# Substreams

Updated Mar 1, 2012 by mkobetic

Substreams are streams embedded in other streams. Often there are multiple substreams embedded sequentially in a parent stream (for example multipart MIME messages). This package allows efficient handling of common types of substreams.

## Fixed size substreams (#limiting:)

Limiting substream starts at the position when the substream is created and ends when the specified number of elements is read or written.

```
('abcdefghijklmno' reading limiting: 10) rest

| stream substream |
stream := String new writing.
substream := stream limiting: 5.
[ substream write: 'Hello World!' ] on: Incomplete do: [].
stream conclusion
```

## Bounded substreams (#ending:/#ending:inclusive:)

Bounded substream ends when the argument matches the content passing through the stream. The argument can take one of the following forms:

- a block - evaluated with each element; the stream ends when the block returns true

```
('abcdefghijklmnopqrstuvxyz' reading ending: [ :e | 'gmt' includes: e ]) rest.

| stream substream |
stream := String new writing.
substream := stream ending: [ :e | 'gmt' includes: e ].
[ substream write: 'abcdefghijklmnopqrstuvxyz' ] on: Incomplete do: [].
stream conclusion
```

- a collection - matched against the matching amount of last elements going through the stream; the stream ends when the collection matches

```
('abcdefghijklmnopqrstuvxyz' reading ending: 'mno' inclusive: true) rest.

| stream substream |
stream := String new writing.
substream := stream ending: 'mno'.
[ substream write: 'abcdefghijklmnopqrstuvxyz' ] on: Incomplete do: [].

stream conclusion
```

- any other object - the stream ends when an equal element passes through the stream"

```
('ab#cd#ef!ABC##' reading ending: $!) rest.

| stream substream |
stream := String new writing.
substream := stream ending: $!.
[ substream write: 'Hello World! Bye World!' ] on: Incomplete do: [].
stream conclusion
```

There is also a longer form #ending:inclusive: which takes an additional Boolean argument determining if the matching elements should be part of the substream content or not. The short form assumes the matching elements should be omitted.

```
('abcdefghijklmnopqrstuvxyz' reading ending: [ :e | 'gmt' includes: e ] inclusive: true) rest.

| stream substream |
stream := String new writing.
substream := stream ending: [ :e | 'gmt' includes: e ] inclusive: true.
[ substream write: 'abcdefghijklmnopqrstuvxyz' ] on: Incomplete do: [].
stream conclusion
```

Finally, closing a substream doesn't close the underlying stream by default. This is usually desirable because another substream is likely to follow so the main stream cannot be closed in that case. So in normal use the main stream is usually closed separately from the substreams. However, closing behavior of any substream can be modified through a custom closeBlock: if different behavior is desired.

```
| limited |
limited := (String new writing limiting: 10) closeBlock: [ :stream | stream destination close ]; yourself.
[ limited write: Object comment ] on: Incomplete do: [ :ex | ].
limited close; terminal
```

## Streams of substreams (slicing and stitching)

Slicing and Stitching is a meta-stream concept. Slicing breaks a stream with content that is delimited in some fashion (limiting:, ending:, etc) in to multiple substreams. The result of slicing is a read stream of substreams. Stitching is the inverse of slicing. It takes a read stream of streams and stiches those back together to look like a single continuous stream. Stitching a slicing stream will normally give the same result as the original underlying stream.

Here is an example that creates a slicing stream that cuts the input up into substreams of size 5.

```
((1 to: 49) reading limiting: 5) slicing collect: [ :slice | slice rest ]
```

Note that only the latest substream is active, previous substream is automatically depleted before next substream is created. Also note that since slicing always creates substreams, it is always a read stream, even if you're slicing a write stream. Output of the slicing stream are the substreams. Again, only the latest substream is active. The previous substream is automatically closed before next substream is created.

```
| samples slices |
samples := String new writing.
slices := (samples limiting: 3) slicing.
Date.MonthNames do: [ :month | [ slices get write: month ] on: Xtreams.Incomplete do: [] ].
samples conclusion
```

In the following example the stream contains multiple messages delimited by $! and each message has multiple parts ending with $#. We want to process each part as a stream of its own (this is a simplified version of how multipart messages are represented in MIME).

```
| messages |
messages := ('ab#cd#ef!ABC##' reading ending: $!) slicing.
messages collect: [ :message |
        (message ending: $#) slicing collect: [ :part | part rest ] ]
```

Note that since only the last slice is active any sort of read-ahead will likely interfere with the expected behavior, e.g. if we used collecting: instead of collect: in the example above, it would not work.

To generate the "hash-bang" style message encoding used in previous example we don't need any sort of end-detecting substream. Instead we need to emit the closing character when the substream is closed. There is a special "closing" substream and corresponding slicer for that as well.

```
| connection messages |
connection := String new writing.
messages := (connection closing: [ connection put: $! ]) slicing.
3 timesRepeat: [ | parts message |
        message := messages get.
        parts := (message closing: [ connection put: $# ]) slicing.
        #(one two three four) do: [ :body | parts get write: body ] ].
connection conclusion
```

Stitching takes a read stream of streams and makes them behave as a single continuous stream. Following example takes the 'chunks' and stitches them together.

```
| chunks |
chunks := (#('abc' 'de' '' 'fghij') collect: [ :c | c reading ]) reading.
chunks stitching rest
```

The stream of streams can be inifinite in which case the stitching stream is inifinite as well. For example the following stitching stream will not end.

```
| main |
main := (1 to: 10) reading.
[ main limiting: 3 ] reading stitching rest
```

The problem is that the underlying stream of streams keeps producing empty limiting streams at the end of the main stream. To make the example end with an empty limiting stream can be done as follows.

```
| main wasEmpty |
main := (1 to: 10) reading.
wasEmpty := false.
[       wasEmpty ifTrue: [ Incomplete zero raise ].
        wasEmpty := true.
        (main doing: [ :e | wasEmpty := false ]) limiting: 3
] reading stitching rest
```

Here the #doing: transform captures the fact that there was in fact an element flowing through the limited: stream and sets the wasEmpty flag accordingly. This way we can detect the first empty limiting: stream and raise Incomplete. Alternatively the block stream can capture the substream in a variable and before creating next one it can check if its position reached the limit.

```
| main current |
main := (1 to: 10) reading.
current := nil.
[       (current notNil and: [ current position < 3 ]) ifTrue: [ Incomplete zero raise ].
        current := main limiting: 3
] reading stitching rest
```

Following example traverses current directory recursively by continuously adding to a queue of directories to search from and using the stitching to combine them together in to one long stream of filenames.

```
| directories files |
directories := Xtreams.ElasticBuffer new: 10 class: Array.
directories put: '.' asFilename.
files :=
        [       directories get reading
                        doing: [:filename | filename isDirectory ifTrue: [directories put: filename]]
        ] reading stitching.
files rest
```

A practical example of stitching write streams is chunking of written content into size-prefixed chunks of some maximum size. This is something that can be often seen in network protocols (e.g. when individual chunks need to be encrypted or signed).

```
| output buffer |
output := ByteArray new writing.
buffer := RingBuffer on: (ByteArray new: 5).
[       (buffer writing limiting: buffer cacheSize)
                closeBlock: [ output put: buffer readSize; write: buffer ];
                yourself
] reading stitching
        write: (1 to: 22); close.
output close terminal
```

## Monitoring

Monitoring transform is useful for gauging throughput and progress. It takes a monitoring block that will be invoked periodically in specified intervals. The monitoring block can take up to three arguments in this order:

- total number of elements processed so far
- increment in the number of elements since the last block invocation
- total microseconds elapsed since the monitoring transform was created

```
| total |
total := 10000000.
(nil writing
        monitoring: [ :processed :delta :elapsed |
                Transcript writing cr;
                        print: (processed / total) floor; write: '% complete, ';
                        print: (processed / elapsed * 1000000) floor; write: ' elements/second' ]
        every: 1 seconds
)       write: ([1] reading limiting: total floor * 100);
```

```
close
```

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

Submit