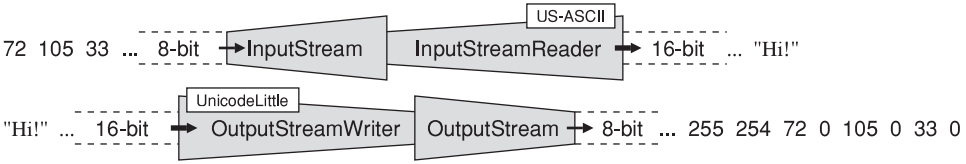

Figure 10.2 Encoding conversion

10.1.1 Correspondences

Table 10.1 lists the character streams that correspond to the various byte streams. There are no character stream equivalents for the byte streams that are concerned with actual binary data; similarly, the byte streams that were used to process text data have now been replaced by more appropriate character streams.

Table 10.1 Character- and byte-stream correspondences

byte streams	character streams
`OutputStream`	`Writer`
`InputStream`	`Reader`
`FileOutputStream`	`FileWriter`
`FileInputStream`	`FileReader`
`ByteArrayOutputStream`	`CharArrayWriter`
`ByteArrayInputStream`	`CharArrayReader`
—	`StringWriter`
`StringBufferInputStream`	`StringReader`
`PipedOutputStream`	`PipedWriter`
`PipedInputStream`	`PipedReader`
`FilterOutputStream`	`FilterWriter`
`FilterInputStream`	`FilterReader`
`BufferedOutputStream`	`BufferedWriter`
`BufferedInputStream`	`BufferedReader`
`PushbackInputStream`	`PushbackReader`
`LineNumberInputStream`	`LineNumberReader`
`PrintStream`	`PrintWriter`
`DataOutputStream`	—
`DataInputStream`	—
`ObjectOutputStream`	—
`ObjectInputStream`	—
`SequenceInputStream`	—
—	`OutputStreamWriter`
—	`InputStreamReader`

10.1.2 Deprecations

Table 10.2 lists the text-oriented methods and classes of the byte streams that have been deprecated by new features of the character stream classes. Note that for compatibility with common legacy systems, the `writeBytes()` method of class `DataOutputStream` remains.

Table 10.2 Deprecated classes and methods

deprecated class/method	replacement
`LineNumberInputStream`	`LineNumberReader`
`PrintStream`	`PrintWriter`
`StringBufferInputStream`	`StringReader`
`DataInputStream.readLine()`	`BufferedReader.readLine()`

10.2 Character encoding

When bridging between a character stream and a byte stream, it is necessary to specify the character encoding used by the byte stream; that is, what characters are represented by each byte or group of bytes (table 10.3). The name of the byte encoding is specified as a `String` that is passed to the constructor of the bridging `OutputStreamWriter` or `InputStreamReader`.

Table 10.3 Some example character encodings

encoding	char	bytes
US-ASCII	!	33
IBM-EBCDIC	!	90
ISO Latin	é	232
ISO Latin 2	č	232
UTF-8	é	195 168

Table 10.4 lists several of the supported encodings; appendix B lists more. The text-oriented methods of the byte streams are equivalent to latin1, which is also known as ISO Latin 1 or ISO 8859-1; this is an 8-bit encoding that matches exactly the first 256 characters of Unicode. Character-encoding names are, in general, case sensitive so you should use the encoding name exactly as it appears in these tables.

Table 10.4 Supported encodings

name	encoding
latin1	ISO 8859-1 (Europe, Latin America, Caribbean, Canada, Africa)
latin2	ISO 8859-2 (Eastern Europe)
latin3	ISO 8859-3 (SE Europe: Esperanto, Maltese, etc.)
latin4	ISO 8859-4 (Scandinavia)
cyrillic	ISO 8859-5 (Cyrillic)
arabic	ISO 8859-6 (Arabic)
greek	ISO 8859-7 (Greek)
hebrew	ISO 8859-8 (Hebrew)
latin5	ISO 8859-9 (ISO 8859-1 with Turkish)
ASCII	7-bit ASCII
Unicode	platform-default marked 16-bit Unicode
UnicodeBig	big-endian marked 16-bit Unicode
UnicodeBigUnmarked	big-endian unmarked 16-bit Unicode
UnicodeLittle	little-endian marked 16-bit Unicode
UnicodeLittleUnmarked	little-endian unmarked 16-bit Unicode
UTF8	Unicode transmission format

If latin1 encoding is used to bridge a character stream to a byte stream then characters 0–255 are transmitted unaltered as a single byte, and all other characters are replaced by the character ?. Similarly, if latin1 is used to bridge a byte stream to a character stream, then every byte that is read is converted directly into a Unicode character.

Alternatively, if UTF8 is used, then during writing, a character is converted into between one and three bytes and during reading, between one and three bytes are read and converted into a 16-bit Unicode character.

10.2.1 Default encoding

All character-byte conversion operations offer a default mode where no character encoding is specified. The actual encoding that is used is determined from the system property `file.encoding`; this is the encoding used to store files on the local platform. If that property is not set, a fallback of ISO Latin 1 is used. In Western locales, under most UNIX systems, the platform-default character encoding is ISO Latin 1 (latin1). Under Windows, as of the latest release of the JDK, it is Windows Latin 1 (Cp1252).

10.2.2 ASCII encoding

ASCII conversion is provided by the encoding named ASCII. Using this converter, the standard 7-bit U.S. ASCII character set maps exactly to Unicode characters 0–127. All other Unicode characters are mapped to ASCII character ? (code 63).

10.2.3 ISO encoding

There are ten supported ISO encodings, from ISO 8859-1 through ISO 8859-9 and ISO 8859-15. These are a standard set of international 8-bit character encodings; the characters from 0–127 are the usual U.S. ASCII characters and the remaining 128 are either control codes, accented characters, or other international characters.

10.2.4 Unicode encoding

There are several Unicode encodings that directly translate between 16-bit characters and two 8-bit bytes (table 10.5). Big- or little-endian conversion can be specified, along with optional *marking*. If the encoding Unicode is specified, then the platform-default endianness is used with marking.

Table 10.5 Unicode encodings

encoding	string	bytes
UnicodeBig	Hë!	254 255 0 72 0 235 0 33
UnicodeSmall	Hë!	255 254 72 0 235 0 33 0
UnicodeBigUnmarked	Hë!	0 72 0 235 0 33
UTF8	Hë!	72 195 171 33

Marking means that a two-byte marker will be written initially, that specifies the endianness of encoding used. Marking is used by the encodings Unicode, UnicodeBig, and UnicodeLittle. To read from such a stream, you can simply use the encoding Unicode and the endianness of the stream can be determined automatically from the marker.

Alternatively, a stream can be written with the encoding UnicodeBigUnmarked or UnicodeLittleUnmarked and no marker will be written. It should then be read with the appropriate UnicodeBig or UnicodeLittle encoding because Unicode cannot automatically determine the encoding used.

The UTF8 encoding is identical to that used by `DataOutputStream` and `DataInputStream` except that the character zero is encoded as per the standard, with a single zero byte.

10.2.5 Network character sets

It is extremely important when using the character streams to communicate with network services, to verify that you are using a valid character encoding *and* valid line terminators. If you fail to verify this, then your application may appear to be correct—it will send valid data—but it will fail to communicate successfully with the remote ser-

vice. More problematically, this problem may only manifest itself when communicating with certain implementations of a network service, or when certain data are transferred.

Gradually, with the introduction of MIME typing to Web and email data, character encodings are being explicitly specified along with data being transported. To communicate with any non-MIME-typed network service, however, you must verify that you are using the character encoding that is specified by the network protocol in use. For most services, the 8-bit character set ISO Latin 1 (latin1) is recognized as the standard, but you should always consult the appropriate documentation before assuming this. There are still a great number of legacy 7-bit protocols or services that will only accept U.S. ASCII (ASCII) data.

Use of the platform-default encoding (i.e., not explicitly specifying a character encoding) is incorrect because your application will behave differently on a UNIX machine than on a Windows machine, where ISO Latin 1 is not the default.

Similarly, use of the platform-default line terminator (i.e., use of any println() methods) is incorrect. Under UNIX, the line terminator is simply LF (ASCII character 10; Java character `\n`). Under Windows, the line terminator is CR-LF (ASCII characters 13 and 10; Java characters `\r\n`). Under MacOS, the line terminator is just CR (ASCII character 13; Java character `\r`). Always consult the appropriate protocol documentation to determine what line terminator is expected by the remote service that you will be communicating with, and always explicitly specify this terminator in calls to `print()`, and so forth.

10.3 JDK 1.1 Class Writer

`Writer` is the superclass of all character output streams. It provides similar methods to `OutputStream`, but oriented around writing characters (figure 10.3).

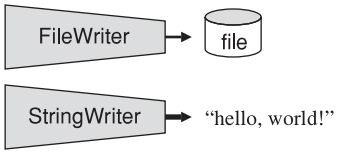

Figure 10.3 Some writers

10.3.1 Constructors

There are two constructors provided by this class. These constructors are `protected`. As with the byte-stream classes, you cannot create just a `Writer`; you must create some subclass.

protected Writer() This constructor creates a `Writer` that uses itself as a lock object for synchronization purposes. This constructor is typically used by *sink* `Writer` classes, that is nonfilters, such as `StringWriter` or `CharArrayWriter`.

protected Writer(Object lock) This constructor creates a `Writer` that uses the specified `Object`, `lock`, as a lock for synchronization purposes. `FilterWriter`s usually

pass their attached stream to this constructor for synchronization purposes. This synchronization mechanism is discussed in detail in the following discussion of the `lock` variable.

10.3.2 Methods

The `Writer` class provides methods to write characters either individually, from an array, or as part of a `String`, in addition to the `flush()` and `close()` methods.

Note that for efficiency, unlike `OutputStream`, the `Writer` class is oriented around writing arrays rather than individual characters. Thus, a subclass need only implement the character-subarray `write()` method, `flush()` and `close()`. Only where efficiency is at an absolute premium need the other `write()` methods be overridden.

void write(int c) throws IOException This method writes the character `c` to the communications channel represented by this stream. The argument is of type `int` but only the bottom 16 bits are actually written. The default implementation of this method creates a single-character array and writes this with the character-subarray `write()` method.

void write(char cbuf[]) throws IOException This method writes the entire array of characters `cbuf` to this stream. The default implementation of this method is to call the character-subarray `write()` method.

abstract void write(char cbuf[], int off, int len) throws IOException This method writes `len` characters from array `cbuf` to the attached stream, starting from index `off`. This method is `abstract` because it must be implemented by a subclass that is attached to an actual communications channel, such as a file or another stream.

void write(String str) throws IOException This method writes the specified `String`, `str`, to this stream. The default implementation of this method is to call the following substring `write()` method.

void write(String str, int off, int len) throws IOException This method writes `len` characters of the `String str` to this stream, starting from offset `off`. The default implementation of this method extracts the substring into a character array and then calls the character-subarray `write()` method.

abstract void flush() throws IOException This method flushes any buffers that the `Writer` may have; that is, it forces any buffered data to be written. This only flushes Java-specific buffers such as are provided by a buffered stream filter; it does not flush any OS buffers.

abstract void close() throws IOException This method closes the attached communications channel. If a `Writer` implements internal buffering, it is appropriate for it to flush its buffers before closing.

10.3.3 Variables

protected Object lock This variable should be used for synchronization by any `Writer` methods that require it. It is more efficient to use this common lock object than to declare synchronized methods.

The reasoning behind the use of this synchronization mechanism is that if a stream filter requires synchronization and it makes a call on an attached stream that also requires synchronization, then ordinarily the run time will obtain two semaphores: one on the stream filter and one on the attached stream. Using the new mechanism, both calls will synchronize on the same object so only one semaphore will be obtained. This efficiency obviously does not extend perfectly to a chain of several filters. However, long chains of character filters are fairly rare.

10.3.4 IOException

If a problem is encountered with the underlying communications channel, any of the methods of `Writer` may throw an exception of type `IOException` or a subclass.

10.4 JDK 1.1 Class Reader

`Reader` is the superclass of all character input streams. It provides similar methods to `InputStream`, but oriented around reading characters (figure 10.4).

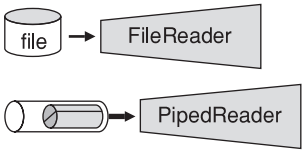

Figure 10.4 Some readers

10.4.1 Constructors

As with the `Writer` class, an optional lock object can be specified in the `Reader` constructor. This is used by subclasses for synchronization purposes.

protected Reader() This creates a `Reader` that uses itself as a lock object. This is usually used by source character streams; that is, nonfilters such as `StringReader`.

protected Reader(Object lock) This creates a `Reader` that uses the specified `Object`, `lock`, as a lock.

10.4.2 Methods

The `Reader` methods parallel those of the `InputStream` class, but the `available()` method has been replaced by a `ready()` method.

int read() throws IOException This method reads a single character from the communications channel represented by this stream and returns this character, or `-1` if the end of file is reached. This method will block if necessary until data are available. The default implementation of this method is to call the character-subarray `read()` method for a unit character-array.

int read(char cbuf[]) throws IOException This method reads as many characters into the array `cbuf` as possible up to the size of the array, returning the number of characters read or `-1` if the end of file is reached before any characters are read. This method will return immediately if some characters are available to read without blocking, and will otherwise block until some become available. The default implementation of this method is to call the character-subarray `read()` method for the whole array.

abstract int read(char cbuf[], int off, int len) throws IOException This method reads as many characters as possible into the subarray of `cbuf` of length `len`, starting from index `off`, returning the number of characters read, or `-1` if the end of file is reached before any characters are read.

This method will fail to read the requested number of characters if either the EOF is reached or some characters could be read without blocking, and attempting to read any more would block. If no characters can be read without blocking, the method will block until characters become available or the EOF is signaled.

This method is `abstract` because it must be implemented by a subclass that is attached to an actual communications channel.

long skip(long n) throws IOException This method attempts to skip the specified number of characters, returning the number successfully skipped. If no characters can be immediately skipped, then the method will block; otherwise it will immediately skip as many as it can, up to the requested number. Upon reaching the EOF, this method returns `0`. The default implementation simply reads and discards the specified number of characters.

boolean ready() throws IOException This method returns whether the stream has data available to be read immediately without blocking. The character streams do not have an `InputStream`-style `available()` method that indicates the number of characters available to read because this is not possible in the presence of nonuniform byte-encodings such as UTF-8. If this method returns `true`, then characters can be read immediately without blocking. On the other hand, if this method returns `false`, then it does not necessarily mean that a call to `read()` will block. There are simply no guarantees being given either way.

abstract void close() throws IOException This method closes the stream and releases any system resources that it holds. Subsequent calls to any of the other methods, including `read()`, will raise an `IOException`. Subsequent calls to `close()` will be ignored.

boolean markSupported() This method returns whether the stream supports the mark/reset methods: not all streams support these methods.

void mark(int readAheadLimit) throws IOException This method marks the current position in the stream. A subsequent call to `reset()` will rewind the stream to start reading again from this point, provided that no more than `readAheadLimit` characters were read before the call to `reset()`. Note that this method may throw an `IOException`, unlike that of `InputStream`.

void reset() throws IOException This method rewinds the stream to start reading from the previously marked position. This method may fail if more than the specified number of characters were read since the call to `mark()`.

10.4.3 Variables

protected Object lock This variable should be used for synchronization purposes by any `Reader` subclass.

10.4.4 IOException

An exception of type `IOException` or a subclass may be thrown by any of the methods of `Reader` but `markSupported()`. This usually indicates that some problem was encountered with the attached communications channel.

10.5 JDK 1.1 *Class OutputStreamWriter*

This class provides a character-oriented `Writer` bridge to a byte-oriented `OutputStream` channel. Characters that are written to this `Writer` class are converted into bytes according to an encoding that is specified in the constructor and then written to the attached `OutputStream` (figure 10.5).

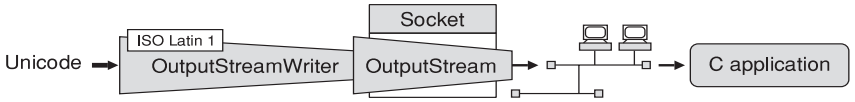

Figure 10.5 Class OutputStream Writer

This class provides internal buffering of the bytes that it writes to the attached stream. However, for efficiency, a `BufferedWriter` also should be attached in front of

this to provide buffering prior to encoding conversion. This increases efficiency by reducing the number of calls to the character-to-byte converter.

10.5.1 Constructors

The encoding that is used by this class to convert characters into bytes is specified in the constructor, or else the platform default is used.

OutputStreamWriter(OutputStream out) This constructor creates an `OutputStreamWriter` attached to the specified `OutputStream`, `out`, that uses the platform default encoding for converting characters into bytes before writing to the attached stream.

This constructor should not in general be used for networked applications. While it is appropriate for writing to files or the console, it is inappropriate to assume that the local platform default is suitable for a network protocol.

OutputStreamWriter(OutputStream out, String enc) throws UnsupportedEncodingException This constructor creates an `OutputStreamWriter`, attached to the `OutputStream` `out`, that uses the specified character encoding, `enc`, such as latin1, to convert characters into bytes for writing to the attached stream.

This is the preferred constructor for use in networked applications. Text-based communications between Java applications should probably use either UTF8 or UnicodeBig. Text-based communications with conventional applications should probably use either latin1 (ISO Latin 1) or ASCII (7-bit US-ASCII). More encodings are listed in appendix B.

10.5.2 Methods

This class provides the usual `Writer` methods, as well as the following:

String getEncoding() This method returns the name of the byte-encoding used to convert characters to bytes. This is the internal name of the encoding and may not be the name specified in the constructor: ISO_8859-1:1978, latin1, and ISO_8859-1 are but three aliases for the encoding that is internally called ISO8859_1.

10.5.3 UnsupportedEncodingException

If the requested encoding is not supported by the Java environment, an `UnsupportedEncodingException` is thrown. This is a subclass of `IOException`, so it will be caught by normal `IOException` handlers.

10.6 JDK 1.1 Class InputStreamReader

This `Reader` represents a character-oriented bridge out of a byte-oriented `InputStream`. Bytes are read from the `InputStream` and converted into characters according to the encoding specified in the constructor (figure 10.6).

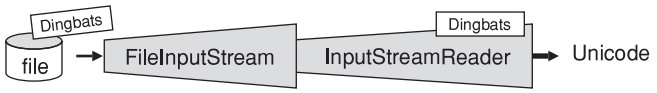

Figure 10.6 Class InputStreamReader

This class provides internal buffering of the bytes that it reads from the attached stream. However, for efficiency, a `BufferedReader` also should be attached after this to provide buffering after encoding conversion. This increases efficiency by reducing the number of calls to the byte-to-character converter.

10.6.1 Constructors

The encoding that is used by this class to convert bytes into characters is specified in the constructor. Otherwise the platform default is used.

InputStreamReader(InputStream in) This constructor creates an `InputStreamReader`, attached to the `InputStream` `in`, which uses the platform-default encoding to convert from bytes to characters.

This constructor should not in general be used for networked applications. It should be restricted to reading from system-specific files or the console.

InputStreamReader(InputStream in, String enc) throws UnsupportedEncodingException This constructor creates an `InputStreamReader`, attached to the `InputStream` `in`, which converts from bytes to characters according to the specified encoding, `enc`.

10.6.2 Methods

String getEncoding() This method returns the name of the byte-encoding used by this stream to convert from bytes to characters. This is the internal name of the encoding and may not be the same name as that specified in the constructor.

10.6.3 UnsupportedEncodingException

If a requested encoding is not supported by the Java environment, an `UnsupportedEncodingException` will be thrown.

10.7 An encoding converter

This simple example demonstrates how the `InputStreamReader` and `OutputStreamWriter` classes can be used to convert a file using one character encoding to one using another (figure 10.7).

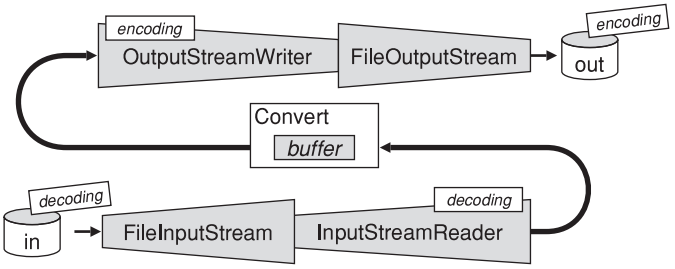

Figure 10.7 An encoding converter

```
import java.io.*;

public class Convert {
  public static void main (String[] args) throws IOException {
    if (args.length != 4)
      throw new IllegalArgumentException
          ("Convert <srcEnc> <source> <dstEnc> <dest>");
    FileInputStream fileIn = new FileInputStream (args[1]);
    FileOutputStream fileOut = new FileOutputStream (args[3]);
    InputStreamReader inputStreamReader =
      new InputStreamReader (fileIn, args[0]);
    OutputStreamWriter outputStreamWriter =
      new OutputStreamWriter (fileOut, args[2]);
    char[] buffer = new char[16];
    int numberRead;
    while ((numberRead = inputStreamReader.read (buffer)) > -1)
      outputStreamWriter.write (buffer, 0, numberRead);
    outputStreamWriter.close ();
    inputStreamReader.close ();
  }
}
```

We first ensure that four arguments have been supplied to this program, throwing an explanatory exception if not. We then create a `FileInputStream`, `fileIn`, that reads from the chosen input file, and a `FileOutputStream`, `fileOut`, that writes to the chosen output file.

We then create an `InputStreamReader`, `inputStreamReader`, that reads from `fileIn` using the chosen input encoding, and an `OutputStreamWriter`, `outputStreamWriter`, that writes to `fileOut` using the chosen output encoding.

We use a simple copy loop that uses an array of characters to copy between the streams. We copy all data from `inputStreamReader` to `outputStreamWriter` and then close both streams.

The `InputStreamReader` performs automatic decoding from the byte-oriented input file to Unicode characters, and then the `OutputStreamWriter` performs automatic encoding from these characters to the byte-oriented output file.

Note that we perform no line-terminator modification; that is a higher-level issue than character encoding.

10.7.1 In practice

To use this example, we must have a source file and specify both an input encoding and an output encoding.

```
java Convert latin1 file.latin1 UnicodeBig file.unicode
```

This command will convert the file `file.latin1` from ISO Latin 1 to the Unicode file `file.unicode`. Examination of the output file will reveal that it consists of a 2-byte endian marker followed by two bytes for every character of the original file.

```
java Convert Unicode file.unicode ASCII file.ascii
```

This command will convert the file `file.unicode` from Unicode to the 7-bit US-ASCII file `file.ascii`. Because we used a marked Unicode encoding, the Unicode decoder can automatically determine the endianness used to encode the source file. If we encoded with UnicodeBigUnmarked, then we would have to decode with UnicodeBig.

10.8 JDK 1.1 *Class FileWriter*

This `Writer` provides a character-streams interface to writing text files using the platform-default character encoding. Note that this is typically *not* 16-bit Unicode; more usually it is ISO Latin 1 or a similar 8-bit encoding (figure 10.8).

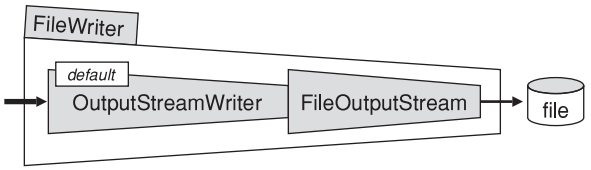

Figure 10.8 Class FileWriter

To manually specify the character encoding that is used to encode a file, simply use an `OutputStreamWriter` attached to a `FileOutputStream`, as in figure 10.8.

10.8.1 Constructors

Creating a `FileWriter` is exactly equivalent to creating an `OutputStreamWriter` using the platform-default character encoding, attached to a `FileOutputStream` constructed with the same parameters. This is how the class is implemented (it actually subclasses `OutputStreamWriter`).

FileWriter(String fileName) throws IOException This creates a `FileWriter` that writes to the specified file, `fileName`, using the platform-default character encoding. Any existing file of that name will be first destroyed.

FileWriter(File file) throws IOException This creates a `FileWriter` that writes to the specified file, `file`, using the platform-default character encoding. If the corresponding file already exists, it will be first destroyed.

FileWriter(String fileName, boolean append) throws IOException This creates a `FileWriter` that writes to the specified file, `fileName`, using the platform-default character encoding. The flag `append` specifies whether data should be appended to an existing file or whether an existing file should be destroyed.

FileWriter(FileDescriptor fd) This creates a `FileWriter` that writes to the specified `FileDescriptor`, `fd`. This must be a valid `FileDescriptor` that is already open for writing.

10.8.2 Methods

The `FileWriter` class provides all the usual methods of `Writer`. Writing characters to a `FileWriter` results in the characters being converted into bytes according to the platform-specific encoding and written to the attached file.

10.8.3 IOException

An `IOException` will be thrown by the methods of `FileWriter` if an error is encountered while writing to the file, or by the constructors if the chosen file cannot be created or written.

10.8.4 SecurityException

All file access is restricted by the current `SecurityManager`. Violation of access restrictions will result in a `SecurityException`.

10.9 JDK 1.1 Class FileReader

This `Reader` provides a character streams interface to reading text files using the platform-default character encoding. This allows you to read text files as streams of Unicode characters, without concern for the local character encoding (figure 10.9).

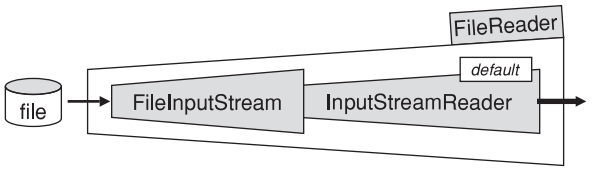

Figure 10.9 Class FileReader

To manually specify the character encoding that is used to decode a file, simply use an `InputStreamReader` attached to a `FileInputStream`, as in figure 10.9.

10.9.1 Constructors

Creating a `FileReader` is exactly equivalent to creating an `InputStreamReader` using the platform-default character encoding, attached to a `FileInputStream` constructed with the same parameter. This is how the class is implemented (it actually subclasses `InputStreamReader`).

FileReader(String fileName) throws FileNotFoundException This creates a `FileReader` that reads from the specified file, `fileName`, using the platform-default character encoding.

FileReader(File file) throws FileNotFoundException This creates a `FileReader` that reads from the specified file, `file`, using the platform-default character encoding.

FileReader(FileDescriptor fd) This creates a `FileReader` that reads from the specified `FileDecriptor`, `fd`. This must be a valid `FileDescriptor` that is already open for reading.

10.9.2 Methods

The `FileReader` class provides all the usual methods of `Reader`. Reading characters from a `FileReader` results in bytes being read from the attached file and converted into characters according to the platform-default encoding.

10.9.3 IOException

As usual, the methods of `FileReader` may all throw exceptions of type `IOException`. The constructors will throw a `FileNotFoundException` if the specified file does not exist or is not readable.

10.9.4 SecurityException

All file access is restricted by the current `SecurityManager`. Violation of access restrictions will result in a `SecurityException`.

10.10 Using the file streams

The following example demonstrates how the `FileReader` and `FileWriter` classes can be used to interface with the console. In earlier examples, when we used methods of `System.in` and `System.out`, we assumed that the console communicated using ISO Latin 1 (8-bit ASCII). Using the `FileReader` and `FileWriter` classes, on the other hand, assumes that it uses whatever is the default platform-specific file encoding.

This example simply echoes keyboard input back to the terminal in uppercase. Data are read and written a buffer at a time. Typically this will be equivalent to echoing every line typed (figure 10.10).

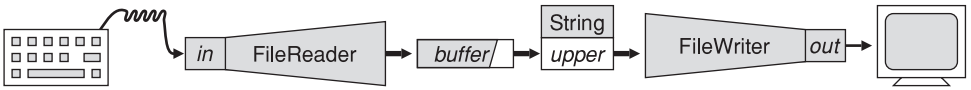

Figure 10.10 Class CharFiles

```
import java.io.*;

public class CharFiles {
  public static void main (String[] args) throws IOException {
    FileReader fileReader = new FileReader (FileDescriptor.in);
    FileWriter fileWriter = new FileWriter (FileDescriptor.out);
    char[] buffer = new char[256];
    int numberRead;
    while ((numberRead = fileReader.read (buffer)) > -1) {
      String upper = new String (buffer, 0, numberRead).toUpperCase ();
      fileWriter.write (upper);
      fileWriter.flush ();
    }
  }
}
```

Here, we attach the `FileReader fileReader` to `FileDescriptor.in`, which is the file descriptor from which keyboard input can be read (`System.in` is attached to this). We also attach the `FileWriter fileWriter` to `FileDescriptor.out`, which is the file descriptor to which console output should be written (`System.out` is attached to this).

We then loop, reading data into the character array `buffer`, converting this to the uppercase `String upper`, and then writing this to `out`. Note that we are not reading a line at a time; we are simply reading as much data as the `read()` method can obtain. Typically, when reading from the keyboard, this will be a line at a time. However this is unique to keyboard input. To properly read lines, see the `readLine()` method of class `BufferReader`. Note also that we are not inserting newlines; newlines in the output simply result from newlines in the input.

We flush after writing every buffer because the `FileWriter` class automatically obtains the output buffering that is provided by its `OutputStreamWriter` superclass.

10.11 Wrapping up

In this chapter we have presented the character streams needed to support character-oriented streams-based communications from Java. `Reader`, `Writer`, and their subclasses parallel closely the byte-oriented `InputStream` and `OutputStream` classes. Additionally,

the `InputStreamReader` and `OutputStreamWriter` classes provide a bridge between character streams and byte streams.

It should not be misconstrued that character streams are always bridged to or from byte streams. In the following chapters we will look at the various character stream filters as well as piped and memory-based character streams that provide a true 16-bit end-to-end communications channel.

Many networked applications that involve text-based communications are only concerned with ASCII or ISO Latin 1 characters, which are easily accessed through the byte streams. However, it is still prudent in many cases to use the character streams for their better text support and increased efficiency, which comes from both their better synchronization support and their better use of internal arrays.