Chapter

13

Streams

Streams and their uses

It's not immediately obvious what a *Stream* is and why you should use one, so let's approach this chapter the opposite way and show how certain things are done. We'll look at streams being used for printing objects, for reading and writing files, and for speeding up string manipulation. Then we'll take a look at how to create streams and the messages you can send them. In what follows, to save space I've put more cascaded messages on a line than I would in a real method.

Streams in printOn:

If you have ever modified the information returned from an object when it's sent the `printString` message, you've had to write or modify the `printOn:` method, which writes to a stream. For example,

```
printOn: aStream
   super printOn: aStream.
   aStream
      crtab; nextPutAll: 'instVar';
      space; print: instVar.
```

You'll notice from this example that streams know the current location in the stream. We haven't had to specify *where* to put anything. Instead, the stream keeps track of its current location and writes the next information starting at that point. There is a lot of built-in sequencing with streams.

You'll also notice that there are some useful messages you can send to a Stream (you can send a lot of the same messages to the Transcript, a global instance of TextCollector). For example, `crtab` writes a carriage return and a tab character, `nextPutAll:` writes a string (actually a collection), `space` writes a space character, and `print:` sends `printOn:` to its parameter.

Streams in file access

We'll talk a lot more about files in Chapter 14, Files, but here we'll look at the stream aspects of reading and writing files. The usual way to access a file is by using the buffered IO mechanism, which is the *Stream* mechanism. For example, if we want to create a file, write to it, then read from it, we can do something as simple as:

```
writeStream := 'myfile' asFilename writeStream.
writeStream nextPutAll: 'Here I am, writing to my file'.
writeStream cr; nextPutAll: 'Here is line two'.
writeStream close.

readStream := 'myfile' asFilename readStream.
Transcript cr; show: (readStream upTo: Character cr).
Transcript cr; show: readStream upToEnd.
readStream close.
```

Note that this code is not very robust, and in Chapter 14, Files, we'll talk more about robustness and the different options available to you when reading and writing files. Here we are simply showing the power of streams for dealing with file access.

Streams on strings

The most common use of a stream with a string is to create a large string out of smaller strings. In the following example, the stream technique is over three times as fast as string concatenation. If you were simply concatenating two strings, it would be more efficient to use the comma message. However, as the number of concatenations increases, so does the performance benefit of using a stream. The main reason for the efficiency is that concatenation creates a new string each time. On the other hand, in the stream technique, we created a large enough string up front so all we have to do is fill the string using the stream messages. If we had created the string using `String new`, the string would have been created with a size of zero, and would have had to grow several times as it was filled up. This is very expensive; if you change the stream example to say `stream := String new writeStream`, the performance is about fifty percent *slower* than using string concatenation.

```
string := 'There are ' , 4 printString, ' apples in ',
   1 printString, ' basket.', (String with: Character cr),
   4 printString, ' of you can have ', 1 printString.

stream := (String new: 100) writeStream.
stream
   nextPutAll: 'There are '; print: 4;
   nextPutAll: ' apples in '; print: 1;
   nextPutAll: ' basket.';
   cr; print: 4;
   nextPutAll: ' of you can have '; print: 1.
string := stream contents.
```

Streams can be created on any sequenceable collection and you will also find them used with Arrays. However, Streams are used with Strings more than with other types of sequenceable collection.

The benefits of a Stream

Why bother using a Stream? As we saw in the examples above, you don't have a choice if you want to write your own `printOn:` methods or if you want to use buffered I/O with files. Leaving these aside, why are Streams useful? Streams provide several benefits. First, they often provide better performance when adding to collections. Second, a Stream can automatically grow a collection to which it is adding, even if the collection is a fixed-size collection such as an Array. Third, Streams provide many more messages for manipulating collections. Fourth, you don't have to keep track of the position in a stream. For example, If you want to move about a string, you have to do operations to discover a position in the string, then do some operation at that position. With a Stream, you can move around and do operations without needing to know precise positions in the collection.

Creating streams

There are two fundamental subclasses of *Stream*: *InternalStream*, which streams over a collection, and *ExternalStream*, which streams over a file, providing buffered access to the file. In Chapter 14, Files, we will look at how to create streams on files; here we will just look at creating streams on collections. *InternalStream* has three subclasses: *WriteStream*, *ReadStream*, and *ReadWriteStream*, which allow write access, read access, and read/write access to the collection.

There are two basic ways of opening a Stream on a collection. You can send the `on:` or `with:` message to the stream class, passing the collection as a parameter, or you can send a message such as `readStream` to the collection. The difference between `on:` and `with:` is that writing to a stream created with `on:` will start writing at the beginning of the collection, overwriting the collection, while writing to a stream created with `with:` will append to the end of the collection. Here are some examples of creating streams of different types. Interestingly, there is no `readWriteStream` message for collections; perhaps because most collection streaming is either reading or writing, but not both.

```
readStream := ReadStream on: aString.
readStream := aCollection readStream.

writeStream := WriteStream on: (Array new: 100).
writeStream := (String new: 100) writeStream.

readWriteStream := ReadWriteStream on: (Array new: 100).
```

In the VisualWorks image there are a few streams on arrays but the great majority of collection based streams are on strings.

Reading, writing, and positioning streams

Writing

Typically we write to streams more than we read from them, so we'll look at writing first. There are relatively few writing operations, and the ones that are used most of the time are `nextPut:`, `nextPutAll:`, and `print:`, plus some character writing methods. There are a few other messages which you can find by browing

the methods for WriteStream and its superclasses. As we mentioned above, you can create a writeStream on an arbitrary collection, although it is more usual to see a writeStream on a String. Note that when you write to a collection, the collection will grow automatically if it needs to.

The `nextPut:` message puts a single object on the stream. You can put a whole collection on the stream as a single collection object using `nextPut:`. The whole collection will be read back as a single object. However, when using a writeStream on a String, `nextPut:` only writes characters. If you try to use `nextPut:` to write a string (a collection), you will get an error.

The `nextPutAll:` message puts a collection on the stream as individual objects. They will be read back as separate objects. When using a writeStream on a String, use `nextPutAll:` to write a string.

The `print:` message puts onto the stream the `printString` representation of an object by sending `printOn` to its parameter. If you are using a writeStream on a String, `print: anObject` is a convenient way to do the same as `nextPutAll: anObject printString`. The following two lines achieve the same result.

```
writeStream nextPutAll: anObject printString.
writeStream print: anObject.
```

There are a few special characters that are difficult to represent using the $ prefix. It's easy to represent the letter X by writing $X, but representing a space or a carriage return is less obvious. Streams provide some messages to make it easier to write these types of character to the stream.

`space`	Write a space
`tab`	Write a tab character
`tab: anInteger`	Write the specified number of tabs
`cr`	Write a carriage return
`crtab`	Write a carriage return followed by a tab
`crtab: anInteger`	Write a carriage return followed by the specified number of tabs
`lf`	Write a line feed.

Reading

There are a lot of messages for reading a collection and we won't go over all of them here. To learn more about the other messages, browse the methods defined by *ReadStream* and its superclasses. Some of the more useful messages are:

`next`	Return the next object on the stream (the next character if this is a string based stream). If we are at the end of the stream, *nil* is returned.
`next:`	Return the next specified number of objects. If this takes us past the end of the stream, an exception is raised.

`nextAvailable:`	Return the next specified number of objects, or the stream up to the end of the stream, whichever is less.
`contents`	Return the entire contents of the stream
`upToEnd`	Return the contents from the current position to the end of the stream
`through:`	Return the objects up to and including the first occurrence of the specified object or the end of the stream, whichever comes first.
`throughAll:`	Return the objects up to and including the first occurrence of the specified collection or the end of the stream, whichever comes first.
`upTo:`	Return the objects up to but *not* including the first occurrence of the specified object or the end of the stream, whichever comes first. The stream will be positioned *after* the specified object.
`upToAll:`	Return the objects up to but *not* including the first occurrence of the specified collection or the end of the stream, whichever comes first. The stream will be positioned *before* the specified collection.

Positioning

The basic positioning messages are:

`atEnd`	Are we at the end of the stream?
`position`	The current position in the stream
`position:`	Set the current position
`readPosition`	The current reading position in the stream
`writePosition`	The current writing position in the stream.
`reset`	Set the position to the start of the stream
`setToEnd`	Set the position to the end of the stream
`skip:`	Skip forward the specified number of elements
`skipSeparators`	Skip foward over separator characters: space, tab, carrriage return, etc.
`skipUpTo:`	Position the stream just *before* the object.
`skipThrough:`	Position the stream just *after* the object.

You can move to different positions in the stream using the `position:` message or one of the `skip` messages. For convenience, `reset` and `setToEnd` set the position to the start and the end of the stream. You can determine if you are at the end of the stream by sending `atEnd`, which returns a *Boolean*. In fact, when reading a stream we often stay in a loop that looks something like like one of the following

```
[readStream atEnd]
   whileFalse: [self myProcessObject: stream next].

[readStream atEnd]
   whileFalse:
      [doneWithStream := self myProcessObject: readStream next.
      doneWithStream ifTrue: [readStream setToEnd]]
```

Example

Here's an example of writing to a stream on an array then reading it back. Notice that we can write out collections either as a collection object using `nextPut:` or as a sequence of individual objects using `nextPutAll:`. When we read back what we wrote, it comes back just as we wrote it. Following the code of the example is the output to the Transcript. (Note that the Transcript, an instance of TextCollector, also responds to many of the same writing messages as a Stream. You can therefore use messages such as `cr`, `nextPutAll:`, `print:`, and `flush` with the Transcript.)

```
stream := ReadWriteStream on: (Array new).
stream nextPut: 2.
stream nextPut: 'How are you'.
stream nextPutAll: 'Alec'.
stream nextPutAll: #('Fine' 'thanks').
stream nextPut: OrderedCollection new.
stream reset.
[stream atEnd]
   whileFalse:
      [Transcript cr; show: stream next displayString]

2
How are you
$A "16r0041"
$l "16r006C"
$e "16r0065"
$c "16r0063"
Fine
thanks
OrderedCollection ()
```