Колекції

Щоб правильно використовувати класи колекцій, читачеві треба знати, принаймні поверхово, широке розмаїття колекцій, які існують, а також їхні спільні риси та відмін­ності. Ось про що цей розділ.

Класи колекцій, підкласи *Collection* і *Stream*, утворюють обширну групу класів загаль­ного призначення. Деякі з підкласів, як *Bitmap* чи *CompiledMethod*, мають спеціальне призначення, їх створено для використання в інших частинах системи або в застосун­ках, відтак в організації системи їх не зачислено до колекцій.

У цьому розділі використовуватимемо термін *ієрархія колекцій*, щоб позначити клас *Collection* і його підкласи, які також зачислено до пакетів, що називаються *Collections-\**. Термін *ієрархія потоків* використовуватимемо, щоб позначити клас *Stream* і його під­класи, які також є у пакеті *Collections-Streams*.

*Object*

*Collection*

*SequenceableCollection*

*ArrayedCollection*

*String*

Array

Text

Symbol

ByteString

OrderedCollection

SortedCollection

Interval

*HashedCollection*

Bag

Set

Dictionary

IdentitySet

PluggableSet

LinkedList

KeyedTree

IdentityDictionary

PluggableDictionary

Рис. 14.1. Ієрархія класів чисел

У цій главі зосередимось головно на підмножині класів колекцій, які зображені на рис. 14.1. Потоки розглянемо окремо у розділі 15 «Потоки».

Pharo за замовчуванням надає достатній набір колекцій. Крім того, проєкт «Containers», доступний на <http://www.github.com/Pharo-Containers/>, пропонує альтернативні реалізації або нові колекції та структури даних.

Почнемо з однієї важливої особливості колекцій у Pharo: їхні API значною мірою вико­ристовують функції вищого порядку. Хоча можна застосовувати цикли *for*, як у старій Java, розробники Pharo здебільшого використовують стиль ітератора, заснований на функціях вищого порядку.

## Функції вищого порядку

Застосування до колекцій функцій вищого порядку замість програмування дій з окремими елементами є важливим способом підвищення рівня абстракції програми. Функція *map* мови Lisp, яка застосовує функцію-аргумент до кожного елемента списку і повертає новий список, що містить результати, є раннім прикладом цього стилю. Наслідуючи свої витоки, мову програмування Smalltalk, Pharo затвердила основним принципом її підхід до програмування – використання колекцій і функцій вищого порядку. Сучасні функціональні мови програмування – ML і Haskell – наслідували приклад Smalltalk і Lisp.

Чому це хороша ідея? Припустимо, що є структура, яка містить колекцію записів про студентів, і потрібно виконати якусь певну дію з усіма студентами, які відповідають заданому критерію. Програмісти, які використовують імперативні мови, одразу ж використають цикл з перевіркою, а програміст на Pharo напише

students

select: [ :each | each gpa < threshold ]

Цей вираз повертає нову колекцію, який містить тільки ті елементи *students*, для яких блок (функція в квадратних дужках) повернув *true*. Блок можна трактувати як лямбда-вираз, що визначає анонімну функцію «*x . x gpa < threshold*». Такий код володіє просто­тою й елегантністю запиту предметно-орієнтованої мови.

Повідомлення *select:* розуміють усі колекції Pharo. Не потрібно визначати, чи структура даних про студентів є масивом чи зв’язним списком, повідомлення *select:* розуміють обидва. Зауважте, що такий код дуже відрізняється від використання циклу, де наперед потрібно знати з масивом, чи зі списком ми працюємо, щоб правильно налаштувати цикл.

Коли хтось говорить про колекцію у Pharo, не називаючи конкретно вид колекції, то він має на увазі об’єкт, який підтримує чітко визначені протоколи для перевірки належ­ності та перебирання елементів. *Усі* колекції розуміють повідомлення перевірки *includes:*, *isEmpty* і *occurrencesOf:*. *Усі* колекції розуміють повідомлення перебирання *do:*, *select:*, *reject:* (протилежне до *select:*), *collect:* (подібне на *map* у Lisp), *detect:ifNone:*, *inject:­into:* (виконує ліву згортку) та багато інших. Саме розповсюдженість цього протоколу, а також різноманітність роблять його таким потужним.

У таблиці систематизовано стандартні протоколи, які підтримуються більшістю класів у ієрархії колекцій. Ці методи визначені, перевизначені, оптимізовані чи іноді навіть заборонені похідними класами *Collection*.

|  |  |
| --- | --- |
| **Протокол** | **Методи** |
| accessing | size, capacity, at:, at:put: |
| testing | isEmpty, includes:, contains:, occurrencesOf: |
| adding | add:, addAll: |
| removing | remove:, remove:ifAbsent:, removeAll: |
| enumerating | do:, collect:, select:, reject: detect:, detect:ifNone:, inject:into: |
| converting | asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection: |
| creation | with:, with:with:, with:with:with:, with:with:with:with:, withAll: |

## Різноманіття колекцій

Попри таку базову уніфікованість, існує багато різних типів колекцій, які підтримують різні протоколи або забезпечують різну поведінку для однакових запитів. Давайте коротко розглянемо деякі ключові відмінності.

*Послідовні колекції*. Екземпляри всіх підкласів *SequenceableCollection* зберігають еле­менти від *першого* до *останнього* у строго визначеному порядку. На противагу їм *Set*, *Bag* і *Dictionary* не є послідовними.

*Впорядковані колекції*. Екземпляр *SortedCollection* зберігає елементи, впорядковані за зростанням або спаданням.

*Індексовані колекції*. Більшість послідовних колекцій індексовані, тобто елемент можна отримати повідомленням «*at: index*». *Array* – звичайна індексована структура да­них фіксованого розміру, масив. Вираз *anArray at: n* повертає *n*-й елемент масиву *anArray*, а вираз *anArray at: n put: v* заміняє його на *v*. Список *LinkedList* – послідов­ний, але не індексований, тому він розуміє повідомлення *first* і *last*, але не *at:*.

*Підтримка ключів*. Екземпляри *Dictionary* та його підкласів підтримують доступ за ключем замість індексу.

*Змінність*. Більшість колекцій змінні за винятком *Interval* і *Symbol*. *Interval* – незмінна колекція, що представляє послідовність цілих, арифметичну прогресію. Наприклад, інтервал *5 to: 16 by: 2* містить елементи *5*, *7*, *9*, *11*, *13*, *15*. Його можна індексувати повідомленням *at: anIndex*, але не можна змінити за допомогою *at: anIndex put: aValue*.

*Зростання розміру*. Екземпляри класів *Array*, *Interval*, *Symbol* завжди мають фіксовану кількість елементів. Колекції інших видів (порядкові, впорядковані та зв’язні) можуть збільшуватися після створення. Клас *OrderedCollection* загальніший ніж *Array*, розмір *OrderedCollection* збільшується на вимогу, він визначає повідомлення *addFirst: anElement* і *addLast: anElement* так само, як *at: anIndex* і *at: anIndex put: aValue*.

*Повторюваність значень*. *Set* відфільтровує дублікати, а *Bag* – ні. Класи *Dictionary*, *Set* і *Bag* використовують метод =, наданий їхніми елементами; Варіанти *Identity* цих класів використовують метод ==, який перевіряє, чи є аргументи тим самим об’єк­том, а варіанти *Pluggable* використовують довільне відношення еквівалентності, надане під час створення колекції.

*Однорідність*. Більшість колекцій можуть містити елементи довільного типу. Проте *String*, *CharacterArray* або *Symbol* містять тільки *Character*. Екземпляр *Array* міститиме суміш довільних об’єктів, а *ByteArray* – тільки байти. Клас *LinkedList* влаштовано так, що його екземпляри містять елементи, які підтримують протокол «*Link >> accessing*».

## Реалізація колекцій

Поділ колекцій на категорії за функціональністю не єдиний наш клопіт. Ми мусимо також розповісти, як реалізовано класи колекцій. Використано п’ять основних способів реалізації.

* *Векторна пам’ять*. Масиви *Array* зберігають свої елементи в індексованій змінній екземпляра. Як наслідок, у них фіксований розмір, а пам’ять для об’єкта виділяється за раз неперервною ділянкою. Векторну пам’ять використовують також *String* і *Symbol*.
* *Векторна пам’ять змінного розміру*. Екземпляри *OrderedCollections* і *SortedCollec­tions* зберігають елементи в масиві, посилання на який містить одна зі змінних екземпляра колекції. Коли збільшення колекції призводить до вичерпання виділеної пам’яті, вкладений масив заміняється більшим. Схожа реалізація у *Text* і *Heap*.
* *Хешована пам’ять*. Різноманітні види множин і словників (*Set*, *IdentitySet*, *Plug­gableSet*, *Dictionary*, *IdentityDictionary*, *PluggableDictionary*) також посилаються на вкладений масив, але використовують його як хеш-таблицю. Контейнери *Bag* і *IdentityBag* використовують вкладені словники, ключами яких є елементи контейнерів, а значеннями – кількості входжень.
* *Зв’язна пам'ять*. Єдиний представник цієї категорії використовує традиційну однозв’язну пам’ять: список складається з ланок, кожна з яких має посилання на наступну, контейнер зберігає посилання на першу й останню ланки.
* *Інтервал* зберігає три числа: кінці інтервалу та крок.

На додаток до цих класів є також слабкі варіанти *Array*, *Set* і різних типів словників. Ці колекції слабко тримають свої елементи, тобто так, що це не перешкоджає збиранню сміття. Віртуальна машина Pharo знає про ці класи й опрацьовує їх спеціально.

## Приклади головних класів

Продемонструємо на простих прикладах використання найбільш звичних і важливих класів колекцій. Головні протоколи колекцій:

* повідомлення *at:*, *at:put:* – для доступу до елемента;
* повідомлення *add:*, *remove:* – для додавання чи вилучення елемента;
* повідомлення *size*, *isEmpty*, *include:* – для отримання деякої інформації про колекцію;
* повідомлення *do:*, *collect:*, *select:* — для перебирання елементів колекції.

Кожна колекція може реалізувати або ні такі протоколи, а якщо так, то наділити їх відповідною семантикою. Пропонуємо дослідити кожен клас, щоб виявити його особливості та інші протоколи взаємодії.

Зосередимося на найбільш вживаних класах колекцій: *OrderedCollection*, *Set*, *Sorted­Collection*, *Dictionary*, *Interval* і *Array*.

## Загальний протокол створення

Існує кілька способів створення екземплярів колекцій. Найзагальніші це *new*, *new: aSize* і *with: anElement*, *with: anElement1…with: anElement6*, *withAll: aCollection*.

* *new* створює порожню колекцію, підходить для колекцій змінного розміру.
* *new: anInteger* застосовують для створення колекції фіксованого розміру:
  + *Array new: anInteger* поверне масив розміру *anInteger*, елементами якого будуть *nil*, згодом їх можна замінити потрібними значеннями;
  + *String new: anInteger* поверне рядок з *anInteger* пропусків;
  + у класах колекцій змінного розміру *new:* діє так само, як унарне *new* – повертає порожню колекцію.
* *with: anObject* створює колекцію і додає до неї об’єкт *anObject*, тобто, створює колекцію з одним елементом; повідомлення *with:with:* з двома селекторами приймає два об’єкти і створює колекцію з двома елементами; можна викорис­товувати такі повідомлення з трьома, чотирма, п’ятьма або шістьма селектора­ми, якщо ж до колекції потрібно додати більше об’єктів, використовують *withAll: aCollection*. Різні колекції по-різному реалізують таку поведінку.

Розглянемо приклади.

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)

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)

До створеної порожньої колекції зручно додавати елементи повідомленням *addAll:*, тільки потрібно пам’ятати, що воно повертає свій аргумент, а не отримувача.

Set new addAll: #(7 3 1 3); yourself

>>> a Set(7 3 1)

(1 to: 5) asOrderedCollection addAll: #(6 7 8); yourself

>>> an OrderedCollection(1 2 3 4 5 6 7 8)

## Array

*Array* – колекція фіксованого розміру, доступ до елементів якої відбувається за ціло­числовими індексами. На відміну від *С*, перший елемент у масивах Pharo має індекс 1, а не 0. Основним протоколом для доступу до елементів є методи *at*: і *at:put:*.

* *at: anInteger* повертає елемент з індексом *anInteger*.
* *at: anInteger put: anObject* розміщує *anObject* в масиві 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 елементам і повертає перший елемент.

| anArray |

anArray := Array new: 5.

anArray at: 1 put: 4.

anArray at: 2 put: 3/2.

anArray at: 3 put: 'Hello'.

anArray at: 1

>>> 4

Є кілька способів створення екземплярів класу *Array*. Можна використовувати пові­домлення *new:*, *with:*, літерал #( ) для створення статичного масиву та запис { . } для конструювання динамічного.

### Створення за допомогою *new:*

Повідомлення *new: anInteger* створює масив розміру *anInteger*. *Array new: 5* створить масив з п’яти елементів.

**Важливо**  Значенням кожного елемента буде *nil*.

### Створення за допомогою *with:*

Повідомлення *with:\** дає змогу записати значення елементів. Код нижче створить масив з трьох елементів: цілого числа *4*, раціонального *3/2* і рядка *'lulu'*.

Array with: 4 with: 3/2 with: 'lulu'

>>> {4. (3/2). 'lulu'}

### Створення літерала #( )

Вираз #( ) задає літерал масиву зі статичними (або *літеральними*) елементами, які мають бути відомі на етапі компіляції ще перед виконанням. Код нижче створить масив розміру 2, в якому перший елемент задано літералом цілого числа *1*, а другий – літералом рядка *'here'*.

#(1 'here') size

>>> 2

Якщо виконати вираз *#(1+2)*, то не отримаємо масив з єдиним елементом *3*, натомість буде масив *#(1 #+ 2)*, тобто з трьома елементами: числом *1*, символом *#+* та числом *2*.

#(1+2)

>>> #(1 #+ 2)

Так відбувається тому, що конструкція #( ) не виконує записаний у ній вираз. Елемен­тами є створені під час розпізнавання виразу об’єкти, тобто літеральні об’єкти. Вираз переглядається і результуючі елементи подаються в новий масив. Літерал масиву може містити числа, *nil*, *true*, *false*, символи, рядки та інші літерали масивів. Під час виконання виразу *#( )* ніякі повідомлення не надсилаються.

### Динамічне створення { . }

І нарешті, динамічний масив можна створити за допомогою конструкції *{ . }*. Вираз *{ a . b }* еквівалентний до *Array with: a with: b*. Зокрема, це означає, що виконається ви­раз, записаний між фігурними дужками, на противагу круглим дужкам літерального масиву.

{ 1 + 2 }

>>> #(3)

{(1/2) asFloat} at: 1

>>> 0.5

{10 atRandom . 1/3} at: 2

>>> (1/3)

### Доступ до елементів

Доступитися до елементів будь-якої послідовної колекції можна за допомогою повідомлень *at: anIndex* і *at: anIndex* *put: anObject*.

| anArray |

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! У попередніх версіях Pharo це могло призвести до тонких помилок. Компілятор зберігає літерали у спеціальному фреймі компільованого методу. Зміна літерального масиву могла змінити вміст цього фрейма. У Pharo немає такої небезпеки, бо літеральні масиви стали незмінними. Наве­дений приклад без «*copy*» не працюватиме – він згенерує виняток.

## OrderedCollection

*OrderedCollection* є однією з колекцій, які можуть збільшуватись, і до яких елементи можна додавати послідовно. Вона підтримує багато методів додавання – *add:*, *addFirst:*, *addLast:* і *addAll:*.

| ordCol |

ordCol := OrderedCollection new.

ordCol add: 'Seaside'; add: 'SmalltalkHub'; addFirst: 'GitHub'.

ordCol

>>> an OrderedCollection('GitHub' 'Seaside' 'SmalltalkHub')

### Вилучення елементів

Метод *remove: anObject* вилучає перше входження *anObject* з колекції. Якщо колекція не містить такого об’єкта, то трапиться виняток.

ordCol add: 'GitHub'.

ordCol remove: 'GitHub'.

ordCol

>>> an OrderedCollection('Seaside' 'SmalltalkHub' 'GitHub')

Існує варіант методу вилучення, який називається *remove:ifAbsent:*. Він дає змогу запмсати другим аргументом блок, який виконуватиметься у тому випадку, коли елемента, якого потрібно видалити, немає в колекції.

result := ordCol

remove: 'zork'

ifAbsent: ['element zork is not in the ordCol'].

result

>>> 'element zork is not in the ordCol'

### Перетворення

За допомогою повідомлення *asOrderedCollection:* можна отримати *OrderedCollection* з *Array* чи будь-якої іншої колекції.

#(1 2 3) asOrderedCollection

>>> an OrderedCollection(1 2 3)

'hello' asOrderedCollection

>>> an OrderedCollection($h $e $l $l $o)

(6 to: 18 by: 3) asOrderedCollection

>>> an OrderedCollection(6 9 12 15 18)

## Interval

Клас *Interval* представляє послідовність цілих чисел. Наприклад, усі цілі від 1 до 100 можна визначити так.

Interval from: 1 to: 100

>>> (1 to: 100)

Результат обчислення *printString* для цього інтервалу засвідчує, що клас *Number* надає зручний метод «*to:*», щоб генерувати інтервали.

(Interval from: 1 to: 100) = (1 to: 100)

>>> true

Можна використовувати *Interval class >> from:to:by:*, або *Number >> to:by:* для того, щоб задати крок між двома послідовними числами в послідовності.

(Interval from: 1 to: 100 by: 0.5) size

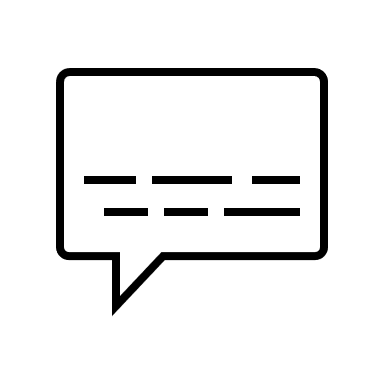
>>> 199

(1 to: 100 by: 0.5) at: 198

>>> 99.5

(1/2 to: 54/7 by: 1/3) last

>>> (15/2)

*Від перекладача.* Варто зауважити, що екземпляр *Interval* представляє не просто числову послі­довність, а *арифметичну прогресію*. У повідомленні *from: start to: endValue by: step* аргумент *start* задає перший член прогресії, аргумент *step* – її різницю, а *endValue* – значення, яке не повинні перевищувати члени прогресії. Якщо існує таке натуральне число *п*, що *endValue* = *start* + *step* × *n*, то *endValue* – останній член прогресії. Якщо для створення колекції використано повідомлення *from:to:*, то різниця прогресії дорівнює 1.

Певним аналогом *Interval* у мові Python є функція *range*(*start*:*excludeEnd*:*step*).

## Dictionary

Словники – це важливі колекції, доступ до елементів яких отримують за ключем. Серед най­більш вживаних повідомлень словника варто зазначити: *at: aKey*, *at: aKey put: aValue*, *at: aKey ifAbsent: aBlock*, *keys* і *values*.

| colors |

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

>>> #(#red #blue #yellow)

colors values

>>> {Color red. Color blue. Color yellow}

Словники порівнюють ключі на рівність. Два ключі вважаються однаковими, якщо їхнє порівняння за допомогою = повертає *true*. Досить поширеною і непростою для виправ­лення помилкою є використання як ключа об’єкта, чий метод = перевизначений, але не перевизначений його метод *hash*. Обидва ці методи використовують в реалізації слов­ника і для порівняння об’єктів.

Спробуйте проінспектувати створений в попередньому прикладі словник. Легко бачити, що екземпляр *Dictionary* реалізовано як колекцію пар (*ключ*; *значення*) – екземплярів класу *Association*, cтворених за допомогою повідомлення «–>»: *key–>value*. Тому можна легко створити словник з колекції асоціацій, або перетворити словник на 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 *Dictionary* використовує результат повідомлень = і *hash* для визначення того, чи два ключі рівні, *IdentityDictionary* використовує перевірку на ідентичність ключів – по­відомлення ==. Це означає що, два ключі вважаються рівними *тільки* тоді, коли вони є тим самим об’єктом.

Часто як ключі використовують символи (екземпляри класу *Symbol*). У цьому випадку природно використовувати *IdentityDictionary*, бо гарантується, що *Symbol* буде глобально унікальним. З іншого боку, якщо ключем є *String*, то краще використовувати звичайний *Dictionary*, бо інакше можуть виникнути проблеми.

| a b trouble|

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, то ключі вважаються різними. Цікаво те, що пам’ять для літерала *'foobar'* виділяється тільки один раз, тому він справді є тим самим об’єк­том, що й *a*. Але ніхто б не хотів, щоб поведінка його коду залежала від таких особли­востей! Звичайний *Dictionary* повертатиме те саме значення для будь-якого ключа зі значенням *'foobar'*.

Ключами *IdentityDictionary* можна робити тільки глобально унікальні об’єкти – *Symbol* чи *SmallInteger*. Ключами звичайного *Dictionary* можуть бути *String*, чи інші об’єкти, для яких визначено = і *hash*.

### Приклад *IdentityDictionary*

Вираз *Smalltalk globals* повертає екземпляр *SystemDictionary*, що є підкласом *IdentityDic­tionary*. Усі його ключі є символами (насправді, екземплярами *ByteSymbol*, підкласу *Symbol*, бо містять тільки 8-ми бітові літери).

Smalltalk globals keys collect: [ :each | each class ] as: Set

>>> a Set(ByteSymbol)

Тут використано повідомлення *collect:as:*, щоб задати тип колекції-результату – *Set*. Множина гарантує, що кожен клас ключа трапиться у підсумку тільки один раз.

## Set

Клас *Set* – це колекція, яка нагадує математичну множину, 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 них за допомогою повідомлення *at:* неможливо. Об’єкти, додані у *Set*, мають реалізувати методи *hash* і =.

| s |

s := Set new.

s add: 10/2; add: 4; add: 5.

s size

>>> 2

Створити множину можна також за допомогою *Set class >> newFrom:* або методом перетворення *Collection >> asSet:*.

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

>>> true

*asSet* пропонує зручний спосіб вилучення дублікатів з колекції:

{Color black. Color white. (Color red + Color blue + Color green) }

asSet size

>>> 2

**Важливо**  *red* + *blue* + *green* = *white*.

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

{Color black. Color white. (Color red + Color blue + Color green) }

asBag size

>>> 3

Операції з множинами *об’єднання*, *перетин*, *перевірка належності* реалізовані у класі *Collection* методами *union:*, *intersection:* і *includes:*, відповідно. Отримувач спершу перетворюється на *Set*, тому ці операції працюють з усіма видами колекцій.

(1 to: 6) union: (4 to: 10)

>>> #(8 5 2 10 7 4 1 9 6 3)

'hello' intersection: 'there'

>>> 'eh'

#Pharo includes: $a

>>> true

Як зазначено нижче, доступ до елементів множини виконують за допомогою ітераторів (див. підрозділ 14.14).

## SortedCollection

На відміну від *OrderedCollection*, *SortedCollection* зберігає елементи впорядкованими. За замовчуванням для впорядкування така колекція використовує повідомлення <=, тому вона може відсортувати екземпляри підкласів абстрактного класу *Magnitude*, що визна­чає протокол порівнюваних об’єктів (методи <, =, >, >=, *between:and:* тощо)(див. розділ 13 «Базові класи»).

Щоб збудувати впорядковану колекцію, можна створити екземпляр *SortedCollection* і додати до нього потрібні елементи.

SortedCollection new add: 5; add: 2; add: 50; add: -10; yourself

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

Однак частіше надсилають повідомлення перетворення *asSortedCollection* до вже існуючої колекції:

#(5 2 50 -10) asSortedCollection

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

'hello' asSortedCollection

>>> a SortedCollection($e $h $l $l $o)

Цей приклад відповідає на таке доволі часте запитання: «Як відсортувати колекцію?» – надіслати їй повідомлення *asSortedCollection*.

Тоді виникає інше питання: як отриманий результат перетворити назад у колекцію початкового типу? Наприклад, у *String*? На жаль, повідомлення *asString* повертає зображення *printString*, а це не те, що нам потрібно.

'hello' asSortedCollection asString

>>> 'a SortedCollection($e $h $l $l $o)'

Правильна відповідь – використати один з методів *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 елемент різних типів, якщо тільки їх можна порівнювати між собою. Наприклад, можна поєднати числа різних типів: цілі, дійсні, раціональні.

{ 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 треба застосувати інший критерій впорядкування. Бажаного можна досягти, якщо надати екземплярові *SortedCollection* бінарний блок – блок, який приймає два аргумен­ти і повертає булеву величину (називається блоком сортування). Наприклад, клас *Color* не наслідує *Magnitude* і не визначає метод <=, але можна визначити блок, який зазначає, що кольори потрібно сортувати відповідно до міри їхньої яскравості.

| col |

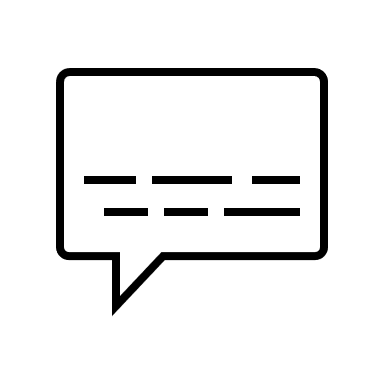
col := SortedCollection

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)

*Від перекладача.* Блоки у Pharo – це *щось*! Уявіть, блок сортування деякої колекції може питати думки користувача щодо того, як впорядковувати елементи. Якщо порівнювати рядки потрібно не в лексикографічному порядку, а за значенням, то такий блок стане в пригоді. Спробуйте виконати фрагмент, наведений нижче.

| assets source |source := #('work' 'money' 'friendship' 'love' 'family' 'Motherland'   
 'honor' 'education').assets:= SortedCollection sortBlock: [ :x :y | UIManager default  
 confirm: 'Is ', x printString, ' more important than ', y printString, '?' ].assets addAll: source.assets.

## Рядки

У Pharo рядок *String* є колекцією літер. Ця колекція послідовна, індексована, змінна й однорідна, бо містить *тільки* екземпляри *Character*. Подібно до масивів рядки мають спеціальний синтаксис. Зазвичай їх створюють як літерали за допомогою одинарних лапок, всередині яких зазначено рядок літер. Але можна використати і стандартні методи створення колекцій.

'Hello'

>>> 'Hello'

String with: $A

>>> 'A'

String with: $h with: $i with: $!

>>> 'hi!'

String newFrom: #($h $e $l $l $o)

>>> 'hello'

Насправді, клас *String* абстрактний. Під час створення екземпляра *String* отримують або *ByteString* з 8-бітними літерами, або *WideString* з 32-бітними. Щоб не ускладнювати міркування без потреби, зазвичай не звертають уваги на різницю, і говорять лише про екземпляри *String*.

Зображення рядка обрамляють апострофами, але рядок може містити апостроф як зви­чайну літеру. Щоб записати апостроф у рядку, його подвоюють, але рядок міститиме тільки один, як у прикладі нижче.

'об''єкт' at: 3

>>> $'

'об''єкт' at: 4

>>> $є

Повідомлення «,» (кома) конкатенує два екземпляри *String*. У одному виразі можна послідовно надіслати кілька таких повідомлень.

| s |

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

s

>>> 'no worries'

Зауважимо, що отримувач і аргумент конкатенації залишаються незмінними, а метод повертає новий екземпляр *String*.

Оскільки *String* змінна колекція, то можна змінювати окремі її літери за допомогою методу *at:put:*. Але з погляду надійного влаштування програм, варто уникати таких змін, бо один рядок може брати участь у виконанні кількох методів.

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

s

>>> 'no hurries'

Варто зауважити, що метод «кома» визначений у *Collection*, тому він працюватиме для будь-якої колекції.

(1 to: 3) , '45'

>>> #(1 2 3 $4 $5)

Існуючий рядок можна модифікувати також за допомогою *replaceAll: oldObject with: newObject* або *replaceFrom: start to: stop with: collection*, як показано нижче. Кількість символів у інтервалі [*start*; *stop*] має бути такою самою, як розмір *collection*.

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: oldSubCollection with: newCollection* створює новий рядок. Цікаво, що аргументами тут є підрядки, а не окремі символи, і їхні розміри можуть не збігатися.

s copyReplaceAll: 'rries' with: 'mbats'

>>> 'No wombats'

Реалізацію цих методів можна знайти в класі *SequenceableCollection*, тому не тільки *String*, а й будь-яка колекція, що його наслідує, розумітиме такі повідомлення. Випробуйте наведений нижче приклад.

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

>>> #(1 2 'three' 'etc.' 6)

### Зіставлення рядків

За допомогою повідомлення *match:* можна легко перевірити, чи певний рядок відпові­дає шаблонові. Шаблон може містити спеціальні літери: \* позначає довільну кількість довільних літер; літера # позначає одну довільну літеру. Треба зауважити, що для перевірки відповідності повідомлення *match:* надсилають шаблону, а не рядку.

'Linux \*' match: 'Linux mag'

>>> true

'GNU/Linux #ag' match: 'GNU/Linux tag'

>>> true

Ширші засоби зіставлення шаблонів доступні у пакеті Regex.

### Підрядки

У класі *SequenceableCollection* визначено низку методів, які можна використовувати для отримання підрядків: *first:*, *allButFirst:*, *copyFrom:to:* тощо.

'alphabet' first

>>> $a

'alphabet' first: 5

>>> 'alpha'

'alphabet' allButFirst: 4

>>> 'abet'

'alphabet' copyFrom: 5 to: 7

>>> 'abe'

'alphabet' copyFrom: 3 to: 3

>>> 'p' "не $p"

Майте на увазі, що різні методи можуть повертати результати різних типів. Більшість методів, пов’язаних із підрядками, повертають екземпляри *String*. Але повідомлення, які завжди повертають один елемент колекції *String*, повертають екземпляр *Character* (наприклад, *'alphabet' at: 6* повертає літеру *$b*). Щоб побачити повний перелік повідом­лень, пов’язаних із підрядками, перегляньте клас *SequenceableCollection* (особливо протокол *accessing*).

Ще один корисний метод – *findString:* та його варіанти.

'GNU/Linux mag' findString: 'Linux'

>>> 5

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

>>> 5

### Предикати

Наведені приклади демонструють, як використовувати повідомлення *isEmpty*, *includes:* і *anySatisfy:*, які визначені не тільки для *String*, а й для інших колекцій.

'Hello' isEmpty

>>> false

'Hello' includes: $l

>>> true

'JOE' anySatisfy: [:c | c isLowercase]

>>> false

'Joe' anySatisfy: [:c | c isLowercase]

>>> true

### Форматування рядків

Для форматування рядків можна використовувати такі повідомлення: *format:*, *expandMacros* і *expandMacrosWith:*.

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

>>> 'Pharo is cool'

'{1} is equal to {2}' format: #( 10 'ten')

>>> '10 is equal to ten'

Повідомлення типу *expandMacros* дають змогу підставляти певні значення замість спе­ціальних позначень у рядку: *<n>* означає переведення каретки; *<t>* – знак табуляції; *<1s>*, *<2s>*, *<3s>* – аргументи повідомлення; *<1p>*, *<2p>* обрамляє рядок апострофами; *<1?value1:value2>* – умовний вибір значення.

'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'

### Деякі допоміжні 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'

### *asString* проти *printString*

Будь-який об’єкт можна перетворити на рядок повідомленням *printString* або *asString*. У загальному випадку вони повертають різні результати: *printString* – рядкове зображен­ня об’єкта, а *asString* – результат перетворення отримувача на рядок. Наводимо прикла­ди, що демонструють цю різницю.

$A printString

>>> '$A'

$A asString

>>> 'A'

#ASymbol printString

>>> '#ASymbol'

#ASymbol asString

>>> 'ASymbol'

Символ подібний на рядок, але гарантовано, що він існує в єдиному примірнику. Саме тому надають перевагу символам, а не рядкам як ключам для словників, зокрема для екземплярів *IdentityDictionary*. Дізнатися більше про *String* і *Symbol* можна в розділі 13 «Базові класи».

## Ітератори колекцій

У Pharo інструкції повторення та галуження – це прості повідомлення до колекції чи іншого об’єкта, як ціле число або блок (див. розділ 9 «Розуміння синтаксису повідом­лень»). Додатково до низькорівневого повідомлення *to:do:*, яке виконує блок з аргумен­том, для кожного числа інтервалу від початкового до кінцевого значення, ієрархія ко­лекцій пропонує багато ітераторів високого рівня. З їх допомогою код можна зробити надійнішим і компактнішим.

### Перебір (*do:*)

Метод *do:* – це базовий ітератор колекцій. Він застосовує свій аргумент, блок з одним параметром, до кожного елемента отримувача. У прикладі усі рядки, що містяться в отримувачі, виводять по одному в консоль.

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

### Варіанти перебору

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

Для індексованих колекцій (*Array*, *OrderedCollection*, *SortedCollection*) використовують метод *doWithIndex:*, що приймає блок з двома параметрами та надає доступ і до поточного елемента, і до його індексу.

#('bob' 'joe' 'toto')

doWithIndex: [:each :i | (each = 'joe') ifTrue: [ ^ i ] ]

>>> 2

Щоб перебрати елементи послідовної колекції у зворотному порядку, використовують метод *reverseDo:*.

Приклад демонструє цікаве повідомлення *do:separatedBy:*, яке приймає два блоки та виконує другий з них тільки між двома елементами.

| res |

res := ''.

#('bob' 'joe' 'toto')

do: [:e | res := res, e ]

separatedBy: [res := res, '.'].

res

>>> 'bob.joe.toto'

Зауважимо, що цей код не дуже ефективний, бо створює проміжні рядки. Було б краще використати потік виведення, щоб записувати результат у буфер (див. розділ 15 «Потоки»).

String streamContents: [:stream |

#('bob' 'joe' 'toto') asStringOn: stream delimiter: '.']

>>> 'bob.joe.toto'

### Перебір словників

Метод *Dictionary >> do:* має особливість: він перебирає не пари (ключ –> значення), а тільки значення, збережені у словнику. Для ітерування словника варто використовува­ти методи *keysDo:*, *valuesDo:* і *associationsDo:*, які перебирають ключі, значення і пари, відповідно.

| colors |

colors := Dictionary newFrom: { #yellow-> Color yellow.

#blue-> Color blue. #red-> Color red }.

colors keysDo: [:key | Transcript show: key; cr]. "друкує ключі"

colors valuesDo: [:value | Transcript show: value; cr]. "друкує значення"

colors associationsDo: [:pair |

Transcript show: pair; cr]. "виводить у консоль асоціації"

" Текст у Transcript: "

red

blue

yellow

Color red

Color blue

Color yellow

#red->Color red

#blue->Color blue

#yellow->Color yellow

## Збір результатів (*collect:*)

Якщо потрібно застосувати певну функцію до кожного елемента деякої колекції і отри­мати нову колекцію, то замість *do:* краще використовувати *collect:* або якийсь інший ітератор. Більшість з них можна знайти у протоколі *enumerating* класу *Collection*, або його підкласів.

Припустимо, потрібно отримати колекцію, що зберігає подвоєні елементи з іншої колекції. Якщо використовувати метод *do:*, то треба написати таке.

| double |

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:* приймає блок, виконує його для кожного елемента отримувача і повертає нову колекцію, що містить результати виконання. Якщо замість *do:* в попередньому прикладі використати *collect:*, то код суттєво спроститься.

#(1 2 3 4 5 6) collect: [ :e | 2 \* e ]

>>> #(2 4 6 8 10 12)

Переваги використання *collect:* над *do:* ще ліпше видно в наступному прикладі, де за колекцією цілих чисел генерують колекцію значень за модулем цих чисел.

integers := #( 2 -3 4 -35 4 -11).

result := integers species new: integers size.

1 to: integers size do: [ :each |

result at: each put: (integers at: each) abs].

result

>>> #(2 3 4 35 4 11)

А тепер порівняємо його з таким кодом:

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

>>> #(2 3 4 35 4 11)

Ще однією перевагою другого підходу є те, що він працюватиме і з неіндексованими колекціями. У більшості випадків можна знайти відповідний ітератор і обійтися без *do:*.

Зауважимо, що повідомлення *collect:* повертає колекцію такого самого типу, як і отри­мувач. Тому наступний код не працюватиме (рядок не може зберігати цілі значення).

'abc' collect: [ :each | each asciiValue ]

>>> "Error: Improper store into indexable object"

Потрібно спершу перетворити *String* на *Array* або *OrderedCollection*.

'abc' asArray collect: [ :each | each asciiValue ]

>>> #(97 98 99)

Насправді, не гарантовано, що *collect:* поверне колекцію такого самого типу, що й отримувач – лише того самого *виду* (*species*). Наприклад, видом *Interval* є *Array*.

(1 to: 5) collect: [ :each | each \* 2 ]

>>> #(2 4 6 8 10)

## Вибір і відхилення елементів

Повідомлення *select:* повертає колекцію елементів отримувача, які задовольняють певну умову.

(2 to: 20) select: [:each | each isPrime]

>>> #(2 3 5 7 11 13 17 19)

Повідомлення *reject:* діє навпаки.

(2 to: 20) reject: [:each | each isPrime]

>>> #(4 6 8 9 10 12 14 15 16 18 20)

### Відшукання елемента за допомогою *detect:*

Повідомлення *detect:* повертає перший елемент отримувача, який задовольняє блок, аргумент повідомлення.

'through' detect: [ :letter | letter isVowel ]

>>> $o

Повідомлення *detect:ifNone:* приймає два блоки та є різновидом повідомлення *detect:*. Якщо жоден елемент не відповідає першому блоку, то видається значення другого.

Smalltalk allClasses

detect: [ :class | '\*cobol\*' match: class asString ]

ifNone: [ nil ]

>>> nil

### Накопичення результатів з *inject:into:*

Мови функціонального програмування часто підтримують функції вищих порядків, які називаються *fold* або *reduce*, для накопичення результату застосування деякої бінарної операції послідовно до кожного елемента колекції. У Pharo це реалізовано через *Collec­tion >> inject:into:*.

Перший аргумент – це початкове значення, другий – це блок з двома параметрами, який обчислюється для кожного отриманого дотепер результату і чергового елемента.

Найпростіше застосування *inject:into:* – обчислити суму елементів колекції чисел. Вираз для обчислення суми перших 100 натуральних чисел у Pharo можна написати так.

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

>>> 5050

Ще один приклад – обчислення факторіала за допомогою блока, що приймає один аргумент.

| factorial |

factorial := [ :n |

(1 to: n)

inject: 1

into: [ :product :each | product \* each ] ].

factorial value: 10

>>> 3628800

## Інші повідомлення вищого порядку

Існує багато інших повідомлень-ітераторів. Можна переглянути клас *Collection*, щоб їх побачити. Ось деякі з них.

***count:*** Повідомлення *count:* повертає кількість елементів, що задовольняють умову. Умову задають унарним блоком, що повертає булеве значення.

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

>>> 10

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

| colors |

colors := {Color white . Color yellow . 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

## Загальна помилка – використання результату *add:*

Наступна помилка є, мабуть, однією з найчастіших у Pharo.

| collection |

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

collection

>>> 2

Тут змінна *collection* містить не щойно створену колекцію, а лише останнє додане число. Так відбулося тому, що метод *add:* повертає не отримувача, а свій аргумент.

Код нижче видає бажаний результат.

| collection |

collection := OrderedCollection new.

collection add: 1; add: 2.

collection

>>> an OrderedCollection(1 2)

Також можна використати повідомлення *yourself*, щоб повернути отримувача каскаду повідомлень.

| collection |

collection := OrderedCollection new add: 1; add: 2; yourself

>>> an OrderedCollection(1 2)

## Загальна помилка – вилучення елемента під час перебору

Легко можна допустити ще одну помилку: вилучити елемент з колекції тоді, коли її перебирають. Таку помилку справді важко виловити, бо порядок перебору може зміню­ватися залежно від того, яку стратегію зберігання елементів використовує колекція.

| range |

range := (2 to: 20) asOrderedCollection.

range do: [:aNumber |

aNumber isPrime ifFalse: [ range remove: aNumber ] ].

Range

>>> "Error: #isPrime was sent to nil"

Вирішенням цієї проблеми є створення копії колекції перед ітеруванням.

| range |

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)

## Загальна помилка – перевизначення = без *hash*

Важко виявити помилку, коли перевизначили = і забули про *hash*. Ознакою помилки є втрата елементів, доданих у множину, або ж інша дивна поведінка колекції. Один загальний спосіб перевизначення *hash* запропонував Кент Бек (Kent Beck): потрібно використовувати *bitXor:* для комбінування хеш-значень складових частин об’єкта.

Розглянемо приклад. Припустимо, що дві книжки будуть однаковими, якщо однакови­ми є їхні автори і назви. Тоді можна перевизначити не лише =, а й *hash*, як записано нижче.

Book >> = aBook

self class = aBook class ifFalse: [^ false].

^ title = aBook title and: [ authors = aBook authors]

Book >> hash

^ title hash bitXor: authors hash

Виникає інша неприємна проблема, якщо використовувати змінний об’єкт, тобто такий, що його *hash*-значення може змінитися протягом часу існування як елемент множини *Set* або ключ словника *Dictionary*. Не робіть цього ніколи, хіба що любите шукати помилки!

## Підсумки розділу

Ієрархія колекцій забезпечує загальний словник для маніпулювання колекціями різних видів.

* Колекції принципово відрізняються способом зберігання елементів: підкласи *SequenceableCollection* зберігають їх у заданому порядку, *Dictionary* і його підкласи зберігають пари (ключ-значення), а *Set* і *Bag* невпорядковані.
* Більшість колекцій можна перетворити до іншого типу, надсилаючи їм повідом­лення *asArray*, *asOrderedCollection* тощо.
* Щоб впорядкувати колекцію, надішліть їй повідомлення *asSortedCollection*.
* Масив літералів (об’єктів, які можна створити без надсилання повідомлень) створюють за допомогою запису #( ... ), динамічний масив – за допомого { ... }.
* Словник *Dictionary* порівнює ключі на рівність, тому найкраще, коли ключами є рядки. Натомість, *IdentityDictionary* перевіряє ідентичність ключів, тому ліпше використовувати як ключі символи.
* *String* реалізує рядки, а також розуміє стандартні повідомлення колекцій. Крім того, *String* підтримує перевірку відповідності шаблонам простого вигляду. Для складніших застосунків потрібно використовувати пакет RegEx.
* Базове повідомлення для перебору колекції – *do:*. Воно корисне для побудови імперативного коду, наприклад, для зміни кожного елемента колекції або для надсилання повідомлення кожному елементу колекції.
* Замість *do:* часто використовують *collect:*, *select:*, *reject:*, *includes:*, *inject:into:* та інші повідомлення вищого рівня для опрацювання колекцій в однаковому стилі.
* Не можна видаляти елементи колекції під час перебору. Якщо необхідно змінити колекцію, то перебирати потрібно копію колекції.
* Якщо перевантажено =, то не забувайте перевантажити і *hash*.