Розділ 15
Потоки

Потоки використовують для перебору послідовностей елементів: послідовних колекцій, файлів, мережевих потоків. Потоки можуть бути або для читання, або для запису, або для обох дій. Читання або запис завжди пов'язані з вказівником потоку. Потоки легко можна перетворити на колекції та навпаки.

15.1. Дві послідовності елементів

Для розуміння потоку добре підходить метафора «двох послідовностей». Потік можна представити у вигляді двох послідовностей елементів: послідовність минулих елементів і послідовність майбутніх. Точка доступу потоку, або його вказівник, розташована на межі між ними. Розуміння цієї моделі важливе, бо всі потокові операції в Pharo покладаються на неї. Тому більшість потокових класів є підкласами *PositionableStream*. Рисунок 15.1 зображає потік, який містить п'ять літер. Він перебуває в початковому стані, тобто, немає ніякого елемента в минулому, вказівник розташовано на початку потоку. Потік можна повернути до такого стану за допомогою повідомлення *reset*, визначеного в *PositionableStream*.

Рис. 15.1. Вказівник потоку розташовано на початку

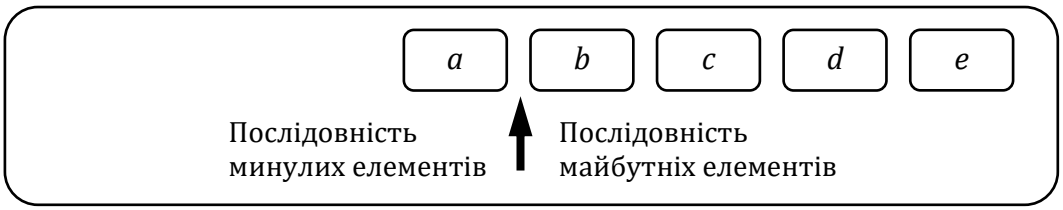

Рис. 15.2. Той самий потік після виконання методу *next*.
Літера *a* тепер у минулому, літери *b*, *c*, *d*, *e* – в майбутньому

«Прочитання елемент потоку» концептуально означає вилучити перший елемент з початку послідовності майбутніх елементів і помістити його в кінець послідовності минулих елементів. Стан потоку після зчитування одного елемента за допомогою повідомлення *next* показано на рис. 15.2. Видно, що вказівник потоку посунувся на один елемент праворуч.

Запис елемента означає заміну першого елемента послідовності майбутніх новим елементом і переміщення його в минуле. На рис. 15.3 зображено стан того ж потоку

після запису '*x*' за допомогою повідомлення *nextPut: anElement*. Вказівник знову посунувся на один елемент праворуч.

Рис. 15.3. Той самий потік після запису *x*

15.2. Потоки проти колекцій

Протокол колекції підтримує зберігання, вилучення та перебір елементів колекції, але не дозволяє змішувати ці операції. Наприклад, якщо елементи деякої *OrderedCollection* опрацьовують методом *do:*, то неможливо додати до неї чи вилучити з неї елементи зсередини блоку *do:*. Так само протокол колекцій не дає змоги перебирати дві колекції одночасно, вибирати, по якій колекції просунутися вперед, а по якій ні. Такі алгоритми потребують, щоби індекс обходу або посилання на позицію в колекції підтримувалися поза самою колекцією: саме це і є призначенням *ReadStream*, *WriteStream* та *ReadWriteStream*.

Ці три класи визначені для потокового доступу до колекції. Наприклад, код нижче спочатку накладає потік на інтервал, а опісля зчитує два елементи.

```
| r |
r := ReadStream on: (1 to: 1000).
r next.
>>> 1

r next.
>>> 2

r atEnd.
>>> false
```

Потоки для запису можуть записувати дані в колекцію.

```
| w |
w := WriteStream on: (String new: 5).
w nextPut: $A.
w nextPut: $B; nextPut: $C.
w contents.
>>> 'ABC'
```

Також можна створювати потоки *ReadWriteStream*, які підтримують обидва протоколи: і читання, і запису.

У наступних параграфах ці протоколи описано детальніше.

15.3. Віддавайте перевагу функціям інтерфейсу

Важливо, щоб ви зосередилися на API, наборі повідомлень, які пропонують потокові класи, а не лише на назвах класів. Чому? Тому що Pharo може запропонувати альтернативні реалізації класів або навіть спеціалізовані потоки для опрацювання інших кодувань, наприклад, двійкових форматів. Такі реалізації, хоча й називатимуться інакше, або навіть не успадковуватимуть базові класи потоків, пропонуватимуть ті самі API. Нарешті, у фрагментах коду цієї глави ми часто посилаємося на класи для створення екземплярів потоку, тоді як у реальному коді ми зазвичай надсилаємо повідомлення (наприклад, *readStream*) екземплярові колекції, щоб накласти на нього потік. Такий підхід робить код більш загальним.

У цьому розділі ми подаємо загальні повідомлення потоку та часто згадуємо кореневий клас, де визначено метод. Однак зауважте, що знати точний клас не так важливо, бо підкласи або інші класи, що надають ті самі API, можуть виконувати власні методи у відповідь на те саме повідомлення.

Потоки справді корисні у роботі з колекціями. Їх можна використовувати для відокремленого в часі читання та запису елементів. Тепер розглянемо можливості потоків для опрацювання колекцій.

15.4. Читання колекцій

Накладання потоку для читання на колекцію по суті надає вказівник на її елемент. Під час читання він автоматично посуватиметься на наступний елемент. Але його можна також перемістити, куди потрібно. Для читання елементів колекції використовують клас *ReadStream*.

У *ReadStream* визначено повідомлення *next* і *next:*. Їх використовують для того, щоб отримати з колекції один або більше елементів.

```
| 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'
```

15.5. Підглядання

Повідомлення *peek*, визначене у *PositionableStream*, використовують, коли потрібно «підглянути» в потік – довідатися, який елемент наступний в потоці, без переміщення вказівника потоку.

```
| stream negative number |
stream := ReadStream on: '-143'.
"Дивимося на перший елемент потоку, але не використовуємо його."
negative := (stream peek = $-).
negative.
>>> true

"Пропускаємо літеру мінус."
negative ifTrue: [ stream next ].
number := stream upToEnd.
number.
>>> '143'
```

Цей код покладає булевій змінній *negative* значення згідно зі знаком числа в потоці і змінній *number* – абсолютну величину значення цього числа. Визначене в *ReadStream* повідомлення *upToEnd* повертає весь вміст потоку від поточної позиції до кінця і встановлює вказівник потоку на його закінчення. Цей код можна спростити за допомогою повідомлення *PositionableStream >> peekFor:*, яке пересуває вказівник потоку вперед, якщо наступний елемент дорівнює параметру, і залишає його на місці в протилежному випадку.

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

stream upToEnd
>>> '143'
```

Повідомлення *peekFor:* також повертає булеве значення, яке сигналізує, чи його аргумент дорівнює елементові потоку.

Ви мали б помітити у наведеному прикладі новий спосіб створення потоку: можна просто надіслати повідомлення *readStream* послідовній колекції (як *String* у прикладі), щоб отримати накладений на неї потік читання.

Рис. 15.4. Вказівник потоку розміщено в позиції 2

15.6. Керування вказівником потоку

Існують повідомлення для позиціонування вказівника потоку. За допомогою *PositionableStream >> position:* можна перейти відразу до місця потоку, заданого індексом.

Поточне розташування вказівника можна довідатися за допомогою запиту *position*. Проте треба пам'ятати, що вказівник позиціонується не на елементі, а між елементами. Початкові потоку відповідає індекс 0.

Ви можете отримати потік у стані, зображеному на рис. 15.4, за допомогою наведеного нижче коду.

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

Щоб помістити вказівник потоку на початок або на закінчення, можна використати повідомлення *reset* або *setToEnd*.

15.7. Пропуск елементів

Повідомлення *skip:* і *skipTo:* використовують, щоб перемістити вказівник потоку вперед від поточної позиції. *skip:* приймає число і пропускає таку кількість елементів потоку від місця розташування. *skipTo:* пропускає всі елементи в потоці доки не знайде елемент, що дорівнює його параметру. Зверніть увагу, що *skipTo:* позиціонує потік після елемента, що збігається.

```
| stream |
stream := 'abcdef' readStream.
stream next
>>> $a
```

Тепер вказівник потоку розташовано відразу після '*a*'.

```
stream skip: 3.
stream position
>>> 4
```

Тепер вказівник після '*d*'.

```
stream skip: -2.
stream position
>>> 2
```

Тепер вказівник після '*b*'.

```
stream reset.
stream position
>>> 0

stream skipTo: $e.
stream next.
```

```
>>> $f
```

Пропуск до заданого елемента позиціонує потік одразу після заданого елемента. Легко бачити, що вказівник потоку було встановлено відразу після літери '*e*'.

```
stream contents
>>> 'abcdef'
```

Повідомлення *contents* завжди повертає копію цілого потоку.

15.8. Предикати

Деякі повідомлення дають змогу перевірити стан потоку: *atEnd* повертає *true* тоді і тільки тоді, коли немає більше елементів, які можна прочитати, тоді як *isEmpty* повертає *true* тоді і тільки тоді, коли в колекції взагалі немає елементів.

Нижче наведено можливу реалізацію алгоритму злиття з використанням *atEnd*, який приймає дві відсортовані колекції та об'єднує їх в нову відсортовану колекцію.

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

"Змінна result міститиме впорядковану колекцію."
result := OrderedCollection new.
[ stream1 atEnd not & stream2 atEnd not ]
   whileTrue: [
      stream1 peek < stream2 peek
         "Менший з елементів вилучаємо з потоку і поміщаємо в результат."
         ifTrue: [ result add: stream1 next ]
         ifFalse: [ result add: stream2 next ] ].

"Один з потоків все ще містить дані. Копіюємо залишок."
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)
```

15.9. Запис у колекцію

Ми вже бачили, як читати колекцію, перебираючи кожен її елемент за допомогою *ReadStream*. Тепер навчимося створювати колекції за допомогою *WriteStream*.

Екземпляри *WriteStream* корисні для додавання великої кількості даних у різних місцях колекції. Їх часто використовують для створення рядків, що складаються з постійної та змінної частин, як у наведеному прикладі.

```
| stream |
stream := String new writeStream.
stream
   nextPutAll: 'This image contains: ';
   print: Smalltalk globals allClasses size;
```

```
        nextPutAll: ' classes.';
        cr;
        nextPutAll: 'This is really a lot.'.

    stream contents.
    >>> 'This image contains: 10672 classes.
    This is really a lot.'
```

Таку методику використовують, наприклад, у різних реалізаціях методу *printOn:*. Якщо вас цікавить тільки вміст потоку, то можна використати простіший і ефективніший спосіб створення рядків.

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

Повідомлення *streamContents:*, надіслане класові колекції (тут *String*), створює екземпляр колекції і накладає на неї потік. Потім воно виконує блок, свій аргумент. Після цього *streamContents:* повертає вміст потоку – новостворену колекцію.

Перелічимо методи *WriteStream*, особливо корисні в цьому контексті.

nextPut: повідомлення *nextPut:* додає в потік свій параметр;

nextPutAll: повідомлення *nextPutAll:* приймає колекцію і додає кожен її елемент до потоку;

print: повідомлення *print:* додає до потоку текстове зображення свого параметра.

Є також зручні повідомлення для друку в потік окремих символів, наприклад, *space*, *tab* і *cr* (переведення каретки). Інший корисний метод *ensureASpace* гарантує, що останній символ у потоці пропуск; якщо це не так, то метод додає його.

15.10. Про конкатенацію рядків

Використання *nextPut:* і *nextPutAll:* з *WriteStream* часто найкращий спосіб дописування літер до рядка. Використання оператора конкатенації кома (,) набагато менш ефективне, що демонструють два приклади, які виконують однакове завдання.

```
    [ | temp |
        temp := String new.
        (1 to: 100000)
            do: [:i | temp := temp, i asString, ' ' ] ] timeToRun
    >>> 0:00:01:54.758
```

```
[ | temp |
   temp := WriteStream on: String new.
   (1 to: 100000)
      do: [:i | temp nextPutAll: i asString; space ].
   temp contents ] timeToRun
>>> 0:00:00:00.024
```

Використання потоку може бути набагато ефективнішим ніж оператора конкатенації, бо кома створює новий рядок, який містить конкатенацію приймача і аргументу, то ж вона мусить скопіювати обох. Якщо багаторазово дописувати до того самого приймача, то він щоразу ставатиме довшим і довшим, тому кількість літер, які потрібно копіювати, зростатиме експотенційно. Це також створює багато сміття, яке треба збирати. Використання потоку замість конкатенації є добре відомою оптимізацією.

Насправді, можна використати вже згадане повідомлення *SequenceableCollection >> streamContents:*, щоб зробити це ще простіше.

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

15.11. Про *printString*

Давайте трохи поговоримо про використання потоку в методах *printOn:*. По суті, метод *Object>>printString* створює потік виведення і передає його як аргумент методу *printOn:*, як подано нижче.

```
Object >> printString
  "Повертає рядок з описом отримувача. За потреби друку без обмежень на
   довжину рядка використовуйте fullPrintString."

  ^ self printStringLimitedTo: 50000
```

```
Object >> printStringLimitedTo: limit
  "Повертає рядок з описом отримувача. За потреби друку без обмежень на
   довжину рядка використовуйте fullPrintString."

  ^self printStringLimitedTo: limit using: [:s | self printOn: s]
```

```
Object >> printStringLimitedTo: limit using: printBlock
  "Повертає рядок з описом отримувача, отриманий у результаті виконання
   printBlock. Переконується, що результат не довший за задане обмеження."

  | limitedString |
  limitedString := String streamContents: printBlock limitedTo: limit.
  limitedString size < limit ifTrue: [^ limitedString].
  ^ limitedString , '...etc...'
```

Видно, що метод *printStringLimitedTo:using:* створює потік і передає його далі.

Якщо в перевизначеному методі *printOn:aStream* у своєму класі надсилати повідомлення *printString* змінним екземпляра свого об'єкта, то створимо ще один потік виведення, а потім скопіюємо його вміст до *aStream*. Нижче наведено приклад.

```
MessageTally >> displayStringOn: aStream
  self displayIdentifierOn: aStream.
  aStream
     nextPutAll: ' (';
     nextPutAll: self tally printString;
     nextPutAll: ')'
```

Тут вираз *self tally printString* запускає такий самий механізм і створює додатковий потік замість того, щоб використати наявний. Це відверто контрпродуктивно. Набагато краще надсилати повідомлення *print:* потокові або *printOn:* змінній екземпляра як у коді нижче.

```
MessageTally >> displayStringOn: aStream
  self displayIdentifierOn: aStream.
  aStream
     nextPutAll: ' (';
     print: self tally;
     nextPutAll: ')'
```

Тепер працює лише потік *aStream*, ніяких додаткових потоків ніхто не створює.

Щоб зрозуміти, як працює метод *print:*, наведемо його оголошення.

```
Stream >> print: anObject
  "Аргумент anObject має надрукувати себе в отримувач."

  anObject printOn: self
```

Інший приклад

Додаткове створення потоку не обмежується методами *printString*. Нижче наведено приклад, знайдений у Pharo (байдуже до класу, в якому знайдено метод), який демонструє таку саму проблему.

```
EpContentStringVisitor >> printProtocol: protocol sourceCode: sourceCode
  ^ String streamContents: [ :stream |
     stream nextPutAll: '"protocol: ';
     nextPutAll: protocol printString;
     nextPut: $"; cr; cr;
     nextPutAll: sourceCode ]
```

Легко бачити, що спочатку створено потік *stream*, а тоді ще один під час виконання виразу «*protocol printString*». Той другий буде знищено після завершення формування рядка, який зображає *protocol*. Кращою реалізацією є зображений нижче варіант.

```
EpContentStringVisitor >> printProtocol: protocol sourceCode: sourceCode
  ^ String streamContents: [ :stream |
     stream nextPutAll: '"protocol: ';
     print: protocol;
     nextPut: $"; cr; cr;
     nextPutAll: sourceCode ]
```

Рис. 15.5. Нова історія порожня. Веб-браузер не показує нічого

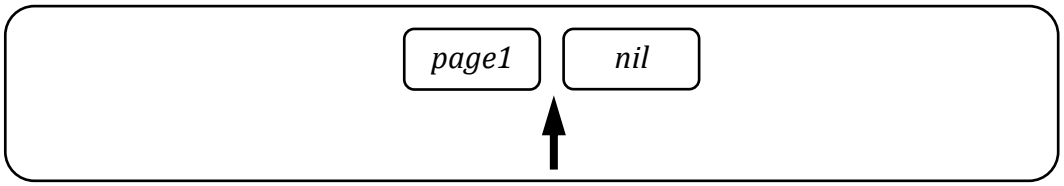

Рис. 15.6. Користувач відкрив першу сторінку

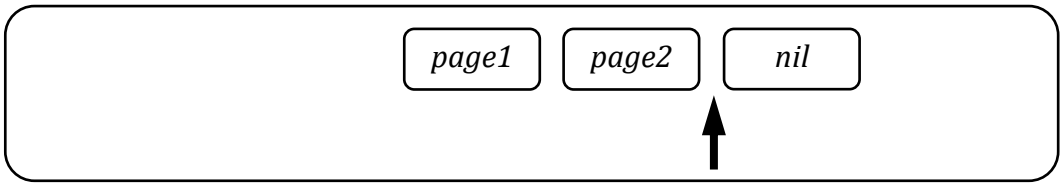

Рис. 15.7. Користувач перейшов за посиланням на другу сторінку

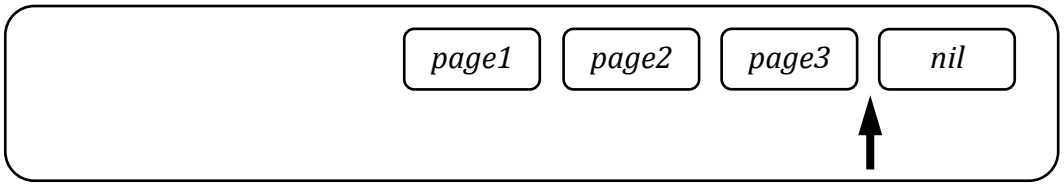

Рис. 15.8. Користувач перейшов за посиланням на третю сторінку

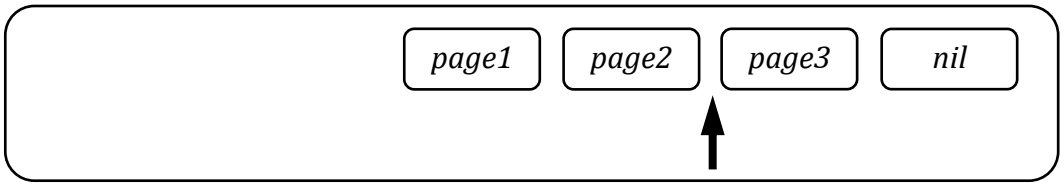

Рис. 15.9. Користувач клацнув кнопку «Назад». Тепер він знову бачить другу сторінку

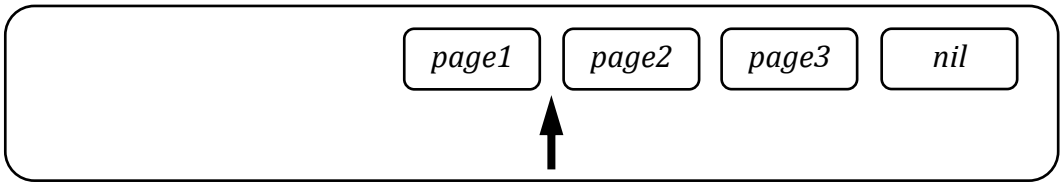

Рис. 15.10. Користувач знову клацнув кнопку «Назад». Відображено першу сторінку

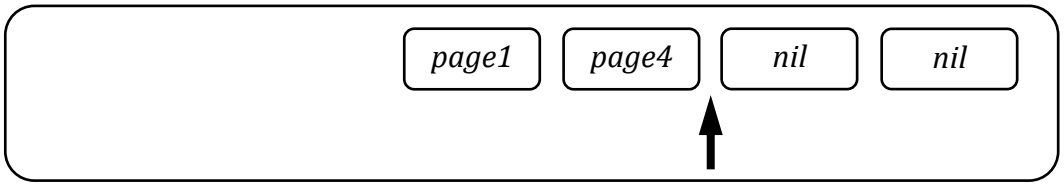

Рис. 15.11. З першої сторінки користувач перейшов за посиланням на четверту. Історія забула другу та третю сторінки

15.12. Одночасні читання та запис

Потік можна використовувати для доступу до колекції для читання і запису одночасно. Припустимо, потрібно створити клас *History*, який керуватиме кнопками «Вперед» і «Назад» у веббраузері. Екземпляр *History* має реагувати на натискання кнопок, як зображено на рис. 15.5 – 15.11.

Таку поведінку можна реалізувати з використанням *ReadWriteStream*.

```
   Object subclass: #History
      instanceVariableNames: 'stream'
      classVariableNames: ''
      package: 'PBE-Streams'

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

Тут нічого складного: визначили новий клас, який містить потік. Потік створюється під час виконання методу *initialize*.

Потрібні також методи, щоб рухатися вперед і назад.

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

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

До цього місця код досить простий. Далі потрібно реалізувати метод *goTo:*, який буде активовано, коли користувач клацне на посилання. Можлива реалізація наведена нижче.

```
History >> goTo: aPage
   stream nextPut: aPage
```

Проте такий варіант незавершений. Коли користувач клацає на посилання, немає ніяких майбутніх сторінок, щоб перейти до них, тобто, кнопку «Вперед» потрібно деактивувати. Щоб зробити це, найпростішим рішенням є записати в потік *nil* відразу після сторінки, щоб зазначити, що браузер перебуває наприкінці історії.

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

Тепер залишилося реалізувати тільки методи *canGoBackward* і *canGoForward*.

Вказівник потоку завжди розташований між двома елементами. Для того, щоб рухатися в зворотному напрямі, має бути дві сторінки перед поточною позицією: поточна сторінка і та, на яку потрібно потрапити.

```
History >> canGoBackward
  ^ stream position > 1

History >> canGoForward
  ^ stream atEnd not and: [ stream peek notNil ]
```

Корисним буде метод, який дає змогу поглянути на вміст потоку.

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

Легко переконатися, що *History* працює належно.

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

15.13. Використання потоків для доступу до файлів[1]

Ми бачили, як використовувати потоки для доступу до елементів колекції. Потоки можна також використати для взаємодії з файлами на жорсткому диску комп'ютера. API файлових потоків такий самий, як у щойно вивчених, особливим є тільки спосіб створення потоку.

Запис до файлу

У наведених нижче прикладах ви не побачите явного вказання класів файлових потоків. Усю роботу щодо створення потоку люб'язно виконають для вас об'єкти системи Pharo. Вам потрібно лише попросити їх про допомогу та виконати такі кроки:

- перетворити рядок, що задає ім'я файлу, на посилання на файл;
- отримати від файлу накладений на нього потік;
- виконати виведення в потік;
- не забути закрити потік після завершення роботи.

```
| hello stream |
hello := 'hello.txt' asFileReference.
hello exists
>>> false   "завжди можна перевірити існування файлу"
stream := hello writeStream.
stream nextPutAll: 'Hello World'.
stream close.
```

[1] Цей параграф додав перекладач книги.

Читання з файлу

Послідовність кроків така сама, як під час читання, тільки для отримання потоку використовують повідомлення *readStream* замість *writeStream*, та читають рядки з потоку, а не надсилають їх туди.

```
| hello stream |
hello := 'hello.txt' asFileReference.
hello exists
>>> true

stream := hello readStream.
stream next.
>>> $H

stream upToEnd.
>>> 'ello World'

stream close
```

Автоматичне закривання потоку

Щоб не турбуватися про своєчасне надсилання повідомлення *close*, можна використовувати методи, які зроблять це за вас. Припустимо *hello* – створене раніше посилання на файл. Тоді запис до нього можна виконати коротко:

```
hello writeStreamDo: [ :stream | stream nextPutAll: 'Hello World'].
```

Читання займе не більше місця:

```
hello readStreamDo: [ :stream | stream contents ]
>>> 'Hello World'
```

15.14. Підсумки розділу

Потоки пропонують кращий порівняно з колекціями спосіб для поступового читання і запису послідовності елементів. Є прості способи, щоб перетворювати потоки на колекції і навпаки.

- Потоки можуть бути лише для зчитування, лише для запису, або і для того, і для іншого.
- Щоб перетворити колекцію на потік, потік накладають на колекцію, тобто створюють повідомленням *on: – ReadStream on: (1 to: 1000)*; або надсилають колекції повідомлення *readStream* тощо.
- Щоб перетворити потік на колекцію, йому надсилають повідомлення *contents*.
- Для конкатенації великих колекцій, замість того, щоб використовувати оператор кома, ефективніше створити потік, додати колекції до нього повідомленням *nextPutAll:* і отримати результат за допомогою *contents*.
- Потоки можна використовувати для доступу до файлів. Зручно надсилати повідомлення, які автоматично закривають потік одразу після завершення роботи.