Przemysław Nieściór

Smalltalk dla praktyków

Kolekcje i strumienie

Kolekcje i strumienie w języku Smalltalk, ich właściwości, najważniejsze metody i praktyczne przykłady użycia.

24.2.2016

Wszystkie prawa zastrzeżone

1 Spis treści

2 Kolekcje

2.1 Podstawowe informacje

W języku Smalltalk wszystkie kolekcje są podklasami klasy abstrakcyjnej `Collection`. W praktyce najczęściej używanymi przez programistów Smalltalka kolekcjami są `OrderedCollection`, `SortedCollection`, `Array`, `Dictionary`, `Set`, `Bag`, `String`, `ByteString`, `Interval` i `Symbol`.

Nadklasa abstrakcyjna `Collection` definiuje podstawowy zestaw metod interfejsu kolekcji. Metody te są redefiniowane, optymalizowane lub zabronione w niektórych podklasach.

Protokół	Metody
dostęp	`size, at: anIndex, at: anIndex put: anElement`
sprawdzanie	`isEmpty, includes: anElement, contains: aBlock, occurrencesOf: anElement`
dodawanie elementów	`add: anElement, addAll: aCollection`
usuwanie elementów	`remove: anElement, remove: anElement ifAbsent: anBlock, removeAll: aCollection`
iteracja	`do: aBlock, collect: aBlock, select: aBlock, reject: aBlock, detect: aBlock, detect: aBlock ifNone: aNoneBlock`
przekształcanie	`asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection: aSortBlock`
tworzenie egzemplarzy	`with: anElement, withAll: aCollection`

Kolekcje można podzielić zależnie od udostępnianych protokołów lub konkretnych implementacji danego protokołu dziedziczonego z nadklasy `Collection` na kilka rodzajów:

- **Sekwencyjne** – elementy w egzemplarzach klas `OrderedCollection`, `SortedCollection`, `Array`, `String`, `ByteString` są scisle uporządkowane według ustalonej kolejnosci od pierwszego elementu. Natomiast egzemplarze klas `Set` i `Dictionary` nie są w ten sposób uporządkowane.
- **Posortowane** – `SortedCollection` przechowuje elementy odpowiednio posortowane.
- **Indeksowane** – większość uporządkowanych kolekcji jest indeksowanych, tzn. dostęp do ich elementów można uzyskać przy pomocy metody `at: anIndex`. Na przykład `anArray at: n` zwraca n-ty element z kolekcji, a `anArray at: n put: aNewElement` zamienia n-ty element na aNewElement.
- **Z dostępem za pomocą kluczy** – do elementów egzemplarzy klasy `Dictionary` i jej podklas dostęp uzyskuje się przy pomocy kluczy, a nie indeksów. Na przyklad `aDictionary at: #marka put: 'Ford'` wstawia pod kluczem oznaczonym symbolem #marka napis 'Ford', a `aDictionary at: #marka` zwraca wartosć umieszczoną pod tym kluczem (w tym wypadku 'Ford').
- **Zmienialne** – większość kolekcji jest zmienialna, ale na przykład `Symbol` nie jest. Nie można zmienić jego zawartosci za pomocą metody `at:put:`.
- **O dynamicznym rozmiarze** – egzemplarze klas `Array` czy `Interval` mają zawsze stałą wielkość, ustaloną podczas tworzenia egzemplarza danej klasy. Niektóre inne kolekcje, na przykład `OrderedCollection` czy `SortedCollection` mogą zmieniac rozmiar po utworzeniu egzemplarza klasy.
- **Dopuszczające duplikaty** – `Set` filtruje duplikaty, a na przykład `Bag` już nie (taki sam obiekt może występować w `Bag` kilkakrotnie). `Dictionary`, `Set` i `Bag` używają metody = udostępnianej przez przechowywane elementy, która sprawdza, czy jej

argumenty są takie same. `IdentityDictionary` i `IdentitySet` używają metody ==, która sprawdza, czy jej argumenty są tym samym obiektem.

- **Heterogeniczne** - większość kolekcji może jednocześnie przechowywać elementy będące egzemplarzami różnych klas. `String` i `Symbol` przechowują tylko elementy klasy `Character`. `Array` może przechowywać dowolne elementy, `ByteArray` tylko SmallInteger, a `IntegerArray` tylko `Integer`.

2.2 Iteracje po kolekcji

W języku Smalltalk iteracje są realizowane za pomocą metod wysyłanych do kolekcji.

2.2.1 Iteracja – metoda do:

Metoda `do:` jest podstawową metodą iteracyjną kolekcji na której oparte są inne tego typu metody. Programiści innych języków piszą najczęściej kod typu:

```
index := 1.
collection := #('Tomek' 'rower' 'Witaj').
[index <= collection size] whileTrue: [
     [Transcript show: (collection at: index)]
```

Tymczasem powyższy kod można zapisać znacznie prościej i krócej przy użyciu metody `do:`

```
#('Tomek' 'rower' 'Witaj') do: [:each | Transcript show: each].
```

Powyższy kod wyświetla wszystkie elementy zawarte w kolekcji (zmienna `each` zawiera przy każdej iteracji kolejny element kolekcji). Jak widać z tego przykładu, metoda `do:` wykonuje zawartość bloku kodu będącego jej argumentem dla każdego elementu kolekcji z osobna.

2.2.2 Selekcja elementów - metoda select:

Metoda `select:` zwraca wszystkie elementy kolekcji spełniające warunek zawarty w bloku kodu będącego jej argumentem.

```
#( 1 2 55 4 8 29 13) select: [:each | each < 8].
```

Wynikiem wykonania powyższego kodu będzie kolekcja #(1 2 4).

Implementacja metody `select:` (i wielu innych metod iteracyjnych) wykorzystuje omówioną w poprzednim rozdziale podstawową metodę iteracyjną `do:`

```
select: aBlock

	| newCollection |
	newCollection := self species new.
	self do: [:each | (aBlock value: each) ifTrue: [newCollection add:
each]].
	^newCollection
```

2.2.3 Selekcja elementów - metoda reject:

Metoda `reject:` jest odwrotnością metody `select:` - zwraca ona wszystkie elementy kolekcji nie spełniające warunku zawartego w bloku kodu będącego jej argumentem.

```
#( 1 2 55 4 8 29 13) reject: [:each | each < 8].
```

Wynikiem wykonania powyższego kodu będzie kolekcja #(55 8 29 13).

2.2.4 Kolekcja wyników - metoda collect:

Metoda `collect:` zbiera wyniki kodu będącego jej argumentem wykonywanego na elementach kolekcji.

```
#(1 2 4 8 13) collect: [:each | each * 2].
```

Wynikiem wykonania powyższego kodu będzie następująca kolekcja #(2 4 8 16 26).

Inny przykład:

```
collectionOfSongs collect: [:each | each title].
```

Wynikiem wykonania powyższego kodu może być następująca kolekcja #('Czerwone korale' 'Kochaj mnie' 'Hej sokoły').

2.2.5 Szukanie elementów – metody detect: i detect:ifNone:

Metoda `detect:` służy do szukania pierwszego elementu w kolekcji spełniającego warunek zawarty w kodzie bloku będącego jej argumentem.

```
#(4 2 8 3 9) detect: [:each | each < 3].
```

Wynikiem wykonania powyższego kodu będzie 2 (pierwszy element mniejszy od 3).

W przypadku, gdy żaden element w kolekcji nie spełnia warunku zdefiniowanego w bloku kodu będącym argumentem metody `detect:` wykonanie tej metody zakończy się błędem „Element not Found" .

```
#(4 2 8 3 9) detect: [:each | each = 240].
```

Wykonanie powyższego kodu zakończy się błędem, ponieważ żaden element nie spełnia warunku zdefiniowanego w bloku kodu będącym argumentem metody detect:.

Jeżeli nie jesteśmy pewni, czy jakikolwiek element kolekcji spełni zadany warunek, to możemy użyć wariantu naszej metody – `detect:ifNone:`

```
#(4 2 8 3 9) detect: [:each | each = 240] ifNone: [nil].
```

Wynikiem wykonania powyższego kodu będzie nil. ponieważ kolekcja nie zawiera liczby 240.

```

```

Powyższy kod możemy w większości dialektów języka Smalltalk zapisać też bez słowa kluczowego nil:

```
#(4 2 8 3 9) detect: [:each | each = 240] ifNone: [].
```

Możemy też zwracać wartości różne od nil:

```
#(4 2 8 3 9) detect: [:each | each = 240] ifNone: [333].
```

Wynikiem wykonania powyższego kodu będzie 333.

2.2.6 Akumulacja wyników – metoda inject:into:

Metoda inject:into: jest odpowiednikiem funkcji wyższego rzędu fold (znanej także jako accumulate, reduce, compress lub inject) występującej w językach funkcyjnych. Przetwarza ona uporządkowaną kolekcję w celu zbudowania końcowego wyniku przy pomocy jakiejś funkcji łączącej elementy tej kolekcji.

Poniższy kod oblicza sumę dziesięciu pierwszych liczb całkowitych większych od zera.

```
(1 to: 10) inject: 0 into: [:sum :each | sum + each].
```

Wynikiem wykonania tego kodu będzie liczba 55.

Wartość początkowa może być dowolna.

```
(1 to: 10) inject: 45 into: [:sum :each | sum + each].
```

Wynikiem wykonania tego kodu będzie liczba 100 (45+55).

Poniżej trochę trudniejszy przykład:

```
„Uporządkowana kolekcja z trzema podkolekcjami typu Array.”
collection := OrderedCollection with: #(1 2 3) with: #(3 4 5) with: #(3 6).

„Dodajemy wszystkie elementy podkolekcji do nowej kolekcji Set”
collection
      inject: Set new
      into: [:result :each | result addAll: each; yourself].
```

Wynikiem wykonania tego kodu będzie Set (1 2 3 4 5 6). Liczba 3 występuje tutaj tylko jeden raz, ponieważ Set filtruje duplikaty.

2.2.7 Ilość elementów spełniających warunek – metoda count:

Metoda `count:` zwraca ilość elementów kolekcji spełniających warunek zdefiniowany w bloku kodu będącym argumentem tej metody.

```
#(1 7 4 3 9) count: [:each | each > 4].
```

Wynikiem wykonania tego kodu będzie 2, ponieważ tylko dwa elementy są większe od 4 (7 i 9)..

2.2.8 Sprawdzanie obecności obiektu w kolekcji – metoda includes:

Metoda `includes:` sprawdza, czy jej argument znajduje się w kolekcji.

```
#(2 6 3 8 9) includes: 8.
```

Wynikiem wykonania tego kodu będzie wartość true Ponieważ kolekcja zawiera liczbę 8.

2.2.9 Sprawdzanie obecności elementu spełniającego warunek – metoda anySatisfy:

Metoda `anySatisfy:` sprawdza, czy kolekcja zawiera przynajmniej jeden element spełniający warunek zdefiniowany w bloku kodu będącym argumentem tej metody.

```
#(2 5 3 8 7) anySatisfy: [:each | each < 3].
```

Wynikiem wykonania tego kodu będzie wartość true ponieważ liczba 2 spełnia zadany warunek.

```
#(2 5 3 8 7) anySatisfy: [:each | each * 10 < 2].
```

Wynikiem wykonania tego kodu będzie wartość false ponieważ żadna liczba z kolekcji pomnożona przez dziesięć nie jest mniejsza od 2.

2.3 Wybrane kolekcje i ich właściwości

Poniżej przedstawionych zostało kilka wybranych, najczęściej używanych kolekcji języka Smalltalk.

2.3.1 OrderedCollection

`OrderedCollection` jest uporządkowaną kolekcją (liniową listą) o dynamicznym (zmiennym) rozmiarze. Jest to najczęściej używana kolekcja do ogólnych zastosowań. Jej elementy są uporządkowane w kolejności ich dodawania.

```
collection := OrderedCollection new.
collection add: 5; add: 3; add: 7.
collection asArray
```

Wynikiem wykonania tego kodu będzie #(5 3 7).

Metody `first` i `last` zwracają odpowiednio pierwszy i ostatni element kolekcji. Jeżeli kolekcja jest pusta, to wywoływany jest sygnał błędu.

```
collection := #(1 4 5 8) asOrderedCollection.
collection first. „Zwraca 1”
collection last. „Zwraca 8”
```

Stos

W języku Smalltalk nie istnieje klasa reprezentująca stos. Funkcjonalność stosu jest natomiast zaimplementowana w klasie `OrderedCollection`. Są to metody `addLast:` (odpowiednik polecenia `push`), `removeLast` (odpowiednik `pop`), `last` (odpowiednik `top`), `size` (odpowiednik `depth`) oraz `isEmpty` (odpowiednik polecenia `empty`).

Programista chcący uzywać składni podobnej do standardowej składni stosu może ją sobie w prosty i szybki sposób zaimplementować:

```
push: aNewObject
      self addLast: aNewObject
```

```
pop
      ^self isEmpty ifFalse: [self removeLast]
```

```
top
      ^self isEmpty ifFalse: [self last]
```

Kolejka

W ten sam sposób jak w przypadku stosu symulujemy w Smalltalku funkcjonalność kolejki: `addLast:` jest odpowiednikiem polecenia `add`, `removeFirst` to odpowiednik `remove`, `first` jest odpowiednikiem `next`, `isEmpty` to odpowiednik `empty`, a `size` to odpowiednik polecenia `length`. Podobnie jak w przypadku stosu, programista chcący uzywać składni podobnej do standardowej składni kolejki może ją sobie w prosty i szybki sposób zaimplementować:

```
add: aNewObject
      self addLast: aNewObject
```

```
delete
      ^self isEmpty ifFalse: [self removeFirst]
```

```
next
	^self isEmpty ifFalse: [self first]
```

2.3.2 SortedCollection

`SortedCollection` zachowuje się podobnie do `OrderedCollection`, z tą różnicą, że jej elementy są posortowane.

```
#(2 4 3 5 1) asSortedCollection asArray.
```

Wynikiem wykonania tego kodu będzie #(1 2 3 4 5).

A co możemy zrobić jeśli chcemy na przykład posortować obiekty typu `String` ze względu na ilość znaków? Użyjemy wtedy metodę `asSortedCollection:`

```
col := #('Tomaszek' 'Zosia' 'Mateusz').
col := col asSortedCollection: [:el1 :el2 | el1 size <= el2 size ].
col asArray
```

Wynik wykonania tego kodu to #('Zosia' 'Mateusz' 'Tomaszek').

Można też zdefiniować własny sposób sortowania za pomocą metody `sortBlock:`.

```
col := #(2 4 3 5 1) asSortedCollection.
col sortBlock: [:el1 :el2 | el1 > el2].
col asArray
```

Wynikiem wykonania tego kodu będzie #(5 4 3 2 1).

Do prawidłowego działania kolekcji `SortedCollection` konieczne jest, aby elementy tej kolekcji posiadały implementację metody <= . Dlatego programista tworząc nową klasę, której egzemplarze będą przetwarzane w kolekcjach, powinien stworzyć własną implementację tej metody. W następnym rozdziale zostanie przedstawiona taka przykładowa implementacja.

2.3.2.1 Sortowanie elementów kolekcji – definiowanie metody <=

Wyobrazmy sobie następującą sytuację: utworzyliśmy nową klasę `Song` z dwoma atrybutami `title` i `artist`, utworzyliśmy też kilka piosenek i dodaliśmy je do kolekcji typu

`OrderedCollection`. Chcemy teraz nasze piosenki posortować. Używamy do tego metody `asSortedCollection`.

```
col := OrderedCollection with: (Song title: 'Piosenka' artist: 'Jas') with:
(Song title: 'Moja etiuda' artist: 'Rysiek Mocart') with: (Song title:
'Przyspiewka' artist: 'Gospodyni').
col asSortedCollection asArray
```

Wykonanie powyższego kodu spowoduje wystąpienie błedu „Message not understood: #<=", ponieważ metoda sortująca używa metody <=. Metoda ta nie jest zdefiniowana w nadklasie `Object` naszej klasy `Song`, w związku z czym musimy ją zdefiniować sami, bezpośrednio w naszej nowej klasie.

```
<= anObject

      ^self class == anObject class and: [self title <= anObject title]
```

Teraz powyższy kod będzie działał poprawnie.

Innym rozwiązaniem, nie wymagającym implementacji metody <= w klasie Song, będzie bezpośrednie przekazanie bloku sortującego za pomocą metody asSortedCollection:

```
col := OrderedCollection with: (Song title: 'Piosenka' artist: 'Jas') with:
(Song title: 'Moja etiuda' artist: 'Rysiek Mocart') with: (Song title:
'Przyspiewka' artist: 'Gospodyni').
(col asSortedCollection: [:el1 :el2 | el1 title <= el2 title]) asArray
```

2.3.3 Array

`Array` jest kolekcją indeksowaną o ściśle określonej wielkości. Jej egzemplarze można tworzyć przy pomocy metody `new: anAmount` lub pisząc bezpośrednio `#()`.

```
Array new: 5. „Kolekcja pięcioelementowa".
#(2 3 4 5) „Kolekcja z trzema elementami 2 3 4 i 5."
```

Elementy kolekcji możemy dodawać lub zmieniać za pomocą metody `at: anIndex put: anElement` oraz odczytywać za pomocą metody `at: anIndex`.

```
col := Array new: 3.
col at: 1 put: 5.
col at: 1.
„Wynikiem wykonania tego kodu będzie 5.”
col at: 1 put: 6.
col at: 1.
„Wynikiem wykonania tego kodu będzie 1 (1 zastąpiło 5 na pierwszej pozycji).”
```

2.3.4 Set

`Set` jest kolekcją nieuporządkowaną, która filtruje duplikaty elementów.

```
col := Set new.
col add: 3; add: 5; add: 4; add: 3. „Dodajemy liczbę 3 dwukrotnie.“
col asArray.
„Wynik działania tego kodu to #(3 4 5). Liczba 3 występuje tylko raz.“
```

W celu usunięcia duplikatów z dowolnej kolekcji wystarczy więc wysłać do niej metodę `asSet`.

```
#( 2 3 5 3 6 8 6 3) asSet asArray.
„Wynikiem tego kodu będzie #(2 3 5 6 8).”
```

Do prawidłowego działania kolekcji `Set` konieczne jest, aby elementy tej kolekcji miały odpowiednie implementacje metod = i `hash`. Dlatego programista tworząc nową klasę, której egzemplarze będą przetwarzane w kolekcjach, powinien stworzyć własną implementację tych metod. W następnym rozdziale zostanie przedstawiona taka przykładowa implementacja.

2.3.4.1 Stosowanie kolekcji Set – definiowanie metod = oraz hash

Wyobrazmy sobie następującą sytuację: utworzyliśmy nową klasę `Song` z dwoma atrybutami `title` i `artist`, utworzyliśmy też kilka piosenek i dodaliśmy je do kolekcji typu `OrderedCollection`. Niektóre piosenki wystepują jednak w naszej kolekcji podwójnie i chcemy usunąć duplikaty. Używamy do tego metody `asSet`.

```
 „Tworzymy kolekcję z trzech elementów, jedna piosenka występuje podwójnie.”
col := OrderedCollection with: (Song title: 'Piosenka' artist: 'Jas') with:
(Song title: 'Moja etiuda' artist: 'Rysiek Mocart') with: (Song title:
'Piosenka' artist: 'Jas').
col asSet asOrderedCollection
```

Nasza kolekcja ma nadal 3 elementy, w tym jeden podwójny. Dlaczego? Ponieważ klasa `Set` używa do rozpoznawania takich samych obiektów dwóch metod: = i `hash`. Spójrzmy na implementację metody = w klasie `Object`

```
= anObject
      "Answer whether the receiver and the argument represent the same object.
      If = is redefined in any subclass, consider also redefining the message
hash."
      ^self == anObject
```

Jak widzimy standardowo metoda = sprawdza czy dany obiekt i jego argument, to ten sam obiekt, a nie czy te obiekty są takie same. My natomiast utworzyliśmy dwa różne obiekty o tym samym tytule i nazwie artysty!

W celu usunięcia duplikatów nie będących tym samym obiektem musimy w naszej klasie `Song` zdefiniować metodę =.

```
= anObject
      ^self class == anObject class
            and: [self title = anObject title
                  and: [self artist = anObject artist]]
```

Tak samo musimy postąpić z metodą `hash`

```
hash
      ^self title hash bitXor: self artist hash
```

Teraz nasz kod będzie już działał poprawnie i odfiltruje duplikaty piosenek.

2.3.5 Dictionary

Dostęp do elementów kolekcji typu `Array` czy `OrderedCollection` jest możliwy za pomocą indeksów, np.: `anArray at: 1`. Kolekcja `Dictionary` natomiast umożliwia dostęp do przechowywanych elementów za pomocą kluczy, przy czym jako klucz może być użyty dowolny obiekt z zaimplementowanymi metodami = i `hash`. Najczęściej jako kluczy używa się egzemplarzy klasy `Symbol`.

Dodawanie elementów do `Dictionary` odbywa się za pomocą metody `at: aKey put: anObject`

```
col := Dictionary new.
col
      at: #niebieski put: ColorValue blue;
      at: #czerwony put: ColorValue red.
```

Odczyt elementu umieszczonego pod danym kluczem jest możliwy za pomocą metody `at:`

```
col at: niebieski. „Zwraca ColorValue blue”.
```

W przypadku gdy nie jesteśmy pewni, czy dany klucz jest już zdefiniowany w `Dictionary` do odczytu z jednoczesnym ewentulnym wstawieniem wartości domyslnej możemy uzyć metody `at:ifAbsentPut:`

```
color := col at: #czarny ifAbsentPut: [ColorValue black].
```

W powyższym przykładzie do `Dictionary` pod kluczem `#czarny` najpierw zostanie dodany nowy element `ColorValue black`, a następnie ten element zostanie przypisany do zmiennej `color`.

```
color := col at: #niebieski ifAbsentPut: [ColorValue yellow].
```

Nasza kolekcja zawiera już klucz `#niebieski`, dlatego obiekt `ColorValue yellow` z bloku `ifAbsentPut:` nie zostaniedo niej wstawiony, a do zmiennej `color` zostanie przypisany już istniejący pod tym kluczem w kolekcji obiekt `ColorValue blue`.

Jak już zostało na wstępie powiedziane, kluczami w Dictionary mogą być dowolne obiekty.

```
aDictionary at: aName put: anAddress.
```

Wymagane jest jednak, żeby obiekt będacy kluczem miał zaimplementowane metody = i `hash`. Opis implementacji tych metod znajduje się w rozdziale poświęconym kolekcji Set, również używającej tych metod dla zarządzania swoimi elementami (rozdział 2.3.4.1).

2.3.5.1 Iteracja po kolekcji Dictionary

Metody iteracyjne `do:`, `select:`, `collect:` itp. nie używają kluczy, a wyłącznie wartości umieszczonych pod nimi.

```
col := Dictionary new.
col
at: #Sprzedawca put: 'Mateusz';
at: #Kierownik put: 'Anna';
at: #Dostawca put: 'Tomek'.
col do: [:each | Transcript cr; show: each ].
```

Na ekranie zostanie wyświetlone:

```
'Mateusz'

'Anna'

'Tomek'
```

W celu umożliwienia iteracji po kluczach i wartościach kolekcja `Dictionary` udostępnia metodę `keysAndValuesDo:`. Zastosujmy ją do powyższego przykładu:

```
col keysAndValuesDo: [:key :value | Transcript cr; show: key; space; show:
value].
```

Na ekranie zostanie wyświetlone:

```
Sprzedawca 'Mateusz'

Kierownik 'Anna'
```

```
Dostawca 'Tomek'
```

`Dictionary` udostępnia jeszcze dwie kolejne przydatne metody – `keys` i `values`. Metoda `keys` zwraca kolekcję `Set` z wszystkimi kluczami zdefiniowanymi w kolekcji `Dictionary`, a metoda `values` zwraca kolekcję `OrderedCollection` z wszystkimi wartościami.

Użyjmy kolekcji z naszego poprzedniego przykładu:

```
Transcript show: col keys printString.
```

Wynik na ekranie to Set (#Sprzedawca #Kierownik #Dostawca).

```
Transcript show: col values printString.
```

Wynik na ekranie to OrderedCollection ('Anna' 'Tomek' 'Mateusz').

2.3.6 Interval

Kolekcja `Interval` reprezentuje pewien zakres liczb. Przykładowo zakres liczb od 1 do 10 jest zdefiniowany następująco:

```
Interval from: 1 to: 10.
```

Można taki `Interval` utworzyć też prościej, wysyłając do egzemplarza klasy `Integer` metodę `to:`

```
(1 to: 10).
```

Można także zdefiniować odstęp pomiędzy poszczególnymi elementami, na przykład chcąc uzyskać kolekcję `Array` z wszystkimi liczbami parzystymi od 0 do 10 możemy napisać:

```
(Interval from: 2 to: 10 by: 2) asArray.
```

lub prościej (metoda `to:by:` wysłana do egzemplarza klasy `Integer`):

```
(1 to: 10 by: 2) asArray.
```

Wynikiem wykonania tego kodu będzie kolekcja `#(2 4 6 8 10)`.

Odstęp pomiędzy elementami klasy `Interval` nie musi byż liczbą całkowitą:

```
(1 to: 20 by: 0.5) at: 2.
```

Wynikiem będzie tutaj liczba 1,5.

2.3.7 String

Klasa `String` reprezentuje kolekcję, ciąg znaków (egzemplarzy klasy `Character`). Kolekcja `String` jest uporządkowana, indeksowana, zmienialna i homogeniczna (może zawierać tylko znaki). Tak jak `Array` może być tworzona bezpośrednio za pomocą specjalnej składni lub tak jak wszystkie inne kolekcje.

```
'Witaj'.
String with: $a.
String with: $O with: $K.
String new.
String newFrom: #($W $i $t $a $j $!).
```

Ponieważ `String` jest kolekcją zmenialną, możemy zmieniać jej elementy używając standardowej metody `at:put:`

```
str = 'Ala ma kota'.
str at: 1 put: $O; at: 11 put: $y.
```

Wynikiem działania tego kodu będzie 'Ola ma koty.'.

Równoważność i identyczność

Dwa takie same ciągi znaków są różnymi obiektami, tzn. porównanie:

```
'aaa' = 'aaa'
```

da wynik `true`, ponieważ obie kolekcje zawierają takie same elementy (są wzajemnie równoważne), ale już:

```
'aaa' == 'aaa'
```

da nam wynik `false`, ponieważ obie kolekcje to dwa różne egzemplarze klasy `String` (nie są identyczne).

Łączenie ciągów znaków

Dwa ciągi znaków, tak jak wszystkie kolekcje, mogą być łączone za pomocą przecinka.

```
'Ala ma', ' kota'.
```

Metoda ta jest jednak bardzo niewydajna. Jeżeli łączymy w kodzie często ciągi znaków, to wskazane jest użyć strumieni (więcej w rozdziale poświęconym strumieniom). Zobaczmy przykład wielokrotnego łączenia ciągów znaków.

Przy pomocy przecinka:

```
str := String new.
1000 timesRepeat: [str := str, 'abcdefg'].
Transcript show: str.
```

Przy pomocy strumieni:

```
stream := WriteStream on: String new.
1000 timesRepeat: [stream nextPutAll: 'abcdefg'].
Transcript show: stream contents.
```

Wykonanie kodu `1000 timesRepeat: […]` trwało na komputerze autora w systemie VisualWorks 7.8 w pierwszym przypadku (z użyciem przecinka) **20** razy dłużej niż w przypadku drugim (z wykorzystaniem strumieni).

2.3.8 Symbol

Kolekcja `Symbol` jest podklasą klasy `String`. Jest on tworzony za pomocą znaku `#`.

```
#JakisSymbol.
```

Każdy symbol istnieje w systemie tylko jeden raz, tzn. w przeciwieństwie do ciągów znaków (klasa `String`) zarówno porównanie równoważności dwóch takich samych symboli

```
#abc = #abc
```

jak i ich identyczności

```
#abc == #abc
```

da nam wynik `true` (mamy doczynienia z tym samym obiektem).

Dzięki tej właściwości porównywanie symboli jest bardzo szybkie (wydajne), dlatego często stosuje się je jako klucze w różnych strukturach danych (np. w `Dictionary`).

Symbole są niemodyfikowalne

Symbole nie mogą być zmieniane po utworzeniu, tzn. metoda `at:put:` jest dla symboli zabroniona. Można natomiast odczytywać poszczególne znaki pod danym indeksem za pomocą metody `at:`

```
#abcd at: 3.
```

Wynik tego kodu to $c (egzemplarz klasy `Character`).

Łączenie symboli

Tak jak w nadklasie `String` (i w innych kolekcjach), egzemplarze symboli mogą być łączone przy pomocy przecinka. Wynikiem jednak nie będzie `Symbol`, lecz ciąg znaków.

```
#abcd, #efgh.
```

Wynik to `'abcdefgh'`.

Chcąc uzyskać jako wynik również symbol należy napisać:

```
(#abcd, #efgh) asSymbol.
```

Wynikiem będzie `#abcdefgh`.

2.4 Kolekcje - praktyczne wskazówki dla programisty Smalltalk

2.4.1 Prawidłowe użycie metody add:

Jedną z najczęstszych pomyłek w programowaniu w języku Smalltalk jest następujące użycie metody `add:`

```
| col |
col := OrderedCollection new add: 2; add: 6.
„Do zmiennej col została przypisana liczba 6, a nie kolekcja.”
```

Zmienna `col` zawiera po przypisaniu liczbę 6, a nie kolekcję jak zapewne chciał programista. Dlaczego? Przyjrzyjmy się implementacji metody `add:`

```
add: newObject
	"Include newObject as one of the receiver's elements.  Answer
newObject."
	^self addLast: newObject
```

Metoda `add:` wywołuje metodę `addLast:`, a ta jest zdefiniowana następująco:

```
addLast: newObject
	"Add newObject to the end of the receiver.  Answer newObject."
	lastIndex = self basicSize ifTrue: [self makeRoomAtLast].
	lastIndex := lastIndex + 1.
	self basicAt: lastIndex put: newObject.
	^newObject
```

Jak widać metoda `addLast:` w klasie `OrderedCollection`,a tym samym także metoda `add:` zwraca jako wynik dodawany obiekt i to właśnie ten ostatnio dodawany obiekt (liczba 6) został przypisany do naszej zmiennej `col`. Metoda `add:` zwraca we wszystkich kolekcjach języka Smalltalk dodawany obiekt. Tak samo działa też metoda `addAll: aCollection` dodająca do kolekcji wszystkie elementy innej kolekcji będącej argumentem tej metody.

Prawidłowo zapisany powyższy kod wyglądałby tak:

```
| col |
col := OrderedCollection new.
col add: 2; add: 6.
```

lub tak:

```
| col |
col := OrderedCollection new add: 2; add: 6; yourself.
```

Metoda `yourself` nie robi nic poza zwróceniem obiektu do którego została wysłana – w tym wypadku egzemplarza klasy `OrderedCollection`. Ten obiekt zostanie następnie przypisany do zmiennej `col`.

2.4.2 Usuwanie elementów z kolekcji podczas iteracji po niej

Często popełnianym błędem jest usuwanie elementów z kolekcji podczas iteracji po niej:

```
col := (2 to: 20) asOrderedCollection.
„Usuwamy wszystkie elementy niepodzielne przez 5”
col do: [:each | (each \\ 5) isZero ifFalse: [col remove: each]].
col asArray
```

W zależności od danej implementacji języka Smalltalk wykonanie powyższego kodu zakończy się błędem (np. w VisualWorks) lub zwróci niewłaściwy wynik (np. w Pharo wynik będzie #(3 5 7 9 10 12 14 15 17 19 20)).

Rozwiązaniem jest skopiowanie kolekcji przed iteracją po niej:

```
col := (2 to: 20) asOrderedCollection.
„Usuwamy wszystkie elementy niepodzielne przez 5”
col copy do: [:each | (each \\ 5) isZero ifFalse: [col remove: each]].
col asArray
```

Teraz wynik będzie już poprawny #(5 10 15 20).

3 Strumienie

Strumienie umożliwiają dostęp do dowolnej uporządkowamej struktury danych, niezależnie od zródła tych danych. Strumienie są używane do odczytu i zapisu zarówno wewnętrznych jak i zewntętrznych struktur danych (m.in. ciągów znaków, plików, zewnętrznych serwerów (połączeń sieciowych), kolekcji, procesów itp.) przy użyciu ustandaryzowanego interfejsu. Są one także używane do dostępu do dużych porcji danych, bez konieczności wczytywania całych objektów do pamięci komputera.

3.1 Tworzenie strumieni

Strumienie są najczęściej tworzone dla konkretnych obiektów:

```
ReadStream on: 'abcd'.
WriteStream on: OrderedCollection new.
'readme.txt' asFilename readStream.
'Ala ma kota' writeStream.
```

3.2 Operacje na strumieniach

Strumienie dzielą się na strumienie tylko do odczytu oraz do zapisu. W strumieniach do odczytu można zmieniać położenie wkaznika kolejnego elementu, w strumieniach do zapisu jest to niemożliwe. Jeżeli programista potrzebuje zmiany położenia wskaźnika w trakcie zapisu (na przykład do nadpisania już istniejących elementów), to musi on użyć strumienia do odczytu i zapisu.

3.2.1 Odczyt

next

Kolejne elementy strumienia odczytujemy za pomocą metody `next`.

```
stream := ReadStream on: 'Ala ma kota.'.
stream next.
```

Powyższy kod zwróci jako wynik znak $A (egzemplarz klasy `Character`). Kolejne wywołanie polecenia stream `next` zwróci $l, kolejne $a itd aż do osiągnięcia końca strumienia. Po osiągnięciu końca strumienia kolejne wywołanie metody `next` spowoduje zależnie od implementacji wystąpienie błędu lub zwrócenie wartości `nil`.

atEnd

To, czy wskaznik kolejnego elementu wskazuje już na koniec strumienia można sprawdzić przy pomocy metody `atEnd`.

```
stream := ReadStream on: ’a’.
stream atEnd. ”false, ponieważ wskaznik pokazuje $a.”
stream next.
stream atEnd. ”true, ponieważ wskaznik pokazuje koniec strumienia.”
```

next:

Można także odczytać dowolną ilość kolejnych elementów za pomocą metody `next:`

```
stream := ReadStream on: 'Ala ma kota'.
stream next: 6.
```

Powyższy kod zwróci ciąg znaków ‘Ala ma’.

Jeżeli w strumieniu nie ma żądanej ilości elementów, to wystąpi błąd. Dlatego dotstępna jest kolejna metoda nextAvailable:

nextAvailable:

Metoda `nextAvailable:` zwraca zadaną ilość elementów strumienia. Jeżeli strumień nie posiada żądanej ilości elementów, to zwróconych zostanie tyle, ile jest dostępnych.

```
stream := ReadStream on: 'Ala ma kota'.
stream nextAvailable: 600.
```

Powyższy kod zwróci ciąg znaków ‘Ala ma kota’.

through: i throughAll:

Meoda `through:` zwraca elementy od aktualnej pozycji wskaznika odczytu do pierwszego wystąpienia obiektu będącego argumentem tej metody, włącznie z tym obiektem.

```
stream := ReadStream on: 'Ala ma kota'.
stream through: Character space.
```

Powyższy kod zwróci ciąg znaków ‘Ala ’ (ze spacją włącznie).

Jeżeli żądany element nie zostanie znaleziony w strumieniu, to metoda ta zwróci zawartość strumienia od bieżącej pozycji wskaznika do końca strumienia.

```
stream := ReadStream on: 'Ala ma kota'.
stream through: $x.
```

Wynikiem wykonania tego kodu będzie ciąg znaków ‘Ala ma kota’.

Wariantem tej metody jest `throughAll:`, jej argumentem jest kolekcja elementów.

```
stream := ReadStream on: 'Ala ma kota'.
stream throughAll: 'ma'.
```

Wynikiem tego kodu będzie ciąg znaków ‘Ala ma’.

Jeżeli żądana kolekcja nie zostanie znaleziona w strumieniu, to metoda ta zwróci zawartość strumienia od bieżącej pozycji wskaznika do końca strumienia.

```
stream := ReadStream on: 'Ala ma kota'.
stream throughAll: 'ona'.
```

Wynikiem wykonania tego kodu będzie ciąg znaków ‘Ala ma kota’.

peekFor:

Jeżeli kolejny obiekt w strumieniu jest równy argumentowi metody `peekFor:`, to metoda ta zwraca wartość `true` i przesuwa wskaznik za ten element. W przeciwnym wypadku metoda ta zwraca wartość `false`, a wskaznik nie ulega zmianie.

```
stream := ReadStream on: 'Ala ma kota'.
stream peekFor: $A.
stream next.
```

W powyższym przykładzie metoda `peekFor:` znalazła znak $A, wskaznik został przesunięty i metoda `next` zwraca kolejny element, czyli znak $l.

```
stream := ReadStream on: 'Ala ma kota'.
stream peekFor: $X.
stream next.
```

W tym przykładzie metoda `peekFor:` nie znajduje znaku $X, wskaznik się nie zmienia, a metoda `next` zwraca znak $A.

upTo:

Meoda `upTo:` zwraca elementy od aktualnej pozycji wskaznika odczytu do pierwszego wystąpienia obiektu będącego argumentem tej metody, ale bez tego obiektu. Wskaznik odczytu zostaje ustawiony za znalezionym obiektem.

```
stream := ReadStream on: 'Ala ma kota'.
stream upTo: Character space.
stream next.
```

W powyższym przykładzie metoda `upTo:` zwraca ciąg znaków ‘Ala’ (bez spacji), a następnie wywołana metoda `next` zwraca znak $m, ponieważ wskaznik został przez metodę `upTo:` przesunięty za spację.

Jeżeli żądany obiekt nie zostanie znaleziony, to metoda `upTo:` zwraca całą pozostałą zawartość strumienia, a wskaznik zostaje ustawiony na jego końcu.

upToEnd

Metoda `upToEnd` zwraca zawartość strumienia od aktualnej pozycji wskaznika odczytu do końca strumienia.

```
stream := ReadStream on: 'Ala ma kota'.
stream upTo: Character space.
stream upToEnd.
```

W powyższym przykładzie metoda `upTo:` ustawia wskaznik odczytu za pierwszą spacją, a metoda `upToEnd` zwraca ciąg znaków ‘ma kota’.

upToAndSkipThroughAll:

Metoda `upToAndSkipThroughAll:` zwraca zawartość strumienia od aktualnej pozycji wskaznika odczytu do pierwszego wystąpienia kolekcji będącej argumentem tej metody, ale bez tej kolekcji. Następnie wskaznik odczytu zostaje ustawiony za ostatnim elementem tej kolekcji w strumieniu. Jeżeli żądana kolekcja nie zostanie znaleziona, metoda `upToAndSkipThroughAll:` zwraca całą zawartość strumienia od aktualnej pozycji wskaźnika do jego końca, a wskaznik zostaje ustawiony na końcu strumienia.

```
stream := ReadStream on: 'Ala ma kota'.
stream upToAndSkipThroughAll: 'kot'.
stream next.
```

W powyższym przykładzie metoda `upToAndSkipThroughAll:` zwróci ciąg znaków ‘Ala ma ’, a następująca po niej metoda `next` zwróci znak $a.

```
stream := ReadStream on: 'Ala ma kota'.
stream upToAndSkipThroughAll: 'Tomek'.
stream atEnd.
```

W tym przykładzie metoda `upToAndSkipThroughAll:` zwróci ciąg znaków ‘Ala ma kota’, a następująca po niej metoda `atEnd` zwróci wartość `true` (tzn. wskaznik odczytu znajduje się na końcu strumienia).

3.2.2 Zapis

nextPut:

Metoda nextPut: wstawia obiekt będący jej argumentem do strumienia, pod aktualną pozycją wskaznika zapisu i zwraca jako wynik ten obiekt.

```
stream := WriteStream on: String new.
stream nextPut: $x.
stream contents.
```

W powyższym przykładzie znak $x jest wstawiany na początku strumienia. Metoda `contents` zwraca kolekcję ‘x’.

nextPutAll:

Metoda `nextPutAll:` wstawia kolekcję będącą jej argumentem do strumienia, pod aktualną pozycją wskaznika zapisu i zwraca jako wynik tę kolekcję.

```
stream := WriteStream on: String new.
stream nextPutAll: 'Witaj'.
stream contents.
```

W powyższym przykładzie kolekcja ‘Witaj’ jest wstawiana na początku strumienia. Metoda `contents` zwraca następnie zawartość strumienia, czyli ciąg znaków ‘Witaj’.

print:

Metoda `print:` wstawia do strumienia wynik metody `printString` zdefiniowanej w obiekcie będącym jej argumentem. W przypadku gdy metoda ta nie jest w danym obiekcie zaimplementowana, zostanie użyta metoda `printString` zdefiniowana w nadklasie `Object`.

```
stream := WriteStream on: String new.
stream
      print: OrderedCollection new;
      print: 725.
stream contents.
```

W powyższym przykładzie metoda `print:` wstawi do strumienia najpierw ciąg znaków ‘OrderedCollection ()‘, a następnie ‘725’. Metoda `contents` zwróci nam ciąg znaków ‘OrderedCollection ()725’.

3.2.3 Zmiana pozycji wskaznika

Po utworzeniu strumienia wskaznik pozycji znajduje się na jego początku (wskazuje pierwszy element strumienia – pozycja 0). Odczyt ze strumienia lub zapis do niego przesuwają odpowiednio ten wskaźnik.

position

Metoda position zwraca aktualną pozycję wskaznika.

```
stream := ReadStream on: 'Ala ma kota'.
stream next.
stream next.
stream position.
```

W powyższym przykładzie metoda `position` zwróci liczbę 2.

position:

Metoda `position:` ustawia aktualny wskaznik zapisu lub odczytu na pozycję oznaczoną liczbą będącą argumentem tej metody. Jeżeli liczba ta będzie większa niż rozmiar strumienia, to wystąpi błąd.

```
stream := ReadStream on: 'Ala ma kota'.
stream position: 4.
stream next.
```

W powyższym przykładzie wskaznik zostanie ustawiony za pierwszą spacją, a następnie wywołana metoda `next` zwróci znak $m.

```
stream := ReadStream on: 'Ala ma kota'.
stream position: stream size - 1.
stream next.
```

W tym przykładzie wskaznik zostanie ustawiony na pozycji 10 (rozmiar strumienia minus jeden), a metoda `next` zwróci znak $a.

reset

Metoda `reset` ustawia wskaznik aktualnej pozycji odczytu lub zapisu na początku strumienia.

```
stream := ReadStream on: 'Ala ma kota'.
stream
      next;
      next;
      next;
      reset;
      next.
```

W powyższym przykładzie pierwsze trzy odczyty przy pomocy metody `next` przesuną wskaznik na pozycję 3 (przed znak spacji po słowie Ala), metoda `reset` cofnie wskaznik na pozycję 0, a kolejne wywołanie metody `next` zwróci znak $A.

setToEnd

Metoda `setToEnd` działa podobnie do metody `reset`, z tym że przesuwa wskaznik aktualnej pozycji odczytu lub zapisu na koniec strumienia.

```
stream := ReadStream on: 'Ala ma kota'.
stream
      position;
      setToEnd;
      position.
```

W powyższym przykładzie pierwsze wywołanie metody position zwróci liczbę 0 (początek strumienia), setToEnd przesunie wskaznik na koniec strumienia, a kolejne wywołanie metody position zwróci liczbę 11.

skip:

Metoda `skip:` przesuwa wskaznik odczytu lub zapisu o liczbę będącą jej argumentem. Liczba ta może być również ujemna.

```
stream := ReadStream on: 'Ala ma kota'.
stream
      skip: 4;
      next.
```

W powyższym przykładzie metoda `skip:` przesuwa wskaznik odczytu o 4 pozycje do przodu (przed słowo ”ma”). Następnie wywołana metoda `next` zwraca znak $m.

skipThrough:

Metoda `skipThrough:` przesuwa wskaznik odczytu lub zapisu za pierwsze wystąpienie obiektu będącego jej argumentem i zwraca swój strumień. Jeżeli żądany obiekt nie zostanie znaleziony, to zwracana jest wartość nil, a wskaznik jest ustawiany na koniec strumienia.

```
stream := ReadStream on: 'Ala ma kota'.
stream
      skipThrough: $k;
      next.
```

W powyższym przykładzie metoda `skipThrough:` przesuwa wskaznik odczytu za znak $k w słowie „kot”. Następnie wywołana metoda `next` zwraca znak $o.

```
stream := ReadStream on: 'Ala ma kota'.
stream
      skipThrough: $x;
      next
```

W tym przykładzie metoda `skipThrough:` przesuwa wskaznik odczytu na koniec strumienia (ponieważ znak $x w nim nie występuje), a metoda `next` zwraca wartość `nil`.

skipThroughAll:

Metoda `skipThroughAll:` przesuwa wskaznik odczytu lub zapisu za pierwsze wystąpienie kolekcji będącej jej argumentem i zwraca swój strumień. Jeżeli żądana kolekcja nie zostanie znaleziona, to zwracana jest wartość `nil`, a wskaznik jest ustawiany na koniec strumienia.

```
stream := ReadStream on: 'Ala ma kota'.
stream
      skipThroughAll: 'ma ko';
      next.
```

W powyższym przykładzie metoda `skipThroughAll:` przesuwa wskaznik odczytu za kolekcję 'ma ko'. Następnie wywołana metoda `next` zwraca znak $t.

```
stream := ReadStream on: 'Ala ma kota'.
stream
      skipThroughAll: 'Tomek';
      next.
```

W tym przykładzie metoda `skipThroughAll:` przesuwa wskaznik odczytu na koniec strumienia (ponieważ kolekcja 'Tomek' w nim nie występuje), a metoda `next` zwraca wartość `nil`.

skipUpTo:

Metoda `skipUpTo:` przesuwa wskaznik odczytu lub zapisu przed pierwsze wystąpienie obiektu będącego jej argumentem.. Jeżeli żądany obekt nie zostanie znaleziony, to zwracana jest wartość `nil`, a wskaznik jest ustawiany na koniec strumienia.

```
stream := ReadStream on: 'Ala ma kota'.
stream
      skipUpTo: $m;
      next.
```

W powyższym przykładzie metoda `skipUpTo:` przesuwa wskaznik odczytu do znaku $m. Następnie wywołana metoda `next` zwraca ten znak.

```
stream := ReadStream on: 'Ala ma kota'.
stream
      skipUpTo: $x;
      next.
```

W tym przykładzie metoda `skipUpTo:` przesuwa wskaznik odczytu na koniec strumienia (ponieważ znak $x w nim nie występuje), a metoda `next` zwraca wartość `nil`.

3.2.4 Zamykanie strumieni

close

Strumienie wewnętrzne nie muszą być zamykane. Strumienie zewnętrzne należy zamykać wysyłając do nich metodę `close`.

commit

Metoda `commit` zapisuje zawartość bufora do strumienia zewnętrznego (np.pliku) Powinna być wysłana do takiego strumienia przed jego zamknięciem, jeżeli wcześniej byly w nim zapisywane jakieś informacje.

3.3 Strumienie zewnętrzne

Strumienie zewnętrzne zapewniają dostęp do zewnętrznych strumieni danych takich jak pliki, bazy danych czy połączenia sieciowe. Podstawowe operacje są tutaj takie same jak w przypadku strumieni wewnętrznych, niektóre metody różnią się jednak zasadniczo.

3.3.1 Tworzenie strumieni zewnętrznych

Strumienie zewnętrzne są tworzone przez wysłanie do obiektu reprezentującego dany typ danych jednej z następujących metod: `readStream`, `writeStream`, `appendStream`, `readWriteStream`, czy `readAppendStream`.

Poniżej przdstawiony jest przykład dodania do pliku tekstowego 'wierszyk.txt' w nowej linii zdania 'Ala ma kota.'.

```
file := 'wierszyk.txt' asFilename.
stream := file appendStream.
stream cr; nextPutAll: 'Ala ma kota'.
stream commit.
```

Metoda `commit` została na końcu użyta w celu zapisu do pliku bufora w systemie operacyjnym. W przypadku zamknięcia dostępu do pliku metodą `close` nie trzeba używać `commit`, bufor jest opróżniany automatycznie.

W przypadku otwierania istniejącego pliku jako strumienia do zapisu, jego zawartość zostanie nadpisana natychmiast po otwarciu, a jego rozmiar zostanie ustawiony na 0.

```
file := 'readme.txt' asFilename.
stream := file writeStream.
```

Dzieje się tak, ponieważ po otwarciu pliku metodą `writeStream` wskaznik pozycji zapisu znajduje się na początku pliku. W celu dopisania czegoś do już istniejącego pliku należy użyć metodę `appendStream`.

Standardowo kolekcje zewnętrzne są otwierane w trybie tekstowym. W celu ustawienia trybu binarnego do strumienia nalezy wysłać polecenie `binary`.

```
file := 'program.exe' asFilename.
stream := file readStream binary.
```

3.3.2 Zapis i odczyt strumieni zewnętrznych

Do zapisu i odczytu strumieni zewnętrznych używa się tych samych metod co w przypadku strumieni wewnętrznych, tzn, `next:`, `nextPut:` itd. Przy odczycie należy sprawdzać, czy nie został już osiągnięty koniec danego strumienia.

Poniżej przedstawiony został przykład kopiowania zawartości pliku tekstowego 'zrodlo.txt' do nowego pliku o nazwie 'nowy.txt'.

```
readStream := 'zrodlo.txt' asFilename readStream.
writeStream := 'nowy.txt' asFilename writeStream.
[readStream atEnd] whileFalse:[ | char |
      char := readStream next.
      writeStream nextPut: char].
```

Przesuwanie wskaznika aktualnej pozycji odczytu lub zapisu (metoda `position:`) nie jest możliwe dla strumieni tylko do zapisu, ponieważ do jego przesunięcia wymagana jest funkcjonalność odczytu.Aby przesunąć wskaznik podczas zapisu należy uzyć strumienia do odczytu i zapisu.

4 Skąd można bezpłatnie pobrać środowisko Smalltalk?

4.1 VisualWorks firmy Cincom

Firma Cincom udostępnia bezpłatnie na własny użytek najpopularniejsze obecnie środowisko i dialekt języka Smalltalk VisualWorks. Można je pobrać na stronie internetowej pod adresem http://www.cincomsmalltalk.com/main/products/visualworks/.

Środowsko to jest używane w średnich i dużych projektach głównie przez firmy przemysłowe i z sektora finansowego (banki, ubezpieczenia) w USA, Niemczech, Wielkiej Brytanii i Holandii.

4.2 Pharo – Open Source Smalltalk

Popularne środowisko Smalltalk rozpowszechniane na zasadzie Open Source. Można je pobrać pod adresem http://pharo.org/.

Środowisko to jest głównie używane do tworzenia aplikacji i stron internetowych. M.in. strona autora jest dynamicznie generowana przez Pharo Smalltalk, z wykorzystaniem web framework Seaside i systemu zarządzania treścią PierCMS. Również strony projektu pharo.org generowane są przez Pharo Smalltalk, z wykorzystaniem Amber Smalltalk (akwiwalent JavaScript) i web framework Marina.