Chapter 10 - Streams, files, and BOSS

Overview

Sequenceable collections are often processed in linear order, one element after another. Although linear access can be performed with collection accessing and enumeration methods, Smalltalk library contains a group of classes called streams that simplify linear access and improve its efficiency. It is important to understand that streams are not a new kind of collection but rather a mechanism for accessing an existing collection.

Smalltalk distinguishes two kinds of streams with largely overlapping protocols - internal and external. Internal streams are used to access sequenceable collections whereas external streams are for file access.

Storing data in a file and reading it back requires two facilities: access to the contents of the file (provided by external streams) and access to the file system itself (for operations such as accessing directories and files, and for creation, naming, and deleting files and directories). Access to the file system is provided by class Filename. Most file operations thus require both an instance of an external stream and an instance of Filename.

External streams provide byte-by-byte access to file contents but no tools to store objects, thus lacking the facility that most Smalltalk programs need. Although every class knows how to convert its instances into text representing executable Smalltalk code, this facility is too inefficient for larger objects. VisualWorks thus provides a special group of classes for storing objects as binary codes. This tool is called the Binary Object Streaming Service (BOSS). Since storage and retrieval of binary objects depend on files and streaming access, the use of BOSS requires understanding of external streams and Filename objects.

10.1 Introduction to streams

Sequenceable collections must often be accessed one element after another with intermediate processing, as if viewed through a window that remembers which element is being viewed (Figure 10.1). Another, and historically more relevant analogy, is that a stream is like a digital magnetic tape whose recordings (collections of sound codes) are read one after another in the order in which they were recorded. This kind of access is called *streaming* and although it can be achieved with standard enumeration methods, the Smalltalk library provides a group of classes that makes streaming access easier and more efficient. A stream is thus a tool for viewing existing sequenceable collections - a collection accessor. A stream is *not* a new kind of collection.

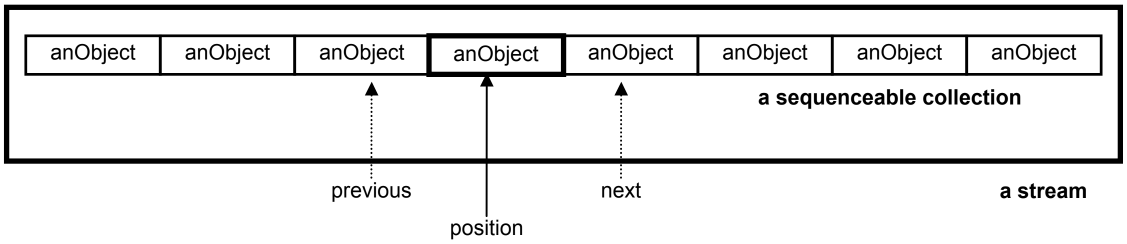


Figure 10.1. Stream is a mechanism for accessing a sequenceable collection via a positionable window.

Here are a few examples of situations that require streaming:

- Construction of text from strings extracted from a file or entered by the user. Examples include creation of reports and form letters.
- Analysis of text such as extraction of words from an article or processing of Smalltalk source code by the compiler.
- Reading and writing of files.

Execution of each of these tasks involves some or all of the following operations:

- Opening a stream on a collection.
- Selecting the starting position.
- Moving to the next or the previous element and examining its contents.
- Replacing the object at the current position with another object.
- Adding an object behind the object processed so far.
- Peeking ahead at the next element without changing the current position.
- Repositioning the stream pointer to the beginning or the end of the stream, or to any location given by an index.
- Testing whether the position pointer is at the end of the stream.
- Accessing file elements.

Since different tasks require different types of streaming access, Smalltalk streams are implemented by a group of classes, a subtree of the abstract class Stream. Stream factors out the shared properties of all streams such as having contents (the underlying collection), testing whether the end of the stream has been reached, and moving the position pointer. Some of these methods are completely defined in the abstract class Stream and possibly overridden at lower levels, others are defined as 'subclass responsibility'.

An example of a stream operation shared by all types of streams is enumeration. Its implementation is the same for all streams and class Stream thus contains its full definition:

```
do: aBlock
"Evaluate aBlock for each of the elements of the receiver."
	[self atEnd] whileFalse: [aBlock value: self next]	“Evaluate block with successive elements.”
```

Method next which is the basis of the method is left as subclass responsibility.

To emphasize the close relationship between streams and collections, all classes in the Stream hierarchy with the exception of Random are defined in category Collections - Streams even though the Stream subtree in the class hierarchy is totally disjoint from the Collection subtree[1]. The whole subtree is as follows:

```
Object
  Stream
    PeekableStream
      PositionableStream
        ExternalStream
          BufferedExternalStream
            ExternalReadStream
              ExternalReadAppendStream
              ExternalReadWriteStream
            ExternalWriteStream
        InternalStream
          ReadStream
          WriteStream
            ReadWriteStream
            TextStream
    Random
```

As we have already suggested, streams can be classified according to several parameters. The first distinction used in the class hierarchy is whether the stream allows reading the next element and returning

[1] Class Random is a subclass of Stream only because its elements are obtained in a linear fashion. Unlike other streams, elements accessed by Random don’t exist independently and are created when requested by message next.

to the original position; in other words, whether it is possible to "peek" ahead without moving the cell (window) pointer. With elements generated by random number generators, this is obviously not possible since a random number generator cannot be asked to recall the random number that it generated before, and this is where Random splits from other streams in the Stream tree. Since we have already covered random numbers, the rest of this chapter deals with peekable collections only.

The fact that we can peek ahead does not imply that we can reposition the window to any place in the stream, in other words, jump from one place to another. This additional property is needed, for example, for random access of files, and its underlying mechanism is defined in class PositionableStream via its instance variable position. Its value is an integer number, an index that points to the current position of the window on the stream, an element in the underlying sequenceable collection. Most stream accessing operations first move the pointer by one position "to the right" (increment the index) and then access the corresponding element. The pointer thus always points before the element that will be accessed by the next stream accessing message. Since the index of the first element in a stream is 1, resetting a stream sets position to 0. To provide control over positioning limits, PositionableStream has two instance variables called readLimit and writeLimit. These two integers determine the current last position accessed by the stream; the first position is always the element at index 1 of the underlying collection.

Class PositionableStream is the root of two sub-trees - internal streams and external streams. *Internal streams* are used for accessing sequenceable collections residing, in principle, in the internal memory of the computer. Smalltalk uses internal streams extensively to construct messages, menu labels, arrays of coordinates of geometric objects, parsing during the compilation of Smalltalk programs, and in other operations. *External streams* are an extension of the stream concept to files. They are used to read or write elements of files stored on external media such as disks or obtained from the network.

One important difference between internal and external streams is in the *kind of objects* stored in their underlying collections (Figure 10.2). Elements of collections accessed by *internal* streams can be any objects such as integers, characters, strings, rectangles, or even other streams. *External streams*, on the other hand, are byte-oriented which means that their elements are individual bytes such as ASCII characters or binary codes with another interpretation. Byte orientation of external streams is due to the fact that files are managed by operating system functions, and operating systems access consecutive elements of files as bytes.

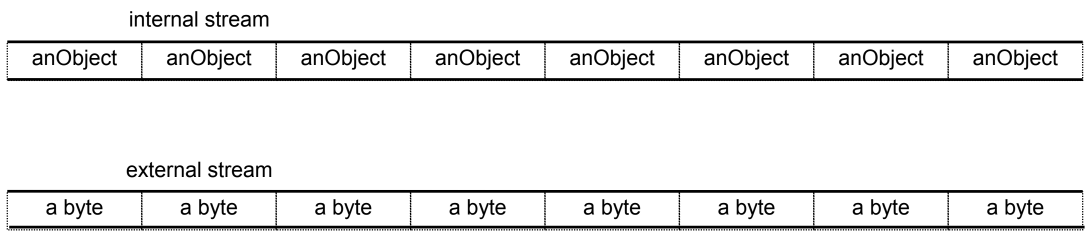


Figure 10.2. *Internal streams* may stream over collections containing any objects but *external stream* access is byte-oriented.

Another difference between internal and external streams is that their hierarchy contains an additional abstract class called BufferedExternalStream. This class implements the concept of a *buffer*, a memory area holding the working copy of a portion of a file (Figure 10.3).

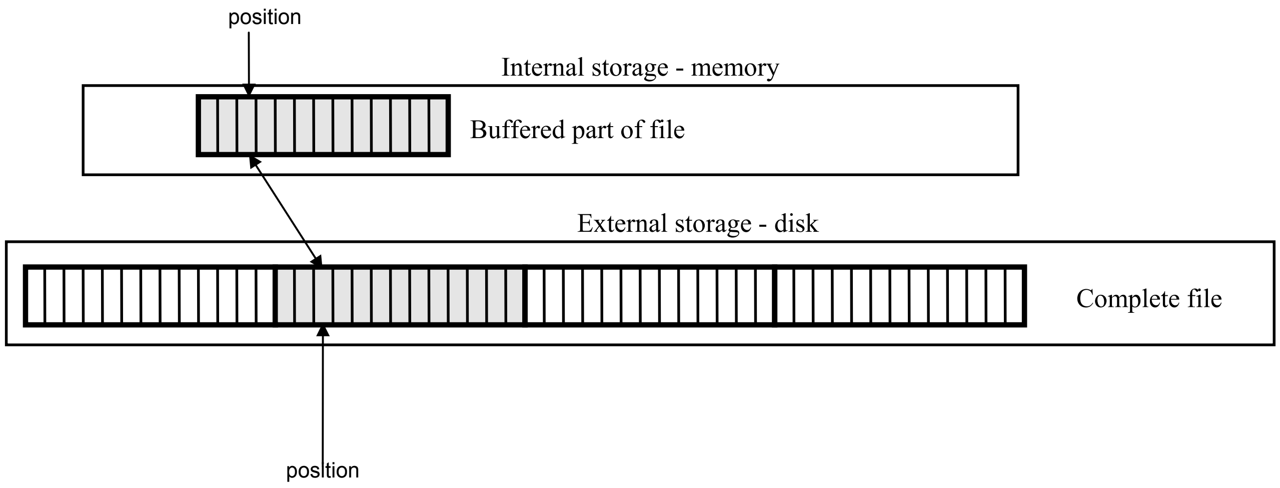


Figure 10.3. Only a part of a file is kept in the memory.

Another difference between internal and external streams is that internal streams include class TextStream specialized for accessing Text objects. External streams, on the other hand, include appendable streams that allow adding information only at the end of a file which does not have an equivalent among internal streams.

Besides the distinction between internal and external streams, we can also distinguish streams that can only be read from streams that can only be written, and streams that can be either read or written. This classification applies both to internal and external streams although there are a few differences in details.

After this brief overview, we will now present internal streams. External streams, files, and related concepts of external storage are covered in the second part of this chapter.

Main lessons learned:

- A stream is an accessor of consecutive elements of sequenceable collections.
- The two main groups of streams are internal and external streams.
- Internal streams are used to access sequenceable collections whose elements may be arbitrary objects stored in memory. External streams are used to access consecutive bytes stored in a file or on the network.
- Besides the distinction between internal and external streams, Smalltalk also distinguishes between read-only, write-only, and read-write streams.
- The class hierarchies of internal and external streams are somewhat different. The hierarchy of external streams includes class BufferedExternalStream which is responsible for hiding the fact that only a part of a file is present in memory at any time, internal streams include TextStream.

10.2 Internal streams

The Smalltalk library uses internal streams a lot but novice programmers often neglect them, probably because their functions can be implemented by operating directly on their underlying collections. Or possibly because there is such an overwhelming number of stream methods, some of them with rather obscure behaviors. This is unfortunate because stream methods considerably simplify frequently needed operations in the same way that specialized enumeration methods simplify specialized enumeration. Moreover, streams may significantly improve performance, for example as an alternative of string concatenation. And finally, most uses of streams depend on only four or five simple messages.

In the rest of this section, we will outline stream protocols, and the next section will give examples of their use. Note that although most streaming methods are shared by all stream classes, some are not: There are methods that only work with external streams, methods that can be used with read streams but not with write streams, and so on. Most of these limitations are obvious and natural.

Creation

Internal streams are usually created with class methods on: with:, or by messages addressed to the underlying sequenceable collections; rarely, streams are created with and on:from:to: and with:from:to:. All these methods create a new stream over the specified collection and initialize the position, readLimit, and writeLimit variables. The details are initially a bit confusing because each method initializes these variables differently but you don't have to think about the details in most cases because the typical behavior is quite natural.

It is interesting to note that creation methods succeed even if the underlying collection is not sequenceable (for example a Set) but any subsequent attempt to access a stream created over such a collection will fail. Now for the details:

aStreamClass on: aCollection, creates a stream over aCollection and positions the pointer *at the start*, to position = 0. The initial settings of the readLimit and writeLimit depend on the kind of stream and the effect is summarized in Figure 10.4.

	position	readLimit	writeLimit
ReadStream	0	end of collection	end - irrelevant
ReadWriteStream	0	0	end of collection
WriteStream	0	0 - irrelevant	end of collection

Figure 10.4. Effect of on: on various types of internal streams.

aStreamClass with: aCollection, creates a stream over aCollection and initializes position, readLimit, and writeLimit to the *last* index, positioning the pointer *at the end*. To remember the difference between with: and on:, use the mnemonic that *the first letter of with: is 'at the end of the alphabet' whereas the first letter of on: is 'at the start of the alphabet'*. The effect of with: is summarized in Figure 10.5. with differences with respect to on: italicized.

	position	readLimit	writeLimit
ReadStream	*end of collection*	end of collection	end - irrelevant
ReadWriteStream	*end of collection*	*end of collection*	end of collection
WriteStream	*end of collection*	*end - irrelevant*	end of collection

Figure 10.5. Effect of with: on various types of internal streams.

The following are examples of the effect of several stream creation messages:

```
ReadStream on: #(1 3 'abc')          "Opens a read stream on array #(1 3 'abc'); position is initialized to 0."
ReadStream with: #(1 3 'abc')        "Opens a read stream on array #(1 3 'abc'); position is initialized to 3."
WriteStream on: (String new: 16)     "Opens a write stream on an empty string; position is initialized to 0."
ReadStream with: ('abcd' ) asSet     “Succeeds but any attempt to access the stream will fail.”
```

Most stream applications use the on: creation message and only a few use with:. Creating a new stream with message new is illegal because it does not specify the underlying collection.

Instead of creating a stream by sending a creation message to a *stream* class, you can also create a stream by sending readStream, writeStream, or readWriteStream to a *sequenceable collection* as in

```
#(12 43 23 67) readStream
```

which produces the same result as

ReadStream on: #(12 43 23 67)

Accessing

This protocol includes many instance messages that return the contents of the stream (the underlying collection), reposition the pointer, or access elements of the underlying collection. The 'setting' messages (various forms of put which add one or more new elements) grow the underlying collection if necessary. Some of the messages in this protocol are:

size - returns the larger of readLimit and position. If position is larger, it increases readLimit to position.

contents - returns a *copy* of the *part* of the underlying collection from the start to the readLimit of the stream. Its definition is

```
contents
"Answer a copy of the receiver's collection from 1 to readLimit."
	readLimit := readLimit max: position.
	^collection copyFrom: 1 to: readLimit
```

next is used for reading the next element. It first moves the pointer to the right by one position (increments position by 1) and returns the element at this position. If the pointer is already at the end of the stream (measured with respect to readLimit or writeLimit), next returns nil and does not change the pointer.

nextPut: anObject - increments the pointer and stores anObject as the next element of the underlying collection. Returns anObject just like other adding messages. Overwrites the existing element of the collection if there was one at this position, and grows the collection if the new element is being added to a full collection. It is important to note that the stream does not work with a copy of the collection but with the collection itself.

nextPutAll: aSequenceableCollection – stores individual elements of aSequenceableCollection of size n as the next n elements of the stream. Compare this with nextPut: which would add the whole collection as a single element (Figure 10.6). The difference is similar to the difference between add: and addAll: collection messages. Another similarity between add methods in collections and nextPut in streams is that they all *return the argument* rather than the modified receiver.

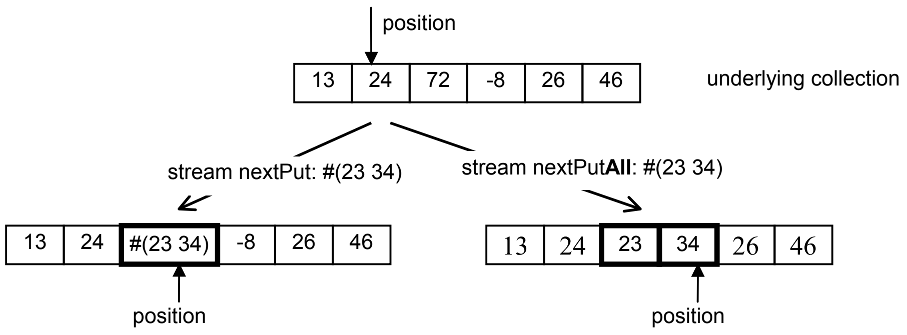


Figure 10.6. Result of nextPut: (left) and nextPutAll: (right). Note which elements are added and which elements are gone.

peek - increments position and returns the element at that position like next, but resets the pointer to its original place.

upTo: anObject - repeats sending next until it reaches the first occurrence of anObject or the readLimit. It returns a *collection* whose elements[2] are the elements retrieved by the consecutive next messages from the start of the iteration up to but *not including* anObject. The pointer is left pointing at anObject so that the next next message will return the item following anObject. If anObject is not found, the message returns a collection containing all elements from the current position up to and including the last element of the receiver stream.

through: anObject - has the same effect as upTo: but anObject *is* included in the returned stream. The final value of position is the same as for upTo:.

Positioning

Methods in this protocol reposition the pointer without retrieving or storing elements.

position: anInteger - changes the value of pointer to anInteger. This method is used mainly for reading and the value of anInteger is usually between 0 and readLimit. Remember that the element accessed by next will be the element at position anInteger + 1.

reset - resets the pointer to 0 to prepare for access to the first element. Same as position: 0.

setToEnd sets pointer to the last element of the stream marked by readLimit.

skip: anInteger - jumps over the specified number of elements without accessing them. Performs

self position: position + anInteger

In other words, method skip: performs *relative* repositioning with respect to the initial position, whereas position: is for *absolute* repositioning. As a consequence, skip: -1 may be legal but position: -1 never is.

skipUpTo: anObject - skips forward to anObject and leaves pointer pointing at it. Next access will thus be to the element following anObject. Returns the receiver stream on success, nil if it does not find anObject.

skipSeparators - skips a sequence of any of the following characters: space, cr, tab, line feed, null, and form feed. This and some other methods hint that internal streams are often used for character processing.

do: - uninterrupted enumeration over the underlying collection until self atEnd returns true. Since it uses next to access the consecutive elements, it *starts at the current position* rather than at the start of the collection. As a consequence, it *may not enumerate over all elements of the collection*.

Testing

Testing messages determine whether the stream is empty, what is the current position in the stream, and whether position points at the end.

atEnd - returns true if position is greater than or equal to readLimit. If the stream is not defined over the whole underlying collection(e.g, on:from:to:), readLimit does not refer to the last element of the collection.

[2] We will use ‘stream elements’ to refer to the elements of the underlying collection.

isEmpty - tests whether position = 0, in other words, refers to how much of the collection has been viewed. This is somewhat confusing because it is not clear what it means that a stream is empty. As an example

```
(ReadStream on: 'abcd') isEmpty
```

returns true although the underlying collection is not empty.

position - returns the current value of the pointer.

The following code fragment illustrates some of these new messages and more examples will be given later:

```
| stream |
stream := ReadStream on: #(13 3 'abc' 'xyz' $a $b).    “Creates new stream over the specified array.”
stream contents.                                  "Returns #(13 3 'abc' 'xyz' $a $b)."
stream position.                                  "Returns 0 - stream is positioned to read the first element."
stream next.                                      "Returns 13, the next element of the underlying collection."
stream skip: 2.                                   "Increments position by 2 and returns receiver stream."
stream next.                                      "Returns 'xyz' and increments position."
stream skip: 20.                                  "Opens an Exception notifier - position out of bounds."
```

Main lessons learned:

- The main stream protocols are creation, accessing, positioning, testing, and enumeration.
- The essential stream messages are on:, with:, next, nextPut:, nextPutAll:, and testing.
- Stream creation messages create a stream over a collection and position a pointer at the start or at the end of the underlying collection.
- A stream may be opened over a sub-range of the underlying collection.
- The values of readLimit and writeLimit represents the effective end of the stream.
- The most common accessing messages are next and nextPut:. Both first increment the pointer and then access (and possibly change) the collection.
- Positioning messages are used for random (non-linear) access.
- Details of stream messages depend on the kind of stream.

Exercises

1. What is the relationship between the position and the index in the underlying collection?
2. Examine what happens to the underlying collection when you add new elements at the end of a write stream.
3. Examine what happens to the underlying collection when you add new elements at the end of a write stream opened over its sub-range.
4. How does on:from:to: work and how does it limit the new stream’s access to a part of the underlying collection?
5. Message upTo: anObject returns a subcollection of the stream’s collection. What happens when you then send nextPut: to this stream?
6. What does skipSeparators return?
7. printString for streams returns only the name of the class. Redefine it to return class name followed by contents, position, and (depending on the kind of stream) the value of readLimit and writeLimit.
8. Can any other enumeration methods in addition to do: be used on streams?
9. Explain the result of each of the following lines:
   ```
   (ReadStream on: 'abcdef') next; next; position: 3; next
   (WriteStream on: Array new) nextPut: $a; nextPut: 13
   (ReadStream on: 'abcdef') peek; peek
   (WriteStream on: String new) nextPut: $a; nextPut: $b; nextPut: 3
   ```

```
(WriteStream with: 'abcd') nextPutAll: 'xyz'; yourself
(WriteStream on: 'abcd') nextPutAll: 'xyz'; yourself
(WriteStream with: 'abcd') nextPutAll: 'xyz'; contents
(ReadWriteStream with: 'abcd') position: 2; nextPutAll: 'xyz'; contents
```

10. How does contents work on writeable streams?

10.3 Examples of operations on internal streams

In this section, we will give several examples of stream behavior and demonstrate some of the most common uses of streams.

Example 1: Stream enumeration

As we already mentioned, stream implementation of do: operates only over the elements following the current position. Because the method does not reset the pointer when it ends, the pointer ends up pointing at the end of the stream. The method returns the receiver stream. As an example,

```
| stream |
stream := ReadStream on: 'abcdefg'.          "Creates a stream on characters; position = 0."
stream skip: 2.                              "Value of position is now 2."
stream do: [:element | Transcript show: (element printString)]
```

prints 'cdefg' and returns the read stream with position = 7.

Example 2: Using streams to edit strings - filtering

Problem: String modification is a typical use of internal streams. Write method replace: char with: aString to replace all occurrences of character char with replacement string aString. As an example,

```
'This is ~ true' replace: $~ with: 'not'          "Should produce 'This is not true'. "
```

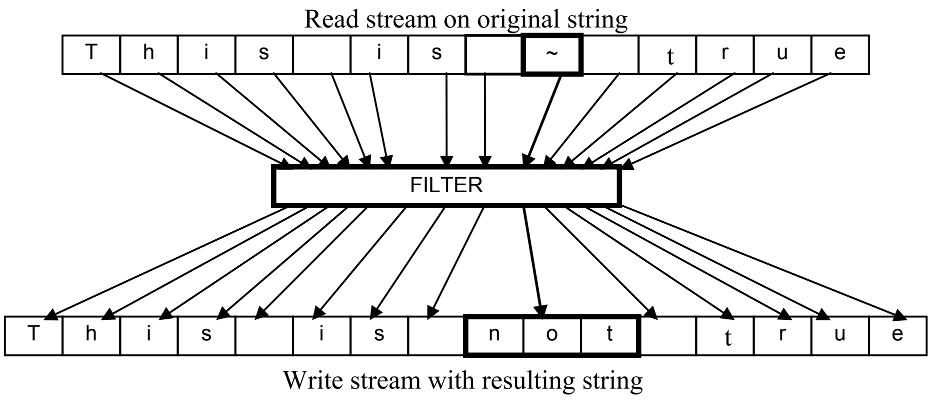

Read stream on original string
T h i s i s ~ t r u e
FILTER
T h i s i s n o t t r u e
Write stream with resulting string

Figure 10.7. Principle of solution of Example 2.

Solution: To solve this problem (Figure 10.7), we will create a ReadStream on the string entered by the user, create an uninitialized WriteStream of suitable size for creating the output, and process the ReadStream one character after another, copying all characters except for $~ into the WriteStream, and replacing every $~ character with 'not'.
We will put replace: char with: aString in class CharacterArray and its definition is as follows:

```
replace: char with: aString
"Replace all occurrences of char with aString."
| output input |
"Open ReadStream on string."
input := ReadStream on: self.
```

```
"Create a WriteStream on a String."
output := WriteStream on: (String new: self size).
input do: [:ch | "Use the stream to build the result."
            ch == char
                        ifFalse: ["Make replacement on match."
                                    output nextPut: ch]
                        ifTrue: ["Leave other characters unchanged."output nextPutAll: aString]].
"Returns processed string."
```

This is a typical use of internal streams - scanning a ReadStream and constructing a WriteStream a piece at a time. Note the use of contents to obtain the resulting string. Since the elements of the underlying String are characters, we use nextPut: to enter the unchanged *characters* but nextPutAll: to enter the *string* ' aString as a sequence of characters.

Example 3: Constructing a string using a stream

Problem: As an experiment with the use of streams, write a code fragment to create a simple personalized letter from a pre-stored template. The letter is a reminder that a book borrowed from the library is overdue and it should have the following form:

May 23, 1997

Dear Ms. Jones,

I would like to remind you that the book borrowed from the Xaviera Library is now **overdue**.

Yours,

Ivan Tomek
Adjunct Librarian

The program should automatically calculate the date, the user selects one of Mr. Mrs. or Ms. from a multiple choice dialog, and the names of the borrower and the Adjunct Librarian are entered by the user. (Unfortunately, I am usually the one who gets the reminders rather than the one who issues them.)
Solution: In this program - another typical application of internal streams - we will use a TextStream because it can handle emphasis and inherits messages for inserting carriage returns, tabs, and other useful characters. We start by opening a TextStream of suitable size, construct the text from strings that are either predefined or calculated or selected by the user, and return the resulting Text object. The principle is simple and the code is as follows:

```
| labels letter |
labels := #('Miss' 'Mr.'  'Mrs.' 'Ms.').
"Create a TextStream on a String of suitable length."
letter := TextStream on: (String new: 170).
"Construct letter."
letter emphasis: nil; cr; cr;
   nextPutAll: (Date today printString); cr; cr;
   nextPutAll: 'Dear ';
   nextPutAll: (Dialog
            choose: 'Which one do you want?'
            fromList: labels
            values: labels
            lines: 4
            cancel: ['']);
            space; nextPutAll: (Dialog request: 'Enter borrower''s name' initialAnswer: '');
   nextPut: $,; cr; cr; tab;
   nextPutAll: 'Please note that the book which you borrowed from our Library is now '; cr;
   crtab: 3; emphasis: #(#bold #underline); nextPutAll: 'overdue';
   emphasis: nil; cr; cr; cr; "Set and clear emphasis"
   nextPutAll: 'Yours,'; cr; cr; cr;
   nextPutAll: (Dialog request: 'Enter Adjunct Librarian''s name' initialAnswer: ''); cr;
   nextPutAll: 'Adjunct Librarian'
```

Test the program and print the letter – the contents of the letter TextStream. Use class Document. Note again that this example is only an illustration of the use of streams. To implement the problem of creating form letters, we would have to create one or more classes to perform the task in a more general context.

Example 4: An example of TextStream methods

As an example of how TextStream handles character oriented operations, the definition of cr inherited from Stream is

```
cr
"Append a return character to the receiver."
self nextPut: Character cr
```

and this is then used with the tab method to define crtab as follows:

```
crtab
"Append a return character, followed by a single tab character, to the receiver."
   self cr; tab
```

Example 5: Skipping up to a specific character

A compiler skips over characters such as spaces and line feeds which don't have any effect on execution. This is implemented by messages such as skipTo:, upTo:, and others. We will now illustrate this principle by reading a string entered by the user and converting it into an array of strings corresponding to sections of the original terminated by $-. As an example, if the user enters the string

```
 'This is-not-my day'
```

the program will convert it to

```
#('This is'   'not'   'my day')
```

The basis of the solution is message upTo: anObject which returns the collection of objects preceding the next occurrence of anObject or the tail of the stream; it returns an empty collection when issued at the end of the collection. The message sets the pointer to anObject so that next access starts just behind it. Our program again first opens a ReadStream on the original string, and then constructs the resulting collection by streaming.

```
|stream  collection string |
“Create OrderedCollection to hold the result – we cannot predict the eventual size.”
collection := OrderedCollection new.
stream := ReadStream on: (Dialog request: 'Enter text using - as separator' initialAnswer: ‘’).
[(string := stream upTo: $-) isEmpty]“Get next piece of string. Stop at end of stream.”
          whileFalse: [collection addLast: string].
collection asArray                          “Convert because the specification required an Array.”
```

Example 6: Using with: to access the whole underlying stream

When you create an instance of ReadWriteStream *on* an existing stream, its position is initialized to 0. As a consequence, a message such as

```
(ReadWriteStream on: 'A string') contents
```

returns an empty string and stream size returns 0. If you then add a new element with nextPut:, it will replace the first element of the original collection, and repeated use of nextPut: will eventually destroy all original data. If you want to be able to access the contents of the whole underlying collection or add elements at the end, use the with: creation message as in

```
| rwStream |
(rwStream := (ReadWriteStream with: 'A string') nextPutAll: '!!!'; yourself) contents.
rwStream nextPutAll:  ' And another string!!!'.
rwStream contents "Returns 'A string!!! And another string!!!'"
```

Example 7: The use of internal streams is not limited to strings

Although internal streams are used mainly for operations on strings, they work with collections of any objects. In fact, enumeration methods such as collect: and select: defined in class SequenceableCollection are based on internal streams. As an example, method reverse which returns a copy of a collection with its elements in reverse order is defined as follows:

```
reverse
"Answer a new sequenceable collection with its elements in the opposite order."
| aStream |
	aStream := WriteStream on: (self species new: self size).
	self size to: 1 by: -1 do:
		[:index | aStream nextPut: (self at: index)].
	^aStream contents
```

Example 8: Streams can make code more readable

Since operations on streams are actually operations on their underlying collections, what do we gain by using streams? One advantage of streams is conceptual clarity and simplicity. As an example, the following two code fragments have exactly the same effect but the second formulation is more natural, simpler and less error prone because we don't have to deal explicitly with the position pointer:

```
“Displaying selected elements of a collection. Implementation with collection.”
|array position|
array := #('a' 'b' 'c' 'd' 'e').
position := 1.
Transcript show: (array at: position); cr.
position := position + 2.
Transcript show: (array at: position); cr.
position := position + 1.
Transcript show: (array at: position); cr.
etc.
```

```
“Displaying selected elements of a collection. Implementation with stream.”
| array stream |
array := #('a' 'b' 'c' 'd' 'e').
stream := ReadStream on: array.
Transcript show: (stream next); cr.
stream skip: 1.     “Note that we had to increment the pointer by 2 in the previous version.”
Transcript show: (stream next); cr.
Transcript show: (stream next); cr.
etc.
```

Example 9: Stream operations are often more efficient

A classical example where streams improve execution speed is concatenation. The following two code fragments produce the same string but the implementation with concatenation is many times slower than the implementation with streams.

```
“Test of concatenation. Implementation with string concatenation.”
Time millisecondsToRun: [
	| string |
	string := 'abcdefg'.
	1000 timesRepeat: [string := string , 'abcd']] “Returns 181 on my laptop.”
```

```
“Test of concatenation. Implementation using internal stream”
Time millisecondsToRun: [
        | string stream |
        string := 'abcdefg'.
        stream := WriteStream on: (String new: 8000).
        stream nextPutAll: string.
        1000 timesRepeat: [stream nextPutAll: 'abcd']] “Returns 5.”
```

The reason why concatenation is very inefficient is that it creates a new string containing a copy of the original and then adds the argument string to it. Don’t use concatenation if you must repeat it more than a few times and if execution speed is important.

Main lessons learned:

- Internal streams are used mainly (but not exclusively) for operations on strings.
- Class TextStream adds emphasis handling to inherited character-oriented text operations.
- Appropriate use of internal streams makes programs simpler and often more efficient.

Exercises

1. Implement the problem in Example 2 with Collection methods and compare the two solutions.
2. Implement the problem in Example 2 with String methods and compare the two solutions.
3. Can you implement Examples 2 and 3 with class StringParameterSubstitution? Note that this implementation of string replacement is also based on streams.
4. Explain the definition of printString with your current background on streams.
5. What will happen if you open a ReadStream and a WriteStream over the same collection and use the two streams alternatively?
6. Browse uses of ReadWriteStream.
7. What happens when you execute nextPut: after reaching the last element and the underlying collection is not large enough?
8. Arrays cannot grow or shrink. What happens when you add an element to a stream whose underlying collection is an array?
9. Write method skipSeparators: aCollection to skip all elements included in aCollection.
10. The Transcript - an instance of TextCollector - is a major application of internal streams. In essence, a TextCollector is a value holder for the Transcript window and its contents are accessed via a write stream. This is why some parts of the Transcript protocol are identical to the protocol of internal streams. Write a short description of TextCollector focusing on its relation to internal streams.

10.4 Example: A Text filter

In Examples 2 and 3 in the previous section, we needed to replace strings, sometimes obtained by evaluating a block. In other words, we needed to filter input text and transform it into new text. This seems like a generally useful functionality and we will now implement it as a new class called TextFilter.

Specification: Class TextFilter takes an initial String object and replaces occurrences of any one of matching substrings with a corresponding String or Text object. Replacement objects are specified as String or Text objects or as blocks that calculate String or Text objects.
Examples of application:

- A form letter could contain 'formal parameters' (in the terminology of StringParameterSubstitution) such as '<name>' and '<date>', and the filtering process would replace the first parameter with a string provided by the user, and the second by an expression calculating today's day.
- A text editor could provide an extended string replacement facility allowing the user to replace not just one string but any number of strings simultaneously.

Scenario
Assume original string = 'abcdefg' and match/replacement pairs pair1 = 'bc'->'xxx', pair2 = 'bed'->'y'.

1. Set current position in string to 1. Compare $a with the first character of pair1 key (no match) and first character of pair2 key (no match).
2. Increment position in string, compare with first character in both pairs, find match in both.
3. Increment position in string, compare with second character in both pairs, find match in both. pair1 match is complete, perform replacement, reset matching for both pairs.
4. Increment position in string, compare with first character in both pairs, and so on.

Preliminary Design: The specification can be implemented with a class-tool and the only questions are how to represent the necessary parameters and how to perform the replacement. We will implement the replacement by scanning the given string character by character and matching it against all match strings at each step. When a match is found, the corresponding replacement is made and the search continues from the next character of the original string. All partial matches are reset at this point.

Considering this principle, we immediately see that the state of processing and additional parameters require the following information:

- The original string and our current place in it.
- The new string as constructed up to this point and our current place in it.
- A collection of match strings and their replacements (strings, texts, or blocks)
- For each match/replacement pair, remember currently reached position in matching.

Design Refinement: We will now decide on the details of the components identified in Preliminary Design, and construct the replacement algorithm.

- The original string is accessed one-element-t-a-time and we will access it through a ReadStream. This also takes care of keeping track of the current location in the string.
- For the same reason, we will access the new string through a WriteStream.
- The obvious storage for strings and their translations is a dictionary with the match string as the key and the replacement string as the value. When we consider that we must also keep track of how much of the match string has been checked, we decide to hold this information also as a part of the value. Altogether the dictionary elements will be match string -> Array (replacement value, position).

The replacement algorithm will be as follows:

1. Create a ReadStream over the original string and a WriteStream over the string being constructed. Initialize the second element of the value array of each element to 0.

```
2. Repeat for each position of the input stream beginning from the start:
   a. For each element of the dictionary do:
      i. Increment current position holder for match string.
      ii. Compare input string and match character.
         1. If no match, reset current position holder for match string to 0.
         2. If match, check if this is the last character to match.
            If this is the last character (match succeeded), make replacement in output stream,
            reset current position holder for match string in all dictionary entries to 0, and repeat
            Step a.
            If this is not the last character (match incomplete), increment current position holder.
```

The intent is to perform filtering is a one-step operation – by submitting a string with all filter parameters, executing the message without interruption, and receiving the result. We will thus never need more than one instance of the filter at a time and we will implement the method as a class method, somewhat like sort: in SequenceableCollectionSorter[3].

We now have all necessary information except for the placement of TextFilter in the class hierarchy. Since there are no related classes, we will make TextFilter a subclass of Object.

Implementation:
The comment of TextFilter is as follows:

I implement general filtering of text. To create an instance, I need the original string and two arrays consisting of strings to be matched, and replacements. Replacement values may be string or text objects or blocks. My filtering method returns the result without affection the original.

Class Variables:

InputStream	<readstream>	streams over input string
Outputstream	<WriteStream>	used to build filtered string
MatchDictionary	<String, Text, Block>	used to do replacements

TextFilter will implement all its functionality via class method filter: aString match: matchArray replace: replaceArray. The definition strictly follows the algorithm outlined above but we will restrict our implementation to string replacements and leave extension to Text and BlockClosure arguments as an exercise. The definition is as follows:

```
filter: aString match: matchArray replace: replacementArray
"I filter aString using matchArray and replacementArray, and return the resulting String ."
	"Initialization."
	MatchDictionary := Dictionary new.
	matchArray with: replacementArray do:
		[:match :replace | MatchDictionary at: match put: (Array with: replace with: 0)].
	InputStream := ReadStream on: aString.
	OutputStream := WriteStream on: (String new: aString size).
	"Filtering."
	[InputStream atEnd] whileFalse: [self matchAndReplace].
	^OutputStream contents
```

Most of the work is done by class method match which takes a single character from the input stream and tries to match it. Its definition is

```
matchAndReplace
"Get next character, match it against all dictionary entries, and do replacement if necessary."
	| ch |
```

[3] Defining behavior via class methods is generally frowned upon by Smalltalk experts because it may complicate specialization via subclassing. In our example, we are following the philosophy of the sorting mechanism in class SequenceableCollectionSorter which serves a similar purpose.

```
        ch := InputStream next.
        “Copy the input character into the output stream for now.”
        OutputStream nextPut: ch.
        “Now try to match against successive entries in the dictionary.”
        MatchDictionary
                keysAndValuesDo:
                        [:key :value |
                        | index |
                        “Get index of next character in this dictionary entry.”
                        index := (value at: 2) + 1.
                        “Check if it equals the input character.”
                        ch == (key at: index)
                                ifTrue: [index = key size       “We have a match. Did we match the whole
                                                                replacement value?”
                                                        ifTrue: “We matched the whole value.”
                                                                “Go back in output stream for replacement.”
                                                                [OutputStream skip: key size negated.
                                                                “Put replacement into output stream.”
                                                                OutputStream nextPutAll: (value at: 1).
                                                                “Reset match positions in all entries.”
                                                                MatchDictionary do: [:valueArray |
                                                                                valueArray at: 2 put: 0].
                                                                                “Done with this character.”
                                                                                ^self]
                                                        ifFalse: “Not end of matching yet - update index.”
                                                                [value at: 2 put: index]]
                                ifFalse:    “No match, reset index in this entry to 0.”
                                            [value at: 2 put: 0]]
```

This seems a bit long but that’s mainly because of our copious comments. To test the method, I executed the following test code

```
|matchArray replacementArray|
matchArray := #('ab' 'eab').
replacementArray := #('xx' 'yy').
TextFilter filter: 'abcdeab' match: matchArray replace: replacementArray .
```

with *inspect* and got 'xxcdexx' which is not quite what I expected - I hoped for the ‘better’ match ‘xxcdyy’. (Essentially, by ‘better’ I mean ‘more compressed’.) What is the problem?

In fact, the problem is with our specification. What is happening is that in our example that the ‘ab’ -> ‘xx’ replacement is made before the method can make the nicer ‘eab’ -> ‘yy’ replacement. We should have said that if several replacements are possible in a given pass, one of those that give the longest replacement will be made. Implementing this specification would have produced the ‘expected’ result. We will formulate a better specification and develop a solution in the next chapter.

Could we have avoided our mistake? If we executed a scenario corresponding to our example in its entirety, we would have noticed the problem. The conclusion is that not only the implementation but also the design and even the specification must be tested.

Main lessons learned:

- When we know that we will never need several instances of a class, we can implement its functionality as a class protocol.
- Before you conclude that your design is incorrect, make sure that your specification is correct and complete. Better still, make sure that your specification is correct before you start design. A good way to obtain this assurance is to completely execute a set of exhaustive scenarios.

Exercises

1. Extend TextFilter to accept blocks as replacement arguments as stated in the specification.

10.5 Example: Circular Buffer

In computing terminology, a *buffer* is a memory area that accepts data from one process and emits it to another process; the two processes work at their own speeds. An example of the use of a buffer is reading a block of data from a file into memory where it is processed one byte at a time by a program. Another example is a computing node on a network that accepts parcels of data arriving in unpredictable amounts and at unpredictable times, processes them one byte at-a-time, and possibly sends the data on to another network node.

The hardware implementation of buffers often has the form of a special memory chip with a fixed number of memory locations with pointers to the first byte to be retrieved, and to the location where the next byte is to be stored as in Figure 10.8. When a new byte arrives, it is stored at the next available location and the pointer is incremented, and when a byte is required from storage, it is removed from the location pointed to and the pointer incremented.

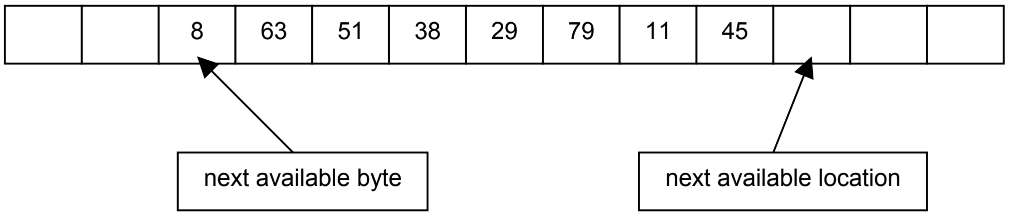


Figure 10.8. Buffer as a fixed size array with pointers to the next available byte and the next available location.

In reality, of course, a byte read from the buffer is not 'removed' and only the changed value of the pointer indicates that the byte has been used. Similarly, an 'empty' position is not really empty but the new value simply overrides the old value. Finally, when a pointer reaches the end of the array, the buffer is treated as if it were circular, as if its beginning were glued to its end, and when the pointer reaches the end, it 'increments' by being repositioned to the start. In mathematical terms, incrementing is performed in modular arithmetic as the remainder of division of the position by the size of the buffer.

The buffer does not, of course, have to be a special hardware chip and, in fact, it usually is not. Instead, it can be just a memory region that emulates the circular buffer area. Implementing this pretend circular buffer structure is the purpose of this section.

Problem. Implement a circular buffer based on a fixed-size array. Instances of the CircularBuffer class implementing this structure must be able to return the next available byte as a result of executing the next message which also updates the internal pointer, and to store a byte in response to nextPut:, again automatically updating the internal pointer. The buffer can also be tested with messages isEmpty and isFull.

Solution. If it wasn't for the very suggestive choice of message names, our first impulse would probably be to implement CircularBuffer as some kind of collection. On second thought, it becomes clear that CircularBuffer is not a collection but rather a mechanism for accessing the collection hidden inside it. Since the access is 'linear', this immediately suggests that CircularBuffer is a stream. We will thus implement it as a part of the Stream hierarchy.

The next question, of course, is where to put it in the Stream tree. To answer this question, let's start from the top and go down only as far as necessary to inherit useful behaviors. Class PeekableStream adds the ability to peak ahead but if we emulate the model of a hardware chip, such functionality should not be present and we conclude that we should subclass CircularBuffer directly to Stream.

The last question before we start implementing the class is what functionality it should implement. According to the specification, we need an accessing protocol (next, nextPut:), a testing protocol (isEmpty, isFull), and it will be useful to implement enumeration (message do:) for consistency with other streams and for printing. A printing protocol is necessary for the inspector and for testing, and initialization is required to back up the creation protocol. Printing obviously enumerates over all elements in the buffer and we will thus need an enumeration protocol. With this, we can now start implementing the class.

The class will need an instance variable for the array that holds the data (array), pointers to the first available location and the first available element (firstLocation and firstElement), and it will be useful to have a variable to hold the state (isEmpty). The modular arithmetic that we will need for updating indices will require the size of the underlying array. We will keep it in an instance variable so that we don't have to retrieve it every time and since the buffer may not be full at all times, we will refer to it as capacity.

The creation message will create an instance with an array of the specified size and initialize the remaining instance variable

```
new: anInteger
  ^self basicNew initialize: anInteger
```

where

```
initialize: anInteger
  array := Array new: anInteger.
  capacity := array size.
  firstIndex := 1.
  lastIndex := 1.
  isEmpty := true
```

initializes the instance variables in an obvious way. A simple test such as

```
CircularArray new: 10
```

executed with *inspect* confirms that everything is OK so far.

What should we implement next? We cannot do anything without nextPut: and next, and these require testing for empty and full so we will first implement the testing methods. Method isEmpty simply returns the value of isEmpty but isFull requires calculation. The buffer is full if the firstLocation has been pushed far enough to coincide with firstElement and so

```
isFull
"Are all slots occupied?"
  ^(firstElement = firstLocation) and: [isEmpty not]
```

because the two pointers will coincide not only when the buffer is full but also when it is empty.

With these two methods, we can now implement next and nextPut:. Method nextPut: adds a new element if the buffer is not yet full. After the test, it then puts the new element into the first available location and updates the pointer:

```
nextPut: anObject
"Add new element if there is room, otherwise execute exception block."
  self isFull ifTrue: [^self error: 'Buffer is full'].
  array at: firstLocation put: anObject.
  self moveFirstLocationIndex.
  ^anObject
```

Moving of the first location pointer is left to another method which increments the pointer using modular arithmetic and adds 1 because modulo n arithmetic counts from 0 to n-1 whereas arrays are numbered from 1 to n-1:

```
moveFirstLocationIndex
"An element has been added, 'increment' firstIndex."
  firstLocation := (firstLocation rem: capacity) + 1.
  isEmpty := false
```

Method next first checks whether the buffer is empty and if it is not, it returns the element at the pointer location and updates the pointer:

```
next
"Return next element and move pointer, return nil if empty."
  ^isEmpty
        ifTrue: [nil]
        ifFalse:
              [| el |
              el := array at: firstElement.
              self moveFirstElementIndex.
              el]
```

Here incrementing is done with modular arithmetic as follows:

```
moveFirstElementIndex
"Element was removed, update firstElement."
  firstElement := (firstElement rem:  capacity) + 1.
  isEmpty := firstElement = firstLocation
```

Finally, we can now implement printing, in other words method printOn: aStream. The desired format is

```
CircularBuffer (13 25 11)
```

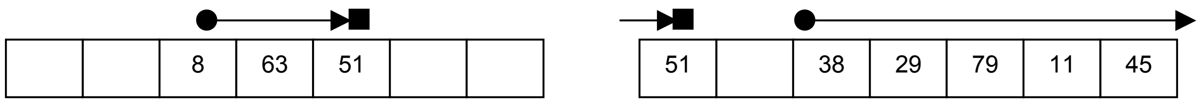
which hides how the data is arranged internally and shows the first element to be retrieved next as the first element inside the brackets, in this case 13. The definition is simple

```
printOn: aStream
"Append to the argument aStream a sequence of characters that identifies the collection."
  | first |
  aStream print: self class; nextPutAll: ' ('.
  first := true.
  self do:  [:element |  first  ifTrue: [first := false]
                                ifFalse: [aStream space].
              element printOn: aStream].
  aStream nextPut: $)
```

if we have a do: message that processes the elements starting with the first available element and ending with the last one. This operation is implemented as follows:

```
do: aBlock
"Evaluate aBlock with each of the receiver's elements as the argument."
        self isEmpty ifTrue: [^self].
        firstElement >= firstLocation
              ifTrue:
                    [firstElement to: capacity do: [:index | aBlock value: (array at: index)].
                    1 to: firstLocation - 1 do: [:index | aBlock value: (array at: index)]]
              ifFalse: [firstElement to: firstLocation - 1 do: [:index | aBlock value: (array at: index)]]
```

The principle of this method is that if the buffer is not empty, the index of the first available element is either less then the index of the last available element or the opposite is true (Figure 10.9). The handling of these two cases can be deduced from the diagram.



firstElement
firstLocation
firstLocation
firstElement

Figure 10.9. The two possible relative positions of firstElement and firstLocation. Arrows point from first available element upward. Filled circle denotes the first lement, filled square is the last element.

Exercises

1. We have cached the value of size and isEmpty in instance variables to avoid the need to recalculate them. Is there any advantage in caching isEmpty? Implement this modification and note that this internal change has no effect on the behavior of CircularBuffer or any other classes the use it.

10.6 Itroduction to files and external streams

External streams are the basis of operations on files and all other data transmission that occurs as a stream of bytes such as network data transmission. We will focus on the use of external streams with files which allows operation on textual data, graphics, sound, and other digital information. In this section, we introduce the basics of file and external streams, and several examples of their use are presented in the following sections.

Smalltalk operations on files and directories are implemented by combining external streams and class Filename (Figure 10.10). The main purpose of *external streams* is to provide byte-oriented streaming access to data, the role of *Filename* is to construct filenames, allow checking whether a file exists, whether a filename name has the proper structure, creating a new directory or file, and perform other file-system related operations.

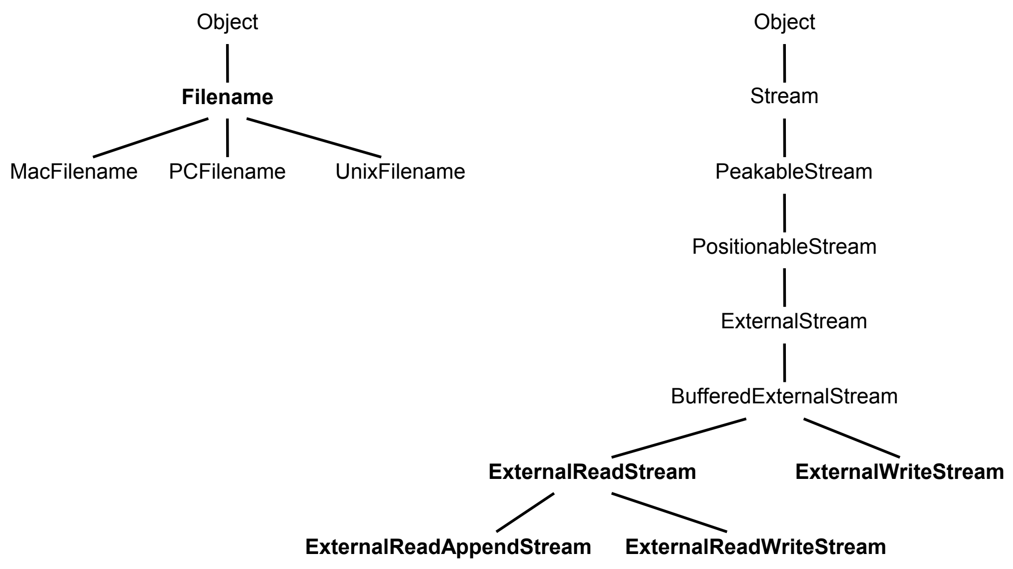


Figure 10.10. Main classes used in file processing.

The cookbook procedure for processing data stored in a file is as follows:

1. Create a Filename object with a filename string. The string is the name of the file that may include its drive/directory path.
2. Create the appropriate kind of external stream and associate the Filename object with it.
3. Perform byte operations on to the stream.
4. Close the stream object; this will close the file too.

Closing a file is very important for two reasons. One is that if a file is not explicitly closed, the data 'written to it' may not be stored on the disk. The second reason is that the operating system assigns to each file that it opens one of a limited number of 'handles'. Failure to close a file means that the handle is not released and if too many handles are in use, new files cannot be open. It may then be impossible even to save your work when leaving Smalltalk!

As a preliminary example demonstrating the above procedure and the role of external streams and Filename objects, the following code fragment opens a file for writing, stores some information in it, and closes the stream and its associated file.

```
|file fileStream|
file := Filename named: 'c:\testfile'.   "Open a file called 'testfile' in the root directory on drive C."
fileStream := file writeStream.          "Attach the file to a write stream (write only access)."
fileStream nextPutAll: 'abc'.            "Store the ASCII codes of 'abc' in the file buffer."
fileStream close                         "Flush buffer to disk and release OS handle."
```

Execute the program and open the file with the file editor to see that the file has indeed been created and contains the string 'abc'.

Although almost all Smalltalk applications use files, direct byte-oriented operations on files via streams as shown above are rare (except when reading data, possibly coming from a network) because Filename and external stream operations cannot directly store *objects*. Smalltalk programmers thus use files and external streams mainly as a vehicle for more powerful object-oriented tools such as BOSS (Section 10.9) and for operations on directories or files as a whole.

Class Filename and various external stream classes contain a large number of methods and we will present only the most important ones. Before we do, however, a few comments about the classes themselves.

Class *Filename* is an abstract class and its concrete subclasses (MaxFilename, PCFilename and its subclasses, and UnixFilename) implement the platform-specific behavior needed on your machine, such as parsing platform-specific syntax of file names. However, you never need to deal with these concrete subclasses because Filename automatically sends all platform-dependent messages to the subclass representing your platform. This is done via Filename's class variable DefaultClass which holds the name of the appropriate Filename class. Removing explicit dependence on one platform makes it possible to write programs that will run on different platforms. This arrangement is similar to the implementation of strings.

External streams perform data transfer operations. Instances of external streams are never created by class messages to external stream classes but by messages to the Filename object as in our example above. The Filename object, in turn, asks class ExternalStream to create and return the appropriate kind of stream; this procedure also opens the file.

After this brief introduction, we will now introduce class Filename and its essential protocols. We will then present external streams.

Main lessons learned:

- Byte-oriented file operations require the combination of Filename and an external stream.
- Filename objects provide interaction with the file system, external streams provide byte-by-byte access to file elements.
- Filename is an abstract class which transparently communicates with appropriate concrete subclass.
- Smalltalk programs rarely perform byte-oriented file access explicitly. To store and retrieve objects in files, use tools such as BOSS or a data base program.

Exercises

1. Examine and describe how Filename achieves passing of messages to its concrete subclass. Compare this with the similar behavior of Character.
2. We created a write stream by sending writeStream to Filename. Examine its definition.

10.7 Class Filename

Class Filename is an interface to the file system and provides access to files and directories. The essence of its comment is as follows:

Class Filename is an abstract class. Instances of its subclasses encapsulate the platform-specific syntax of OS file path names. This class can almost be used as a concrete class, except name syntax is not interpreted. There is standard protocol provided to do most of the things that OS's can do with references to files -- deleting, renaming, etc.

The best way to understand the role of Filename is to examine its protocols and file-related protocols in other classes.

Creating Filename objects

Filename objects can be created in two ways:

- By sending named: aString to Filename as in Filename named: 'prog'.
- By sending asFilename to a string as in 'c:\st\prog' asFilename.

In both cases, the string may be either a 'relative' specification (the first example) or an 'absolute' specification (the second example). In other words, the filename string may refer either to a file in the currently active directory or specify the complete path. As another example of relative specification, 'file.st' refers to the file called 'file.st' in the current directory, whereas 'c:\smalltalk\examples\example.1' specifies the path including the disk drive. Certain messages (but not creation messages) allow wildcard characters # (any single character) and * (any group of characters) inside a filename. As an example, in some contexts 'story.1#' refers to any string consisting of 'story.1' followed by a single character (such as 'story.12'), whereas 'story.1*' refers to any string starting with 'story.1' followed by zero or more characters (as in 'story.1', 'story.12' or 'story.123').

If you are developing an application that should run on several different platforms, you must consider that different operating systems use different separators between directories and file names in the filename path (in our example, we assumed the PC platform which uses \). To get the appropriate separator for an arbitrary platform, use message Filename separator; this way, the program can construct path name at run time in the appropriate way. Remember, however, that different platforms also have different rules for the maximum filename length. To get the maximum filename length for a platform, execute Filename maxLength. You can ignore these details if your application is designed to run on one platform only.

Filename prompts in class Dialog

Class Dialog provides several powerful requestFileName: messages in the *file name dialogs* protocol. All these methods prompt the user for a file name and return a *string* which can then be used to construct the Filename object as explained above. These messages also allow you to specify, for example, whether the file should be new (succeeds only if the file does not yet exist) or old (succeeds only if the file already exists). Some of these messages repeat prompting until the desired condition is satisfied, and some allow you to specify a block to be executed when the message fails. The simplest of these messages is requestFileName: which displays a prompt . It can be used as in

```
| file |
file := (Dialog requestFileName: 'Enter file name') asFilename
```

This message behaves just like the familiar Dialog request: but allows wildcard characters * and # in the answer. If the user enters a string with wildcard characters, the method displays a pop up menu containing the names of all matching filenames and allows the user to make a selection, try again, or abort by clicking *Cancel* (Figure 10.11). In the last case, the message returns an empty string and this must be kept in mind to prevent asFilename from crashing.

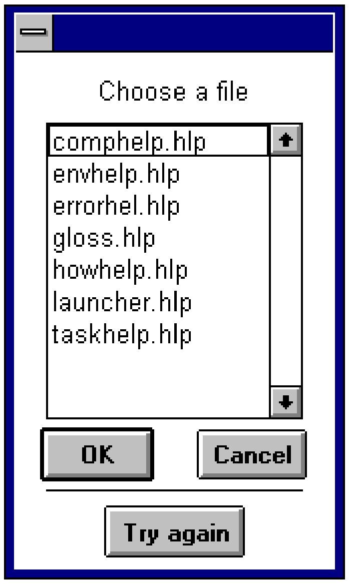


Figure 10.11. Possible result of typing '*.hlp' in response to Dialog requestFileName:.

The proper use of the combination of requestFileName: and asFilename should thus be something like

```
| file name |
name := Dialog requestFileName: 'Enter file name'.
name isEmpty ifTrue: [^self].
file := name asFilename.
etc.
```

A more powerful filename dialog message has the form requestFilename:default:. This message allows the specification of an initial filename as in

```
Dialog requestFileName: 'Select a file' default: '*.st'
```

An even more powerful version is requestFileName:default:version: which lets you specify not only the default filename but also its type. The version: argument may be #mustBeNew (user is asked how to proceed if the filename already exists on the specified path), #mustBeOld (user is asked what to do if the name is not found on the specified path), #new (user is warned if the file exists), #old (user is warned if the file does not exist), or #any. Yet another version of file prompt messages is the message requestFileName:default:version:ifFail: which includes an exception block to be executed when the 'version' condition fails in the case of #mustBeNew or #mustBeOld.

<u>Accessing operations</u> are scattered across several protocols and include the following methods:

contentsOfEntireFile - opens an external read stream on the file, gets its contents, returns it as a String, and closes the stream and the file. The user is not aware of the read stream created and closed during the operation. Note that we can also access the contents of a file by attaching it to an external stream and sending contents to the stream. However, message contentsOfEntireFile saves you from creating an external stream and closing it explicitly. The following example creates a new file, stores some data in it, closes the file, and gets and displays its contents.

```
|file fileStream|
“Create a file, put some text in it, and close it.”
file := Filename named: 'c:\testfile'.
fileStream := file writeStream.
fileStream nextPutAll: 'abc'.
fileStream close.
“Display file contents in the Transcript”
Transcript cr; show: (Filename named: 'c:\testfile') contentsOfEntireFile   “Displays the string ‘abc’.”
```

directory - returns the directory containing the file corresponding to the Filename receiver. As an example,

```
| file |
file := 'c:\abc\xyz’ asFilename.
file directory
```

returns an object such as a FATFilename (a concrete subclass of abstract class PCFilename for the MS-DOS operating system). Note that both files and directories are instances of Filename.

Class message defaultDirectoryName returns the *String* describing the full path of the current directory as in

```
Filename defaultDirectoryName                    " Returns, for example, 'c:\visual\image'"
```

The related class message currentDirectory returns the corresponding *Filename* directory object.

<u>Deleting, copying, moving, renaming, and printing files</u>

delete - as in fileName delete - deletes the Filename object (a file or a directory). As an example of its use, the following fragment creates and opens a file called 'test' in the root directory of drive C, stores data in it, closes it, displays the file's contents, and deletes the file:

```
|file fileStream|
"Create, initialize, and close a file."
file := Filename named: 'c:\testfile'.
fileStream := file writeStream.          “Create write stream on the file.”
fileStream nextPutAll: 'abc'.
fileStream close.
"Display file contents in the Transcript."
Transcript cr; show: (Filename named: 'c:\testfile') contentsOfEntireFile.
"Delete the file."
(Filename named: 'c:\testfile') delete
```

Note that delete must be sent to the Filename object - the stream does not understand delete.

renameTo: pathName renames the receiver Filename object, *and moves it* to a new directory if the new path is different from the old one; the original name is deleted. As an example,

```
| filename |
filename := Filename named: 'test'.
filename renameTo: 'c:\smalltalk\examples\example.1'.          “Renames and moves the file.”
```

copyTo: pathNameString creates a copy of the receiver under a new name, possibly in a new location. The original file and its name remain unchanged.

To *print* a text file, print its String contents. To print a PostScript file, use class Document.

Testing

exists - checks whether the receiver Filename exists and returns true or false. Note again that the receiver may be a file or a directory. As an example,

```
Filename defaultDirectoryName asFilename exists
```

returns true.

isDirectory - tests whether the Filename receiver is a directory or a file. Returns true for a directory, false for a file. As an example,

```
Filename currentDirectory isDirectory
```

returns true.

Directory operations

makeDirectory - creates a directory according to the specification in the Filename receiver as in

```
(Filename named: 'new') makeDirectory             "Creates subdirectory 'new' of the current directory."
(Filename named: 'c:\dos\new') makeDirectory      "Creates directory 'new' with the specified path."
```

dates returns an IdentityDictionary containing the dates of creation, last modification, and last access of the receiver - if these parameters are supported by the operating system. As an example,

```
| file |
file := (Dialog requestFileName: 'Enter file name' default: '*.st') asFilename.
file dates
```

returns an instance of IdentityDictionary with date information on a file selected by the user. On PC platforms, for example, this fragment will return something like

```
IdentityDictionary (#statusChanged->nil #modified->#(6 April 1993 1:59:50 pm ) #accessed->nil )
```

where nil values indicate that the corresponding parameter is not supported on the current platform.

directoryContents returns an array of strings, the names of files and subdirectories in the current directory. As an example,

```
Filename currentDirectory directoryContents
```

could return something like #('VISUAL.IM' 'VISUAL.SOU' 'VISUAL.CHA' 'WORKSP.2')

Main lessons learned:

- Class Filename supports operations such as deletion, renaming, copying, and closing of files and directories. It also provides tests and access to internal parameters such as the length of a file and the contents of a directory.
- Some file operations require only Filename, others also require an external stream.
- Filename can be used to create directories but creation of files requires an external stream.
- Class Dialog provides several file-related dialogs that search the directory for the specified filename, allowing wildcard characters and specification of the type of the desired file.

Exercises

1. Try requestFileName:default:version: with various values of version: and names of files that already exist/don't yet exist.
2. What happens when you send message directory to a Filename object and the directory with the specified name does not exist?
3. Create a table of all essential messages introduced in this section. For each method, specify whether it is a class or an instance method, what are its arguments, what object it returns, what are its preconditions, and what is its effect.
4. Why are defaultDirectoryName and currentDirectory class methods?
5. Define method deleteDirectory: aString ifFail: aBlock which checks whether aString is a directory name, deletes the directory if appropriate, and executes aBlock otherwise.

10.8 Examples of the use of file operations that don't require external streams

External streams are needed only for byte-by-byte access to files. Operations on directories, and operations on the contents of a file treated as a string do not require explicit use of external streams. This section gives several examples of such operations.

Example : List alphabetically all files in the current directory and their sizes
Solution: As we know, there are two messages to access to current directory. Message currentDirectory returns the current directory as a Filename object, and message currentDirectoryString returns a String containing the filename path of the current directory. Since we need the cntents of the file, we need the Filename object. We will thus use the currentDirectory message.

If you examine the Filename protocols, you will find that you can get the contents of a Filename directory object by sending it the directoryContents message. This message returns an Array of strings - names of the files and subdirectories in the receiver - and to sort it alphabetically, we will convert it to a SortedCollection. To obtain information on the corresponding files, we must create Filename objects over the individual string elements and ask them about their size using the instance message fileSize (returns the size of the file in bytes). The whole program is as follows:

```
|  names |
“Extract names and convert to sorted collection.”
names := Filename currentDirectory directoryContents asSortedCollection.
Transcript cr.
“Convert names individually to filenames and extract and print the desired information.”
names do: [:name | Transcript show: name; tab; show: name asFilename fileSize printString; cr]
```

Note that the program does not check whether the extracted names are names of files or directories and lists them all. We leave it to you to correct this imperfection.

Example 2: Test whether two files (two directories) entered by the user have the same contents
Solution: To check whether two files contain the same data, we don't need an external stream because we can compare the contentsOfEntireFile of both files:

```
| file1 file2 text1 text2 |
"Let the user select two files from the current directory."
file1 := (Dialog requestFileName: 'Select the first file.' default: '*.*') asFilename.
file2 := (Dialog requestFileName: 'Select the second file.' default: '*.*' ) asFilename.
text1 := file1 contentsOfEntireFile.
text2 := file2 contentsOfEntireFile.
text1 = text2
```

If the files are large, this program will work with two large objects and take a long time to execute. Using streams explicitly may then be preferable.

Example 3: Let user delete a file from a list

Problem: Implement a method to display the file names in the current directory in a multiple choice dialog, and allow the user to delete a file

Solution: This problem does not require a specific Filename and we will implement it as a class method in Filename, following the example of several existing *fromUser methods. The method will obtain the current directory, display its contents in a multiple choice dialog asking the user which file to delete, and delete the file if the user makes a selection. The implementation is as follows:

```
deleteFromUser
"Display dialog with names in current directory and allow user to delete one."
	| choice fileNames |
	"Display dialog with names of all files in the current directory."
	fileNames := Filename currentDirectory directoryContents asSortedCollection.
	choice := Dialog
		choose: 'Which file do you want to delete?'
		fromList: fileNames
		values: fileNames
		lines: 20
		cancel: [''].
	"If the user selected a file, delete it."
	choice isEmpty ifFalse: [choice asFilename delete]
```

Main lessons learned:

- File and directory operations that don't require explicit byte-oriented access can be performed without external streams. These operations include operations on entire contents of a file, deleting, renaming, accessing contents, and similar operations.

Exercises

1. Refine Example 1 to distinguish between files and subdirectories. Your version of the program should print 'directory' instead of the size for those filenames that are directories.
2. Define a new method called = to test whether two files or directories have the same contents.
3. Why does Example 2 take so long to execute for larger files?
4. The method in Example 3 is not a safe way to delete files and it does not distinguish between files and directories. Write a new version that will request a confirmation and ask the user whether to delete a subdirectory if it is not empty.
5. When you use named: to create a new Filename object on a PC platform, the name is reduced to at most 8 characters. How does this happen? Since this can be a problem with names of drives on networks, can it be avoided? (Hint: Try another creation method, possibly inherited.)

10.9 External streams

We have seen that Filename and external stream functionalities somewhat overlap. If you find it confusing, the rule of thumb is that creating external streams and attaching them to Filename objects is necessary only to access the contents of the file in a streaming fashion or to store or access objects created by BOSS. External streams are not necessary for operations on whole files and directories.

We have already seen that the attachment of a stream to a file is performed by asking a Filename to create a stream of the desired kind. The following messages are available for this purpose: appendStream, newReadAppendStream, newReadWriteStream, readStream, readAppendStream, readWriteStream, and writeStream. Each of them creates a different kind of stream over the same file and will now explain them briefly. A summary table is provided below.

Creating external streams

- **appendStream** opens an 'append stream', a file that allows only sequential writing at the end. As an example, create file 'test' containing the string 'abc' using the file editor. The following program

```
|file fileStream|
file := 'test' asFilename.
fileStream := file appendStream.              "Attach file to an append stream."
fileStream nextPutAll: 'xyz'.          "Store the ASCII codes of 'xyz' in the file buffer."
fileStream close                              "Close file via its associated stream."
```

  opens the file for appending, writes the three characters 'xyz' at the end, and closes the file. The file now contains 'abcxyz'. Check this by opening an editor on the file.
- **newReadAppendStream** opens an ExternalReadWrite stream at the beginning of the file for unrestricted reading, but writing is restricted to appending at the end. For reading, the file can be positioned with position: but this has no effect on writing. The message clears (erases) all original contents if the file already exists; this is suggested by the word new in the name of the method.
- **newReadWriteStream** opens a new read/write stream that can be randomly positioned for both reading and writing using position:. The word new in the name of the method indicates that if the file existed before the message was sent, the original contents are deleted. Writing to a position within the file replaces the old byte with the new value.
- **readAppendStream** has the same properties as newReadAppendStream but does not reset and clear the receiver file.
- **readStream** can only read an existing file and is fully positionable.
- **readWriteStream** opens a read/write stream on a new or existing file without deleting the old contents. This is indicated by the absence of new in the name of the message. The message opens the file at the beginning and allows arbitrary positioning. It behaves as newReadWriteStream in all other respects.
- **writeStream** opens a purely sequential write stream at the beginning of an existing or new file. If the file existed, all data is lost. The stream does not understand any positioning messages and cannot be read.

	readable	writeable	positionable	append only	clears original
appendStream	no	yes	no	yes	no
newReadAppendStream	yes	yes	reading	yes	yes
newReadWriteStream	yes	yes	yes	no	yes
readAppendStream	yes	yes	reading	yes	no
readStream	yes	no	yes	n/a	no
readWriteStream	yes	yes	yes	no	no
writeStream	no	yes	yes	no	no

Table 10.1. Properties of external streams.

To understand how stream messages work, it is useful to examine the definition of newReadAppendStream which is as follows:

```
newReadWriteStream
"Answer a new readWrite stream connected to the file represented by the receiver."
    ^ExternalReadWriteStream on:
            (FileConnection
                    openFileNamed: self
                    mode: #readWrite
                    creationRule: #truncateOrCreate)
```

This explains how the limited number of external stream classes (Figure 10.8) can provide such a variety of accessing modes - the type of access is controlled by an instance of FileConnection. The other stream creation messages are similar.

Since a file and its mode of access are two separate things, a file initially accessed via one kind of stream may be closed and accessed again via another type of stream. As an example, we have already seen that you may open a file for writing, store some data in it, close it, and open it for reading later.

Operations on external streams

The following are the main operations on external streams:

Accessing

Includes reading or writing of individual bytes to or from the buffer and control of the buffer itself. The operation of accessing messages depends on the principle of the interface between Smalltalk's external streams and the operating system, and between the operating system and disk storage. This will be explained next.

As we already mentioned, external streams are 'buffered' which means that the stream object holds on to the part of the file which it is currently accessing via its instance variable ioBuffer. When accessing operations fill the stream buffer, its contents are automatically sent to the operating system and the buffer is reset. You can also perform this operation explicitly by sending flush, commit, or close to the stream (see below). For read-only streams, the buffer is just a multi-byte window into a file stored on the disk.

In addition to the buffer kept by Smalltalk, the operating system maintains its own buffer which operates in a similar way but is under the control of the operating system. Sending the contents of the stream buffer to the operating system thus writes to the *operating system's buffer* but it does not guarantee that the contents of the buffer is written to the disk ('committed'). Messages close and commit perform even this task.

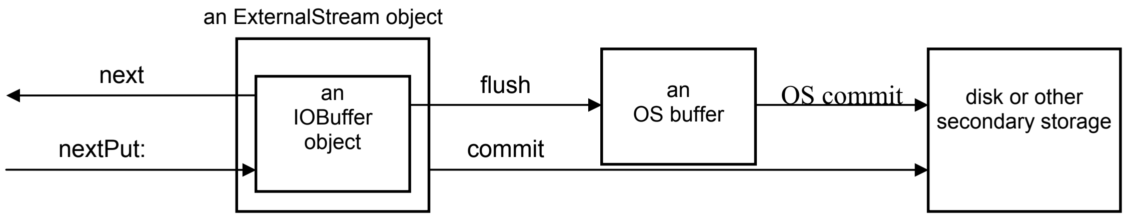


Figure 10.12. Data transfers resulting from various stream messages.

The buffer flushing, committing, and accessing messages are defined on writeable streams (Figure 10.12). Note again that these operations work on external streams, not on FileName objects! Note also that flush and commit are needed only for explicit buffer operations and that these operations happen automatically when the corresponding buffers become full.

- flush - sends the bytes accumulated in the stream's memory buffer to the *operating system*.
- commit - writes the contents of stream's buffer to the *disk*.
- next, nextPut: - streaming access in the style of next and nextPut: messages of internal streams. Operate on the contents of the stream buffer and flush or refill it when necessary.

We have already mentioned that external streams are normally read one character at a time; in other words, they are byte-oriented. They can, however, also be accessed in a bit-wise fashion. To access external streams one *bit* at a time, send message binary to the stream. To change bit access back to byte access, send text to the stream. Note that although you can change stream access from byte-oriented to bit-oriented and vice versa while the file is open. On the other hand, you cannot change the type of stream (for example from write only to read only); to do this, you must close the file and create the desired new kind of stream.

Positioning

- position, position:, setToEnd – work the same way as for internal streams.

In closing, we will now illustrate the difference between the various writeable external streams on a short example.

Example: Behavior of writeable external streams

In this example, we assume that the underlying file called 'test' already exists and contains the string '123456789'. Each example is executed from this initial state.

- After executing the following code fragment that uses **appendStream**

```
| stream |
stream := (Filename named: 'test') appendStream.
stream nextPutAll: 'abc'.                    "Store characters $a, $b, $c at the end of the file."
stream close
```

  the contents of the file become '123456789abc'. The new data have been appended at the end, the old data have not changed. Neither positioning nor reading are possible.
- When you change the previous program to use **newReadAppendStream** as in

```
| stream |
stream :=(Filename named: 'test') newReadAppendStream.
stream nextPutAll: 'abc'.             " Store characters $a, $b, $c in the file ."
stream close
```

  the contents of the file become '123456789' to 'abc'. The old data is thus lost. The stream can be positioned with position: but positioning affects only reading. Writing always occurs at the end of the file.
- With **newReadWriteStream**, writing erases the original contents of the file. The pointer can be repositioned for reading and for writing within the limits of the new contents.

```
| stream |
stream := (Filename named: 'test') newReadWriteStream.
stream nextPutAll: 'abc'.             "The stream now contains three characters $a, $b, $c."
stream position: 1; nextPut: $X. "Replaces the second element."
stream close
```

  changes the contents from '123456789' to 'aXc', erasing the original contents. Message nextPut: issued after setting position to 1 overwrites the element in position 2 because the value of position is incremented before writing takes place.
- With **readWriteStream**, we don't lose the original contents of the file.

```
 |stream |
stream := (Filename named: 'test') readWriteStream.
stream position: stream size.    "Position at end."
stream nextPutAll: 'abc'.
stream position: 1.
stream nextPut: $X.
stream close
```

  changes the original contents '123456789' to '1X3456789abc'. The stream is fully positionable.
- Finally, a **writeStream** allows positioning and writing starts from the beginning of the file. The original contents of the file are lost.

```
| stream |
stream := (Filename named: 'test') writeStream.
stream nextPutAll: 'abc'.
stream close
```

changes the contents of the file to 'abc'.
Finally, note that we have been careful to close the file stream when it was no longer needed.

Main lessons learned:

- Several kinds of external streams can be opened by sending the appropriate stream creation message to the Filename object.
- External streams differ in the kind of access (read-only, write-only, read-write) and the kind of positioning (random, sequential only, append only) that they provide.
- Stream messages operate on a part of the file stored in a buffer. The contents of the buffer is flushed to the operating system or committed to the disk only when the buffer fills, when a flush or commit message is sent, or when the file is closed.

Exercises

1. How is it that the reading position of a ReadAppendStream can be changed but writing always occurs at the end of the file?
2. We have seen that different kinds of file access are obtained by collaboration with class FileConnection. Write a short descripion of this class.
3. For each task listed below, write the message that will open the file for the specified purpose assuming that the file is named 'test' and is stored in directory c:\binary.
   a. Append new data to the end of the file.
   b. Empty the file and write new data into it in sequential order.
   c. Open the file, add data at the end, and read data anywhere in the file.
   d. Open the file without losing the existing data and write new data anywhere in the file. No reading is anticipated.
   e. Same as the previous situation but you want to be able to read the data randomly too.
4. One of the numerous suggestions for extensions of VisualWorks tools is adding *save* and *load* commands to the <operate> menu in the Workspace. Implement these extensions as described below. (Hint: Use the Resource Finder tool to examine the menu bar of the Visual Launcher to find how it opens a Workspace.)
   a. Command *save* opens a file dialog and when the user *accepts*, the contents of the whole Workspace window is saved in the specified file. The *load* command works similarly but adds the contents of the file to the current Workspace contents.
   b. Add command *save it* to save only the currently selected part of the Workspace.
5. Define an internal read-append stream that stores its contents in an external stream and resets itself when it reaches a specified size.

10.10 Storing objects with BOSS

BOSS - Binary Object Streaming Service - is a very important tool for converting most types of Smalltalk objects into compact binary representation that requires relatively little memory space. Although BOSS is used mainly to store objects in a file and retrieve them, it can also be used for other purposes such as sending objects across a network. BOSS is the essence of all programs that store data in a file.

BOSS is implemented by a group of Smalltalk classes in category System-Binary Storage. It is a very powerful tool that can, for example, store both classes and their instances, help converting from one version of a class to another, and read objects in the sequence in which they were stored or in random

order. In this section, we will limit ourselves to the simplest but most important use of BOSS - storing class instances and accessing them sequentially. For more sophisticated use, refer to User's Guide.
The typical BOSS usage pattern is as follows:

1. Create an instance of class BinaryObjectStorage and open it on an external stream associated with a file.
2. Read the stored objects from the stream using next or write them to the stream using nextPut:.
3. Close the BOSS object; this closes the file too.

The main BOSS protocols are as follows:

Creating and closing BOSS

The following two class methods are used to create BOSS objects and tie them to streams:

- onOld: aStream creates a BOSS object for reading the stream argument associated with an existing file or for appending to it. The stream must, of course, be capable of the desired type of access.
- onNew: aStream creates a BOSS object for writing to aStream starting at the beginning of the file. The file does not have to be new but will be treated as if it were. The stream must be capable of desired type of access.
- close closes the stream and the file associated with the BOSS object.

Accessing - reading and storing objects, changing position

BOSS is based on streams and its accessing messages are a subset of Stream accessing messages. The most important ones are

next	reads and reconstructs the next object from the BOSS stream
nexPut: anObject	increments the position pointer and stores anObject in the stream
nextPutAll: aCollection	stores all elements of a collection of objects, one after another
position	returns the current position in the stream
position:	changes the current position
reset	resets position to start
setToEnd	resets position to end
atEnd	tests whether the stream is positioned at the end

The following example shows how to use BOSS to *store* an object in a *new file* called 'c:\boss.tst':

```
| boss |
"Create a BOSS object."
boss := BinaryObjectStorage onNew: (Filename named: 'c:\boss.tst') writeStream.
"Store object in file using BOSS."
boss nextPut: #('string1' 'string2').
"Close BOSS object and the file."
boss close
```

The following complementary program *reads* the object back and recreates it. Note that it is not necessary to specify that the object is an array, this information is recovered by BOSS. Using BOSS is thus simple and the only thing you must watch is to assign the retrieved objects to the correct variables when you read the objects back.

```
| array boss|
"Create a BOSS object."
boss := BinaryObjectStorage onOld: (Filename named: 'c:\boss.tst') readStream.
"Read the array previously stored in the file."
array := boss next.
```

```
"Close the file."
boss close
```

In most situations, you will use BOSS to store complete objects by a single operation rather than storing each component separately. As an example, store a whole collection as one object rather than storing the elements as separate objects one after another - and read it back without reconstructing it laboriously by enumeration. If, however, BOSS is used to access and frequently modify a large collection of objects, and if this access is not always sequential, you may want to store the objects one after another.

Another point to realize is that when inter-related objects are 'bossed' to one file, no duplication occurs and object relationships are preserved. This does not happen if the objects are stored in separate files. The following example illustrates the difference.

Example: Store network of inter-related objects in one file

Consider the simplest group of inter-related objects – two arrays sharing one element (Figure 10.13). Clearly, this group consists of three objects – the two arrays, and the shared fraction.

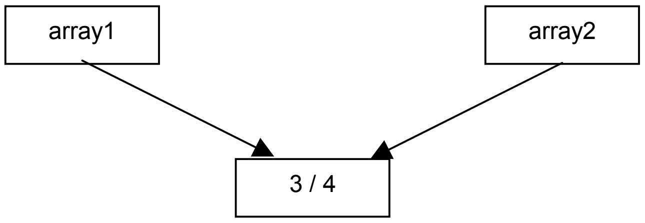


Figure 10.13. Two arrays sharing one element.

The following program creates the arrays, bosses them in two separate files,, and reads them back. When you execute it with *inspect*, you will find that the test at the end returns false, indicating that the two reconstituted arrays *do not* share the fraction that the original arrays did (Figure 10.14). This is not surprising because we did not boss out any inverse references from the fraction to the other array.

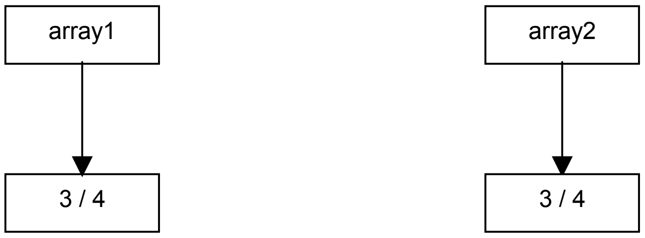


Figure 10.13. The two arrays after bossing out into two separate files, and bossing in again.

```
| array1 array2 boss x |
x := 3/4.
"Create two arrays sharing one object and boss each to its own file."
array1 := Array with: x.
array2 := Array with: x.
boss := BinaryObjectStorage onNew: (Filename named: 'c:\boss.tst1') writeStream.
boss nextPut: array1.
boss close.
boss := BinaryObjectStorage onNew: (Filename named: 'c:\boss.tst2') writeStream.
boss nextPut: array2.
boss close.
"Read the two objects back."
boss := BinaryObjectStorage onOld: (Filename named: 'c:\boss.tst1') readStream.
array1 := boss next.
```

```
boss close.
boss := BinaryObjectStorage onOld: (Filename named: 'c:\boss.tst2') readStream.
array2 := boss next.
boss close.
"Check whether the two arrays still share the element."
(array1 at: 1) = (array2 at: 1) “Returns false.”
```

If we now modify the program to write both arrays to the *same* file

```
| array1 array2 boss x |
x := 3/4.
"Create two arrays sharing one object and boss both to the same file."
array1 := Array with: x.
array2 := Array with: x.
boss := BinaryObjectStorage onNew: (Filename named: 'c:\boss.tst1') writeStream.
boss nextPut: array1.
boss nextPut: array2.
boss close.
"Read the two objects back."
boss := BinaryObjectStorage onOld: (Filename named: 'c:\boss.tst1') readStream.
array1 := boss next.
array2 := boss next.
boss close.
"Check whether the two arrays still share the element."
(array1 at: 1) = (array2 at: 1) “Returns true.”
```

we find that the two reconstituted arrays now *do* share the fraction, as they did before being bossed out. We conclude that storing multiple objects in one file preserves the original structure of their relationship. In fact, the second version also saves space because it stores the shared fraction object only ones.

In closing, we want to repeat that BOSS is one of the most important Smalltalk tools and if you don’t have a data base program, you will probably store all your data using it. The small amount of space that we dedicated to BOSS is a tribute to the simplicity of its basic use and does not reflect its importance. We will use BOSS in all our applications to store persistent data.

Main lessons learned:

- BOSS - a collection of built in Smalltalk classes - is the standard tool for storing objects in files.
- BOSS is one of the most important VisualWorks tools.
- To use BOSS, create an instance of BinaryObjectStorage on a suitable external stream, perform the storage or retrieval operation, and close the BOSS object.
- Upon reading an object, BOSS recognizes its type automatically.
- Store compound objects as single entities rather than component by component.
- Store interrelated objects in the same file.

Exercises

1. What happens to the value of a variable associated with a BinaryObjectStorage when you close the BOSS object?
2. Must the stream used by BOSS be an external stream?
3. Write a program to use BOSS to store an array containing the factorials of all integers from 1 to 20 in file ‘test’ in directory c:\. Write another program to read the object back and print it in the Transcript.
4. Open a file editor on the BOSS file created in the previous exercise.
5. BOSS can be used as a simple database system by storing elements of a collection in consecutive locations and accessing them by position, for example through some translation table (a dictionary). Explain how this would be done on the example of an inventory of items with unique Ids. Explain the BOSS accessing methods suitable for this use.

6. Write a program that creates two arrays called array1 and array2, both containing element x = 5/6. 'Boss' array1 to file test.1a and array2 to file test.1b, and then boss the two objects back in, storing them in variables array3 and array4. The original arrays array1 and array2 shared the same element x, in other words, there were originally three objects – array1, array2, and x. Arrays array3 and array4, on the other hand, each have their own copy of 5/6 corresponding to four objects altogether. The objects bossed back are thus an imprecise representation of the original objects.
7. Repeat the previous exercise but write the two arrays to the same BOSS file. What do you get when you read the two arrays back? Compare with the previous exercise and state a conclusion about storing networks of inter-related objects.

10.11 Other ways of storing objects

As you know, parts of the class library can be stored using *file out* and restored using *file in*. The file out procedure saves the source code and adds a few extra characters to separate, for example, one method from another. *File in* uses these extra characters and the compiler to recompile the code and save it back in the library. As a simple example, the file out of the following method in protocol accessing

```
firstName: aString
  firstName := aString
```

is the following ASCII file:

```
'From VisualWorks(R), Release 2.5 of September 26, 1995 on July 11, 1997 at 12:56:20 am'!



!Name methodsFor: 'accesing'!

firstName: aString
  firstName := aString! !
```

As we have seen in the previous section, classes can also be stored by BOSS but their restoration requires BOSS classes rather than the compiler.

Classes and their instances can also be stored and restored by methods storeOn: and readFrom:, both defined in class Object and redefined in several classes at lower levels of the class hierarchy. This approach is independent of BOSS but much less efficient and limited, and it is never used in Smalltalk applications. We introduce it only because its implementation is an interesting example of the use of polymorphism and delegation, and because it is the basis of the automatic saving of changes in your library code.

Message storeOn: aStream constructs anASCII string describing the receiver and adds it to the specified stream. Message readFrom: aStream then reconstructs the original object from it as in

```
| dictionary stream |
dictionary := Dictionary new.
dictionary add: 'Saleem' ->'Khan'; add: 'Ke'->'Qiu'.
stream := WriteStream on: (String new: 20).
"Store the Dictionary object in the stream using storeOn:."
dictionary storeOn: stream.
"Produces  stream on '((Dictionary new) add: (''Ke'' -> ''Qiu''); add: (''Saleem'' -> ''Khan''); yourself)' ."
"Create a copy of the original Dictionary object using readFrom:."
Object readFrom: (ReadStream on: stream contents)
```

as you can see when you execute this fragment with *inspect*.

If the stream in which the string is stored is external, this approach can be used to store an object in a file and reconstruct it but the representation is bulky.

The basic definition of storeOn: in Object simply generates messages to create a new instance of the receiver and further messages to initialize its variables. The interesting part of the definition is that it asks each component of the receiver to store itself. Typically, this results in the component asking its own components to store themselves, and so on. You can see how this can create problems if the structure is circular. The definition of storeOn: is as follows:

```
storeOn: aStream
"Append to aStream an expression whose evaluation creates an object similar to the receiver.  This is
appropriate only for smaller simpler objects and it cannot handle arbitrary circular references of objects.”
aStream nextPut: $(.
self class isVariable
        ifTrue: “For instances of classes with indexable elements.”
                [aStream nextPutAll: '(', self class name, ' basicNew: '; : self basicSize; nextPutAll: ') ']
        ifFalse: “For instances of classes with named instance variables”
                [aStream nextPutAll: self class name, ' basicNew'].
“Get values of instance variables of the receiver object being stored and ask them to store themselves using
store:.”
1 to: self class instSize do:
         [:i | aStream nextPutAll: ' instVarAt: '; store: i;
        nextPutAll: ' put: '; store: (self instVarAt: i); nextPut: $;].
1 to: self basicSize do:       “Get values of indexed variables”
        [:i | aStream nextPutAll: ' basicAt: '; store: i; store: (self basicAt: i); nextPut: $;].
aStream nextPutAll: ' yourself)'
```

If the object being stored has some special properties, we may be able to store the object more efficiently. As an example, Array redefines storeOn: to take advantage of literal arrays as follows:

```
storeOn: aStream
"Use the literal form if possible."
self isLiteral
        ifTrue:  [aStream nextPut: $#; nextPut: $(.
                self do: [:element | storeOn: aStream.  space].
                aStream nextPut: $)]
        ifFalse: [super storeOn: aStream]     “Use general implementation if not literal.”
```

and the definition of storeOn: in class Point is

```
storeOn: aStream
        aStream nextPut: $(;
        nextPutAll: self species name;
        nextPutAll: ' x: ';
        store: x;
        nextPutAll: ' y: ';
        store: y;
        nextPut: $).
```

Note that the basic definition of storeOn: depends on method store: which is defined in class Stream as follows:

```
store: anObject
"Have anObject print itself on the receiver for rereading."
        anObject storeOn: self
```

This interesting definition simply exchanges the receiver and the argument of storeOn: (anObject storeOn: aStream is equivalent to aStream store: anObject) to make the definition of storeOn: simpler. Since store: both uses and is used by storeOn:, the definition of storeOn: is recursive - when it stores the values of instance variables of an object, it asks them to store themselves (Figure 10.13).

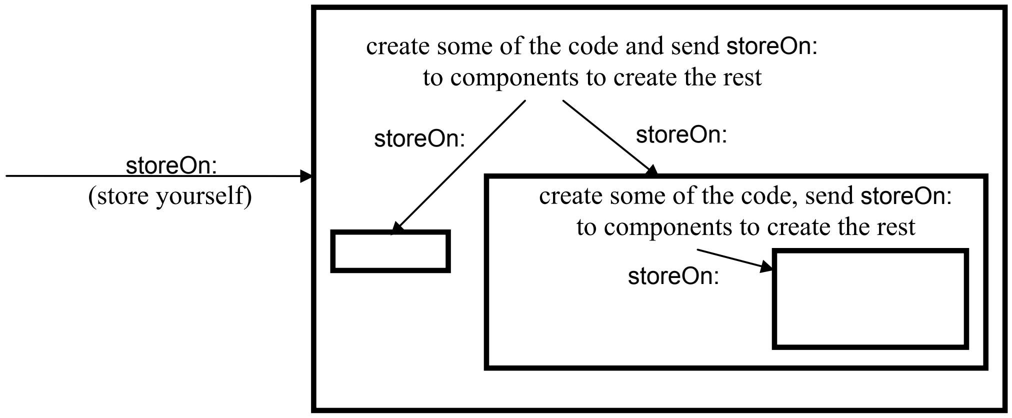

create some of the code and send storeOn:
to components to create the rest
storeOn:
storeOn:
storeOn:
(store yourself)
create some of the code, send storeOn:
to components to create the rest
storeOn:

Figure 10.13. The definition of storeOn: is recursive.

As an illustration of the operation of this recursive definition, consider using storeOn: on a literal array containing string elements: The storeOn: method creates the code to create a literal array and asks the string elements to create their own description of how they are stored. As a result, when you *inspect*

```
| stream |
stream := WriteStream on: (String new: 16).
#('ab' 'cd' 'ef') storeOn: stream.
stream
```

you will find that the stream's contents are

'#(''ab'' ''cd'' ''ef'')

where the underlined parts were created by the string elements of the array, and the rest by storeOn: in Array. When BOSS stores compound objects, it operates the same way.

As a closing note, the simple nature of storeOn: does not allow it to handle circular structures – unlike BOSS which does.

Main lessons learned:

- Methods storeOn: and readFrom: can store and reconstruct any object that does not have circular structure.
- Both storeOn: and readFrom: are used by the system but applications use either BOSS or a data base system to store objects in files.
- Method storeOn: is recursive and delegates the storage of the components of the object being stored to the components themselves.
- Method storeOn: cannot handle circular structures.

Exercises

1. What is the difference between printOn: and storeOn:?
2. Examine and explain the result of executing storeOn: on the object created with message
   Array with: 'abc' with: (Array with: with: 13 $x with: 5 factorial) with: (Dictionary with: ('key' -> 'value')).
   Test that readFrom: reconstructs the original object.
3. Execute the previous exercise with an external stream and file and open a file editor on the file. Compare the contents of the file with the contents of the equivalent BOSS file.
4. Explain the definition of readFrom: in three selected classes.
5. Rewrite the definition of storeOn: in Object without using store: to appreciate the gain in simplicity.

6. Explain the definitions of storeOn: in the following classes: Character, Collection, Date, and Time.
7. How would a Point be stored by the original storeOn: method defined in Object? How is it stored by its special storeOn: method? (Hint: Redefine storeOn: in Point to use super storeOn:.)
8. Find references to storeOn: and readFrom:.
9. Compare the speed and storage requirements of storeOn: and readFrom:, and BOSS, by storing several arrays of increasing size. Plot the results in terms of speed and file size as a function of the size of the arrays.

Conclusion

Sequenceable collections are often accessed linearly - one element after another. When a loop executing identical statements for each element is desired, this is best implemented with enumeration. When access is irregular, for example dispersed over several consecutive statements, streaming (use of streams for accessing) is preferable because it eliminates the need to maintain the current position within the collection. This becomes almost essential when the access is distributed over several methods.

All streams are subclasses of the abstract class Stream and can be divided into three groups: class Random, internal streams, and external streams. In this chapter, we dealt with internal and external streams and their three main forms - read-only, write-only, and read-write streams.

Internal streams are used for streaming over sequenceable collections, mainly strings and ordered collections. Their main uses are for string processing and their advantage is increased clarity of programs, simplification of programming, and sometimes increased execution speed. Internal streams are heavily used by the system and experienced programmers but novice programmers often don't take advantage of them and access collection elements by their index when streaming would be preferable.

External streams are used for accessing files and networks in a byte-by-byte fashion. In VisualWorks, files are implemented as instances of class Filename. Class Filename implements Smalltalk's interface to the platform's file system and executes various file-oriented and disk-oriented operations without explicit cooperation of an external stream. (Some of these operations use an external stream but hide it.) When the operation requires explicit access to the elements of the file, an external stream of the appropriate kind must first be created by sending a stream-creation message to the Filename object. Filename objects themselves are created with a string specifying the name of the file or directory. Class Dialog provides several powerful file-request messages that make obtaining the name of a file easier.

To use external streams and files properly, one must understand that external streams use an intermediate buffer object to hold a working copy of a part of the file or transmitted data. Sending flush to the stream sends the contents of the Smalltalk buffer to the operating system's buffer, commit sends it directly to the disk. The buffer is also flushed or committed whenever it becomes full, and committed when the file is closed by sending the close message to the stream.

The Binary Object Streaming Service (or BOSS) stores and restores objects efficiently and with minimum effort on the part of the programmer. To use BOSS, specify the file, create an appropriate external stream over it, create a BinaryStorageObject over the stream, perform the required operation, and close the BinaryStorageObject object. We have only covered basic storage and retrieval of class instances; more sophisticated uses are described in the User Manual. There are only a few occasions when an application might need to deal with a file directly, such as when you want to read a file containing a digitized picture rather than a Smalltalk object.

Two other ways of storing and restoring objects are the use of a data base system (VisualWorks library does not contain one), and the storeOn: and readFrom: messages. These two messages are heavily used by the system to save changes to the library but not by applications because they are very inefficient.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

BufferedExternalStream, ExternalStream, ***ExternalReadAppendStream***, ***ExternalReadStream***, ***ExternalReadWriteStream***, ***ExternalWriteStream***, **Filename**, InternalStream, PositionableStream, **ReadStream**, **ReadWriteStream**, Stream, **TextStream**, **WriteStream**.

Terms introduced in this chapter

append stream - stream allowing adding elements only at the end
binary object storage - storage of objects in binary form rather than as printable ASCII codes
buffer - area in memory holding data such as a part of a file
commit - save contents of a buffer on the disk
external stream - a stream designed for file or network access
file handle - a binary number used by the operating system to refer to a file
internal stream - accessor of sequenceable collections such as strings and ordered collections
stream - an accessor of linearly arranged data
streaming - linear access of sequentially organized data using a stream