Transforms

Updated Nov 20, 2013 by mkobetic

Transforms are streams on top of other streams. A stack of transforms on top of each other with a terminal stream at the bottom is called a **stream stack**. The usual purpose of a transform stream is transformation of the elements that are passing through it. Alternative it can be some sort of a side-effect while passing the elements through unchanged, e.g. counting, logging, hashing. A transform may also cause the number of elements to change, e.g. compression, encoding, filtering, etc.

Collection enumeration analogs

Following set of transforms emulates the popular collection enumerators, providing the same sort of benefits. Primary difference is that the enumeration happens incrementally, as elements are being written into or read from the stream. This avoids some of the overhead that the collection API incurs improving scalability of these operations on large datasets.

Element Conversion (#collecting:)

This is the simplest form of transformation where elements are transformed one for one, i.e. the number of elements before and after transformation is the same.

```
"squaring read input"
((1 to: 10) reading collecting: [ :e | e * e ]) rest

"normalizing output"
(String new writing collecting: #asLowercase) write: 'abcABC'; conclusion
```

Element Filtering (#selecting:/#rejecting:)

These transformations eliminate some of the elements flowing through the stream based on provided criteria. The number of elements after the transformation is same or smaller than before transformation.

```
((1 to: 50) reading selecting: [ :e | e odd ]) rest

(String new writing rejecting: [ :c | c isLowercase ]) write: 'abABcC'; conclusion
```

Additional Processing of Elements (#doing:)

The #doing: transformation allows to perform additional activity with each element flowing through the stream. The elements can be counted, logged, copied into a different stream, etc.

```
count := 0.
('abcdef' reading doing: [ :e | count := count + 1 ]) rest.
count

log := String new writing.
(String new writing doing: [ :e | log put: e ]) write: 'abcdef'.
log conclusion
```

Per-Element Accumulation (#injecting:into:)

The #inject:into: idiom is used to simplify the expression of a running value computed over a set of elements. In the streaming variant the resulting stream provides the subsequent states of the running value (not the elements).

```
"e series"
x := 1.
```

```
        integers := [ x := x + 1 ] reading.
        ((integers injecting: 1 into: [ :sum :next | sum + next factorial reciprocal ]) collecting: #asDouble) read: 2
```

```
        (Array new writing injecting: 0 into: [ :total :each | each + total ]) write: (1 to: 10); conclusion
```

Specialized Transformations

These are specific, commonly used transformations. They are often optimized via a dedicated stream class, but that is usually hidden behind the conventional creation methods on ReadStream.

Character encoding (#encoding:)

Character encoding is a process of turning a sequence of characters into a sequence of bytes. The inverse process is called decoding. There are different kinds of character encoding often tailored to specific alphabets. Some are able to encode only a certain subset of characters, others are able to encode any character. These specific kinds of encodings are usually identified with their names, e.g. 'ASCII', 'UTF-8', 'ISO 8859-1'. When creating an encoding stream, the encoding name has to be specified. Any encoding name recognized by class Encoder can be used.

```
        (ByteArray new writing encoding: #iso8859_1) write: 'Hello'; conclusion
```

Encoding write stream consumes characters and converts them into bytes. Consequently the underlying write stream must be binary (i.e accept bytes (0..255)). The encoding stream also performs automatic conversion of CRs (Character cr) into the native line-end convention of the underlying platform, unless set into a different line-end convention mode.

```
        (ByteArray new writing encoding: #utf16)
                setLineEndCRLF;
                write: 'Hello World!\Bye World!' withCRs;
                conclusion
```

An encoding read stream reads bytes from an underlying binary stream, decodes the bytes according to the pre-configured encoding and produces characters. It also performs automatic line end conversion from arbitrary platform convention to CRs, unless set into a transparent mode.

```
        (#[65 66 67 13 10 68 69 70 10 71 72 73 ] reading encoding: #ascii) rest
        (#[65 66 67 13 10 68 69 70 10 71 72 73 ] reading encoding: #ascii) setLineEndTransparent; rest
```

Base-64 encoding (#encodingBase64)

Base-64 encoding is a commonly used encoding of bytes into characters, usually for the purpose of transporting bytes over text based protocols (e.g. MIME or XML). This is different from character encoding, the input and output type is inverted. The encoding has to be able to handle arbitrary sequence of bytes, but not every character sequence is valid base-64 encoding. Consequently the write stream consumes bytes and writes characters into the underlying stream. Note that base-64 encoding requires specific treatment at the end of the encoding sequence, so base-64 write stream must be closed at the end.

```
        encoded := String new.
        encoded writing encodingBase64 write: (ByteArray withAll: (0 to: 255)); close.
        encoded
```

The read stream reads characters from underlying stream and converts the characters into bytes.

```
        encoded reading encodingBase64 rest
```

The read stream ignores any intervening white space that might be used to format the base-64 characters for transport (e.g. base-64 encoding is often wrapped to certain line length with new-line characters interspersed with the base-64 characters. The read stream will automatically end when it encounters the base-64 termination sequence ($=). Note however that the sequence is not always present, so it cannot be used as a reliable stream terminator.

General Transformations (#transforming:)

This is the most general form of transformation. The block argument receives two streams, input and output. The block can read arbitrary number of elements from input (including none) and write arbitrary number of elements into the output (including none).

With read streams the input is the underlying source stream and output is a virtual stream of elements to be produced by the stream. The block will be invoked as many times as necessary to produce the requested number of elements (requested by a read call), or until an Incomplete is raised. Consequently if the block handles Incomplete from the input, it has to raise an Incomplete at some point, otherwise the stream will never end (improper Incomplete handling can result in tight infinite loops).

```
        "Multiply each pair of input elements together and return the result"
        ((Array withAll: (1 to: 20)) reading transforming: [ :in :out | out put: in get * in get ]) rest
```

Note that if the contentSpecies of the source doesn't fit the output of the transformation, the contentsSpecies of the transform stream has to be set explicitly.

```
"hex encoding: #[0 1 2 ... 255] -> '000102...FF' "
bytes := ByteArray withAll: (0 to: 255).
digits := '0123456789ABCDEF'.
byte2hex := [ :in :out || byte |
        byte := in get.
        out put: (digits at: (byte bitShift: -4) + 1).
        out put: (digits at: (byte bitAnd: 15) + 1) ].
(bytes reading transforming: byte2hex)
        contentsSpecies: String;
        rest
```

In case of write streams, the input of the transform block is a virtual stream of written elements and the output is the underlying destination stream. The block will be invoked as many times as necessary to consume any written elements, or until an Incomplete is raised by the destination. Again depending on the specifics of the transformation the contentsSpecies may have to be set explicitly (the collection used by the buffer backing the virtual input stream must be able accommodate any written elements).

```
(String new writing transforming: byte2hex)
        contentsSpecies: ByteArray;
        write: bytes;
        conclusion
```

Note that in order to allow writing transformation blocks for write streams the same way as for read streams, the write stream has to be backed up by a background process, which allows suspending the execution of the transformation block in case it needs to wait for more elements to be written. The benefit is that the same transformation block can be used on both read and write stream without change to perform the same transformation. This highly elaborate implementation has its costs and complexities which are better to be avoided if possible. Therefore the transform stream should be used only easy tasks and prototyping. Especially for simple transformations it is highly recommended to create a dedicated transform stream class to simplify the runtime characteristics. Creating custom stream classes is intentionally very simple for these exact reasons.

Duplicating

Duplicating streams copy the contents flowing through into a provided write stream. The duplicate write stream is specified as an argument at creation.

```
| copy |
copy := ByteArray new writing.
((0 to: 15) reading duplicating: copy) rest -> copy conclusion
```

```
| original copy |
original := Array new writing.
copy := ByteArray new writing.
(original duplicating: copy) write: (0 to: 15).
original conclusion -> copy conclusion
```

CType Import/Export (Interpreting)

Interpreting streams convert specific types of objects to/from bytes of corresponding primitive C-type, i.e. they assign designated "interpretation" to the bytes (terminology borrowed from UninterpretedBytes and associated suite of primitives employed by these streams). Supported C-types are defined by the Interpretations shared variable on class InterpretedBytes. The streams are created with message #interpreting: with argument being a Symbol denoting the C-type in the Interpretations registry, e.g. #float, #double, #unsignedChar, etc.

InterpretedWriteStream converts consumed elements into bytes of pre-configured (primitive) CType, e.g. float, long etc. The type of written elements must match the C-type and the underlying destination must be binary. InterpretReadStream performs the inverse transformation, converting bytes from a binary source according to the pre-configured (primitive) CType, e.g. float, long etc. It produces elements of corresponding class, e.g. #float -> Float, #double -> Double, etc.

```
doubles := [ Random new next ] reading.
bytes := (ByteArray new writing interpreting: #double)
        write: 10 from: doubles;
        conclusion.
(bytes reading interpreting: #double) read: 10
```

Marshaling

Marshaling streams are used to encode arbitrary smalltalk objects into a sequence of bytes suitable for binary storage or transport. The format of the binary encoding is defined by an ObjectMarshaler and is identified by particular version ID. Custom marshalers can be derived from ObjectMarshaler. Custom marshaling formats must declare their own (unique) version ID.

```
        object := 5 @ 5 extent: 5 @ 5.
        bytes := ByteArray new writing marshaling put: object; conclusion.
        bytes reading marshaling get
```

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

Submit