# Core

Updated Mar 1, 2012 by mkobetic

Xtreams is a generalized stream/iterator framework providing simple, unified API for reading from various kinds of **sources** and writing into various kinds of **destinations** (Collections, Sockets, Files, Pipes, etc). Streams themselves can be sources or destinations as well, allowing to **stack** streams on top of each other. At the bottom of such stack is some kind of non-stream (e.g. a collection), we will call it a **terminal**. Directly above it is a specialized stream providing a streaming facade over the terminal. The rest of the streams in the stack we'll call **transforms**. Their primary purpose is to perform some kind of transformation on the elements that are passing through. Application code interacts with the top stream of the stack the same way it would with any other stream (or stream stack) producing/consuming the same elements. The goal of the framework is to provide consistent behavior of different stacks so that the application code can treat them the same way regardless of what exactly is the ultimate source or destination. For example an application analyzing binary data should be able to treat the source stream the same way if it is a simple stream over a ByteArray or if it is a stack providing contents of a specific binary part of a mulitpart, gziped and chunked HTTP response coming through a socket. Xtreams is an attempt to achieve this goal in a scalable and efficient manner.

Note that the sequence of streams in a stack cannot be completely arbitrary. Each transform constrains the type of input it accepts and the type of output it produces. Since the adjacent streams in a stack are linked so that input of one is connected to the output of the other, they need to produce/consume same type of elements . For example, a "character encoding" read transform converting bytes to characters can only sit on top of a stream that produces bytes (0..255). These constraints are not checked when the stack is being constructed, it is up to the application code to make sure the streams in the stack are compatible with their adjacent streams.

The framework is split into several packages:

| Package Xtreams- | Notes |
|---|---|
| Core | Defines the API and core classes |
| Terminals | Streams for all supported terminals (Collections, Blocks, Files, Sockets, Pipes, etc) |
| Transforms | Core transform streams (collection style, encodings, marshaling, etc) |
| Substreams | Streams embedded in other streams, slicing and stitching streams |
| Multiplexing | A substreaming protocol for sharing a paired read/write stream |
| Crypto | Cryptographic transforms (hashing, encryption, etc) |
| Xtras | Additional non-core transforms (e.g. compression) |
| Parsing | PEG parsing for Xtreams with a collection of parsers/generators (PEG, Smalltalk, Javascript, Wiki, etc) |

## Stream Creation

Xtreams strictly separate read and write streams. This allows for simpler API with better consistency. If necessary it would be quite simple to create a wrapper that holds a separate read and write stream and dispatches the read operations to one and write operations to the other.

Streams are primarily created by messages with -ing suffix on the first keyword (it often is a simple unary message). A read stream on top of an arbitrary **source** terminal is created by sending message #reading to it. The source can be a collection, an IOAccessor (socket, pipe), aFilename,

etc. For an exhaustive list of supported types of sources see package Xtreams-Terminals. If #reading is sent to a read stream, the same stream is returned.

Similarly, a write stream is created by sending #writing to a **destination** terminal. Generally the same kinds of objects can serve as sources or destinations, Xtreams-Terminals provides further details on what those are. Message #writing sent to a write stream returns the same stream.

## Basic Operations

Here are few simple examples showing equivalent code in classic streams and xtreams:

```
"classic"       'hello world' readStream next: 5.
"xtreams"       'hello world' reading read: 5.

"classic"       String new writeStream nextPutAll: 'hello'; contents
"xtreams"       String new writing write: 'hello'; close; terminal
```

Nothing terribly surprising here, just some API changes. Note that xtreams do not provide the #contents message. It's a read operation and as such it doesn't really fit a write stream. Consequently it only works in specific circumstances (e.g. above), but not in a case like this for example:

```
(ByteArray new withEncoding: #ascii) writeStream nextPutAll: 'Hello'; contents
```

Instead, in Xtreams, if you close a write stream on a collection it will trim the underlying collection down to the actual content. So the collection itself corresponds to stream #contents after closing. This provides equivalent capability within the constraints of generally applicable write stream API. Message #terminal can be used to retrieve the underlying terminal from a stream or stream stack. Message #conclusion can be used as a convenient shortcut combining #close and #terminal together.

Here's a rough correspondence table of the most common API

| classic | xtreams |
|---|---|
| next | get |
| nextPut: | put: |
| next: | read: |
| next:into:startingAt: | read:into:/read:into:at: |
| nextPutAll: | write: |
| next:putAll:startingAt: | write:from:/write:from:at: |
| upToEnd | rest |
| skip: | ++ |

## Stream Copying

Another useful difference is that read streams can be used as arguments of write: and write:from:. This provides a simple and efficient way of copying from stream to stream. Providing this capability at the API level allows for interesting optimizations, for example copying between two collection streams will eventually boil down to a single copy operation between the underlying collections. So the classic (slow) stream copying pattern

```
source := 'Hello World' readStream.
destination := String new writeStream.
[ source atEnd ] whileFalse: [ destination nextPut: source next ].
destination contents
```

becomes simple and optimal

```
String new writing write: 'Hello World' reading; conclusion
```

## End Of Stream

Another difference that you'll probably notice early on is that reading past the end of stream is not a notification which returns nil by default. With Xtreams it is always an exception, Incomplete, the same way you get IncompleteNextCountError with #next:/#next:into:startingAt: (but not with #next). Incomplete always carries count of elements that were read or written successfully, and optionally also the destination or source collection and start parameters to provide easy access to whatever was successfully processed. Similarly, the positioning APIs report the actual change of position, with collection and start being nil. As usual the stream that raises the exception is the originator.

```
[ 'Hello World!' reading ++ 6; read: 50 ] on: Incomplete do: [ :ex | ex contents ]
```

Another significant difference is that xtreams do not provide the #atEnd test. Depending on the type of stream, implementing such test may require an actual read attempt to be able to answer. Once you perform a read, you'll need to cache the element if the stream is not positionable. This gets even more complicated with complex stream stacks. Instead xtreams provide several alternatives each suitable for different circumstances. Message #rest reads everything that is left in the stream and automatically handles the Incomplete exception at the end.

```
'Hello World!' reading read: 6; rest
```

If you don't want to collect all the elements in the collection, xtreams also support collection style iteration protocols

```
'Hello World!' reading
        read: 6;
        inject: 0 into: [ :count :each | count + 1 ]
```

For comparison here's a common "read line by line" example in both classic and xtream style

```
text := 'line1\line2\line3\line4' withCRs readStream.
lines := OrderedCollection new
```

```
        lines := OrderedCollection new.
        [ text atEnd ] whileFalse: [ lines add: (text upTo: Character cr) ].
        lines


        text := 'line1\line2\line3\line4' withCRs reading.
        slicer := (text ending: Character cr) slicing.
        slicer collect: [ :line | line rest ]
```

The xtreams example relies on some features that are yet to be discussed, but the point we're trying to make here is that there's often better way to handle the classic pattern. For example if you want only up to first 3 characters from each line, just change the 'line rest' bit into '(line limiting: 3) rest'. The slicing layer will take care of aligning the underlying stream for you automatically, whereas in the classic case the code would have to handle it explicitly. It's also worth noting that the xtreams solution will do it efficiently. Each line can be processed as a stream of its own avoiding creation of unnecessary and potentially costly garbage.

Obviously, if you can't find more suitable solution, any use of atEnd can be replaced by an Incomplete handler. For example the usual looping pattern

```
        [ stream atEnd ] whileFalse: [ ... ]
```

can be converted to

```
        [ [ ... ] repeat ] on: Incomplete do: [ :ex | ]
```

The stream can be ended in a way that it wasn't closed, but an error was raised. Usually this is a kind of OsError and it is not the same as knowing that a stream closed. The Incomplete exception does not raise when an OsError occurs on the underlying terminal. An example of where this distinction matters would be HTTP/1.0 where the body contents are sent to you and you know that the body is completely sent when the connection closes - however, if you receive an OsError before the Incomplete then you know you did not receive the entire body. Better protocols of course give you an indication of how much content there should be, so usually you want to catch OsError at a higher level of your application.

## Enumeration

As was alluded to above, read streams provide all the traditional, collection style enumeration API: #collect:, #select:, #inject:into:, etc. Including some of the more recent additions like #do:separatedBy: and #groupedBy. This allows certain level of polymorphism with Collections. It's not clear at this point how far should we take this. For example we also provide concatenation of read streams with #, as an experimental feature (in Xtreams-Xtras).

## Closing

It is beneficial to get into the habit of always closing a stream when you're done with it. However, it is worth noting, you should never close a stream that you did not create. If you are handed a stream from a framework or library, you should expect the framework or library to close it when execution control returns to it. Even though in particular cases it may end up being a noop, not making that assumption in your algorithm may extend its applicability to wider range of streams and stream stacks. Closing the stream isn't necessarily just a matter of releasing resources. Especially with write streams it may be important to perform additional activity to properly terminate the stream. Some of these might be unobvious implementation aspects that may need to be properly cleared out. For example, current implementation of generic TransformWriteStream involves a background process which won't be terminated if the stream is not closed. Given that even read transforms may involve arbitrary side-effects, similar precautions apply to read streams as well. The best approach is to always close a stream when you're done with it.

## Positioning / Seeking

Not all streams are naturally positionable. Traditionally, an algorithm that required positionability was restricted to naturally positionable streams. Xtreams attempt to bridge this gap by providing positioning for non-positionable streams via a specialized buffering transform (PositionRead/WriteStream). To make sure a stream is positionable, send it #positioning. It returns the same stream if it already is positionable.

```
        Random new reading positioning read: 5; -- 2; read: 2
```

The positioning API imitates pointer arithmetic. Only skipping forward (++) and skipping to the end (-= 0) is available on both positionable and non-positionable streams.

| message | notes |
|---|---|
| position | returns current position from the beginning(1) of the stream |
| position: | sets current position from the beginning(1) of the stream |
| available | how many elements are left from current position to the end(1) |
| length | total size of the stream (position + available) |
| ++ | skips forward specified number of elements |
| -- | skips backward specified number of elements |

| | |
|---|---|
| -= | skips backward specified number of elements from the end(1) of the stream |
| += | skips forward specified number of elements from the beginning(1) of the stream |

Some streams, such as sequenceable collection write streams and file write streams, can be positioned past the end of their current size. In this situation, #position:, #-= and #+= can all position the stream beyond the current size. The size of the stream will not change until an actual write is performed. It is possible to use -= with a negative argument to move from the end of the stream in to its future. This mode of operation is called "sparse writing".

(1) Note that interpretation of beginning of the stream is slightly different in the case of the augmented non-positionable streams. The true beginning of the stream may not be reachable by the time the stream is wrapped in the positioning layer, so the beginning refers to the position at the time the stream is wrapped (unless specific buffering strategy, e.g. a ring buffer, moves the beginning of the buffer forward, in which case the beginning of the stream moves with it). This applies equally to read and write streams.

The backward skipping operators -- and -= cannot skip past the beginning of the stream. Incomplete will be raised if such request is made leaving the stream positioned at the beginning. Similarly the forward skipping operators += and ++ cannot skip past the end of the stream, and raises Incomplete if such request is made leaving the stream positioned at the end.

```
[ Random new reading positioning read: 5; -- 8 ] on: Incomplete do: [ :ex | ex count ]
```

There are several things to keep in mind when using the positioning/buffering transform. Some methods of the positioning API need to discover the end of the underlying stream, specifically #-=, #available and #length. These will cause the underlying stream to seek to the end immediately (filling up the positioning buffer along the way). Consequently, if the underlying stream is infinite, these operations will not return (potentially exhausting available memory, if the buffering strategy in use keeps growing the buffer).

Once added the positioning layer cannot be removed arbitrarily. In the case of read streams the underlying stream is always aligned with the end of the positioning buffer. With write stream the underlying destination stream is aligned with the beginning of the positioning buffer, i.e. the elements are written when they either fall out of the bottom of the buffer (e.g. in case of fixed size ring buffer) or the stream is explicitly flushed (all buffered elements are written and buffer is reinitialized). While it should be safe to remove the positioning layer when its position is aligned with the underlying stream, it's best to avoid it altogether. Consequently, if an algorithm requires a positionable stream, it should be the caller's responsibility to pass in a positionable stream, not the algorithm attempting to turn arbitrary stream into a positionable one, because the positioning layer usually cannot be passed back to the caller.

Buffering requires memory and different algorithms may require different buffering strategies. There are several Buffer classes available that are used throughout the framework for various purposes. Streams that employ buffers allow to replace the default buffer with a different one via the #buffer: message. For example if an algorithm only needs to be able to peek up to five elements ahead, RingBuffer of size 5 is efficient and doesn't waste memory.

## Exploring

Any positionable stream can be explored. The #explore: method is an enhancement of the classic #peek. A 'stream peek' can be replaced with

```
stream explore: [ stream get ]
```

The advantage of #explore: is that the block allows arbitrary activity with the stream. The stream will return back to its original position when the block completes. For example you can peek for arbitrary number of elements, not just one:

```
actions at: (stream explore: [ stream read: 3 ]))
        ifAbsent: [ "not an action ID, do something else with the stream" ]
        ifPresent: [ :action | stream ++ 3. action value: stream ]
```

Positionable write streams can also be explored. The motivating use case is attempting a complex write and being able to abandon it if it doesn't work out.

```
String new writing
        write: 'Hello ';
        explore: [ :stream | stream write: 'Fooled!' ];
        write: 'World!';
        conclusion

String new writing
        write: 'Hello ';
        explore: [ :stream | stream write: 'World!' ];
        -= 0;
        conclusion
```

The most common kind of exploration is to 'peek' ahead by one element only. This is a short hand for aStream explore: get, eg:

```
'Hello' reading peek.
```

## Write vs Insert

At times it is useful to be able to insert elements into the existing content of a positionable write stream. All the #write:... methods have an #insert:... equivalent for that purpose. Insert at the end of the stream has the same effect as write, similarly insert: on non-positionable stream is the same as write:.

```
String new writing
        write: 'Hello World!';
        += 5;
        insert: ', Hello';
        -= 0;
        conclusion
```

## Buffered Writing

When writing to a write stream, it is the habit of Xtreams to always send data through immediately. There are cases where sending data immediately (eg: through a socket or protocol) could cause a protocol to become "chatty". When you know where it is you can safely flush to keep a protocol happy, it is useful to buffer up write data before transmitting it.

```
(ByteArray new writing buffering: 10)
        write: #[ 1 2 3 4 5 ];
        write: #[ 6 7 8 9 0 ];
        put: 11;
        flush;
        conclusion.
```

Comment by aro...@gmail.com, Oct 11, 2014

The use of ++ -- += -= here is **most** unfortunate. The names are not intention-revealing and are widely used for other purposes.

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

Submit