Глава 1

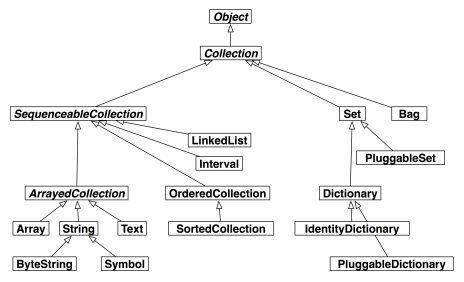
**Колекції**

**1.1 Вступ**

Класи колекцій формують групу класів призначених для загальних потреб, похідних від Collection і Stream. Група класів з Blue Book налічує 17 похідних класів від Collection і 9 похідних від Stream. Вона була декілька разів переглянута перед релізом системи Smalltalk-80. Ця група класів часто розглядається як парадигматичний приклад об’єктно-орієнтованого проектування.

У Pharo існує 101 похідний клас від абстрактного класу Collection, та 50 від абстрактного класу Stream, але багато з них (наприклад Bitmap, FileStream чи CompiledMethod)   
мають спеціальне призначення і створені щоб використовуватись в інших частинах системи чи в програмах, і відтак не віднесені до категорії колекцій в системній організації. У цій главі ми використовуватимемо термін Collection Hierarchy щоб позначити клас Collection і 47 його похідних класів, які також розташовані у пакетах, позначених Collections-\*. Термін Stream Hierarchy використовуватимемо щоб позначити клас Stream і 9 його похідних класів, які також є у пакетах Collections-Streams. Ці 56 класів реагують на 982 повідомлень і визначають в загальному 1609 методів!

У цій главі ми зосередимось в основному на підмножині класів колекцій, які показані на *Малюнку 1*. Потоки будуть розглянуть окремо у Главі ??:Потоки.



Малюнок 1. Деякі основні колекції y Pharo.

**1.2 Різноманіття колекцій**

Щоб ефективно використовувати класи колекцій, читачу необхідно мати хоча б поверхневі знання широкого різноманіття колекцій, які вони реалізовують, їхні схожі властивості та різницю.

Програмування з використанням колекцій, а не окремими елементами є важливим способом підвищення рівня абстракції програми. Розподіл функцій Lisp, який застосовує аргумент функції для кожного елементу списку і повертає новий список, що містить результати, є раннім прикладом цього стилю, але Smalltalk-80 прийняв програмування, що базується на колекціях, за основний принцип. Сучасні функціональні мови програмування, такі як ML і Haskell послідували прикладу Smalltalk.

Чому це хороша ідея? Припустимо, що ви маєте структуру,яка містить колекцію записів про студентів, і хочете виконати якусь дію над всіма студентами, за певним критерієм. Програмісти, які використовують імперативні мови, одразу ж використають цикл. Але програміст Smalltalk напише:

students select: [ :each | each gpa < threshold ]

Ця конструкція створює нову колекцію, яка містить конкретно тих студентів, для яких умова у квадратних дужках виконується (під виразом у дужках може розумітись лямбда-вираз, який описує функцію x. x gpa *<* threshold). Код Smalltalk є простим і елегантним, як спеціалізовані мови запитів.

Повідомлення select: розуміють всі колекції Smalltalk. Не потрібно визначати, чи структура даних про студентів є масивом чи зв’язним списком, повідомлення select: розуміють обоє. Зауважте, що таке використання відрізняється від циклу, де наперед потрібно знати з масивом, чи зі списком ми працюємо.

Коли хтось говорить про колекцію у Smalltalk, не вказуючи конкретно вид колекції, то він має на увазі об’єкт, який підтримує чітко визначені протоколи для тестування членів та перерахунку елементів. Усі колекції розуміють повідомлення для перевірки: includes:, isEmpty і occurrencesOf:.

Усі колекції розуміють повідомлення перерахунку: do:, select:, reject: (протилежне до select:), collect: (таке ж як map у Lisp), detect:ifNone:, inject:into: (вставляє зліва) та багато інших. Повсюдність цього протоколу, так само, як і різноманіття, роблять його таким потужним.

Наступна таблиця підсумовує стандартні протоколи, які підтримуються більшістю класів у ієрархії колекцій. Ці методи визначені, перевизначені, оптимізовані чи іноді навіть заборонені похідними класами Collection.

**Протокол Методи**

*доступ* size, capacity, at:, at:put:

*перевірка* isEmpty, includes:, contains:, occurrencesOf:

*додавання* add:, addAll:

*видалення* remove:, remove:ifAbsent:, removeAll:

*перерахунок* do:, collect:, select:, reject:, detect:, detect:ifNone:, inject:into:

*конвертація* asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection:

*створення* with:, with:with:, with:with:with:, with:with:with:with:, withAll:

Поза цією базовою однорідністю існує багато різних видів колекцій, які або підтримують різні протоколи, або забезпечують різну поведінку для однакових запитів. Давайте коротко розглянемо основні відмінності:

**Визначення порядку послідовності:** Екзампляри всіх похідних класівCollection починають з першого елемента і продовжують в чітко визначеному порядку до останнього елементу. Навпаки, екземпляри класів Set, Bag і Dictionary, не допускають визначення порядку послідовності елементів.

**Сортування:** SortedCollection зберігає елементи у відсортованому порядку.

**Індексування:** Більшість колекцій з визначеним порядком послідовності елементів допускають також індексування, тобто елемент може бути повернений за допомогоюat:. Масив є знайомою структурою даних з індексуванням та фіксованим розміром; anArray at: n повертає n-ий елемент з anArray, і anArray at: n put: v змінює n-ий елемент на v. Списки LinkedList та SkipList допускають визначення порядку послідовності, але не індексування, тому вони розуміють first і last, але не at:.

**Ключі:** До екземплярів класу Dictionary і похідних від нього можна доступатись за ключами замість індексів.

**Модифікація:** Більшість колекцій, окрім класів Interval і Symbol можна змінювати. Interval є незмінною колекцією, яка представляє ряд чисел Integer. Наприклад 5 to: 16 by: 2 це інтервал, який містить елементи 5, 7, 9, 11, 13 і 15. Він індексується за допомогою at:,але не можезмінюватись за допомогою at:put:.

**Зростання:** Екземпляри класівInterval і Array завжди є фіксованого розміру. Інші типи колекцій (відсортовані, впорядковані і зв’язні списки) можуть рости після створення. Клас OrderedCollection є більш загальним, ніж клас Array; Розмір OrderedCollection зростає при потребі і цей клас розуміє повідомлення addFirst: і addLast: так само як at: і at:put:.

**Допуск однакових елементів:** Клас Set відфільтрує копії, а Bag ні. Dictionary, Set і Bag використовують метод = визначений для елементів; для ідентичних елементів цих класів використовується метод ==, який перевіряє, чи аргументи це той самий об’єкт, вбудовані варіанти використовують відношення еквівалентності, яке обрав той, хто створив колекцію.

**Неоднорідність:** Більшість колекцій зберігають будь-який тип елементів. Прооте класиString, CharacterArray і Symbol, зберігають лише елементи класу Character. Клас Array зберігатиме будь-який склад об’єктів, але клас ByteArray зберігає лише елементи Byte, IntegerArray лише елементи Integer, аFloatArray лише елементи Float. Клас LinkedList побудований так, щоб зберігати елементи, які підтримують Link *>* протокол доступу.

**1.3 Реалізація колекцій**

Категоризація за функціональністю є не єдиною турботою, також потрібно брати до уваги як класи колекцій реалізовані. Як показано у *Таблиці 1*. використовується п’ять основних технік.



**Таблиця 1. Деякі класи колекцій можна розбити на категорії за технікою реалізації.**

* Масиви зберігають свої елементи в індексованих екземплярах змінних самої колекції; як висновок, масиви повинні бути фіксованого розміру, але можуть бути створені за допомогою одного виділення на це пам’яті.
* Екземпляри класів OrderedCollection і SortedCollection зберігають елементи у масиві, доступ до якого відбувається через один із екземплярів змінних колекції. Зрештою внутрішній масив може бути замінений більшим, якщо розмір колекції перевищує вмістимість внутрішнього об’єкту зберігання.
* Різні типи Set I Dictionary також звертаються до додаткового масиву для зберігання, але використовують його як хеш-таблицю. Клас Bag використовує допоміжний клас Dictionary, з елементами як ключами і кількістю появи як значеннями.
* Клас LinkedList використовує стандартне одно-зв’язне представлення.
* Клас Interval представляється трьома цілими числами, які зберігають межі і розмір кроку.

На додачу до цих класів є ще нестійкі варіанти класів Array, Set і Dictionary. Вони зберігають елементи слабко, тобто таким чином, що збірник сміття може вилучити їх. Віртуальна машина Pharo знає про ці класи і виконує операції над ними особливим чином.

Читачі, зацікавлені у вивченні додаткової інформації про колекції Smalltalk можуть звернутись до чудової книжки Ла Лонде та Пуга.

**1.4 Приклади ключових класів**

До зараз ми розглянули найбільш звичні та важливі класи колекцій використовуючи прості приклади. Головними операціями над колекціями є: at:, at:put: — для доступу до елемента, add:, remove: — для додавання та видалення, size, isEmpty, include: — для деякої інформації про колекцію, do:, collect:, select: — для ітерації над колекцією. Кожна колекція може реалізувати або ні ці протоколи, а якщо так, то вони представляють їх з потрібною семантикою. Ми пропонуємо дослідити кожен клас щоб виявити особливості та інші способи взаємодії.

Ми зосередимося на найбільш вживаних класах колекцій: OrderedCollection, Set, SortedCollection, Dictionary, Interval і Array.

**Загальне створення.** Існує кілька способів створення екземплярів колекцій. Найзагальніші це new: і with:. new:

anInteger створює колекцію розміру anInteger, елементами якої будуть nil. with: anObject створює колекцію і додає об’єкт anObject до неї. Різні колекції по-різному реалізують таку поведінку.

Array with: 1 -> #(1)  
Array with: 1 with: 2 -> #(1 2)  
Array with: 1 with: 2 with: 3 -> #(1 2 3)  
Array with: 1 with: 2 with: 3 with: 4 -> #(1 2 3 4)  
Array with: 1 with: 2 with: 3 with: 4 with: 5 -> #(1 2 3 4 5)  
Array with: 1 with: 2 with: 3 with: 4 with: 5 with: 6 -> #(1 2 3 4 5 6)

Також можна використати addAll: додати всі елементи одного типу колекції до іншого типу:

(1 to: 5) asOrderedCollection addAll: '678'; yourself -> an OrderedCollection (1 2 3 4 5 $6 $7 $8)

Також зверніть увагу, що addAll: повертає аргументи, а не отримувача повідомлення!  
Можна також створити багато колекцій за допомогою withAll: або newFrom:.

Array withAll: #(7 3 1 3) -> #(7 3 1 3)  
OrderedCollection withAll: #(7 3 1 3) -> an OrderedCollection(7 3 1 3)  
SortedCollection withAll: #(7 3 1 3) -> a SortedCollection(1 3 3 7)  
Set withAll: #(7 3 1 3) -> a Set(7 1 3)  
Bag withAll: #(7 3 1 3) -> a Bag(7 1 3 3)  
Dictionary withAll: #(7 3 1 3) -> a Dictionary(1->7 2->3 3->1 4->3 )

Array newFrom: #(7 3 1 3) -> #(7 3 1 3)   
OrderedCollection newFrom: #(7 3 1 3) -> an OrderedCollection End of statement list encountered ->(7 3 1 3)  
SortedCollection newFrom: #(7 3 1 3) -> a SortedCollection(1 3 3 7)  
Set newFrom: #(7 3 1 3) -> a Set(7 1 3)  
Bag newFrom: #(7 3 1 3) -> a Bag(7 1 3 3)  
Dictionary newFrom: {1 -> 7. 2 -> 3. 3 -> 1. 4-> 3} -> a Dictionary(1->7 2->3 3->1 4->3 )

Зауважте, що ці два методи не ідентичні. Зокрема, Dictionary *>>*withAll: інтерпретує аргументи як колекцію значень, тоді як Dictionary *>>*newFrom: очікує колекцію пар.

**Array**

Array є колекцією фіксованого розміру, доступ до елементів якої відбувається за допомогою цілочисельних індексів. На відміну від С, перший елемент в масивах Smalltalk має позицію 1, а не 0. Основним протоколом для доступу до елементів є метод at: і at:put:. at:anInteger повертає елемент на позиції anInteger. at:anIntegerput:anObject розміщує anObjeсt за індексом anInteger. Масиви мають фіксований розмір, тому не можна додати або видалити елементи з кінця масиву. Наступний код створює масив розміру 5, задає значення першим трьом позиціям і повертає перший елемент.

anArray := Array new: 5.  
anArray at: 1 put: 4.  
anArray at: 2 put: 3/2.  
anArray at: 3 put: 'ssss'.  
anArray at: 1 -> 4

Існує декілька способів створення екземплярів класу Array. Можна використовувати new:, with:, і конструкції #() та {}.

**Створення за допомогою new:** new: anInteger створює масив розміру anInteger. Arraynew: 5 створить масив розміру 5.

**Створення за допомогою with:** Метод with: дозволяє вказати значення елементів. Наступний код створить масив з трьох елементів, який містить число 4, дріб 3/2 і рядок ‘lulu’.

Array with: 4 with: 3/2 with: 'lulu' -> {4. (3/2). 'lulu'}

**Створення літералів за допомогою #():** #() створює масиви літералів зі статичними (або літеральними) елементами, які повинні бути відомі при компілюванні виразу, а не при його виконанні. Наступний код створить масив розміру 2, в якому перший елемент є (літеральним) числом 1 і другий (літеральним) рядком ‘here’.

#(1 'here') size -> 2

При виконанні #(1+2) ви не отримаєте масив з єдиним елементом 3, натомість буде масив #(1 #+ 2), тобто з трьома елементами: 1, символ #+ та число 2.

#(1+2) -> #(1 #+ 2)

Так відбувається тому, що конструкція #() змушує компілятор тлумачити вираз в масиві буквально. Вираз переглядається і елементи результату подаються в новий масив. Масиви літералів містять числа, nil, true, false, символи та стрічки.

**Динамічне створення за допомогою {}:** І нарешті, динамічний масив можна створити, використовуючи конструкцію {}. { a . b } еквівалентно до Arraywith: a with: b. Зокрема це означає,що виконається вираз, поміщений в { та }.

{1+2} -> #(3)  
{(1/2) asFloat} at: 1 -> 0.5  
{10 atRandom . 1/3} at: 2 -> (1/3)

**Доступ до елементів.** Доступитися до елементів можна за допомогою at: та at:put:.

anArray := #(1 2 3 4 5 6) copy.  
anArray at: 3 -> 3  
anArray at: 3 put: 33.  
anArray at: 3 -> 33

Будьте обережні з кодом, який модифікує масиви літералів! Компілятор намагається виділяти місце для масивів літералів лише раз. Якщо ви не копіюєте масив, при повторному оцінюванні коду ваш масив літералів може містити не ті значення, які ви очікуєте.

**OrderedCollection**

OrderedCollection є однією з колекцій, які можуть збільшуватись, і до яких елементи можна додавати послідовно. Вона пропонує багато методів, таких як add:, addFirst:, addLast:, іaddAll:.

ordCol := OrderedCollection new.  
ordCol add: 'Seaside'; add: 'SqueakSource'; addFirst: 'Monticello'.  
ordCol -> an OrderedCollection('Monticello' 'Seaside' 'SqueakSource')

**Видалення елементів.** Метод remove: anObject видаляє перше входження об’єкта в колекції. Якщо колекція не містить такого об’єкту, видається помилка.

ordCol add: 'Monticello'.  
ordCol -> an OrderedCollection('Monticello' 'Seaside' 'SqueakSource' 'Monticello')  
ordCol remove: 'Monticello'.  
ordCol -> an OrderedCollection('Seaside' 'SqueakSource' 'Monticello')

Існує варіант remove: з назвою remove:ifAbsent:, який дозволяє вказати другим аргументом блок, який виконуватиметься у випадку, коли елемент, який потрібно видалити, відсутній в колекції:

res := ordCol remove: 'zork' ifAbsent: ['element zork is not in the ordCol'].  
res -> 'element zork is not in the ordCol'

**Перетворення.** Можна отримати OrderedCollection з Array (чи будь-якої іншої колекції), надсилаючи повідомлення asOrderedCollection:

#(1 2 3) asOrderedCollection -> an OrderedCollection(1 2 3)  
'hello' asOrderedCollection -> an OrderedCollection($h $e $l $l $o)

**Interval**

Клас Interval представляє діапазон чисел. Наприклад інтервал чисел від 1 до 100 можна визначити наступним чином:

Interval from: 1 to: 100 -> (1 to: 100)

Повідомлення printString для цього інтервалу показує, що клас Number забезпечує нас відповідним методом to: щоб генерувати інтервали:

(Interval from: 1 to: 100) = (1 to: 100) -> true

Ми можемо використовувати Interval class>>from:to:by: для того щоб визначити крок між двома послідовними числами так як показано нижче:

(Interval from: 1 to: 100 by: 0.5) size -> 199   
(1 to: 100 by: 0.5) at: 198 -> 99.5  
(1/2 to: 54/7 by: 1/3) last -> (15/2)

**Dictionary**

Dictionary це така колекція в якій до елементів можна доступатися використовуючи ключі. Серед найбільш вживаних повідомлень словника можна знайти такі: at:, at:put:, at:ifAbsent:, keys і values.

colors := Dictionary new.  
colors at: #yellow put: Color yellow.  
colors at: #blue put: Color blue.  
colors at: #red put: Color red.  
colors at: #yellow -> Color yellow  
colors keys -> #(#yellow #red #blue)  
colors values -> {Color yellow. Color red. Color blue}

Dictionary порівнює ключі на рівність. Два ключі вважаються однаковими, якщо вони при порівнянні з допомогою = повертають true. Досить поширеною і непростою для вирішення помилкою є використання в ключі об’єкта чий метод = є перевизначений, але не перевизначений його метод hash. Ці обидва методи використовуються в реалізації словника і при порівнянні об’єктів.

Клас Dictionary чітко демонструє, що ієрархія колекцій базується на наслідуванні, а не на побудові підтипів. Не зважаючи на те, що Dictionary є підкласом Set, ми зазвичай не використовуємо Dictionary там де варто використати Set. Хоча, в своїй реалізації Dictionary складається з множини пар (ключ значення) , що створюються за допомогою повідомлення ->. Можна створити Dictionary з колекції пар, або перетворити словник в масив пар.

colors := Dictionary newFrom: { #blue->Color blue . #red->Color red . #yellow->Color yellow }.  
colors removeKey: #blue.  
colors associations -> {#yellow->Color yellow. #red->Color red}

**IdentityDictionary**. Тоді як словник використовує результат повідомлення = і hash для визначення чи два ключі є рівними, IdentityDictionary використовує перевірку на ідентичність ключів (повідомлення ==), а не значень, тобто, два ключі вважаються рівними якщо вони є тим самим об’єктом.

Часто символи (Symbol) використовують як ключі, і в цьому випадку природно використовувати IdentityDictionary, оскільки гарантується, що Symbol буде глобально унікальним. З іншої сторони, якщо ключем є String, краще використовувати звичайний Dictionary, бо інакше можуть виникнути проблеми:

a := 'foobar'.  
b := a copy.  
trouble := IdentityDictionary new.  
trouble at: a put: 'a'; at: b put: 'b'.  
trouble at: a -> 'a'  
trouble at: b -> 'b'   
trouble at: 'foobar' -> 'a'

Оскільки a і b є різними об’єктами, вони вважаються різними об’єктами. Цікаво те, що літерал ‘foobar’ локалізується лише раз, отже він справді є тим же ж об’єктом, що і a. Та не завжди є бажаним, щоб код саме таким чином залежав від поведінки! Звичайний Dictionary буде повертати те ж значення для будь-якого ключа, що рівний ‘foobar’.

Слід використовувати тільки глобально унікальні об’єкти (такі як Symbol чи SmallInteger), як ключі для IdentityDictionary, і String (чи інші об’єкти), як ключ для звичайного Dictionary.

Варто запам’ятати те, що глобальна змінна Smalltalk є елементом SystemDictionary, що є підкласом IdentityDictionary, оскільки всі його ключі є символами (насправді ByteSymbol, які містить тільки 8-ми бітові символи).

Set newFrom: (Smalltalk keys collect: [ :each | each class ]) -> a Set(ByteSymbol)

Результатом виклику повідомлення keys чи values до Dictionary є Set, цей клас ми розглянемо далі.

**Set**

Клас Set — це колекція, поведінка якої нагадує поведінку математичної множини, тобто це колекція, яка не містить повторюваних елементів, і ці елементи розміщені без жодного порядку. У Set елементи можна додавати за допомогою повідомлення add:. Доступитися до елементів, використовуючи повідомлення at:, неможливо. Об’єкти, додані у Set, повинні реалізовувати методи hash і =.

s := Set new.  
s add: 4/2; add: 4; add:2.  
s size -> 2

Створити множину можна ще за допомогою Set class>>newFrom: або повідомлення Collection>>asSet:

(Set newFrom: #( 1 2 3 1 4 )) = #(1 2 3 4 3 2 1) asSet -> true

asSet пропонує зручний спосіб вилучення дублікатів із колекції:

{Color black. Color white. Color white} asSet size -> 2

Варто зауважити, що red + blue + green = white.

Колекція Bag подібна до Set. Різниця полягає в тому, що у Bag елементи можуть повторюватися.

{Color black. Color white. Color white} asBag size -> 3

Такі операції на множинах, як об’єднання, перетин і належність множині, реалізовані у колекції за допомогою повідомлень union:, intersection: і includes:. Одержувач спершу перетворений до Set, так що ці операції працюють для всіх видів колекцій.

(1 to: 6) union: (4 to: 10) -> #(1 2 3 4 5 6 7 8 9 10)

'hello' intersection: 'there' -> 'eh'

#Smalltalk includes: $k -> true

Як зазначено нижче, доступ до елементів множини здійснюється за допомогою ітератора (див. Розділ 1.5).

**SortedCollection**

На відміну від OrderedCollection, SortedCollection зберігає елементи у відсортованому порядку. За замовчуванням, щоб встановити порядок сортування, така колекція використовує повідомлення <=, тому вона може відсортувати екземпляри підкласів абстрактного класу Magnitude, що визначає протокол порівнюваних об’єктів(>, <, =, >=, between:and:...)(див. Розділ 1: Basic Classes).

Можна створити SortedCollection шляхом створення нового екземпляру і додавання до нього елементів:

SortedCollection new add: 5; add: 2; add: 50; add: -10; yourself -> a SortedCollection(-10 2 5 50)

Частіше, однак, надсилають повідомлення перетворення asSortedCollection до вже існуючої колекції:

#(5 2 50 -10) asSortedCollection -> a SortedCollection(-10 2 5 50)

Цей приклад відповідає на наступне доволі часте запитання:

Питання: Як відсортувати колекцію?

Відповідь: надіслати їй повідомлення asSortedCollection.

'hello' asSortedCollection -> a SortedCollection($e $h $l $l $o)

Тоді виникає питання: як отримати назад із результату String? На жаль, asString повертає вигляд printString, а це не те, що нам потрібно:

'hello' asSortedCollection asString -> 'a SortedCollection($e $h $l $l $o)'

Правильно використовувати String class>>>newFrom:, String class>>>withAll: або Object>>>as::

'hello' asSortedCollection as: String -> 'ehllo'  
String newFrom: ('hello' asSortedCollection) -> 'ehllo'  
String withAll: ('hello' asSortedCollection) -> 'ehllo'

У SortedCollection можна зберігати різні типи елементів, якщо всі вони можуть бути порівнюваними. Наприклад, можна поєднати різні типи чисел, такі як integers, floats і fractions:

{ 5. 2/ -3. 5.21 } asSortedCollection -> a SortedCollection((-2/3) 5 5.21)

Тепер уявіть, що потрібно відсортувати об’єкти, які не визначають методу <=, або які потрібно відсортувати за різними критеріями. Це можна зробити за допомогою блоку, який приймає два аргументи(називається блоком сортування). Наприклад, клас Color не є Magnitude і не реалізовує методу <=, але ми можемо визначити блок, який зазначає, що кольори повинні бути відсортовані відповідно до міри їхньої яскравості:

col := SortedCollection sortBlock: [:c1 :c2 | c1 luminance <= c2 luminance].  
col addAll: { Color red. Color yellow. Color white. Color black }.  
col -> a SortedCollection(Color black Color red Color yellow Color white)

**String**

У Smalltalk String представляє колекцію з Characters. Вона забезпечує обхід елементів, доступ за індексом, можливість динамічно змінювати вміст, містить однотипні елементи, які є екземплярами Character. Подібно до Arrays, String має спеціальний синтаксис, і, як правило, створюється за допомогою одинарних лапок, всередині яких зазначено рядок символів. Проте спрацює і стандартний метод створення колекцій:

'Hello' -> 'Hello'  
String with: $A -> 'A'  
String with: $h with: $i with: $! -> 'hi!'  
String newFrom: #($h $e $l $l $o) -> 'hello'

Насправді, String є абстрактною. Під час ініціалізації String ми насправді отримуємо або 8-бітний ByteString, або 32-бітний WideString. Щоб не ускладнювати собі життя, ми зазвичай не звертаємо уваги на різницю, і говоримо лише про екземпляри String.

За допомогою коми можна конкатенувати два екземпляри String:

s := 'no', ' ', 'worries'.  
s -> 'no worries'

Оскільки String є динамічною колекцією, існує можливість змінювати її за допомогою методу at:put:.

s at: 4 put: $h; at: 5 put: $u.

s -> 'no hurries'

Слід зауважити, що попередній метод визначений у Collection, тому він працюватиме для будь-якої колекції.

(1 to: 3) , '45' -> #(1 2 3 $4 $5)

Можна також модифікувати існуючий рядок за допомогою replaceAll:with: або replaceFrom:to:with: (як показано нижче). Кількість символів у інтервалі повинна відповідати розміру.

s replaceAll: $n with: $N.  
s -> 'No hurries'  
s replaceFrom: 4 to: 5 with: 'wo'.  
s -> 'No worries'

На відміну від методів, описаних вище, метод copyReplaceAll: створює новий рядок(цікаво, що аргументами тут є скоріше підрядки, ніж окремі символи, і їхні розміри не повинні співпадати)

s copyReplaceAll: 'rries' with: 'mbats' -> 'No wombats'

Якщо глянути на реалізацію цих методів, можна зауважити, що вони визначені не лише для Strings, але й для будь-якого виду SequenceableCollection, тому наступний код також буде правильним:

(1 to: 6) copyReplaceAll: (3 to: 5) with: { 'three'. 'etc.' } -> #(1 2 'three' 'etc.' 6)

**String відповідність.** Існує можливість перевіряти, чи певний патерн відповідає рядку, за допомогою повідомлення match:. Патерн може містити \*, що відповідатиме довільній кількості певних символів, і #, що відповідатиме точно одному символу. Слід зауважити, що для перевірки відповідності match: надсилається патерну, а не рядку.

'Linux \*' match: 'Linux mag' -> true  
'GNU/Linux #ag' match: 'GNU/Linux tag' -> true

Ще одним корисним методом є findString:.

'GNU/Linux mag' findString: 'Linux' -> 5  
'GNU/Linux mag' findString: 'linux' startingAt: 1 caseSensitive: false -> 5

Більш передові патерни на відповідність, що пропонують можливості Perl, доступні в пакеті Regex.

**Деякі перевірки для String.** Наступні приклади показують, як використовувати повідомлення isEmpty, includes: і anySatisfy:, які визначені не тільки для Strings, але й для інших колекцій.

'Hello' isEmpty -> false  
'Hello' includes: $a -> false  
'JOE' anySatisfy: [:c | c isLowercase] -> false  
'Joe' anySatisfy: [:c | c isLowercase] -> true

**Шаблони String.** Існують три повідомлення, які допомагають керувати шаблонами String: format:, expandMacros і expandMacrosWith:.

'{1} is {2}' format: {'Pharo' . 'cool'} -> 'Pharo is cool'

Повідомлення типу expandMacros дозволяють робити підстановку змінних. Використовуємо <n> для переходу на наступний рядок, <t> - для табуляції, <t> , <1s>, <2s>, <3s> для аргументів, і <1?value1:value2> для умови.

'look-<t>-here' expandMacros -> 'look- -here'  
'<1s> is <2s>' expandMacrosWith: 'Pharo' with: 'cool' -> 'Pharo is cool'  
'<2s> is <1s>' expandMacrosWith: 'Pharo' with: 'cool' -> 'cool is Pharo'  
'<1p> or <1s>' expandMacrosWith: 'Pharo' with: 'cool' -> '''Pharo'' or Pharo'  
'<1?Quentin:Thibaut> plays' expandMacrosWith: true -> 'Quentin plays'  
'<1?Quentin:Thibaut> plays' expandMacrosWith: false -> 'Thibaut plays'

**Інші корисні методи.** Клас String пропонує багато інших корисних методів, зокрема повідомлення asLowercase, asUppercase і capitalized.

'XYZ' asLowercase -> 'xyz'  
'xyz' asUppercase -> 'XYZ'  
'hilaire' capitalized -> 'Hilaire'  
'1.54' asNumber -> 1.54  
'this sentence is without a doubt far too long' contractTo: 20 -> 'this sent...too long'

Варто звернути увагу: загалом існує різниця між надсиланням об’єкту повідомлення printString , щоб отримати String-вигляд об’єкта, і його перетворенням до String за допомогою повідомлення asString. Нижче наведено приклад, що демонструє цю різницю:

#ASymbol printString -> '#ASymbol'  
#ASymbol asString -> 'ASymbol'

Symbol подібний до String, але гарантовано, що він буде глобально єдиним. Саме тому надають перевагу Symbol, а не String, у якості ключа для Dictionary, зокрема для екземплярів IdentityDictionary. Для детальної інформації про String і Symbol див. Розділ 1 : Basic Classes.

**1.5 Ітератори колекцій.**

У SmallTalk цикли і оператори умови – це прості повідомлення, надіслані колекції чи іншому об’єкту, такі як цілі числа або блоки(див. Розділ Синтаксис Nutshell). Крім низькорівневого повідомлення to:do:, який визначає блок з аргументом, що змінюється від початкового до кінцевого значення, ієрархія колекцій SmallTalk пропонує багато інших літераторів вищого рівня. З використанням цих ітераторів користувач може зробити код більш надійним і компактним.

**Ітерування(do:)**

Метод (do:) є базовим ітератором для колекцій. Він “застосовує” свій аргумент(блок, що приймає один аргумент) до кожного елемента одержувача. У наступному прикладі виводяться усі String, що містяться в одержувачі, до транскрипту.

#('bob' 'joe' 'toto') do: [:each | Transcript show: each; cr].

**Модифікації.** Є декілька варіантів do:, наприклад do:without:, doWithIndex: і reverseDo:.

Для колекцій, доступ до елементів яких здійснюється через індекс (Array, OrderedCollection, SortedCollection), використовують метод doWithIndex: що дає доступ до поточного індексу. Цей метод нагадує метод to:do: , визначений у класі Number.

#('bob' 'joe' 'toto') doWithIndex: [:each :i | (each = 'joe') ifTrue: [ ^ i ] ] -> 2

Щоб розмістити елементи впорядкованої колекції у зворотному порядку, використовують метод reverseDo:.

Наступний код демонструє цікаве повідомлення do:separatedBy:, яке виконує блок тільки між двома елементами:

res := ''.  
#('bob' 'joe' 'toto') do: [:e | res := res, e ] separatedBy: [res := res, '.'].  
res -> 'bob.joe.toto'

Цей код не є дуже ефективним, оскільки він створює проміжні String, і краще було б написати потік, щоб записувати результати у буфер(див. Розділ Потоки).

String streamContents: [:stream | #('bob' 'joe' 'toto') asStringOn: stream delimiter: '.'] -> 'bob.joe.toto'

**Dictionaries**. Коли до Dictionary надіслано повідомлення do:, то елементами, які тоді залучаються, є значення, а не зв’язки. Слід використовувати методи keysDo:, valuesDo:, і associationsDo:, які ітерують за ключами, значеннями і зв’язками відповідно.

colors := Dictionary newFrom: { #yellow-> Color yellow. #blue-> Color blue. #red-> Color red }.  
colors keysDo: [:key | Transcript show: key; cr]. "показує ключі"  
colors valuesDo: [:value | Transcript show: value;cr]. "показує значення"  
colors associationsDo: [:value | Transcript show: value;cr]. "показує відношення"

**Результати колекцій(collect:)**

Якщо потрібно виконати певні дії з елементами колекції, і в результаті утвориться нова колекція, замість do: краще використовувати collect: або якийсь інший ітератор, які можна знайти у протоколі enumerating Collection або підкласів Collection.

Нехай нам потрібна колекція, що зберігає дублікати елементів в іншій колекції. Якщо використовувати метод do:, слід написати наступне:

double := OrderedCollection new.  
#(1 2 3 4 5 6) do: [:e | double add: 2\*e].  
double -> an OrderedCollection(2 4 6 8 10 12)

Метод collect: виконує свій блок аргументів для кожного елементу і повертає нову колекцію, що містить результати. Але, якщо використати collect:, код стає набагато простіший:

#(1 2 3 4 5 6) collect: [:e | 2\*e] -> #(2 4 6 8 10 12)

Переваги використання collect: over do: видно ще краще в наступному прикладі, де ми беремо колекцію цілих чисел, і в якості результату генеруємо колекцію значень за модулем цих чисел:

aCol := #( 2 -3 4 -35 4 -11).  
result := aCol species new: aCol size.  
1 to: aCol size do: [ :each | result at: each put: (aCol at: each) abs].  
result -> #(2 3 4 35 4 11)

А тепер порівняємо його з наступним кодом:

#( 2 -3 4 -35 4 -11) collect: [:each | each abs ] -> #(2 3 4 35 4 11)

Ще однією перевагою останнього прикладу є те, що він працюватиме і для Set і Bag.

Якщо не потрібно надсилати повідомлення кожному елементу колекції, то, загалом, варто уникати використання do:.

Надсилаючи повідомлення collect:, ми отримуємо той самий тип колекції, як і отримувач повідомлення. Тому наступний код не працюватиме(String не може зберігати цілих значень).

'abc' collect: [:ea | ea asciiValue ] "Error: Importer store into indexable object"

Потрібно спершу конвертувати String до Array або OrderedCollection:

'abc' asArray collect: [:ea | ea asciiValue ] -> #(97 98 99)

Не гарантовано, що collect: поверне той самий тип колекції, що й отримувач повідомлення, - лише того самого роду. Наприклад, у випадку Interval, родом є Array.

(1 to: 5) collect: [ :ea | ea\*2 ] -> #(2 4 6 8 10)

**Вибір і відкидання елементів**

select: повертає елементи одержувача, які задовольняють певну умову:

(2 to: 20) select: [:each | each isPrime] -> #(2 3 5 7 11 13 17 19)

reject: діє навпаки:

(2 to: 20) reject: [:each | each isPrime] -> #(4 6 8 9 10 12 14 15 16 18 20)

**Ідентифікація елементу за допомогою detect:**

Метод detect: повертає перший елемент одержувача, що співпадає з блоком аргументів:

'through' detect: [:each | each isVowel] -> $o

Метод detect:ifNone: є різновидом методу detect. Якщо жоден елемент не відповідає першому блоку, то видається значення другого:

Smalltalk allClasses detect: [:each | '\*cobol\*' match: each asString] ifNone: [ nil ] -> nil

**Накопичення результатів з inject:into:**

Мови функціонального програмування часто забезпечують функції вищих порядків, що називаються fold або reduce, для накопичення результату через якусь ітераційну бінарну операцію над всіма елементами колекції. У Pharo це реалізовано завдяки Collection>>inject:into:.

Перший аргумент – це початкове значення, другий – це блок, який приймає два аргументи і реалізовується для кожного результату, отриманого до цього часу, і кожного елемента.

Елементарне застосування inject:into: - це отримати суму елементів колекції чисел. Як і в Gauss, в Pharo ми можемо написати цей вираз для обчислення перших 100 цілих чисел:

(1 to: 100) inject: 0 into: [:sum :each | sum + each ] -> 5050

Ще одним прикладом є обчислення факторіалу за допомогою блоку, що приймає один аргумент:

factorial := [:n | (1 to: n) inject: 1 into: [:product :each | product\*each ] ].

**Інші повідомлення**

**count:** Повідомлення count: повертає кількість елементів, що задовольняють умову. Умова реалізована у формі булівського блоку.

Smalltalk allClasses count: [:each | 'Collection\*' match: each asString ] -> 6

**includes:** Повідомлення includes: перевіряє, чи аргумент є елементом колекції.

colors := {Color white . Color yellow. Color red . Color blue . Color orange}.  
colors includes: Color blue. -> true

**anySatisfy:** Повідомлення anySatisfy: повертає true, якщо хоча б один елемент колекції задовольняє умову, задану аргументом.

colors anySatisfy: [:c | c red > 0.5] -> true

**1.6 Деякі підказки у використанні колекцій**

**Помилка з add:** . Наступна помилка є, мабуть, найчастішою в Smalltalk.

collection := OrderedCollection new add: 1; add: 2.  
collection -> 2

Тут змінна collection містить не щойно створену колекцію, а лише останнє додане число. Це все тому, що метод add повертає не одержувача, а доданий елемент.

Наступний код видає бажаний результат:

collection := OrderedCollection new.  
collection add: 1; add: 2.  
collection -> an OrderedCollection(1 2)

Також можна використовувати повідомлення yourself, щоб повернути одержувача каскаду повідомлень:

collection := OrderedCollection new add: 1; add: 2; yourself -> an OrderedCollection(1 2)

**Видалення елемента колекції з використанням ітерації**. Іншою ймовірною помилкою є видалення елемента з колекції під час ітерації.

range := (2 to: 20) asOrderedCollection.  
range do: [:aNumber | aNumber isPrime ifFalse: [ range remove: aNumber ] ].  
range -> an OrderedCollection(2 3 5 7 9 11 13 15 17 19)

Очевидно, що результат є неправильним, оскільки 9 і 15 повинні були відфільтруватися.

Вирішенням цієї проблеми є створення копії колекції перед ітеруванням.

range := (2 to: 20) asOrderedCollection.  
 range copy do: [:aNumber | aNumber isPrime ifFalse: [ range remove: aNumber ] ].  
range -> an OrderedCollection(2 3 5 7 11 13 17 19)

Перевизначення обох = і hash. Важко виявити помилку, коли ви перевизначаєте = , а не hash. Наслідком є втрата елементів, доданих у множину, або ж інше дивне поводження колекції. Розв’язок проблеми запропонував Кент Бек: потрібно використовувати xor: для перевизначення hash. Нехай ми маємо дві книжки, які будуть однаковими, якщо однаковими є їхні автори і назви. Тоді ми перевизначимо не лише =, але й hash:

**Скрипт 1.1: Перевизначення = і hash.**

Book»= aBook  
self class = aBook class ifFalse: [^ false].  
^ title = aBook title and: [ authors = aBook authors]  
Book»hash  
^ title hash xor: authors hash

Виникає інша неприємна проблема, якщо використовувати об’єкт, який може змінюватися динамічно, тобто об’єкт, hash значення якого змінюється протягом часу, наприклад, елемент колекції Set або Dictionary. Не використовуйте цього, хіба що ви любите шукати помилки!

**1.7 Підсумки**

Ієрархія колекцій Smalltalk забезпечує загальний словник для маніпулювання різноманітних видів колекцій.

* Значна різниця полягає між SequenceableCollections, що зберігають їхні елементи у заданому порядку, Dictionary і підкласах Dictionary, які зберігають зв’язки між ключами і значеннями, та невпорядкованими Set і Bag.
* Більшість колекцій можна перетворити до іншого типу, надсилаючи їм повідомлення asArray, asOrderedCollection тощо.
* Щоб посортувати колекцію, слід використовувати повідомлення asSortedCollection.
* Маси літералів створюється за допомогою спеціального синтаксису #( ... ). Динамічний масив створюється за допомого синтаксису { ... }.
* Dictionary порівнює ключі на рівність, тому найкраще, коли ключі є екземплярами String. Натомість, IdentityDictionary використовує ідентичність об’єкта для порівняння ключів, тому краще використовувати Symbols у якості ключів, або ж звертатися до значень.
* String також відповідає на стандартні повідомлення колекцій. Крім того, String підтримує просту форму відповідності шаблонів. Для більш складного застосування слід використовувати пакет RegEx.
* Базовим повідомленням для здійснення ітерації є do:. Воно корисне для імперативного коду, наприклад, зміни кожного елементу колекції або надсилання повідомлення кожному елементу колекції.
* Замість do:, часто використовують collect:, select:, reject:, includes:, inject:into: та інші повідомлення вищого рівня для конкретних дій з колекціями.
* Не можна видаляти елементи колекції під час ітерації колекції. Якщо необхідно змінити колекцію, слід проводити ітерацію над копією колекції.
* Під час перевантаження = слід перевантажувати і hash.