Chapter

14

Files

In this chapter we will be looking at buffered reading and writing of files, which is all you will need for most file access. Non-buffered I/O is beyond the scope of this book; however, if your application needs to use non-buffered I/O, take a look at the subclasses of *IOAccessor*.

Filename

The class *Filename* is associated with files. Filename provides various utilities for finding out about files, and is also used to open files. There is a lot of behavior associated with Filename, both on the class side and the instance side. We'll show some of it here, but it's worth browsing the class to see what else is available.

To create an instance of Filename there are two basic approaches. You can send a message to the Filename class, or you can send a message to a string. For example,

```
fileName := Filename named: 'c:\baseDir\file.ext'.
fileName := 'c:\baseDir\file.ext' asFilename.
```

If you have a directory, and want to create a Filename from the directory and the name of a file, there are two ways to do it. If the directory is already a Filename, you construct another Filename from the directory and the name of the file. If the directory is a string, you concatenate the directory, a separator, and the name of the file to create the full name of the file, then send the `asFilename` to the new string. (By convention, the name of a directory does not end in the separator character.) So, you would do one of the following.

```
fileName := directoryFilename construct: 'myFile.ext'.
fileName := (directoryPathString, (String with: Filename separator),
'myFile.ext') asFilename.
```

To ensure portability across platforms, you may want to add a new class side method to Filename that creates a Filename from a list of directories and a file. Here's an example of what it might look like.

```
Filename class>>construct: aFileString directories:
anArrayOfDirectoryStrings
   | stream |
   stream := (String new: 30) writeStream.
```

```
anArrayOfDirectoryStrings do:
   [:each | stream nextPutAll: each; nextPut: Filename separator].
aFileString == nil ifFalse: [stream nextPutAll: aFileString].
^self named: stream contents
```

Because you can open a Filename but not a String, there's a difference between a Filename and a string containing the name of a file. It can get confusing to talk about file names because it's hard to know whether we are talking about a string or an instance of Filename. A convention that is often adopted is to refer to strings containing names as *paths*, so you might see something like the following in code.

```
filePath := '/baseDir/subDir1/subDir2/file'.
fileName := filePath asFilename.
directoryPath := 'c:\baseDir\subDir1\subDir2'.
directoryName := directoryPath asFilename.
```

If you have an instance of Filename, you can find the various component parts by sending messages to it. Similarly, if you have a string that contains the name of a file, you can find the various components by sending messages to the class side of Filename.

```
fileName := 'c:\baseDir\subDir\myFile.ext' asFilename.
fileName directory.                 a FATFilename('c:\basedir\subdir')
fileName head.                      'c:\basedir\subdir'
fileName tail.                      'myfile.ext'

filePath := 'c:\baseDir\subDir1\fileName.ext'.
Filename components: filePath.      OrderedCollection ('c:' 'baseDir'
                                    'subDir1' 'fileName.ext')
Filename breakup: filePath.         OrderedCollection (#('c' ':\')
                                    #('baseDir' '\') #('subDir1' '\')
                                    #('fileName' '.') #('ext' ''))
Filename splitExtension: filePath.  #('c:\baseDir\subDir1\fileName'
                                    'ext')
Filename splitPath: filePath.       #('c:\baseDir\subDir1\'
                                    'fileName.ext')
```

Opening files

The stream classes associated with file access are *ExternalReadStream*, *ExternalWriteStream*, *ExternalReadWriteStream*, and *ExternalReadAppendStream*. However, it’s not usual to send messages directly to these classes to create a stream because they expect to be passed an already open file connection. Instead, the usual way to open a file and create a stream on the file is to send the appropriate message to a Filename. So, for example, if you send the `readStream` message to a Filename, it will open the file in the correct mode, then create an ExternalReadStream on the opened file. The following messages sent to a Filename open the underlying disk file in different modes and give back a stream of the appropriate class. Table 14-1 summarizes the information.

- The message `readStream` opens an existing file for reading.
- The message `appendStream` opens a file for appending. If the file already exists, its contents are left untouched. If the file doesn't exist, it will be created. Reading is not allowed, and writing will always append to the end.

- The message `readAppendStream` opens a file for reading and appending. If the file already exists, its contents are left untouched. If the file doesn't exist, it will be created. The stream can be positioned anywhere for reading, but writing will always append to the end.
- The message `newReadAppendStream` opens a file for reading and appending. If the file already exists, its contents will be deleted. If the file doesn't exist, it will be created. The stream can be positioned anywhere for reading, but writing will always append to the end.
- The message `writeStream` opens a file for writing. If the file already exists, its contents will be deleted. If the file doesn't exist, it will be created. This stream cannot be positioned since it always writes to the end of the stream.
- The message `readWriteStream` opens a file for reading and writing. If the file already exists, its contents are left untouched. If the file doesn't exist, it will be created. This stream can be positioned anywhere for reading and writing.
- The message `newReadWriteStream` opens a file for reading and writing. If the file already exists, its contents will be deleted. If the file doesn't exist, it will be created. This stream can be positioned anywhere for reading and writing.

Table 14-1. Summary of file opening messages.

Message	Delete Contents	Create File	Read	Write
readStream	n	n	a	n
writeStream	y	y	n	e
appendStream	n	y	n	e
readAppendStream	n	y	a	e
newReadAppendStream	y	y	a	e
readWriteStream	n	y	a	a
newReadWriteStream	y	y	a	a

Key: y = Yes, n = No, a = Anywhere, e = at End

Closing files

When you have finished reading a file, it's important to close the file. It's easy to close a file — you simply send the `close` message to the stream. However, if you rely on the garbage collector to dispose of the stream and don't explicitly close the file, the file remains open to the operating system. Since operating systems allow only a limited number of open files, you'll eventually run out of file descriptors and won't be able to open any more files. The simplest way to make sure that you close a file is to do something like the following.

```
writeStream := 'c:\temp\myfile' asFilename writeStream.
[writeStream
   nextPutAll: 'Some data';
   cr; nextPutAll: 'More data']
      valueNowOrOnUnwindDo: [writeStream close]
```

The `valueNowOrOnUnwindDo:` message makes sure that the code in the parameter block is always executed, whether the receiver block terminates naturally or an exception is raised. (If you wanted to execute a

block of code *only* if the block is terminated by an exception, you would send `valueOnUnwindDo:`.) The unwind aspect of the message refers to the fact that the exception unwinds the stack. In the event of an exception, the exception handler is invoked before the block parameter is executed.

Sometimes, however, you can't wrap the reading or writing code with a `valueNorOrOnUnwindDo:` because you want to leave the file open. For example, if you create a log file, you may decide for performance reasons to leave it open, rather than open and close it every time you want to log something. In situations like this, you need a different mechanism to close the file. For more information on how you might do this, see Chapter 18, Cleaning up at Termination.

What files are open?

If you forget to close a file by sending `close` to the stream, you may find yourself out of files descriptors and unable to file out your changes. Fortunately, there is a way to find out what files are open, and then to close them. To find out what files are open, you can inspect the following. *OpenStreams* is a class variable of *ExternalStream* that contains all the open files (assuming they were opened using the normal mechanisms related to a subclass of ExternalStream).

```
(ExternalStream classPool at: #OpenStreams) copy inspect.
```

The reason for the `copy` message is that if you close a file, the collection held by OpenStreams will shift under you, while a copy of the collection will continue to point to the files correctly. To close a file, open an inspector on the element of the collection corresponding to the file, then in the right hand pane, evaluate `self close`. If you want a more automatic way to close the files, you can do the following. In Chapter 31, Customizing your Environment, we show a way to put this in the VisualLauncher menu.

```
(ExternalStream classPool at: #OpenStreams) copy do:
   [:each | each name = 'someFilePath' ifTrue: [each close]].
```

Reading from and writing to files

Reading text files

Many applications read files containing data that is used to set parameters and direct behavior. These files often use the pound (#) symbol to denote a comment. Here is some code that reads a line from a file, and strips off the comments. We've added the method `nextUncommentedLine` to the Stream class to make it available to anyone who needs it. It requires the `trimWhiteSpace` method that we showed in Chapter 12, Strings. There are three things to note in this example: we've wrapped the opening of the file in a signal handler to trap errors associated with the file not existing or not being readable; we keep reading the file until we are at the end, which we discover by sending the `atEnd` message; and we are wrapping all the read operations with `valueNowOrOnUnwindDo:` to make sure that the file gets closed.

```
Stream>>nextUncommentedLine
   "Read a line, remove any comment, then trim the white space"
   ^((self upTo: Character cr)
      copyUpTo: $#) trimWhiteSpace
```

```
MyClass>>readFile: aFilename
   | readStream |
   OSErrorHolder errorSignal
      handle: [:ex | ex restartDo: [^self myHandleError: ex]]
      do: [readStream := aFilename readStream].
   [[readStream atEnd]
      whileFalse: [self myProcessLine: readStream
nextUncommentedLine]]
         valueNowOrOnUnwindDo: [readStream close]
```

Reading and writing binary data

Besides reading and writing text data, ExternalStream provides the ability to read and write binary data. If you want to store binary data, first send the stream the `binary` message to tell it to expect binary rather than text data. To tell if a stream is a binary stream you can send `isBinary`, which returns a Boolean.

To write a 32-bit integer to the next four bytes of the stream, send `nextLongPut:` and to read the next four bytes as a 32-bit integer, use `nextLong`. To write and read the next two bytes as an integer, use `nextWordPut:` and `nextWord`. To get the next two bytes as a signed integer, use `nextSignedInteger`. You can specify the number of bytes to treat as an integer when writing or reading by sending `nextNumber:put:` or `nextNumber:`. You can write out a string, where the string's length will be written in the first one or two bytes (depending on the number of bytes in the string, then read the same string, with the messages `nextStringPut:` and `nextString`. Here's an example.

```
stream := 'c:\temp\binfile' asFilename readWriteStream.
stream binary.
[stream nextStringPut: 'Hi Alec'.
stream nextLongPut: 999999.
stream nextWordPut: 33.
stream reset.
Transcript cr; show: 'String = ', stream nextString.
Transcript cr; show: 'Long = ', stream nextLong printString.
Transcript cr; show: 'Word = ', stream nextWord printString]
   valueNowOrOnUnwindDo: [stream close].
```

Positioning

Besides the normal stream positioning messages, file-based streams support some additional positioning and padding messages. Without going into much detail, they include the ability to skip to the end of the next chunk of data, assuming that the stream consists of a series of chunks of the specified size (`padTo:`), to skip to the end of the next chunk writing the specified character in the skipped positions (`padTo:put:`), to skip to the end of the current word, assuming that the stream consists of two-byte words (`padToNextWord`), to skip to the end of the current word and replace the skipped characters with the specified character (`padToNextWordPut:`), to skip the specified number of two-byte words (`skipWords:`), and to get and set the current position in the stream, assuming the stream consists of two-byte words (`wordPosition` and `wordPosition:`).

End of line character

You can discover what convention the file uses to indicate end-of-line characters by sending the `lineEndConvention` message to the stream. The line end convention for the different platforms is:

Macintosh — LineEndCR, PC — LineEndCRLF, Unix — LineEndLF. When you are simply in the Smalltalk image, you can send the `cr` message to a stream or a TextCollector and it will do the correct thing (from my experience, you can substitute the `lf` message for `cr` within Smalltalk and get the same results). However, if you are sending text to another process or writing text to a file, you need to make sure that the end-of-line character is appropriate for the reader of the text, which you can do by setting the line end convention for the stream by sending it the `lineEndConvention:` message.

Flushing buffers to disk

The `flush` message makes sure that the stream contents are flushed to the disk file buffers. Some operating systems, such as Unix, manage the synchronization of disk buffers and files on disk, so flushing Smalltalk streams to disk buffers doesn't necessarily guarantee that the data gets to the disk, only that the disk file buffers contain the data. To get the data to disk, send `commit`, which goes to the operating system and asks it to sync the buffers with the disk, assuming the operating system provides this capability.

Utilities

Class side utilities

Some of the more useful utilities on the class side of Filename are shown below. The `isCaseSensitive` message tells you whether file names are case sensitive. For example, on Windows, there is no distinction between 'fileName.ext' and 'FILENAME.EXT'. However, on Unix, these are very different files. The `defaultDirectory` message tells you the default directory you are in. The confusingly named `currentDirectory` tells you the relative directory name for constructing relative paths. The `volumes` message tells you what volumes are reachable. Finally, you can find all the files that match a specified pattern.

```
Filename isCaseSensitive.          false
Filename currentDirectory.         a FATFilename('.')
Filename defaultDirectory.         a FATFilename('c:\visual\image')
Filename volumes.                  #('A:\' 'B:\' 'C:\' 'D:\')
Filename filesMatching: '*.im'.    OrderedCollection ('visual.im'
                                   'alec.im' 'math.im' 'mkim.im')
```

Instance side utilities

Directories

If the instance of Filename is a directory, you can find its contents, or get all the files that match a certain wildcard pattern. For example,

```
dirName := Filename defaultDirectory.
dirName directoryContents.         #('VISUAL.IM' 'VISUAL.SOU'
                                   'VISUAL.CHA' 'ST' 'ALEC.IM'
                                   'ALEC.CHA' 'SYSCHANG' 'MATH.IM'
                                   'MKIM.IM' 'MYFILE' 'MKIM.CHA'
                                   'GOODIES')
dirName filesMatching: '*.im'      OrderedCollection
                                   ('c:\visual\image\visual.im'
```

```
'c:\visual\image\alec.im'
'c:\visual\image\math.im'
'c:\visual\image\mkim.im')
```

You can also create a directory by sending the `makeDirectory` message. Of course, this may fail if there is already a file with that name or the parent directory is unwritable. You can check for these conditions beforehand, or wrap the directory creation code with a signal handler.

Contents of files

You can read the contents of a file into a string by sending `contentsOfEntireFile` (which closes the file after reading it). You can even bring up an editor on a file by sending `edit`.

Information about files

You can find the various access and modification dates associated with a file by sending the `dates` message. You can check to see if a file exists by sending either `exists`, or `definitelyExists`, which handles inacessible network files differently. The size of the file in bytes can be found by sending `fileSize`. You can determine if a file is readable or writable by sending `isReadable` and `isWritable` (for a directory to be writable, it must be possible to create and rename files in it). To find if the file is a true writeable file (ie, not a directory), send `canBeWritten`. If you want to know if a file is a directory, send `isDirectory`. You can tell if the file has an absolute or relative path by sending `isAbsolute` or `isRelative`. Let's look at some examples of these on a Windows machine.

```
dirName := 'c:\baseDir' asFilename.
fileName := dirName construct: 'myfile'.
fileName exists.                                true
dirName definitelyExists.                       true
fileName fileSize.                              20
dirName fileSize.                               0
fileName isReadable.                            true
dirName isWritable.                             true
fileName canBeWritten.                          true
dirName canBeWritten.                           false
fileName isDirectory.                           false
dirName isDirectory.                            true
fileName isAbsolute.                            true
dirName isRelative.                             false
```

Manipulating files

Among other utilities, you can delete a file (`delete`), you can copy a file to another file (`copyTo:`), and you can rename a file by moving it (`moveTo:`, which copies the file then deletes the original). If your platform supports it and it is a meaningful operation — for example, you are not trying to rename across disks — you can rename a file using the much more efficient `renameTo:` message.