Розділ 8
Два слова про синтаксис

Синтаксис Pharo дуже близький до синтаксису його предка Smalltalk.

Синтаксис мови спроєктовано так, що код програми під час читання звучить як спрощена англійська. Приклад синтаксису демонструє наведений нижче метод класу *Week*. Він перевіряє чи *DayNames* містить аргумент, тобто чи аргумент є правильною назвою дня тижня. Якщо так, то цей аргумент присвоюють змінній класу *StartDay*, а в протилежному випадку генерують повідомлення про помилку.

```
startDay: aSymbol

   (DayNames includes: aSymbol)
      ifTrue: [ StartDay := aSymbol ]
      ifFalse: [ self error: aSymbol,
                 ' is not a recognized day name' ]
```

У синтаксисі Pharo є мінімум вимог. По суті, весь синтаксис полягає у надсиланні повідомлень (у побудові виразів). Вирази складаються з невеликої кількості примітивних елементів (надсилання повідомлень, присвоєння, замикання, повернення результату тощо). Є лише шість зарезервованих слів, що позначають псевдозмінні, і немає спеціального синтаксису побудови структур керування чи створення нових класів. Замість цього використовують надсилання повідомлень об'єктам.

Наприклад, замість структури керування «*if-then-else*» галуження зображають повідомленням (таким як *ifTrue*:) до об'єкта екземпляра класу *Boolean*. Новий клас створюють надсиланням повідомлення його базовому класу. Будь-який потік керування у Pharo просто виражають повідомленнями, тому цей розділ не зможе охопити всіх, щоб перелік не став занадто довгим. Увагу зосереджено на найважливішому, тому читач мав би прочитати наступний розділ про булеві значення, колекції та блоки.

8.1. Синтаксичні елементи

Вирази складають з таких синтаксичних елементів.

1. Шість *псевдозмінних*: *self*, *super*, *nil*, *true*, *false* та *thisContext*.
2. Константні вирази або зображення об'єктів-*літералів*: числа, літери, рядки, символи та масиви.
3. Оголошення змінної (тимчасової змінної).
4. Присвоєння.
5. Блок коду (синтаксичне замикання).
6. Повідомлення.
7. Повернення результату виконання методу.

Нижче наведено таблицю з прикладами різних синтаксичних елементів.

Вираз	**Що означає**
`startPoint`	Ім’я змінної
`Transcript`	Ім’я глобальної змінної
`self`	Псевдозмінна
`1`	Десяткове ціле число
`2r101`	Двійкове ціле число
`1.5`	Дійсне число
`2.4e7`	Дійсне число в експотенційному записі
`$a`	Літера '*a*'
`'Hello'`	Рядок «*Hello*»
`#Hello`	Символ «*Hello*»
`#(1 2 3)`	Літерал масиву
`{ 1 . 2 . 1 + 2 }`	Динамічний масив
`"a comment"`	Коментар
`| x y |`	Оголошення змінних *x* та *y*
`x := 1`	Присвоєння значення 1 змінній *x*
`[ :x | x + 2 ]`	Блок з параметром, що обчислює *x* + 2
`<primitive: 1>`	Анотація методу (тут – примітивного)
`3 factorial`	Унарне повідомлення *factorial*
`3 + 4`	Бінарне повідомлення +
`2 raisedTo: 6 modulo: 10`	Ключове повідомлення *raisedto:modulo:*
`^ true`	Повернення значення *true*
`x := 5 . y := x * 2`	Два вирази розділено крапкою (.)
`Transcript show: 'Hello'; cr`	Два каскадні повідомлення відокремлено крапкою з комою (;)

Локальні змінні. *startPoint* – це ім’я змінної або ідентифікатор. За домовленістю ідентифікатори утворюють зі слів у «горбатому регістрі», тобто кожне слово, крім першого, починається з великої літери. Перша літера імені змінної екземпляра, аргументу методу чи блока, тимчасової змінної має бути малою. Це підказує читачеві, що змінна перебуває в приватній області видимості.

Спільні змінні. З великої літери починаються ідентифікатори глобальних змінних, змінних класу, спільних словників і назви класів. *Transcript* є глобальною змінною, екземпляром класу *TranscriptStream*.

Отримувач повідомлення. Псевдозмінна *self* посилається на об’єкт, що отримав повідомлення. Щоб опрацювати повідомлення, об’єкт виконує певний метод, у тілі якого цей об’єкт і є *self*. Це дає нам змогу надсилати в тілі методу нові повідомлення отримувачу. Ми називаємо *self* «отримувачем», тому що цей об’єкт отримує повідомлення, що призводить до виконання методу. Нарешті, *self* називається «псевдозмінною», оскільки ми не можемо безпосередньо змінити її значення або виконати присвоєння.

Цілі числа. В доповнення до звичайних десяткових цілих чисел, як *42*, наприклад, Pharo підтримує запис із зазначенням основи системи числення. *2r101* – це 101 у двійковій системі числення (бінарне ціле), що дорівнює десятковому числу 5.

Дійсні числа містять у записі десяткову крапку. Їх можна також записувати у форматі з плаваючою крапкою: *2.4e7* – це $2{,}4 \times 10^7$.

Літери – екземпляри класу *Character*. Перед літерою записують знак долара. Наприклад, *$a* є зображенням літери '*a*'. Недруковані літери можна отримати за допомогою відповідного повідомлення класу *Character*, таких як *Character space* або *Character tab*.

Рядки є екземплярами класу *String*. Для позначення їх беруть в одинарні лапки. Якщо потрібно помістити одинарну лапку всередину рядка, то її треба продублювати, наприклад, *'G''day'*.

Символи схожі на рядки, бо містять послідовності літер. Проте, на відміну від рядків, символи унікальні: кожен з них гарантовано відрізняється від усіх інших. У системі може бути тільки один об'єкт *#Hello*, екземпляр класу *Symbol*, а рядків, екземплярів класу *String*, зі значенням *'Hello'* – скільки завгодно.

Масиви, які створюються на етапі компіляції, визначають за допомогою *#( )*, що обрамляють послідовність літералів, розділених пропусками. Між дужками можна записувати виключно константи, які можна створити на етапі компіляції. Наприклад, *#(27 (true) abc 1+2)* є літералом масиву, що складається з шести елементів: цілого числа 27, масиву етапу компіляції з одного елемента *true* – незмінного логічного об'єкта, символу *#abc*, цілого *1*, символу *#+* і цілого *2*. Цей самий приклад можна записати по-іншому: *#(27 #(true) #abc 1 #+ 2)*.

Масиви, що створюються на етапі виконання. Фігурні дужки *{ }* визначають динамічний масив, елементами якого є вирази, розділені крапкою. Наприклад, *{ 5 factorial . 99 . 7 * 8 }* визначає масив з трьох елементів: 120, 99 і 56 – перший та останній отримано в результаті обчислення виразів.

Коментарі обрамляють подвійними лапками. *"hello"* – це коментар, а не рядок. Компілятор Pharo ігнорує всі коментарі. Вони можуть займати кілька рядків, але не можуть бути вкладені.

Визначення локальних змінних. Вертикальні риски | | обрамляють оголошення однієї чи кількох локальних змінних на початку методу чи тіла блока.

Присвоєння позначається парою символів *:=*. Після виконання виразу *variable := object* змінна *variable* містить посилання на об'єкт *object*.

Блоки. За допомогою квадратних дужок *[ ]* визначають блок, також відомий як блокове замикання або лексичне замикання. Блок є об'єктом першого класу[1], що представляє функцію. Згодом побачимо, що блоки можуть приймати аргументи (*[:i | ...]*) і можуть мати локальні змінні (*[| x | ...]*). Блоки замикають імена зі свого середовища визначення, тобто вони можуть посилатися на змінні, які були доступні на момент їх визначення.

[1] *Об'єктом першого класу* називають сутність програми (не обов'язково об'єкт у сенсі ООП), яку можна присвоїти змінній, передати у функцію як аргумент, отримати з функції як результат, порівняти на ідентичність (*прим. – Ярошко С.*).

Прагми та примітиви. *<primitive: ...>* є анотацією методу. Наведений приклад позначає виклик примітиву[2] віртуальної машини. Якщо після примітиву записано програмний код, то він або пояснює, що робить примітив (для основного примітиву), або виконується лише тоді, коли примітив завершився помилкою (для необов'язкового примітиву). Той самий синтаксис «повідомлення в кутових дужках < >» використовують також для інших типів анотацій методів, які ще називають прагмами.

Унарні повідомлення складаються з одного слова (наприклад, *factorial*), яке надсилають отримувачу (наприклад, об'єкту *5*). У виразі «*5 factorial*» *5* – отримувач, а *factorial* – селектор повідомлення.

Бінарні повідомлення. Це повідомлення з одним аргументом і селектором, що схожий на математичний оператор, наприклад, +. У виразі *3 + 4* отримувач *3*, селектор повідомлення + і аргумент *4*.

Ключові повідомлення мають селектор, що складається з одного чи кількох слів (наприклад, *raisedTo: modulo:*), кожне з яких закінчується двокрапкою і приймає один аргумент. У виразі «*2 raisedTo: 6 modulo: 10*» селектор повідомлення «*raisedTo:modulo:*» приймає два аргументи *6* і *10*, кожен з яких розташовано після двокрапки. Повідомлення надіслано отримувачу *2*.

Послідовності тверджень. Крапка (.) є розділювачем тверджень. Якщо між двома виразами поставити крапку, то вони перетворяться на два незалежні твердження або інструкції. Наприклад, у фрагменті «*x := 5 . y := x * 2*» ми спочатку присвоюємо значення *2* змінній *x*, а тоді подвоюємо його і поміщаємо результат в *y*.

Каскадні повідомлення. Крапку з комою (;) використовують для об'єднання в каскад кількох повідомлень тому самому отримувачу. У виразі «*Transcript show: 'hello'; cr*» спочатку надсилаємо ключове повідомлення *show: 'hello'* отримувачу *Transcript*, а тоді надсилаємо унарне повідомлення *cr* тому ж отримувачу.

Повернення з методу. Оператор ^ використовують для повернення значення з методу.

Базові класи *Number*, *Character*, *String* і *Boolean* описані у розділі 13.

8.2. Псевдозмінні

У Pharo є шість псевдозмінних: *nil*, *true*, *false*, *self*, *super* та *thisContext*. Їх називають *псевдозмінними* тому, що їхні значення наперед визначені і не можуть бути змінені. *true*, *false* і *nil* є константами, а значення *self*, *super* та *thisContext* змінюються динамічно залежно від виконуваного коду.

- *true* і *false* – єдині екземпляри класів *True* та *False* відповідно, а ті є підкласами класу *Boolean*. Детальніше вони описані у розділі 13 «Базові класи».
- *self* завжди посилається на отримувача повідомлення і позначає об'єкт, для якого буде виконано відповідний метод. Тому значення *self* динамічно змінюється під час виконання програми, але йому нічого не можна присвоїти в коді.
- *super* також посилається на отримувача повідомлення, але механізм пошуку методу для опрацювання повідомлення, надісланого до *super*, працює інакше: він

[2] Частина методів Pharo заради ефективності реалізовані як примітиви віртуальної машини (*прим. – Ярошко С.*).

починає пошук *з надкласу* того класу, в методі якого трапилося повідомлення. За детальнішою інформацією зверніться до розділу 10 «Об'єктна модель Pharo».

- *nil* – це невизначений об'єкт. Він єдиний екземпляр класу *UndefinedObject*. Змінні екземплярів, класів і локальні змінні за замовчуванням під час ініціалізації отримують значення *nil*.
- *thisContext* – це псевдозмінна, яка представляє вершину стеку викликів і надає доступ до поточної точки виконання. Для більшості програмістів змінна *thisContext* зазвичай не цікава, але її суттєво використовують для створення інструментів розробки, наприклад, налагоджувача та для опрацювання винятків і реалізації відкладених обчислень.

8.3. Повідомлення і надсилання повідомлень

Як ми вже писали, у Pharo є три типи повідомлень з визначеним пріоритетом. У різних типів повідомлень пріоритети зроблено різними, щоб зменшити кількість обов'язкових дужок.

Тут наведемо короткий огляд типів повідомлень, способів їхнього надсилання та виконання. Детальний опис можна знайти в розділі 9 «Розуміння синтаксису повідомлень».

1. *Унарні* повідомлення не мають аргументів. У виразі «*1 factorial*» об'єктові *1* надсилають повідомлення *factorial*. Селектори унарних повідомлень складаються з букв, цифр, літери '_' і починаються з малої букви.
2. *Бінарні* повідомлення завжди приймають один аргумент. У виразі «*1 + 2*» об'єктові *1* надсилають повідомлення *+* з аргументом *2*. Селектори бінарних повідомлень складаються з однієї або більше літер з набору « *+ - / * ~ < > = @ % | & ? ,* ».
3. *Ключові* повідомлення приймають довільну кількість аргументів. У виразі «*2 raisedTo: 6 modulo: 10*» об'єктові *2* надсилають повідомлення, яке складається з селектора *raisedTo:modulo:* і аргументів *6* та *10*. Селектори ключових повідомлень складаються з одного або більше ключових слів, кожне з яких складається з букв і цифр, починається з малої букви і закінчується двокрапкою.

Пріоритет повідомлень

Унарні повідомлення мають найвищий пріоритет, далі йдуть бінарні, а у ключових повідомлень пріоритет найнижчий. Щоб змінити порядок обчислення виразів, визначений пріоритетами, використовують круглі дужки.

Отже, у наведеному нижче прикладі ми спочатку надсилаємо повідомлення *factorial* об'єктові *3*, що дає нам *6*. Далі надсилаємо *+ 6* об'єктові *1*, звідки отримуємо *7*. І нарешті, надсилаємо об'єкту 2 повідомлення *raisedTo: 7*.

```
2 raisedTo: 1 + 3 factorial
>>> 128
```

Повідомлення одного типу мають однаковий пріоритет, їх завжди виконують зліва направо. Тому наведений нижче вираз, що містить два бінарні повідомлення, поверне *9*, а не *7*.

```
1 + 2 * 3
```

```
>>> 9
```

Для зміни порядку обчислення використовують круглі дужки:

```
1 + (2 * 3)
>>> 7
```

8.4. Послідовності та каскади

Усі вирази можна об'єднувати в послідовності, відокремлюючи їх крапкою. Надсилання повідомлень також можна об'єднувати в каскад, записавши їх через крапку з комою. Кожен вираз у послідовності відокремлених крапкою виразів виконується почергово, один після одного, як окрема *інструкція*.

```
Transcript cr.
Transcript show: 'Hello world'.
Transcript cr.
```

Тут перша інструкція надсилає повідомлення *cr* об'єкту *Transcript*, друга надсилає до *Transcript* повідомлення *show: 'Hello world',* і остання знову надсилає *cr*.

Коли послідовність повідомлень надсилають *тому самому* отримувачу, то це легше виразити *каскадом* повідомлень. Отримувача зазначають один раз, а послідовність повідомлень записують через крапку з комою, як у прикладі нижче.

```
Transcript
   cr;
   show: 'Hello world';
   cr
```

Цей каскад виконує таку саму роботу, як і послідовність у попередньому прикладі.

8.5. Синтаксис методу

Ми вже знаємо, що вирази можна виконувати будь-де у Pharo (наприклад, у Пісочниці, Робочому вікні, Оглядачі класів, Налагоджувачі). Методи ж зазвичай визначають в Оглядачі класів, або в Налагоджувачі. Методи також можна завантажувати з файлів, але це не поширений спосіб програмування у Pharo.

Розробка програми полягає у визначенні класів і методів: по одному методу за раз. Метод визначають у межах певного класу. Новий клас створюють надсиланням повідомлення наявному класові з проханням утворити підклас, тому визначення класів не потребує спеціального синтаксису.

Нижче наведено метод *lineCount*, визначений у класі *String*. За домовленістю для ідентифікації методу ми використовуємо запис *ІмʼяКласу >> імʼяМетоду*, отже, у прикладі йдеться про метод *String >> lineCount*. Нагадаємо, що запис *ClassName >> methodName* не є частиною синтаксису Pharo, а лише домовленістю, яку використовуємо в книзі для зрозумілого вказання на метод з зазначенням класу, в якому він визначений.

```
String >> lineCount
  "Answer the number of lines represented by the receiver, where
    every cr adds one line."
```

```
| cr count |
cr := Character cr.
count := 1 min: self size.
self do: [:c | c == cr ifTrue: [count := count + 1]].
^ count
```

Синтаксично метод складається з:

1) заголовка методу, що охоплює ім'я (тут *lineCount*) і аргументи (немає в цьому прикладі);
2) коментарів, які можна розташовувати в будь-якій частині коду, але за домовленістю коментар з поясненнями, що робить метод, записують після заголовка;
3) визначення локальних змінних (тут *cr* і *count*);
4) довільної кількості тверджень або інструкцій, розділених крапками (у цьому прикладі їх – чотири).

Виконання будь-якої інструкції, перед якою стоїть літера ^ («шапочка», або стрілочка догори, яку на більшості клавіатур можна набрати комбінацією [*Shift* + 6]) призведе до виходу з методу і повернення як результат значення виразу, записаного після ^. Виконання методу може завершитися і без явної інструкції повернення результату просто, коли буде виконано його останню інструкцію. У цьому випадку метод поверне об'єкт *self*.

Імена аргументів і локальних змінних треба завжди розпочинати з малої літери. Імена, що починаються з великої літери, використовують для глобальних змінних. Імена класів, як наприклад, *Character*, є звичайними глобальними змінними, що посилаються на об'єкт, який представляє клас.

8.6. Синтаксис блока

Блоки (лексичні замикання) надають механізм відкладеного виконання інструкцій. Насправді блок є анонімною функцією, визначеною у певному контексті. Щоб виконати блок, йому надсилають повідомлення *value*. Результатом обчислення блока зазвичай є значення останнього виразу в його тілі, якщо тільки немає оператора явного повернення результату (^). Якщо ж є, то блок поверне значення виразу, зазначеного після оператора повернення.

```
[ 1 + 2 . 44 ] value
>>> 44
```

```
[ 3 = 3 ifTrue: [ ^ 33 ]. 44 ] value
>>> 33
```

Блоки можуть мати параметри. Оголошення кожного з них розпочинається з двокрапки. Оголошення параметрів відокремлюють від тіла блока вертикальною рискою. Щоб виконати блок з одним параметром, потрібно надіслати йому повідомлення *value:* з одним аргументом. Якщо блок має два параметри, то йому надсилають повідомлення *value:value:* з двома аргументами, і так далі аж до чотирьох аргументів.

```
[ :x | 1 + x] value: 2
>>> 3
```

```
[ :x: :y | x * y] value: 11 value: 4
>>> 44
```

Якщо ваш блок має більше ніж чотири параметри, то для його обчислення треба використати повідомлення *valueWithArguments:* та передати йому *один* масив, який містить *всі* потрібні аргументи.

```
[ :a :b :c :d :e | a + b + c + d + e ]
   valueWithArguments: #[ 1 2 3 4 5 ]
>>> 15
```

Так можна робити, проте наявність блока з великою кількістю параметрів є ознакою неправильної архітектури.

Всередині блока можна оголошувати локальні змінні так само, як в методі, обрамляючи їх вертикальними рисками. Оголошення локальних змінних розташовують після параметрів блока і риски-розділювача перед тілом блока. У прикладі нижче *x* та *y* є параметрами, а *z* – локальною змінною.

```
[ :x :y |
  | z |
  z: = x + y.
  z] value: 1 value: 2
>>> 3
```

Блоки є лексичними замиканнями, тому можуть посилатися на змінні зі свого оточення. У наступному прикладі блок посилається на змінну *x*, розташовану поза ним у методі, де оголошено блок.

```
| x |
x: = 1.
[ :y | x + y ] value: 2
>>> 3
```

Блоки є екземплярами класу *BlockClosure*. Це означає, що вони є об'єктами, тому їх можна присвоювати змінним, їх можна передавати як аргумент повідомлення як і будь-який інший об'єкт.

8.7. Галуження і повторення

Pharo не потребує спеціального синтаксису для структур керування. Їх можна виразити через надсилання повідомлень логічним значенням, числам, чи колекціям з аргументами блоками. Крім того, програмісти у Pharo також можуть визначати свої власні галуження, ітераційні та арифметичні цикли за допомогою розширення можливостей базових об'єктів (булевих значень, цілих чисел, колекцій або блоків). Далі описано часто вживані повідомлення, але це далеко не повний перелік.

Деякі галуження

Галуження виражають за допомогою надсилання одного з повідомлень *ifTrue:*, *ifFalse:* або *ifTrue:ifFalse:* чи *ifFalse:ifTrue:* результатові обчислення логічного виразу. Доклад-

ніше про побудову логічних виразів і логічні величини написано в розділі 13 «Базові класи».

```
(17 * 13 > 220)
  ifTrue: [ 'bigger' ]
  ifFalse: [ 'smaller' ]
>>> 'bigger'
```

Деякі цикли

Повторення виражають через надсилання повідомлень блокам, цілим числам або колекціям. Під час виконання ітераційного циклу умову завершення потрібно обчислювати на кожному кроці, тому її треба зображати блоком, а не логічним виразом. Нижче наведено приклад типового циклу.

```
n: = 1.
[ n < 1000 ] whileTrue: [ n: = n * 2 ].
n
>>> 1024
```

whileFalse: дає змогу використати протилежну умову:

```
n: = 1.
[ n >= 1000 ] whileFalse: [ n: = n * 2 ].
n
>>> 1024
```

timesRepeat: надає простий спосіб повторити тіло циклу певну кількість разів.

```
n: = 1.
10 timesRepeat: [ n: = n * 2 ].
n
>>> 1024
```

Ми також можемо надіслати повідомлення *to:do:* будь-якому числу. Так утворимо арифметичний цикл з лічильником, початковим значенням якого буде одержувач, кінцевим значенням – перший аргумент повідомлення, а тілом циклу – другий аргумент, блок з параметром. Лічильник циклу змінюватиметься з кроком 1 і надсилатиметься блокові як аргумент на кожній ітерації циклу. Щоб задати довільний крок зміни параметра циклу, використайте повідомлення *to:by:do:*.

```
result: = String new.
1 to: 10 do: [ :n | result: = result, n printString, ' '].
result
>>> '1 2 3 4 5 6 7 8 9 10 '
```

Ітератори вищих порядків

До колекцій належить велика кількість класів, багато з яких підтримують однаковий протокол. Найважливішими повідомленнями для перебирання вмісту колекції є *do:*, *collect:*, *select:*, *reject:*, *detect:* і *inject:into:*. Ці повідомлення надають ітератори[3], які допомагають писати дуже стислий і виразний код.

[3] *Ітератором* у Pharo називають *метод* перебирання елементів колекції, на відміну, наприклад, від C++, де ітератором є об'єкт (*прим. – Ярошко С.*).

Interval – колекція, яка дає змогу перебирати послідовність чисел від початкового до кінцевого. *1 to: 10* реалізує інтервал від 1 до 10. Оскільки інтервал є колекцією, то йому можна послати повідомлення *do:*. Його аргументом є блок з параметром, блок буде виконано з кожним елементом колекції.

```
result: = String new.
(1 to: 10) do: [ :n | result := result, n printString, ' '].
result
>>> '1 2 3 4 5 6 7 8 9 10 '
```

collect: створює нову колекцію такого ж розміру і типу, що й отримувач. Її вміст формують результати застосування блока, аргументу повідомлення, до кожного елемента отримувача. Ви можете трактувати *collect:* як *Map* у моделі програмування MapReduce.

```
(1 to: 10) collect: [: each | each * each]
>>> #(1 4 9 16 25 36 49 64 81 100)
```

select: і *reject:* створюють нову колекцію, що охоплює підмножину елементів ітерованої колекції, які, відповідно, задовольняють, чи ні умову, задану логічним виразом у блоці, аргументі повідомлення.

detect: повертає перший елемент колекції, який задовольняє умову.

Варто нагадати, що рядок також є колекцією (літер), тому його можна ітерувати по літерах.

```
'hello there' select: [ :char | char isVowel ]
>>> 'eoee'
```

```
'hello there' reject: [ :char | char isVowel ]
>>> 'hll thr'
```

```
'hello there' detect: [ :char | char isVowel ]
>>> $e
```

Нарешті, ви маєте знати, що колекції також підтримують функціональний оператор згортки методом *inject:into:*. Ви також можете трактувати його як *Reduce* у моделі програмування MapReduce. Метод дає змогу накопичувати результат за допомогою виразу, який починає обчислення з заданого значення та враховує кожен елемент колекції. Типовими прикладами є обчислення сум і добутків.

```
(1 to: 10) inject: 0 into: [ :sum :each | sum + each ]
>>> 55
```

Це те саме, що й 0+1+2+3+4+5+6+7+8+9+10.

Більше про колекції можна дізнатися з розділу 14 «Колекції».

8.8. Анотації методів: примітиви і прагми

У Pharo перед тілом методу можна записувати анотацію. Її обрамляють кутовими дужками: < та >. Анотації застосовують у двох випадках: для виконання примітивів та оголошення метаданих.

Примітиви

У Pharo все є об'єктом і все відбувається через надсилання повідомлень. Проте в певних ситуаціях ми впираємося в природні обмеження мови. Деякі об'єкти можуть виконувати роботу лише за допомогою примітивів віртуальної машини – базових примітивів, які не можна виразити термінами Pharo.

Наприклад, всі далі перелічені повідомлення реалізовані як примітиви: виділення пам'яті (*new*, *new:*), побітові операції (*bitAnd:*, *bitOr:*, *bitShift:*), вказівники та арифметика цілих чисел (+, -, <, >, *, /, =, ==, …), доступ до масивів (*at:*, *at:put:*).

Коли виконується метод з примітивом, то замість методу виконується код примітиву. Метод, що викликає примітив, може також містити код Pharo, який виконається, якщо примітив завершиться помилкою (таке можливо для необов'язкових примітивів).

Нижче показано код методу *SmallInteger >> +*. Якщо примітив завершиться помилкою, то виконається наступне (по тексту) повідомлення *super + aNumber*, яке й поверне результат.

```
+ aNumber
"Primitive. Add the receiver to the argument and answer with the result
if it is a SmallInteger. Fail if the argument or the result is not a
SmallInteger Essential No Lookup. See Object documentation
whatIsAPrimitive."

<primitive: 1>
^ super + aNumber
```

Прагми

Обрамлення кутовими дужками у Pharo використовують також для анотації методу, яку називають прагмою. Після того, як метод анотовано за допомогою прагми, анотації можна зібрати за допомогою колекції (див. клас PragmaCollector).

8.9. Підсумки розділу

- У Pharo є лише шість зарезервованих ідентифікаторів, відомих як псевдозмінні: *true*, *false*, *nil*, *self*, *super* та *thisContext*.
- Є п'ять типів об'єктів, які можна задати літералом: числа (*5*, *2.5*, *1.9e15*, *2r111*), літери (*$a*), рядки (*'Hello'*), символи (*#hello*) та масиви (статичні #('hello' #hi) або динамічні { *5 factorial . 7 * 8 . 1 + 2* }).
- Рядки обмежують апострофами, коментарі – подвійними лапками. Щоб записати апостроф усередині рядка, його подвоюють.
- На противагу рядкам символи завжди унікальні.

- Щоб задати масив літералом на етапі компіляції, використовують позначення *#(...)*. За допомогою позначення *{ ... }* визначають динамічний масив на етапі виконання. Зауважте, що *#(1 + 2) size* >>> *3*, але *{ 1 + 2 } size* >>> *1*. Щоб зрозуміти, чому, порівняйте *#(1 + 2) inspect* і *{ 1 + 2 } inspect*.
- Є три види повідомлень: унарні (наприклад, *15 asString*; *Time now*), бінарні (наприклад, *3 + 4*; 'hi ', 'there'), ключові (*1 to: 15 by: 2*).
- Надсилання каскаду повідомлень діє як надсилання послідовності повідомлень до того самого отримувача. Повідомлення в каскаді відокремлюють крапкою з комою: *OrderedCollection new add: #calvin; add: #hobbes; size* >>> *2*.
- Оголошення локальних змінних обмежують вертикальними рисками. Символом := позначають оператор присвоєння. | *x* | *x* := *1*.
- Вирази складаються з надсилання повідомлень, каскадів і присвоєнь, виконуються зліва направо, можуть бути згруповані круглими дужками. Інструкція або твердження – це вираз, що закінчується крапкою. Твердження відокремлюють крапками.
- Блокове замикання – це вираз у квадратних дужках. Блоки можуть приймати аргументи та містити локальні змінні. Вирази у блоці не будуть виконані доки він не отримає повідомлення *value* з правильною кількістю аргументів – *value*, *value:arg*, *value:arg1 value:arg2* тощо. *[:x | x + 2] value: 4*.
- Немає спеціального синтаксису для структур керування, натомість надсилають повідомлення з аргументами-блоками, які за певних умов можуть виконувати ці блоки.