Базові класи

Pharo – це справді проста, але потужна мова програмування. Частина її сили полягає не в самій мові, а в її бібліотеках класів. Щоб ефективно програмувати нею, потрібно дізна­тися, як бібліотеки класів підтримують мову і середовище. Бібліотеки класів повністю написані на Pharo і легко можуть бути доповнені. Доповнення можна зібрати в окремий пакет: пакет може додати нові методи до класу навіть, якщо він не визначає цей клас.

Завдання розділу не описати всі бібліотеки класів Pharo аж до нудних деталей, а швидше розповісти про ключові класи та методи, які доведеться використовувати (або наслідувати чи перевантажувати), щоб ефективно програмувати. У розділі описано основні класи, які будуть потрібні майже для кожного застосунку: *Object*, *Number* і його підкласи, *Character*, *String*, *Symbol* і *Boolean*.

## Object

На всі випадки життякоренем ієрархії наслідуванняє *Object*. Хоча в Pharo справжній корінь ієрархії – це *ProtoObject*, який використовують для визначення мінімальних сутностей, замаскованих під об’єкти, але деякий час цю обставину можна ігнорувати.

*Object* визначає майже 400 методів. Іншими словами, кожен клас, який ви визначили, автоматично підтримуватиме всі ці методи. Зауважимо, що кількість методів у класі можна порахувати програмно, як показано нижче.

Object selectors size "Лічить методи екземпляру Object"

Object class selectors size "Лічить методи класу"

Клас *Object* забезпечує спільну для всіх звичайних об’єктів поведінку за замовчуван­ням – доступ, копіювання, порівняння, опрацювання помилок, надсилання повідомлень і рефлексія. Тут також визначено допоміжні повідомлення, на які повинні реагувати всі об’єкти. *Object* не має змінних екземпляра, і не мав би їх мати. Це пов’язано з тим, що *Object* наслідують кілька класів, які мають спеціальну реалізацію (наприклад, *SmallInte­ger* і *UndefinedObject*). Віртуальна машина знає про них, та вони залежать від структури і влаштування деяких стандартних класів.

Якщо переглянути протоколи методів на стороні екземпляра *Object*, то можна побачити головні риси поведінки, яку він надає.

## Друк об’єкта

Кожен об’єкт здатний повертати свій друкований вигляд. Можна позначити будь-який вираз у вікні редактора тексту і вибрати команду «*Print it*» контекстного меню. Вона виконає вираз і просить обчислений об’єкт надрукувати себе. Насправді об’єктові, що повертається, надсилається повідомлення *printString*. Метод *printString* – це шаблонний метод, який всередині надсилає отримувачу повідомлення *printOn:*. Повідомлення *printOn:* – зачіпка, яку можна спеціалізувати в своїх класах.

Видається, що метод *Object >> printOn:* – один з тих, які перевантажують найчастіше. Він приймає як аргумент екземпляр *Stream*, в який буде записано зображення об’єкта у вигляді рядка *String*. Реалізація методу за замовчуванням тільки записує ім’я класу з артиклем *'a'* або *'an'*. *Object >> printString* повертає записаний рядок.

Наприклад, у класі *OpalCompiler* не перевантажено метод *printOn:*, і надсилання повідомлення *printString* екземплярові класу виконує визначені в *Object* методи.

OpalCompiler new printString

>>> 'an OpalCompiler'

Клас *Color* демонструє приклад спеціалізації методу *printOn:*.

Color >> printOn: aStream

| name |

(name := self name).

name = #unnamed

ifFalse: [

^ aStream

nextPutAll: 'Color ';

nextPutAll: name ].

self storeOn: aStream

Він друкує ім’я класу, а слідом – ім’я методу класу, використаного для створення цього кольору.

Color red printString

>>> 'Color red'

### *printOn:* проти *displayStringOn:*

Потрібно враховувати, що метод *printOn:* призначено для зрозумілого відображення об’єктів під час розробки. Справді, під час використання інспектора або налагоджувача набагато інформативніше бачити точний опис об’єкта замість загального. Водночас *printOn:* не призначено для побудови інтерфейсу користувача, наприклад, для гарного відображення об’єктів у списках, оскільки зазвичай потрібно відображати інший тип інформації. Для цього треба використовувати *displayStringOn:*. Стандартна реалізація *displayStringOn:* викликає *printOn:*.

Зауважте, що метод *displayStringOn:* запроваджено нещодавно, тому багато бібліотек ще не враховують цієї різниці. Насправді це не проблема, але коли пишете новий код, то маєте знати про це.

### *printOn:* проти *storeOn:*

Зверніть увагу, що повідомлення *printOn:* це не те ж саме, що й *storeOn:*. Метод *storeOn:* записує в свій потік-аргумент вираз, який можна використати для відтворення отри­мувача. Такий вираз виконається, коли прочитати його з потоку повідомленням *readFrom:*. На противагу *storeOn:*, повідомлення *printOn:* повертає лише текстове зображення приймача. Звичайно, може трапитися так, що воно зображатиме приймача виразом, придатним до виконання.

## Зображення і самовідтворення

У функціональному програмуванні вирази після виконання повертають значення. У Pharo повідомлення (вирази) повертають об’єкти (значення). Деякі об’єкти мають таку чудову властивість, що їхнім значенням є вони самі. Наприклад, значенням об’єкта *true* є об’єкт *true*. Такі об’єкти називають *свмовідтворюваними*. Друковану версію значення об’єкта можна побачити під час друку об’єкта в Робочому вікні. Ось деякі приклади таких самовідтворюваних виразів.

True

>>> true

3@4

>>> (3@4)

$a

>>> $a

#(1 2 3)

>>> #(1 2 3)

Color red

>>> Color red

Зауважимо, що самовідтворюваність деяких об’єктів залежить від того, які об’єкти во­ни містять. Наприклад, масив булевих величин самовідтворюваний, а масив об’єктів типу *Person* – ні. З прикладів нижче видно, що динамічні масиви самовідтворювані, якщо такі їхні елементи.

{10@10. 100@100}

>>> {(10@10). (100@100)}

{OpalCompiler new . 100@100}

>>> an Array(an OpalCompiler (100@100))

Нагадаємо, що літерали масивів можуть містити тільки літерали. Тому масив нижче містить не дві точки, а шість літералів.

#(10@10 100@100)

>>> #(10 #@ 10 100 #@ 100)

Багато спеціалізацій методу *printOn:* реалізують самовідтворювану поведінку. Наприк­лад, реалізації *Point>>printOn:* та *Interval>>printOn:* самовідтворювані (знайдіть їх за допомогою Оглядача класів).

1 to: 10

>>> (1 to: 10) "інтервали самовідтворювані"

## Ідентичність і рівність

У Pharo повідомлення «=» перевіряє *рівність* об’єктів, тоді як «==» перевіряє їхню *ідентичність*. Тобто, перше з них перевіряє, чи представляють два об’єкти те саме значення, а друге – чи результати обчислення двох виразів є тим самим об’єктом.

Реалізація за замовчуванням перевірки рівності об’єктів є перевіркою ідентичності.

Object >> = anObject

"Відповідає, чи отримувач і аргумент представляють той самий об’єкт.

Якщо в якомусь підкласі перевизначають =, потрібно також перевантажити

hash."

^ self == anObject

Якщо в класі перевантажують =, то треба розглянути можливість перевантажити *hash*. Якщо екземпляри такого класу коли-небудь стануть ключами в словнику, то потрібно переконатися, що екземпляри, які вважаються рівними, мають однакове хеш-значення.

Методи = і *hash* перевантажують разом, а метод == не перевантажують *ніколи*. Семантика ідентичності об’єктів однакова для всіх класів. Повідомлення == реалізує примітивний метод класу *ProtoObject*.

Зауважте, що Pharo має дивну поведінку рівності порівняно з іншими реалізаціями Smalltalk. Наприклад, символ і рядок можуть бути рівними. (Ми вважаємо це помилкою, а не корисною особливістю).

#'lulu' = 'lulu'

>>> true

'lulu' = #'lulu'

>>> true

'lulu' = #lulu

>>> true

## Належність до класу

Кілька методів дають змогу довідатися клас об’єкта.

### *class*

Будь-який об’єкт можна запитати про його клас за допомогою повідомлення *class*.

1 class

>>> SmallInteger

### *isMemberOf:*

З іншого боку, можна запитати чи об’єкт є екземпляром конкретного класу:

'lulu' isMemberOf: Symbol

>>> false

### *isKindOf:*

*Object>> isKindOf:* відповідає, чи клас отримувача є класом-аргументом, або його підкласом.

1 isKindOf: SmallInteger

>>> true

1 isKindOf: Integer

>>> true

1 isKindOf: Number

>>> true

1 isKindOf: Object

>>> true

1 isKindOf: String

>>> false

1/3 isKindOf: Number

>>> true

1/3 isKindOf: Float

>>> false

1/3 екземпляр класу *Fraction* і різновид *Number*, бо клас *Number* надклас класу *Fraction*. Але 1/3 не є дійсним чи цілим.

### *respondsTo:*

*Object >> respondsTo:* відповідає, чи отримувач розуміє повідомлення, чий селектор задано аргументом.

1 respondsTo: #,

>>> false

## Про isKindOf: і respondsTo:

Зауваження про використання *isKindOf:* і *respondsTo:*. Зазвичай це погана ідея питати об’єкт про його клас, або запитувати його, які повідомлення він розуміє. Замість того, щоб приймати рішення, які ґрунтуються на класі об’єкта, потрібно відправити об’єкто­ві повідомлення і дозволити йому самому вирішити (опираючись на свій клас), як він має себе поводити. Клієнт об’єкта не мав би запитувати його, щоб вирішити, яке пові­домлення надіслати. Наріжний камінь об’єктно-орієнтованого проєктування – «не питай, а кажи». Тому будьте обережні, якщо вам знадобиться використовувати ці повідомлення.

## Поверхневе копіювання об’єктів

Копіювання об’єктів порушує деякі складні питання. Оскільки змінні екземпляра зберігають посилання на значення, то *поверхнева копія* об’єкта поділятиме їх зі змінними оригіналу.

a1 := { { 'harry' } }.

a1

>>> #(#('harry'))

a2 := a1 shallowCopy.

a2

>>> #(#('harry'))

(a1 at: 1) at: 1 put: 'sally'.

a1

>>> #(#('sally'))

a2

>>> #(#('sally')) "вкладений масив спільний!"

*Object >> shallowCopy* – примітивний метод, який створює поверхневу копію об’єкта. Оскільки *a2* – тільки поверхнева копія *a1*, то обидва масиви поділяють посилання на вкладений масив, який вони містять.

## Глибоке копіювання об’єктів

Є два способи розв’язати проблему спільних посилань, яка виникає під час поверхнево­го копіювання: (1) використати *deepCopy*, (2) перевизначити *postCopy* і використовува­ти *copy*.

### *deepCopy*

*Object >> deepCopy* робить як завгодно глибоку копію об’єкта.

a1 := { { { 'harry' } } }.

a2 := a1 deepCopy.

(a1 at: 1) at: 1 put: 'sally'.

a1

>>> #(#('sally'))

a2

>>> #(#(#('harry')))

Проблема з *deepCopy* полягає в тому, що він не завершиться, коли застосовується до взаємно рекурсивної структури.

a1 := { 'harry' }.

a2 := { a1 }.

a1 at: 1 put: a2.

a1 deepCopy

>>> !''... does not terminate!''!

### *copy*

Альтернативним рішенням є використання повідомлення *copy*. Метод *Object >> copy* реалізовано так.

Object >> copy

"Повертає інший екземпляр, схожий на отримувача. Підкласи зазвичай

перевантажують postCopy і не перевантажують shallowCopy."

^ self shallowCopy postCopy

Object >> postCopy

^ self

Метод *copy* надсилає повідомлення *postCopy* результатові поверхневого копіювання. За замовчуванням *postCopy* повертає *self*. Це означає, що за замовчуванням *postCopy* робить те саме, що й *shallowCopy*, але кожен підклас може вирішити перевантажити *postCopy*, який відіграє роль зачіпки. Потрібно перевантажити *postCopy*, щоб скопіювати ті значення змінних екземпляра, які не можна поділяти. Крім того, *postCopy* завжди мав би надсилати *super postCopy*, щоб впевнитися, що стан надкласу також скопійовано.

## Налагодження

Клас *Object* визначає кілька методів, що стосуються налагодження.

### *halt*

Найважливішим серед них є *halt*. Щоб встановити в методі точку переривання, вставте «*self halt.*» в потрібному місці тіла методу. Зауважте, що можна також написати «*1 halt.*», бо метод визначено в *Object*.

Після надсилання повідомлення виконання перерветься і відкриється Налагоджувач у тому місці програми.

Можна також використовувати *Halt once*, або *Halt if: aCondition*. Перегляньте клас *Halt* – це виняток, призначений для налагодження.

### *assert:*

Наступне важливе повідомлення – *assert:*. Його аргументом є блок. Якщо обчислення блока поверне *true*, то виконання продовжиться. У протилежному випадку виникне виняток *AssertionFailure*. Якщо його не перехоплено програмно, то в тому місці відкриється Налагоджувач. Повідомлення *assert:* особливо корисне для підтримки *проєктування за контрактом*[[1]](#footnote-1). Його найбільш типове використання – перевірити нетривіальні попередні умови для загальнодоступних методів об’єкта.

З його допомогою можна було б легко реалізувати метод *Stack >> pop*, наприклад, так:

Stack >> pop

"Повертає перший елемент і вилучає його зі стека."

self assert: [ self isNotEmpty ].

^ self linkedList removeFirst element

Це визначення тільки гіпотетичний приклад, у системі Pharo 9.0 використано інше.

Не плутайте *Object >> assert:* з *TestCase >> assert:*, яке використовують у системі модуль­ного тестування SUnit (див. розділ 12 «SUnit: модульне тестування у Pharo»). Перший з них приймає аргумент блок (насправді, він може приймати будь-який об’єкт, який розуміє *value*, у тім числі екземпляр *Boolean*), водночас другий розраховує на логічне значення. Хоча обидва корисні для налагодження, кожен з них вирішує зовсім інші завдання.

## Опрацювання винятків

Цей протокол містить кілька методів для повідомлення про виникнення помилок на етапі виконання.

### *doesNotUnderstand:*

Повідомлення *doesNotUnderstand:* (для його позначення в обговореннях зазвичай вико­ристовують абревіатуру *DNU* або *MNU*) надсилається щоразу, коли пошук методу зазнає невдачі. Реалізація за замовчуванням, тобто метод *Object >> doesNotUnderstand:*, відкриє в цьому місці Налагоджувача. Буває корисно перевантажити цей метод, щоб задати якусь альтернативну поведінку.

### *error*

Загальні методи *Object >> error* і *Object >> error:* можна використовувати для запуску винятків. Загалом краще запускати винятки, написані власноруч, щоб легше відрізняти помилки, які виникають у своєму коді, від винятків з класів ядра.

### *subclassResponsibility*

За домовленістю тілом абстрактного методу є вираз «*self subclassResponsibility*». Якщо випадково буде створено екземпляр абстрактного класу, то виклик абстрактного методу призведе до виконання *Object >> subclassResponsibility*.

Object >> subclassResponsibility

"Повідомлення налаштовує поведінку підкласів, повідомляючи, що вони

мали б реалізувати метод."

SubclassResponsibility signalFor: thisContext sender selector

Класичні приклади абстрактних класів – *Magnitude*, *Number* і *Boolean*. Їхній короткий огляд трохи згодом в цьому розділі.

### *shouldNotImplement*

Повідомлення *self shouldNotImplement* за домовленістю надсилають, щоб зазначити, що успадкований метод непритаманний підкласові. Загалом це ознака того, що не все гаразд з проєктом ієрархії класів. Однак через обмеження, які накладає просте наслі­дування, буває важко уникнути використання таких обхідних шляхів.

Типовим прикладом є метод *Collection >> remove:*, успадкований класом *Dictionary* і позначений в ньому як нереалізований. Замість нього словник підтримує метод *Dictionary >> removeKey:*.

### *deprecated:*

Надсилання «*self deprecated:*» сигналізує, що поточний метод не мали б більше викорис­товувати, бо він застарілий. Увімкнути чи вимкнути використання застарілих методів можна в розділі *Debugging* оглядача налаштувань. Аргумент повідомлення мав би пропонувати альтернативу. Знайдіть відправників повідомлення *deprecated:*, щоб побачити приклади (*Collection >> detectSum: aBlock* – один з них).

## Тестування

Методи тестування не мають нічого спільного з модульним тестуванням! Метод тестування дає змогу запитати про стан отримувача та повертає у відповідь логічне значення.

Численні методи тестування надає клас *Object*. Серед них *isArray, isBoolean, isBlock, isCollection* тощо. Зазвичай потрібно уникати таких методів, бо перевірка об’єкта на тип є формою порушення інкапсуляції. Їх часто використовують замість *isKindOf:*, проте обмеження в проєктуванні класів у них такі самі. Замість того, щоб тестувати об’єкт на клас, потрібно просто надіслати повідомлення і дозволити об’єкту вирішити, як його опрацювати.

Проте деякі з методів тестування, безсумнівно, корисні. Найкориснішими, ймовірно, є *ProtoObject >> isNil* і *Object >> notNil*. Шаблон проєктування Null Object може позбавити від необхідності використовувати навіть ці методи, але часто так зробити неможливо або неправильно.

Лістинг .. Загальний *initialize* – метод зачіпка

ProtoObject >> initialize

"Підкласи мали б перевизначити цей метод, щоб ініціалізувати створені

екземпляри"

Лістинг .. Метод *new* – шаблонний метод на стороні класу

Behavior >> new

"Повертає новий ініціалізований екземпляр отримувача (який є класом) без

індексованих змінних. Завершається невдачею, якщо клас індексований."

^ self basicNew initialize

## Ініціалізація

Завершальний важливий метод, але визначений не в *Object*, а в *ProtoObject* – *initialize*.

Причина, чому це важливо, полягає в тому, що у Pharo стандартний метод *new*, визначе­ний для кожного класу в системі, надсилатиме *initialize* новоствореним екземплярам.

Це означає, що достатньо перевантажити метод-зачіпку *initialize*, щоб екземпляри нових класів автоматично ініціалізувалися. Зазвичай метод *initialize* мав би виконувати *super initialize*, щоб визначити інваріант класу для будь-яких успадкованих змінних екземпляра.

## Числа

Числа в Pharo не примітивні значення даних, а справжні об’єкти. Звичайно, числа ефективно реалізовані у віртуальній машині, але ієрархія *Number* так само доступна і розширювана, як і будь-яка інша частина ієрархії класів.

Абстрактним коренем цієї ієрархії є клас *Magnitude*, який представляє всі види класів, що підтримують оператори порівняння. *Number* додає різні арифметичні й інші опера­тори здебільшого як абстрактні методи. *Float* і *Fraction* представляють, відповідно, числа з плаваючою комою і раціональні числа. Підкласи *Float* (*BoxedFloat64* і *Small­Float64*) представляють *Float* у певних архітектурах. Наприклад, *BoxedFloat64* доступний тільки для 64-розрядних систем. Клас *Integer* також абстрактний, об’єднує різні підкласи: *SmallInteger*, *LargePositiveInteger* і *LargeNegativeInteger*. Здебільшого користувачі можуть не турбуватися про відмінності між трьома класами цілих, бо значення за потреби перетворюються автоматично.

*Object*

*Magnitude*

*Number*

Fraction

ScaledDecimal

Float

*Integer*

BoxedFloat64

SmallFloat64

SmallInteger

*LargeInteger*

LargePositiveInteger

LargeNegativeInteger

Рис. .. Ієрархія класів чисел

## Magnitude

Клас *Magnitude* базовий не тільки для класів чисел, а й для інших класів, що підтриму­ють операції порівняння, таких як *Character*, *Duration*, *Timespan* тощо.

Методи < і = абстрактні. Решта операторів визначені в загальному випадку. Наприклад,

Magnitude >> < aMagnitude

"Відповідає, чи отримувач менший за аргумент."

^ self subclassResponsibility

Magnitude >> > aMagnitude

" Відповідає, чи отримувач більший за аргумент."

^ aMagnitude < self

## Можливості чисел

Подібно *Number* визначає +, –, \* і / як абстрактні, але всі інші арифметичні оператори визначені в загальному випадку.

Всі числа підтримують різні методи *перетворення*, як *asFloat* і *asInteger*. Існують також численні методи *швидкого створення*, які генерують екземпляри *Duration*, наприклад, *hour*, *day* і *week*.

Числа безпосередньо підтримують загальні математичні функції – *sin*, *log*, *raiseTo:*, *squared*, *sqrt* тощо.

Метод *Number >> printOn:* реалізовано в термінах абстрактного методу *Number >> print­On: base:*. (За замовчуванням для основи числення використовується значення 10).

Методи тестування – *even*, *odd*, *positive* і *negative*. *Number* закономірно перевантажує *isNumber*. Цікавіше, що визначено метод *isInfinite*, який повертає *false*.

Методи *усікання* охоплюють *floor*, *ceiling*, *integerPart*, *fractionPart* тощо.

1 + 2.5

>>> 3.5 "Додавання двох чисел"

3.4 \* 5

>>> 17.0 "Множення двох чисел"

8 / 2

>>> 4 "Ділення двох чисел"

10 - 8.3

>>> 1.7 "Віднімання двох чисел"

12 = 11

>>> false "Рівність двох чисел"

12 ~= 11

>>> true "Перевірка, чи числа відрізняються"

12 > 9

>>> true "Більше ніж"

12 >= 10

>>> true "Більше або дорівнює"

12 < 10

>>> false "Менше ніж"

100@10

>>> (100@10) "Створення точки (Point)"

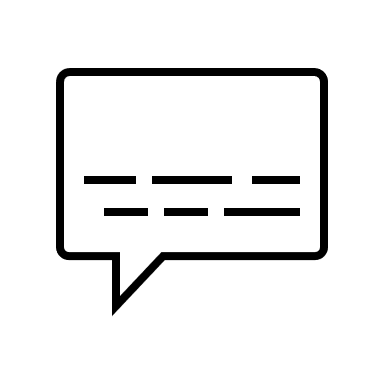
Наступний приклад на диво добре працює в Pharo:

1000 factorial / 999 factorial

>>> 1000

Зверніть увагу, що 1000! справді обчислюється, що в багатьох інших мовах може бути досить складно зробити. Це прекрасний приклад автоматичного приведення типу та точного опрацювання числа.

**Виконайте**  Спробуйте вивести результат обчислення *10000 factorial*. Потрібно більше часу для відображення, ніж для його обчислення!

*Від перекладача.* Сучасна реалізація Pharo напрочуд ефективна, а комп’ютери швидкі, тому відчути різницю можна тільки на справді великих числах. Щоб мати точніше уявлення про тривалість обчислень і перетворень, виконайте двічі, наприклад, такий фрагмент.

| start end |start := Time now."Transcript show: 10000 factorial printString."10000 factorial. "закоментуйте замість верхнього рядка"end := Time now.end asDuration - start asDuration "Print it"

## Дійсні

*Float* реалізує абстрактні методи класу *Number* для чисел з плаваючою комою.

Цікавіше, що клас *Float class* (тобто метаклас класу *Float*) надає методи для отримання таких констант: *е*, *infinity*, *nan* та *pi*.

Float pi

>>> 3.141592653589793

Float infinity

>>> Float infinity

Float infinity isInfinite

>>> true

## Раціональні

Екземпляри *Fraction* зберігають змінні для чисельника і знаменника, які мають бути цілими числами. Раціональні зазвичай створюють за допомогою ділення цілих (частіше ніж за допомогою методу класу *Fraction class >> numerator:denominator:*).

6/8

>>> (3/4)

(6/8) class

>>> Fraction

Множення раціонального числа на ціле або на інше раціональне може дати ціле число.

6/8 \* 4

>>> 3

## Цілі

*Integer* – абстрактний базовий клас для трьох конкретних реалізацій цілих чисел. Крім конкретної реалізації багатьох абстрактних методів класу *Number*, він також додає кілька методів, специфічних для цілих чисел, таких як *factorial*, *atRandom*, *IsPrime*, *gcd:* та багато інших.

*SmallInteger* особливий тим, що його екземпляри представлені в пам’яті компактно: замість того, щоб зберігати посилання, число кодується безпосередньо, в бітах, які в іншому випадку використовували б для зберігання посилання. Перший біт посилання на об’єкт інформує, чи він *SmallInteger*, чи ні. Віртуальна машина приховує це від користувача, тому його не можна побачити під час інспектування об’єкта.

Методи класу *minVal* і *maxVal* повідомляють діапазон значень *SmallInteger*. Зауважимо, що він залежить від розрядності образу системи і може бути або *(2 raisedTo: 30) – 1* для 32-розрядної архітектури, або *(2 raisedTo: 60) – 1* для 64-розрядної.

SmallInteger maxVal = ((2 raisedTo: 60) - 1)

>>> true

SmallInteger minVal = (2 raisedTo: 60) negated

>>> true

Коли значення *SmallInteger* виходить з цього діапазону, він автоматично перетво­рюється на *LargePositiveInteger* або *LargeNegativeInteger*, відповідно до потреби.

(SmallInteger maxVal + 1) class

>>> LargePositiveInteger

(SmallInteger minVal - 1) class

>>> LargeNegativeInteger

Так само великі цілі числа у разі потреби перетворюються назад в малі.

Як і в більшості мов програмування, цілі числа можуть бути корисні для задання повто­рень. Визначено спеціальний метод *timesRepeat:* для багаторазового виконання блока. Ми вже бачили подібний приклад в розділі 8 «Синтаксис у двох словах».

| n |

n := 2.

3 timesRepeat: [ n := n \* n ].

n

>>> 256

## Літери

*Character* визначено підкласом *Magnitude*. Друковані символи записують у Pharo у вигляді *$<літера>*, наприклад, *$a*, *$b*, *$5*, *$P*, *$+*.

Недруковані символи можна генерувати різними методами класу. *Character class >> va­lue:* приймає цілочислове значення Unicode (або ASCII) як аргумент і повертає відповід­ну літеру. Протокол «*accessing untypeable characters*» містить багато зручних методів конструювання: *arrowRight*, *backspace*, *cr*, *escape*, *space*, *tab* тощо.

Character space = (Character value: Character space asciiValue)

>>> true

Метод *printOn:* досить розумний, щоб знати, який з трьох способів генерування літер підходить найкраще:

Character value: 1

>>> Character home

Character value: 2

>>> Character value: 2

Character value: 32

>>> Character space

Character value: 97

>>> $a

Вбудовано різні зручні методи тестування: *isAlphaNumeric*, *isCharacter*, *isDigit*, *isLowercase*, *isVowel* тощо.

Щоб перетворити літеру на рядок, який містить тільки її, надсилають повідомлення *asString*. У цьому випадку *asString* і *printString* дають різні результати.

$a asString

>>> 'a'

$a

>>> $a

$a printString

>>> '$a'

Як і *SmallInteger*, екземпляр *Character* є безпосереднім значенням, а не посиланням на об’єкт. У більшості випадків ви не побачите ніякої різниці і зможете використовувати об’єкти класу *Character*, як і будь-які інші. Але це означає, що символи з однаковими значеннями завжди ідентичні.

(Character value: 97) == $a

>>> true

*Object*

*Collection*

*SequenceableCollection*

*ArrayedCollection*

*String*

Array

Text

Symbol

ByteString

Рис. .. Ієрархія класу *String*

## Рядки

*String* – це індексована *Collection*, яка містить тільки *Character*.

Насправді клас *String* абстрактний, а рядки Pharo є екземплярами конкретного класу *ByteString*.

'hello world' class

>>> ByteString

Інший важливий підклас *String* – *Symbol*. Головна відмінність в тому, що існує тільки один екземпляр *Symbol* з заданим значенням. Це іноді називають *властивістю уні­кального екземпляра*. Навпаки, два окремо побудовані рядки, які містять ту саму послідовність літер, часто будуть різними об’єктами.

'hel','lo' == 'hello'

>>> false

('hel','lo') asSymbol == #hello

>>> true

Інша важлива відмінність полягає в тому, що вміст екземпляра *String* можна змінювати, а екземпляр *Symbol* – незмінний. Зауважимо також, що в Pharo 9.0 літерали рядків ста­ли незмінними. Це добре, бо літерал може залучатися до виконання кількох методів, і зміна в одному з них могла б спричинити проблеми в іншому.

(String fromString: 'hello') at: 2 put: $u; yourself

>>> 'hullo'

#hello at: 2 put: $u

>>> Error: symbols can not be modified.

Про незмінність легко забути, бо, оскільки рядки є колекціями, то вони розуміють ті самі повідомлення, що й інші колекції.

#hello indexOf: $o

>>> 5

Хоча *String* не наслідує *Magnitude*, він підтримує звичайні методи порівняння <, = тощо. Крім того, *String >> match:* корисний для деяких базових шаблонів зіставлення в стилі glob[[2]](#footnote-2).

'\*or\*' match: 'zorro'

>>> true

*String* підтримує досить значну кількість методів перетворення. Багато з них є метода­ми конструювання екземплярів інших класів: *asDate*, *asInteger* тощо. Є також багато корисних методів для перетворення рядка в інший рядок, наприклад, *capitalized* і *translateToLowercase*.

'256-th day of the year' asInteger

>>> 256

'hello, world!' capitalized

>>> 'Hello, world!'

Додаткові відомості про рядки та колекції в наступному розділі.

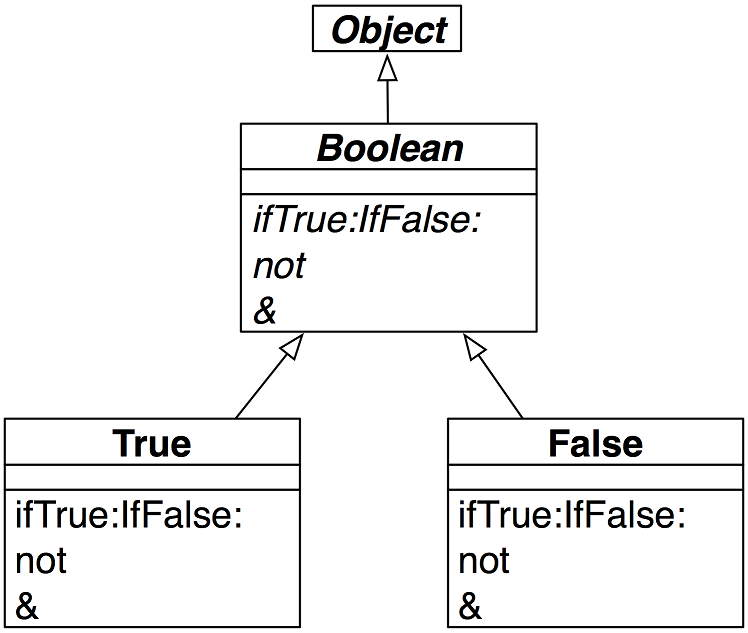


Рис. .. Ієрархія класу *Boolean*

## Булеві величини

Клас *Boolean* пропонує чудову нагоду довідатися, скільки мови Pharo було перенесено в бібліотеку класів. *Boolean* – це абстрактний надклас одноелементних класів *True* і *False*.

Більшу частину поведінки булевих величин можна зрозуміти, розглянувши метод *ifTrue:ifFalse:*, який приймає два блоки як аргументи.

4 factorial > 20

ifTrue: [ 'bigger' ]

ifFalse: [ 'smaller' ]

>>> 'bigger'

Метод *ifTrue:ifFalse:* у класі *Boolean* абстрактний. Його реалізації в обох підкласах дуже прості.

True >> ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock

^ trueAlternativeBlock value

False >> ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock

^ falseAlternativeBlock value

Кожен з них виконує правильний блок залежно від отримувача повідомлення. Фактично, це сама суть ООП: коли повідомлення надсилають об’єкту, сам об’єкт визначає, який метод використати для відповіді. В цьому випадку екземпляр *True* виконує *істинну* альтернативу, а екземпляр *False* – *помилкову*. Всі абстрактні методи класу *Boolean* реалізовані в *True* і *False* за таким самим принципом. Наприклад, так реалізовано заперечення (повідомлення *not*).

True >> not

"Заперечення--відповідає false, бо отримувач true."

^ false

False >> not

" Заперечення--відповідає true, бо отримувач false."

^ true

*Boolean* пропонує ще кілька часто вживаних методів для організації галужень: *ifTrue:*, *ifFalse:* й *ifFalse:ifTrue:*. Також можна вибирати між ретельною і лінивою версіями обчислення кон’юнкції та диз’юнкції.

У першому прикладі будуть обчислені обидва логічних підвирази, оскільки & приймає булеву величину. Хоча очевидно, що *(1 > 2)* повертає *false*, і немає потреби перевіряти *(3 < 4)*, однаково метод & обчислює свій аргумент.

( 1 > 2 ) & ( 3 < 4 )

>>> false "Ретельні обчислення. Буде виконано і отримувача, і аргумент"

У другому і третьому прикладах виконується тільки вираз-отримувач. Він повертає *false*, тому аргумент-блок виконано не буде. Зауважте, що аргументом повідомлення *and:* має бути блок. У третьому прикладі блок *[1 / 0]* не виконується і не генерує виняток, бо метод *and:* виконує свій аргумент тільки, якщо отримувачем є *true*.

( 1 > 2 ) and: [ 3 < 4 ]

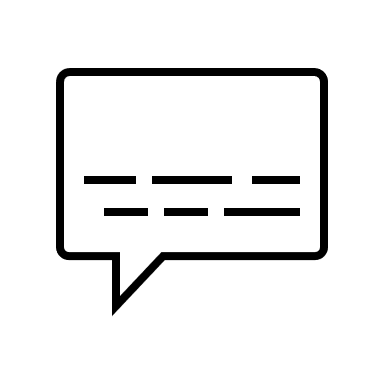
>>> false "Ліниві обчислення, виконано тільки отримувача"

( 1 > 2 ) and: [ 1 / 0 ]

>>> false "Аргумент не виконується, тому немає винятку"

Лінивий метод *or:* демонструє схожу поведінку. Він виконує свій аргумент тільки тоді, коли отримувач *false*.

Спробуйте уявити, як реалізовані *and:* і *or:*. Перевірте реалізації в *Boolean*, *True* і *False*.

*Від перекладача.* Підійміть руку, хто виявив, що *True>>or:* і *True>>|* реалізовані однаково. То чому ж тоді перший з них забезпечує ліниві обчислення, а другий – ретельні?

Знайдіть пояснення цьому феномену.

*Підказка.* Поміркуйте про пріоритети повідомлень різних видів.

## Підсумки розділу

* Якщо ви перевантажили =, то повинні перевантажити також *hash*.
* Перевизначайте *postCopy* для правильної реалізації копіювання ваших об’єктів.
* Використовуйте «*self halt*», щоб задати точку переривання.
* Повертайте «*self subclassResponsibility*», щоб оголосити метод абстрактним.
* Перевантажте *printOn:*, щоб надати об’єкту рядкове зображення.
* Перевантажте метод-зачіпку *initialize* для правильної ініціалізації екземплярів.
* Методи класу *Number* автоматично підлаштовуються під дійсні, цілі та раціо­нальні числа.
* Екземпляри класу *Fraction* представляють раціональні числа, а не дійсні.
* Усі літери, екземпляри класу *Character*, можна трактувати як унікальні.
* Рядки, екземпляри класу *String*, змінювані, символи (*Symbol*) – ні. Однак не пробуйте змінювати рядкові літерали!
* Символи унікальні, рядки – ні.
* Рядки та символи є колекціями, тому підтримують звичайні методи *Collection*.
* Методи класу *Boolean* і його підкласів – ключ до розуміння того, як влаштовано і як функціонує Pharo.

1. [Проєктування за контрактом – Вікіпедія (wikipedia.org)](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%94%D0%BA%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%B7%D0%B0_%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BE%D0%BC) [↑](#footnote-ref-1)
2. [glob (programming) – Wikipedia](https://en.wikipedia.org/wiki/Glob_(programming)) [↑](#footnote-ref-2)