Terminals

Updated Nov 20, 2013 by mkobetic

Terminals are the objects at the bottom of stream stacks. These are the ultimate sources of elements produced by read streams or ultimate consumers of elements written into write streams. This package provides support for various kinds of terminals available in VisualWorks. Eventual ports to other dialects may support different set of terminals although there will likely be significant overlap. However, implementations may have subtle differences as well. For example streams in Squeak/Pharo do not use become: when they grow the underlying collection of a collection stream. Therefore code assuming that identity of a collection stream terminal will be preserved may not be portable.

Collections

Collections are traditional stream terminals. Read streams are created by sending #reading to a collection.

```
(1 to: 10000) reading ++ 1000; read: 5
```

Similarly write streams are created by sending #writing to a collection. The collection is grown automatically to accommodate any elements written. Closing a collection write stream will truncate the collection to the current stream position. This behavior is useful as a replacement for the traditional #contents message. The contents can be accessed with the #terminal message after the stream is closed. The advantage is that the sequence #close and #terminal (or the shortcut #conclusion) can be sent to arbitrary stream stack and behaves consistently.

```
String new writing write: 'Hello World'; -- 6; conclusion
```

Note that while the read streams generally require a sequenceable collection, the write streams allow non-sequenceable terminals as well.

```
Bag new writing write: 'abbcccdaabdda'; conclusion
```

Block Closures

Streaming over block closures is proving to be "xtremely" versatile tool, allowing to stream over result sets of arbitrary computations, modelling infinite streams, etc.

```
"inifinite stream of ones"
[ 1 ] reading read: 20

"Fibonacci"
| a b | a := 0. b := 1.
[ | x | x := a. a := b. b := x + a. x ] reading ++ 500; get

"Streaming over ObjectMemory"
x := ObjectMemory someObject.
[ x := ObjectMemory nextObjectAfter: x ] reading read: 5
```

Single argument blocks can be used as read terminals too, which turn an iterative process in to a stream.

```
"In this example, we have a -hard- loop, not using collection protocol, which will only run one element at a tir
[:out | 1 to: 10 do: [:i | out put: i]] reading read: 5.
```

Single argument blocks can be used as write terminals.

```
"Transcript as an xtream"
[ :x | Transcript nextPut: x ] writing write: 'Hello World!'

"/dev/null"
[ :x | ] writing write: 'Hello World!'
```

Combined with stream transforms like #doing: and #limiting:, a block closure stream can be used to transform arbitrary loop into a stream and the streaming API can provide fine grained control over the execution of the loop body.

Files

File read streams are created by sending #reading to a Filename. However the actual terminal is an IOAccessor. The original filename is accessible through the IOAccessor.

```
| file |
file := ObjectMemory imageName asFilename reading.
[ file read: 13 ] ensure: [ file close ]
```

As a convenience, sending #reading to a Filename of a directory will create a stream of all filenames in that directory.

```
'/tmp' asFilename reading rest
```

File write streams can be created via the usual #writing message or via #appending which opens the file in appending mode. In appending mode, you cannot position the stream before the end of the file contents, so you can never overwrite existing contents. In writing mode, the file will be truncated at stream's current position when #close is called. To keep the entire contents of the file, use -= 0 to skip to the end before closing. This behavior is different from the classic streams which would erase the contents of the file on open.

```
| file |
file := '/dev/shm/xtreams-test' asFilename.
[       file writing write: 'Hello'; close.
        file appending write: ' World!'; close.
        file contentsOfEntireFile.
] ensure: [ file delete ]
```

It is also possible to send #reading or #writing to a pre-opened IOAcccessor if some other opening mode configuration is desirable. For example to emulate the classic write stream opening behavior, you can use the following:

```
(IOAccessor openFileNamed: '/dev/shm/xtreams-test'
        direction: IOAccessor writeOnly
        creation: IOAccessor truncateOrCreate
) writing close
```

Bare file streams are always binary. An encoding transform has to be applied to translate bytes into characters, which is discussed in more detail in [Xtreams-Transforms]. However for some types of encoding it is possible to use ByteStrings instead of ByteArrays to get similar effect. E.g reading from a file into a ByteString is like applying ISO8859-1 encoding on the fly. It can also be quite a bit faster than full blown encoding layer.

However it does not do line-end translation and only few single-byte encodings are available (see the ByteString hierarchy). Similarly writing into a bare file stream from a ByteString works as if the characters were automatically encoded with ISO-8859-1. This is again without line end translation, so normally you would end up with CRs (code 13) in your file, which these days does not match any OS convention (Unixes and MacOS use LF, and Windows uses CRLFs).

```
| file header |
file := ObjectMemory imageName asFilename reading.
header := ByteString new writing.
[ header write: 20 from: file ] ensure: [ file close ].
header conclusion
```

Sockets and Pipes

Sockets and Pipes are very similar, both accessed via IOAccessors again. The examples below show how data can be sent through a TCP or pipe connection by setting up a read stream and a write stream on a pair of connected accessors.

```
[ :in :out |
        [       out writing write: 'Hello'; close.
                in reading read: 5
        ] ensure: [ in close. out close ]
] valueWithArguments: SocketAccessor openPair


[ :in :out |
        [       out writing write: 'Hello'; close.
                in reading read: 5
        ] ensure: [ in close. out close ]
] valueWithArguments: OSSystemSupport concreteClass pipeAccessorClass openPair
```

Same rules regarding binary vs text reading/writing applies to socket and pipe streams as was discussed with regards to file streams above.

There is also a variation of these streams that allows to attach a specific timeout value to the stream. If a given read or write operation doesn't return within the specified timeout a Timeout exception is signaled. These streams are created with #reading: or #writing: message where the argument is the timeout duration.

```
pipe := OSSystemSupport concreteClass pipeAccessorClass openPair.
[       (pipe first reading: 1 seconds) get
] ensure: [ pipe do: #close ]
```

Finally, note that unlike pipes, sockets are full-duplex, i.e. a socket is really two independent pipes one for each direction. Consequently closing a socket read or write stream does not close and release the socket itself. A stream close just shuts down the corresponding pipe (using the appropriate socket shutdown: call). It is important to explicitly close the socket itself when it is not needed anymore.

Buffers

Buffers are used internally by Xtreams, but they could be useful in other circumstances too. There are several kinds and they can be used as both read and write stream terminals at the same time (not concurrently by multiple processes though, process synchronization has to managed explicitly via an external semaphore if necessary). For example writing into ElasticBuffer will grow it as much as necessary, but reading frees the space back up and it is immediately reused for further writes. So with interleaved reads and writes the size of the buffer will only grow to accommodate the maximum size written but not read yet.

```
buffer := ElasticBuffer on: String new.
bufferIn := buffer writing.
bufferOut := buffer reading.

100000 timesRepeat: [ bufferIn write: 'Hello World'. bufferOut read: 11 ].
buffer cacheSize
```

SharedQueues

SharedQueues are primarily used for data transfer between processes, their API is rather spartan though. Streams over SharedQueues provide the convenience of full streaming API.

```
queue := SharedQueue new.
in := queue reading.
out := queue writing.
received := Array new writing.
done := Semaphore new.
consumer :=
        [ | size |
                [       (size := in get) isZero
                ] whileFalse: [ | word |
                        word := ByteString new: size.
                        in read: size into: word.
                        received put: word ].
                done signal.
        ] fork.
#(one two three four) do: [ :word | out put: word size; write: word ].
out put: 0.
done wait.
received conclusion
```

Exotic streams

There are also a few special purpose terminal streams that could be useful at times.

A random number read stream can be easily created from an instance of Random.

```
Random new reading read: 10
```

An efficient stream of a constant (identical) object can be created with message #repeating. This stream is more efficient than equivalent block stream.

```
String new writing write: 100 from: $- repeating; close; terminal
```

A write stream on Transcript allows using Xtreams APIs for simple transcript logging.

```
Transcript writing cr; write: 'Hello World!'
```

A special read stream on top of a listening socket provides a new socket for each incoming connection.

```
| socket |
socket := SocketAccessor newTCP.
socket listenFor: 1.
[       socket accepting do: [:client | client close]
```

```
        ] ensure: [socket close]
```

A write stream on nil is a highly efficient bit bucket.

```
        nil writing write: Object comment
```

A reading stream on nil is the canonical empty stream

```
        nil reading rest
```

A read stream on a Context walks up the sender chain allowing for easy stack traversal.

```
        thisContext reading rest
```

A read stream on ObjectMemory enumerates all non-immediate objects in the image.

```
        ObjectMemory reading inject: 0 into: [ :total :object | total + 1 ]
```

Enter a comment: Hint: You can use Wiki Syntax.

Submit