Streams are used to iterate over sequences of elements such as sequenced collections, files, and network streams. Streams may be either readable, or writeable, or both. Reading or writing is always relative to the current position in the stream. Streams can easily be converted to collections, and vice versa.

1.1 Two sequences of elements

A good metaphor to understand a stream is the following. A stream can be represented as two sequences of elements: a past element sequence and a future element sequence. The stream is positioned between the two sequences. Understanding this model is important, since all stream operations in Pharo rely on it. For this reason, most of the Stream classes are subclasses of PositionableStream. Figure 1.1 presents a stream which contains five characters. This stream is in its original position, i.e., there is no element in the past. You can go back to this position using the message reset defined in PositionableStream.

Reading an element conceptually means removing the first element of the future element sequence and putting it after the last element in the past.

![Figure 1.1: A stream positioned at its beginning.](image)
Figure 1.2: The same stream after the execution of the method next: the character a is in the past whereas b, c, d and e are in the future.

Figure 1.3: The same stream after having written an x.

element sequence. After having read one element using the message next, the state of your stream is that shown in Figure 1.2.

Writing an element means replacing the first element of the future sequence by the new one and moving it to the past. Figure 1.3 shows the state of the same stream after having written an x using the message nextPut: anElement defined in Stream.

1.2 Streams vs. collections

The collection protocol supports the storage, removal and enumeration of the elements of a collection, but does not allow these operations to be intermingled. For example, if the elements of an OrderedCollection are processed by a do: method, it is not possible to add or remove elements from inside the do: block. Nor does the collection protocol offer ways to iterate over two collections at the same time, choosing which collection goes forward and which does not. Procedures like these require that a traversal index or position reference is maintained outside of the collection itself: this is exactly the role ofReadStream,WriteStream andReadWriteStream.

These three classes are defined to stream over some collection. For example, the following snippet creates a stream on an interval, then it reads two elements.

```
| r |
r := ReadStream on: (1 to: 1000).
r next.
>>> 1
r next.
>>> 2
r atEnd.
>>> false
```
1.3 Streaming over collections

WriteStreams can write data to the collection:

```smalltalk
| w |
w := WriteStream on: (String new: 5).
w nextPut: $a.
w nextPut: $b.
w contents.
>>> 'ab'
```

It is also possible to create ReadWriteStreams that support both the reading and writing protocols.

Streams are not only meant for collections, they can be used for files or sockets too. The following example creates a file named test.txt, writes two strings to it, separated by a carriage return, and closes the file.

```smalltalk
StandardFileStream
fileNamed: 'test.txt'
do: [:str | str
        nextPutAll: '123';
        cr;
        nextPutAll: 'abcd' ].
```

The following sections present the protocols in more depth.

1.3 Streaming over collections

Streams are really useful when dealing with collections of elements, and can be used for reading and writing those elements. We will now explore the stream features for collections.

Reading collections

Using a stream to read a collection essentially provides you a pointer into the collection. That pointer will move forward on reading, and you can place it wherever you want. The classReadStream should be used to read elements from collections.

Messages next and next: defined inReadStream are used to retrieve one or more elements from the collection.

```smalltalk
| stream |
stream := ReadStream on: #(1 (a b c) false).
stream next.
>>> 1
stream next.
>>> #(a b c)
stream next.
>>> false
```
| stream |
stream := ReadStream on: 'abcdef'.
stream next: 0.
>>> ''
stream next: 1.
>>> 'a'
stream next: 3.
>>> 'bcd'
stream next: 2.
>>> 'ef'

The message peek defined in PositionableStream is used when you want to know what is the next element in the stream without going forward.

stream := ReadStream on: '-143'.
"look at the first element without consuming it."
negative := (stream peek = $-).
negative.
>>> true
"ignores the minus character"
negative ifTrue: [ stream next ].
number := stream upToEnd.
number.
>>> '143'

This code sets the boolean variable negative according to the sign of the number in the stream, and number to its absolute value. The message upToEnd defined in ReadStream returns everything from the current position to the end of the stream and sets the stream to its end. This code can be simplified using the message peekFor: defined in PositionableStream, which moves forward if the following element equals the parameter and doesn’t move otherwise.

| stream |
stream := '-143' readStream.
(stream peekFor: $-) 
>>> true
stream upToEnd 
>>> '143'

peekFor: also returns a boolean indicating if the parameter equals the element.

You might have noticed a new way of constructing a stream in the above example: one can simply send the message readStream to a sequenceable collection (such as a String) to get a reading stream on that particular collection.

Positioning

There are messages to position the stream pointer. If you have the index, you can go directly to it using position: defined in PositionableStream. You
1.3 Streaming over collections

![Diagram of stream elements](image)

**Figure 1.4:** A stream at position 2.

You can request the current position using `position`. Please remember that a stream is not positioned on an element, but between two elements. The index corresponding to the beginning of the stream is 0.

You can obtain the state of the stream depicted in 1.4 with the following code:

```plaintext
| stream |
stream := 'abcde' readStream.
stream position: 2.
stream peek
>>> $c
```

To position the stream at the beginning or the end, you can use the message `reset` or `setToEnd`. The messages `skip:` and `skipTo:` are used to go forward to a location relative to the current position: `skip:` accepts a number as argument and skips that number of elements whereas `skipTo:` skips all elements in the stream until it finds an element equal to its parameter. Note that it positions the stream after the matched element.

```plaintext
| stream |
stream := 'abcdef' readStream.
stream next.
>>> $a "stream is now positioned just after the a"
stream skip: 3. "stream is now after the d"
stream position.
>>> 4
stream skip: -2. "stream is after the b"
stream position.
>>> 2
stream reset.
stream position.
>>> 0
stream skipTo: $e. "stream is just after the e now"
stream next.
>>> $f
stream contents.
>>> 'abcdef'
```

As you can see, the letter e has been skipped.

The message `contents` always returns a copy of the entire stream.
Testing

Some messages allow you to test the state of the current stream: atEnd returns true if and only if no more elements can be read, whereas isEmpty returns true if and only if there are no elements at all in the collection.

Here is a possible implementation of an algorithm using atEnd that takes two sorted collections as parameters and merges those collections into another sorted collection:

```smalltalk
| stream1 stream2 result |
stream1 := #(1 4 9 11 12 13) readStream.
stream2 := #(1 2 3 4 5 10 13 14 15) readStream.

"The variable result will contain the sorted collection."
result := OrderedCollection new.
[stream1 atEnd not & stream2 atEnd not ]
whileTrue: [
    stream1 peek < stream2 peek
    "Remove the smallest element from either stream and add it to the result."
    ifTrue: [result add: stream1 next ]
    ifFalse: [result add: stream2 next ] ].

"One of the two streams might not be at its end. Copy whatever remains."
result
    addAll: stream1 upToEnd;
    addAll: stream2 upToEnd.

result.

>>> an OrderedCollection(1 1 2 3 4 4 5 9 10 11 12 13 13 14 15)
```

Writing to collections

We have already seen how to read a collection by iterating over its elements using a ReadStream. We’ll now learn how to create collections using WriteStreams.

WriteStreams are useful for appending a lot of data to a collection at various locations. They are often used to construct strings that are based on static and dynamic parts, as in this example:

```smalltalk
| stream |
stream := String new writeStream.
stream
    nextPutAll: 'This Smalltalk image contains: ';
    print: Smalltalk allClasses size;
    nextPutAll: ' classes.';
    cr;
    nextPutAll: 'This is really a lot.'.

6
1.3 Streaming over collections

stream contents.
>>> 'This Smalltalk image contains: 2322 classes.
This is really a lot.'

This technique is used in the different implementations of the method printOn:, for example. There is a simpler and more efficient way of creating strings if you are only interested in the content of the stream:

```smalltalk
| string |
string := String streamContents:
  [:stream |
    print: #(1 2 3);
    space;
    nextPutAll: 'size';
    space;
    nextPut: $=;
    space;
    print: 3. ].
string.
>>> '#(1 2 3) size = 3'
```

The message streamContents: defined SequenceableCollection creates a collection and a stream on that collection for you. It then executes the block you gave passing the stream as a parameter. When the block ends, streamContents: returns the contents of the collection.

The following WriteStream methods are especially useful in this context:

- **nextPut**: adds the parameter to the stream;
- **nextPutAll**: adds each element of the collection, passed as a parameter, to the stream;
- **print**: adds the textual representation of the parameter to the stream.

There are also convenient messages for printing useful characters to a stream, such as space, tab and cr (carriage return). Another useful method is ensureASpace which ensures that the last character in the stream is a space; if the last character isn’t a space it adds one.

About String Concatenation

Using nextPut: and nextPutAll: on a WriteStream is often the best way to concatenate characters. Using the comma concatenation operator (,) is far less efficient:

```smalltalk
[[| temp |
  temp := String new.
  (1 to: 100000)
  do: [:i | temp := temp, i asString, ' ' ] ] timeToRun
>>> 115176 "(milliseconds)"
```
Streams

Figure 1.5: A new history is empty. Nothing is displayed in the web browser.

Figure 1.6: The user opens to page 1.

The reason that using a stream can be much more efficient is that using a comma creates a new string containing the concatenation of the receiver and the argument, so it must copy both of them. When you repeatedly concatenate onto the same receiver, it gets longer and longer each time, so that the number of characters that must be copied goes up exponentially. This also creates a lot of garbage, which must be collected. Using a stream instead of string concatenation is a well-known optimization.

In fact, you can use the message `streamContents:` defined in `SequenceableCollection` class (mentioned earlier) to help you do this:

```smalltalk
String streamContents: [:tempStream |
  (1 to: 100000)
    do: [:i | tempStream nextPutAll: i asString; space ]
]

>>> 1262 "(milliseconds)"
```

Reading and writing at the same time

It’s possible to use a stream to access a collection for reading and writing at the same time. Imagine you want to create a History class which will manage backward and forward buttons in a web browser. A history would react as in figures 1.5 to 1.11.

This behaviour can be implemented using a ReadWriteStream.

```smalltalk
Object subclass: #History

instanceVariableNames: 'stream'
```
Figure 1.7: The user clicks on a link to page 2.

Figure 1.8: The user clicks on a link to page 3.

Figure 1.9: The user clicks on the Back button. They are now viewing page 2 again.

Figure 1.10: The user clicks again the back button. Page 1 is now displayed.

Figure 1.11: From page 1, the user clicks on a link to page 4. The history forgets pages 2 and 3.
Streams

classVariableNames: ''
package: 'PBE-Streams'

History >> initialize
    super initialize.
    stream := ReadWriteStream on: Array new.

Nothing really difficult here, we define a new class which contains a stream. The stream is created during the initialize method.

We need methods to go backward and forward:

History >> goBackward
    self canGoBackward
    ifFalse: [ self error: 'Already on the first element' ].
    stream skip: -2.
    ^ stream next.

History >> goForward
    self canGoForward
    ifFalse: [ self error: 'Already on the last element' ].
    ^ stream next

Up to this point, the code is pretty straightforward. Next, we have to deal with the goTo: method which should be activated when the user clicks on a link. A possible implementation is:

History >> goTo: aPage
    stream nextPut: aPage.
    ^ stream nextPut: nil.
    stream back.

This version is incomplete however. This is because when the user clicks on the link, there should be no more future pages to go to, i.e., the forward button must be deactivated. To do this, the simplest solution is to write nil just after, to indicate that history is at the end:

History >> goTo: anObject
    stream nextPut: anObject.
    stream nextPut: nil.
    stream back.

Now, only methods canGoBackward and canGoForward remain to be implemented.

A stream is always positioned between two elements. To go backward, there must be two pages before the current position: one page is the current page, and the other one is the page we want to go to.

History >> canGoBackward
    ^ stream position > 1

History >> canGoForward
    ^ stream atEnd not and: [stream peek notNil ]
Let us add a method to peek at the contents of the stream:

```
History >> contents
  ^ stream contents
```

And the history works as advertised:

```
History new
goTo: #page1;
goTo: #page2;
goTo: #page3;
goBackward;
goBackward;
goTo: #page4;
contents
>>> strncmp(#page1 #page4 nil nil)
```

### 1.4 Using streams for file access

You have already seen how to stream over collections of elements. It’s also possible to stream over files on your hard disk. Once created, a stream on a file is really like a stream on a collection; you will be able to use the same protocol to read, write or position the stream. The main difference appears in the creation of the stream. There are several different ways to create file streams, as we shall now see.

#### Creating file streams

To create file streams, you will have to use one of the following instance creation messages offered by the class `FileStream`:

- **fileNamed**: Open a file with the given name for reading and writing. If the file already exists, its prior contents may be modified or replaced, but the file will not be truncated on close. If the name has no directory part, then the file will be created in the default directory.

- **newFileNamed**: Create a new file with the given name, and answer a stream opened for writing on that file. If the file already exists, ask the user what to do.

- **forceNewFileNamed**: Create a new file with the given name, and answer a stream opened for writing on that file. If the file already exists, delete it without asking before creating the new file.

- **oldFileNamed**: Open an existing file with the given name for reading and writing. If the file already exists, its prior contents may be modified or replaced, but the file will not be truncated on close. If the name has no directory part, then the file will be created in the default directory.

- **readOnlyFileNamed**: Open an existing file with the given name for reading.
Streams

You have to remember that each time you open a stream on a file, you have to close it too. This is done through the `close` message defined in `FileStream`.

```plaintext
| stream |
    stream
        nextPutAll: 'This text is written in a file named ';
        print: stream localName.
    stream close.

stream := FileStream readOnlyFileNamed: 'test.txt'.
    stream contents.
>>> 'This text is written in a file named '"'test.txt"'"
    stream close.

The message `localName` defined in class `FileStream` answers the last component of the name of the file. You can also access the full path name using the message `fullName`.

You will soon notice that manually closing the file stream is painful and error-prone. That's why `FileStream` offers a message called `forceNewFileNamed:do:` to automatically close a new stream after evaluating a block that sets its contents.

```plaintext
| string |
FileStream
    forceNewFileNamed: 'test.txt'
    do: [ :stream |
        stream
            nextPutAll: 'This text is written in a file named ';
            print: stream localName ].
string := FileStream
    readOnlyFileNamed: 'test.txt'
    do: [ :stream | stream contents ].

>>> 'This text is written in a file named '"'test.txt"'"
```

The stream creation methods that take a block as an argument first create a stream on a file, then execute the block with the stream as an argument, and finally close the stream. These methods return what is returned by the block, which is to say, the value of the last expression in the block. This is used in the previous example to get the content of the file and put it in the variable `string`.

**Binary streams**

By default, created streams are text-based which means you will read and write characters. If your stream must be binary, you have to send the message `binary` to your stream.
When your stream is in binary mode, you can only write numbers from 0 to 255 (1 Byte). If you want to use `nextPutAll:` to write more than one number at a time, you have to pass a `ByteArray` as argument.

```plaintext
FileStream
forceNewFileNamed: 'test.bin'
do: [ :stream |
    stream binary;
    nextPutAll: #(145 250 139 98) asByteArray ].

FileStream
readOnlyFileNamed: 'test.bin'
do: [ :stream |
    stream binary.
    stream size.
    >>> 4
    stream next.
    >>> 145
    stream upToEnd.
    >>> #[250 139 98 ]
].
```

Here is another example which creates a picture in a file named `test.pgm` (portable graymap file format). You can open this file with your favorite drawing program.

```plaintext
FileStream
forceNewFileNamed: 'test.pgm'
do: [ :stream |
    stream
    nextPutAll: 'P5'; cr;
    nextPutAll: '4 4'; cr;
    nextPutAll: '255'; cr;
    binary;
    nextPutAll: #(255 0 255 0) asByteArray;
    nextPutAll: #(0 255 0 255) asByteArray;
    nextPutAll: #(255 0 255 0) asByteArray;
    nextPutAll: #(0 255 0 255) asByteArray ]
```

This creates a 4x4 checkerboard as shown in 1.12.

1.5 Chapter summary

Streams offer a better way (compared to collections) to incrementally read and write a sequence of elements. There are easy ways to convert back and forth between streams and collections.

- Streams may be either readable, writeable or both readable and writeable.
• To convert a collection to a stream, define a stream on a collection, e.g., ReadStream on: (1 to: 1000), or send the messages readStream, etc. to the collection.

• To convert a stream to a collection, send the message contents.

• To concatenate large collections, instead of using the comma operator, it is more efficient to create a stream, append the collections to the stream with nextPutAll:, and extract the result by sending contents.

• File streams are by default character-based. Send binary to explicitly make them binary.