Розділ 10
Об’єктна модель Pharo

Модель програмування Pharo розроблена під значним впливом об’єктної моделі Smalltalk. Вона проста й однорідна: все є об’єктом і об’єкти взаємодіють тільки через надсилання повідомлень одне одному. Змінна екземпляра приватна для об’єкта. Всі методи доступні, зв’язуються на етапі виконання (пізнє зв’язування застосовується завжди).

У розділі описано основоположні концепції об’єктної моделі Pharo. Параграфи розділу впорядковано так, щоб спочатку обговорити найважливіше. Ще раз пояснено поняття *self* і *super* і точно визначено їхню семантику. Потім з’ясовано наслідки того, що класи також є об’єктами. Детальніше про них йтиметься в розділі 17 «Класи та метакласи».

10.1. Правила базової моделі

Об’єктна модель заснована на наборі простих правил, які діють завжди й усюди без жодних винятків. Ось ці правила.

Правило 1. Кожна сутність є об’єктом.

Правило 2. Кожен об’єкт є екземпляром якогось класу.

Правило 3. У кожного класу є надклас.

Правило 4. Усе відбувається через надсилання повідомлень.

Правило 5. Алгоритм відшукання методу перебирає ланцюжок наслідування.

Правило 6. Класи також є об’єктами і діють за тими ж правилами.

Давайте розглянемо кожне з цих правил детальніше.

10.2. Все є об’єктом

Мантра «*все є об’єктом*» дуже заразна. Незадовго після початку роботи з Pharo ви здивуєтеся, як це правило спрощує все, що ви робите. Цілі числа, наприклад, також справжні об’єкти, тому їм можна надсилати повідомлення так само, як усім іншим.

Розглянемо два приклади.

```
"повідомлення + 4, надіслане до 3, поверне 7"
3 + 4
>>> 7
```

```
"повідомлення factorial, надіслане до 20, поверне велике число"
20 factorial
>>> 2432902008176640000
```

Об’єкт *7* відрізняється від результату обчислення виразу «*20 factorial*»: 7 – екземпляр класу *SmallInteger*, а *20 factorial* – класу *LargePositiveInteger*. Але вони обидва поліморфні об’єкти, бо вміють відповідати на однаковий набір повідомлень, тому ніякий код, навіть реалізація *factorial*, не повинні знати, що вони різні.

Мабуть один з найбільш значущих наслідків правила *все є об’єктом* – це класи також об’єкти! Класи є об’єктами першого класу, не другого, тому можна надсилати їм повідомлення, інспектувати і змінювати їх, як звичайні об’єкти.

З погляду надсилання повідомлень немає різниці між екземпляром класу і класом. З прикладу бачимо, що можна надіслати повідомлення *today* класові *Date*, щоб отримати від операційної системи поточну дату.

```
Date today printString
>>> '24 July 2022'
```

Можна запитати клас про змінні його екземплярів, як у прикладі нижче. Зверніть увагу, що повідомлення *allInstVarNames* повертає всі змінні екземпляра з успадкованими включно.

```
Date allInstVarNames
>>> #(#start #duration)
```

Важливо Класи – це також об’єкти. З класами взаємодіють так само, як з іншими об’єктами: за допомогою надсилання повідомлень.

10.3. Кожен об’єкт є екземпляром класу

Кожен об’єкт має свій клас. Можна легко довідатись який, надіславши йому повідомлення *class*.

```
1 class
>>> SmallInteger
```

```
20 factorial class
>>> LargePositiveInteger
```

```
'hello' class
>>> ByteString
```

```
(4@5) class
>>> Point
```

```
Object new class
>>> Object
```

Клас визначає *структуру* своїх екземплярів оголошенням змінних екземпляра та їхню поведінку визначенням методів. Кожен метод має назву, яку називають *селектором*, унікальну в межах класу.

З правил *класи є об’єктами* і *кожен об’єкт є екземпляром якогось класу* випливає, що клас також має бути екземпляром якогось класу. Клас, екземпляр якого є класом,

називають *метакласом*. Щоразу, коли створюють клас, система автоматично створює метаклас. Метаклас визначає структуру і поведінку класу, який є його екземпляром. 99% часу вам не доведеться думати про метакласи, тому можете з успіхом їх ігнорувати. Детальніше розглянемо метакласи в розділі 17 «Класи і метакласи».

10.4. Структура та поведінка екземпляра

Розглянемо коротко, як визначити структуру та поведінку екземплярів класу.

Змінні екземпляра

До змінної екземпляра можна звернутися за іменем у будь-якому методі екземпляра в межах класу, в якому вона оголошена, а також в методах екземпляра підкласів. Це означає, що змінні екземпляра Pharo подібні на *захищені* (protected) змінні класів C++ і Java. Проте ми вважатимемо, що змінні екземпляра приватні, бо пряме звертання до змінної з методу підкласу вважається у Pharo поганим стилем програмування.

Інкапсуляція, заснована на екземплярі

Поля екземпляра в Pharo приватні для *екземпляра*. Це відрізняє Pharo від Java і C++, у яких дозволено доступ до змінних екземпляра (які ще називають *полями* або *змінними-членами*) будь-якому іншому об'єкту, який просто виявився екземпляром того самого класу. Можна сказати, що межами інкапсуляції об'єктів у Java і C++ є клас, а в Pharo – екземпляр.

У Pharo два екземпляри одного класу не можуть отримати доступ до змінних екземплярів один одного, якщо клас не визначає методи доступу. Мова не має такого синтаксису, який би надавав прямий доступ до полів екземпляра будь-якого об'єкта. Насправді, існує механізм, який називають рефлексією, що дає змогу запитати інший об'єкт про значення його змінних. Рефлексія є основою метапрограмування, призначеного для написання інструментів програмування таких, наприклад, як інспектор об'єктів.

Лістинг 10.1. Обчислення відстані між двома точками

```
Point >> distanceTo: aPoint
  "Поверне відстань між aPoint і отримувачем."

  | dx dy |
  dx := aPoint x - x.
  dy := aPoint y - y
  ^ ((dx * dx) + (dy * dy)) sqrt
```

Приклад інкапсуляції екземпляра

Метод *distanceTo*: класу *Point* обчислює відстань між отримувачем й іншою точкою (див. лістинг 10.1). До змінних *x* і *y* отримувача доступаються напряму в тілі методу. До змінних іншої точки, аргументу можна доступитися тільки через повідомлення *x* і *y*.

```
1@1 distanceTo: 4@5
>>> 5
```

Головна перевага інкапсуляції, заснованої на екземплярі, над інкапсуляцією класу в тому, що вона дає змогу співіснувати різним реалізаціям тої самої абстракції.

Наприклад, метод *distanceTo*: може не знати і не турбуватися, чи аргумент *aPoint* є екземпляром того самого класу, що й приймач.

Об’єкт-аргумент може бути заданий у полярних координатах, бути записом в базі даних, або існувати на іншому комп’ютері в розподіленій системі – незалежно від цього, доки він відповідатиме на повідомлення *x* і *y*, код методу *distanceTo*: працюватиме.

Методи

Усі методи *відкриті* та *віртуальні* (пошук методу відбувається на етапі виконання). У Pharo немає статичних методів. Методи мають доступ до всіх змінних екземпляра об’єкта. Деякі розробники вважають, що для доступу до змінних екземпляра потрібно використовувати тільки методи доступу. Така практика має право на існування, але вона засмічує інтерфейс класу, ба більше, надає доступ до приватного стану об’єкта.

Для легшої орієнтації в класі методи групують в *протоколи*, які позначають їхнє призначення. З погляду мови програмування протоколи не мають семантичного навантаження, це просто папки для зберігання методів. Кілька загальних назв протоколів уже вважають стандартними: *accessing* для всіх методів доступу, *initialization* для методів налаштування належного початкового стану об’єкта.

Протокол *private* інколи використовують для групування методів, які не мали б викликати ззовні. Правду кажучи, ніщо не забороняє вам надсилати повідомлення, реалізоване таким «приватним» методом, але «приватність» означає, що розробник завжди може змінити або вилучити такий метод.

10.5. Кожен клас має надклас

Кожен клас у Pharo наслідує поведінку й опис структури з якогось єдиного *надкласу*. Це означає, що Pharo підтримує просте наслідування.

З прикладів видно, як можна дослідити ієрархію наслідування.

```
SmallInteger superclass
>>> Integer
```

```
Integer superclass
>>> Number
```

```
Number superclass
>>> Magnitude
```

```
Magnitude superclass
>>> Object
```

```
Object superclass
>>> ProtoObject
```

```
ProtoObject superclass
>>> nil
```

За традицією коренем дерева класів є клас *Object*, бо все є об'єктом. Більшість класів наслідують *Object*, який визначає багато додаткових повідомлень, які майже всі об'єкти розуміють і відповідають на них.

Лістинг 10.2. Визначення класу *Point*

```
Object subclass: #Point
   instanceVariableNames: 'x y'
   classVariableNames: ''
   package: 'Kernel-BasicObjects'
```

У Pharo, насправді, кореневим класом є *ProtoObject*, але ви зазвичай не будете звертати на нього ніякої уваги. *ProtoObject* інкапсулює мінімальний набір повідомлень, що мали б бути в кожного об'єкта. Також *ProtoObject* влаштовано так, щоб запускати всі винятки, які тільки можна (для підтримки патерну проєктування посередник). Під час створення класів для своєї програми треба наслідувати клас *Object* (або його підкласи), хіба що виникнуть дуже вагомі причини зробити інакше.

Новий клас зазвичай створюють за допомогою повідомлення «*subclass:instanceVariableNames:...*» наявному класові (див. лістинг 10.2). Для створення класів є ще кілька інших методів. Щоб побачити, які саме, подивіться на клас *Class* і його протокол *subclass creation*.

10.6. Усе відбувається через надсилання повідомлень

Це правило охоплює саму суть програмування у Pharo.

У процедурному програмуванні яку процедуру виконати вибирає той, хто її викликає. Прив'язування адреси виклику відбувається *статично* на етапі компіляції за іменем процедури. Подібні механізми діють і в деяких об'єктно-орієнтованих мовах, наприклад, стосовно виклику статичного методу в Java або звичайного (не віртуального) методу в C++: кого викликати, вирішує користувач, прив'язування відбувається на етапі компіляції, і ніякі алгоритми пошуку методу чи динамічного виклику на етапі виконання не діють.

Автор повідомлення не вирішує, який метод буде виконано, він тільки *просить* об'єкт зробити щось, надіславши йому *повідомлення*. Повідомлення це ніщо інше, як назва і список аргументів. *Отримувач* повідомлення сам вирішує, як реагувати, вибравши певний свій *метод*, щоб виконати те, про що його попросили. Оскільки різні об'єкти можуть мати різні методи, щоб відповідати на однакове повідомлення, то метод для виклику має вибиратися динамічно, коли повідомлення отримано.

Як наслідок, можна надсилати *одне повідомлення* різним об'єктам, кожен з яких має *власний метод* для відповіді на нього.

```
"надсилання повідомлення + з аргументом 4 цілому числу 3"
3 + 4
>>> 7
" надсилання повідомлення + з аргументом 4 точці (1@2)"
(1@2) + 4
>>> 5@6
```

У наведених прикладах не ми вирішуємо, як *SmallInteger 3* чи *Point (1@2)* відповідатимуть на повідомлення *+ 4*. Ми покладаємось на вибір об’єктів: кожен з них має власний метод для повідомлення *+* і належно відповість на *+ 4*.

Термінологія

У Pharo *не кажуть* «викликати метод» натомість *надсилають повідомлення*. Здається, невелика відмінність у термінології, але вона має важливе значення, бо змінює межі відповідальності. Це означає, що не користувач класу вибирає, який метод виконати, а отримувач знаходить відповідний метод, щоб опрацювати повідомлення.

Інші обчислення

Майже все у Фаро відбувається через надсилання повідомлень. Але не все. У якийсь момент мають відбуватися й інші дії.

- *Оголошення змінних* не виконується надсиланням повідомлень. Насправді, воно навіть не є виконуваним виразом. Оголошення змінної просто спричиняє виділення пам’яті для неї за адресою посилання на об’єкт.
- *Доступ до змінної* – це просто доступ до значення змінної.
- *Присвоєння* не надсилає повідомлень. Присвоєння змінній призводить до оновлення її значення: з іменем змінної буде пов’язано результат обчислення виразу в правій частині присвоєння.
- *Повернення* (^) не надсилає повідомлень. Воно просто повертає обчислений результат відправнику.
- *Прагми* не є надсиланнями повідомлень. Це анотації методів.

Усе, крім цих кількох винятків, тобто, майже все решта справді відбувається через надсилання повідомлень.

Про об’єктно-орієнтоване програмування

Одним з наслідків такої моделі надсилання повідомлень у Pharo є те, що вона заохочує стиль програмування, за якого об’єкти мають маленькі методи і делегують завдання іншим об’єктам, замість того, щоб реалізувати великі процедурні методи, що беруть на себе забагато відповідальності.

Джозеф Перлайн (Joseph Pelrine) висловлює цей принцип так.

> **Важливо** «*Не роби нічого, що ти можеш передати комусь іншому*».

Багато об’єктно-орієнтованих мов підтримують і статичні, і динамічні виклики методів. У Pharo є тільки динамічний спосіб надсилання повідомлень. Наприклад, замість того, щоб підтримувати статичні операції з класами, ми просто надсилаємо повідомлення класам, які є звичайними об’єктами.

Зокрема, оскільки в Pharo немає *відкритих полів*, то єдиний спосіб оновити значення змінної екземпляра іншого об’єкта – це надіслати повідомлення з запитом до об’єкта, щоб він змінив своє власне поле. Звісно, оголошення методів читання і запису всіх змінних екземпляра не є добрим прикладом об’єктно-орієнтованого стилю, бо надає клієнтові доступ до внутрішнього стану об’єкта. Джозеф Перлайн висловив це дуже вдало.

> **Важливо** «*Не дозволяй нікому гратися з твоїми даними*».

10.7. Надсилання повідомлення – двокроковий процес

Що насправді відбувається, коли об'єкт отримує повідомлення?

Це двоетапний процес: *пошук методу* і *виконання методу*.

- **Пошук.** Спочатку знаходиться метод, який має таку саму назву, як і повідомлення.
- **Виконання.** Потім знайдений метод застосовується до отримувача з аргументами повідомлення. Коли метод знайдено, аргументи зв'язуються з параметрами методу, і віртуальна машина виконує його.

Пошук методу досить простий.

1. Клас отримувача повідомлення шукає метод, щоб опрацювати повідомлення.
2. Якщо клас отримувача не має такого методу, то він запитує свій надклас і так далі по ієрархії класів.

Все справді так просто, як написано вище. Але є ще кілька запитань, з якими треба уважно розібратися.

- *Що відбувається, якщо метод явно не повертає значення?*
- *Що стається, коли клас перевантажує метод свого надкласу?*
- *Яка різниця між надсиланням до self і super?*
- *Що стається, коли потрібного методу не знайдено?*

Згадані вище правила пошуку методу концептуальні. Розробники віртуальних машин використовують всі можливі хитрощі й оптимізації, щоб пришвидшити пошук методів.

Спочатку розглянемо базову стратегію пошуку, а потім перейдемо до решти питань.

Лістинг 10.3. Реалізований у класі метод

```
EllipseMorph >> defaultColor

   "Answer the default color/fill style for the receiver"
   ^ Color yellow
```

Лістинг 10.4. Успадкований метод

```
Morph >> openInWorld

   "Add this morph to the world."
   self openInWorld: self currentWorld
```

10.8. Алгоритм пошуку методу перебирає ланцюжок наслідування

Припустимо, що ми створили екземпляр класу *EllipseMorph*.

```
anEllipse := EllipseMorph new.
```

Якщо ми зараз надішлемо йому повідомлення *defaultColor*, то отримаємо відповідь *Color yellow*.

```
anEllipse defaultColor
>>> Color yellow
```

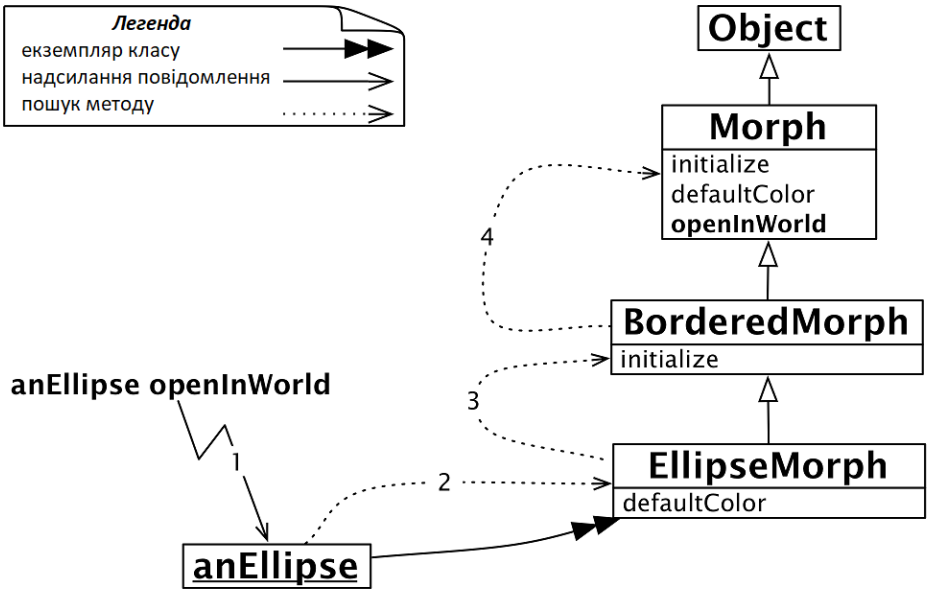

Рис. 10.1. Пошук методу в ланцюжку наслідування

Клас *EllipseMorph* реалізує метод *defaultColor*, тому потрібний метод знайдено тут же (див. лістинг 10.3).

На противагу цьому, якщо ми надішлемо до *anEllipse* повідомлення *openInWorld*, то метод знайдеться не одразу, бо клас не реалізовує *openInWorld*. Пошук продовжиться в надкласі *BorderedMorph* і вище по ієрархії, доки метод *openInWorld* не буде знайдено в класі *Morph* (див. рис. 10.1).

Лістинг 10.5. Ще один визначений в класі метод

```
EllipseMorph >> closestPointTo: aPoint
  ^ self intersectionWithLineSegmentFromCenterTo: aPoint
```

10.9. Виконання методу

Пригадаємо, що опрацювання повідомлень двоетапний процес.

- **Пошук.** Спочатку знаходиться метод, який має таку саму назву, як і повідомлення.
- **Виконання.** Потім знайдений метод застосовується до отримувача з аргументами повідомлення. Коли метод знайдено, аргументи зв’язуються з параметрами методу, і віртуальна машина виконує його.

Пояснимо, як відбувається другий етап – виконання методу.

Коли потрібний метод знайдено, псевдозмінна *self* у його тілі пов’язується з отримувачем повідомлення, а параметри методу – з аргументами повідомлення. Тоді система виконує тіло методу. Так відбувається щоразу, коли знайдено метод, який потрібно виконати. Уявімо, що ми надіслали повідомлення *EllipseMorph new closestPointTo: 100@100*, і знайдено метод, зображений на лістингу 10.5.

Змінна *self* вказуватиме на щойно створений еліпс, параметр *aPoint* – на точку *100@100*.

Такі ж прив’язування відбуваються і в тому випадку, коли метод для виконання знайдено в надкласі. Коли надсилають повідомлення *EllipseMorph new openInWorld*, метод *openInWorld* знаходиться в класі *Morph*, але змінна *self* все одно пов’язується з новоствореним еліпсом. Тому кажуть, що *self* завжди представляє отримувача повідомлення, незалежно від класу, в якому знайдено метод для виконання.

Отож є два різні етапи опрацювання повідомлення: пошук відповідного методу в ієрархії класів і виконання знайденого методу з отримувачем повідомлення.

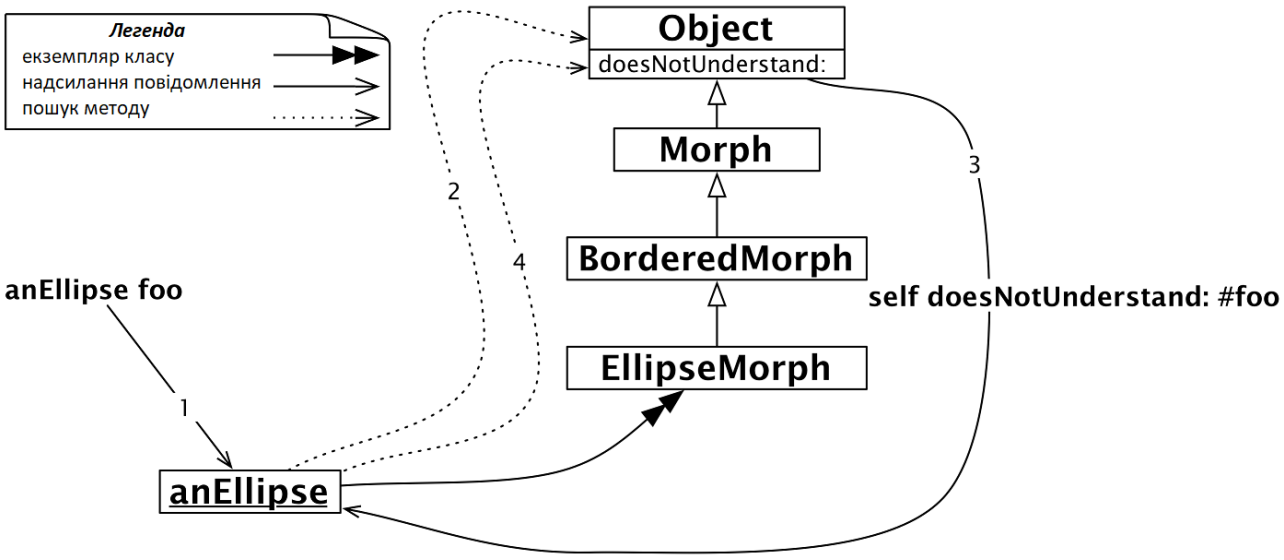

Рис. 10.2. Еліпс не зрозумів повідомлення *foo*

10.10. Об’єкт не зрозумів повідомлення

Що відбудеться, якщо шуканий метод не буде знайдено?

Припустимо, вже знайомому еліпсу надіслано повідомлення *foo*. Спочатку звичний пошук методу пройде по всій ієрархії аж до класу *Object* (або, навіть, до *ProtoObject*). Коли методу *foo* не буде знайдено, віртуальна машина змусить еліпс надіслати повідомлення *self doesNotUnderstand: #foo* (див. рис. 10.2).

Це звичне надсилання повідомлення до *self*, тому пошук почнеться знову з класу *EllipseMorph*, але цього разу – методу *doesNotUnderstand:*. Виявляється, *Object* реалізує метод *doesNotUnderstand:*. Цей метод створить екземпляр класу *MessageNotUnderstood*, який запускає налагоджувач у поточному контексті виконання.

Чому використано такий заплутаний шлях, щоб опрацювати таку просту помилку?

Він надає розробникам можливість легко перехоплювати такі помилки і виконувати альтернативні дії. Можна перевантажити метод *Object>>doesNotUnderstand* у будь-якому підкласі і реалізувати інший спосіб опрацювання помилок.

Насправді, це досить простий спосіб реалізувати автоматичне делегування повідомлень одного об’єкта іншому. Делегуючий об’єкт може перенаправляти всі свої незрозумілі повідомлення до іншого об’єкта, відповідальність якого опрацювати їх або самому згенерувати помилку!

10.11. Про повернення *self*

Зауважте, що метод *defaultColor* класу *EllipseMorph* явно повертає *Color yellow*, тоді як метод *openInWorld* класу *Morph* не повертає нічого.

Насправді метод у відповідь на повідомлення *завжди* повертає значення, яке звісно є об’єктом. Відповідь можна визначити за допомогою оператора ^ в тілі методу, але якщо виконання методу завершилось, без виконання ^, метод все одно поверне значення – той об’єкт, що отримав повідомлення. Ми зазвичай говоримо, що метод *повертає self*, тому що в Pharo псевдозмінна *self* представляє отримувача повідомлення, подібно до ключового слова *this* у Java. Інші мови, такі як Ruby, за замовчуванням повертають значення останнього виразу в методі. У Pharo це не так, натомість можна уявляти, що метод без явного повернення результату закінчується виразом «^ *self*».

Важливо *self* завжди представляє отримувача повідомлення.

Це означає, що *Morph>>openInWorld* з лістингу 10.4 еквівалентний визначеному нижче методу *openInWorldReturnSelf*.

```
Morph >> openInWorldReturnSelf
   "Add this morph to the world."
   self openInWorld: self currentWorld
   ^ self
```

Чому явне написання «^ *self*» – не те, що варто робити?

Коли ви повертаєте щось явно, то наголошуєте, що повертаєте, щось корисне відправнику. Коли ви явно повертаєте *self*, то висловлюєте сподівання, що відправник використає це значення. Але не у випадку *Morph>>openInWorld*, тому краще не повертати явно *self*. Об’єкт *self* повертають тільки в окремих випадках, щоб наголосити, що метод повертає отримувача повідомлення.

Кент Бек (Kent Beck) назвав цю загальну ідіому Pharo *поверненням цікавого значення*: «*Повертай значення тільки тоді, коли хочеш, щоб відправник повідомлення використав його*».

Важливо За замовчуванням, якщо не визначено іншого, метод повертає отримувача повідомлення.

10.12. Перевантаження та розширення

Погляньмо ще раз на ієрархію класу *EllipseMorph*, зображену на рис. 10.1. Видно, що обидва класи *Morph* та *EllipseMorph* реалізують метод *defaultColor*. Справді, якщо відкрити нову морфу (*Morph new openInWorld*), то отримаємо прямокутник синього кольору, водночас еліпс за замовчуванням буде жовтим.

Кажуть, що *EllipseMorph перевантажує* наслідуваний від *Morph* метод *defaultColor*. З погляду *anEllipse* успадкований метод більше не існує.

Інколи потрібно не стільки перевантажити наслідувані методи, скільки *розширити* – доповнити їх новою функціональністю. Треба мати можливість викликати перевантажений метод *додатково* до нової визначеної в підкласі функціональності. У Pharo, як і в багатьох інших об’єктно-орієнтованих мовах, що підтримують просте наслідування, це можна виконати за допомогою повідомлення до *super*.

Часте застосування цього механізму можна бачити в методі *initialize*. Щоразу, коли ініціалізують новий екземпляр класу, дуже важливо ініціалізувати всі успадковані змінні екземпляра. Проте, вміння робити це правильно вже є в методах ініціалізації кожного з надкласів у ланцюжку наслідування. Підклас не має навіть пробувати ініціалізувати успадковані змінні екземпляра!

Отже, доброю практикою є надсилати *super initialize* в методі ініціалізації перед тим, як проводити хоч якусь ініціалізацію, як записано в методі нижче.

```
BorderedMorph >> initialize
  "Ініціалізувати стан отримувача"
  super initialize.
  self borderInitialize
```

Надсилання повідомлення до *super* потрібне, щоб компонувати наслідувану поведінку з власною, яка в іншому випадку була б перевантажена.

Важливо Доброю практикою є надсилати «*super initialize*» на початку методу ініціалізації.

10.13. Надсилання до *self* і *super*

Псевдозмінна *self* представляє отримувача повідомлення і пошук методу розпочинається з класу отримувача. Поміркуйте, що таке *super*? Псевдозмінна *super* – це *не* надклас! Якщо ви так подумали, то допустили звичайну помилку, яку всі допускають. Також помилково думати, що пошук методу розпочинається з надкласу класу отримувача повідомлення.

Важливо *self* представляє отримувача повідомлення, і пошук методу розпочинається в класі отримувача.

Як надсилання повідомлення до *self* відрізняється від надсилання до *super*?

Як і *self*, *super* представляє отримувача повідомлення. Так, ви все правильно прочитали! Змінився тільки спосіб пошуку методу. Замість того, щоб розпочати пошук в класі отримувача, він розпочинається в *надкласі того класу, де визначений метод, у тілі якого трапилося надсилання до super*.

Важливо *super* представляє отримувача повідомлення, і пошук методу розпочинається в надкласі того класу, в якому визначений метод, у тілі якого трапилося надсилання до *super*.

Лістинг 10.6. Надсилання повідомлення об'єктові *self*

```
Morph >> fullPrintOn: aStream
  aStream nextPutAll: self class name, ' new'
```

Лістинг 10.7. Інше надсилання повідомлення об'єктові *self*

```
Morph >> constructorString
  ^ String streamContents: [ :s | self fullPrintOn: s ]
```

Лістинг 10.8. Комбінування надсилання до *self* і *super*

```
BorderedMorph >> fullPrintOn: aStream
  aStream nextPutAll: '('.
  super fullPrintOn: aStream.
  aStream
     nextPutAll: ') setBorderWidth: ';
     print: borderWidth;
     nextPutAll: ' borderColor: ', (self colorString: borderColor)
```

Пояснимо докладно на прикладі, як це працює. Уявіть, що визначено три методи, зображені на лістингах 10.6–10.8.

Спочатку в лістингу 10.6 у класі *Morph* визначено метод *fullPrintOn*, що лише виводить у потік ім’я класу отримувача та рядок ' *new*' слідом. Ідея в тому, що інтерпретація отриманого рядка призведе до створення екземпляра того ж класу, що й отримувач.

Потім визначено метод *constructorString*, який надсилає до *self* повідомлення *fullPrintOn:* з аргументом – потоком виведення, накладеним на рядок (див. лістинг 10.7).

Наостанок у класі *BorderedMorph*, що є надкласом *EllipseMorph*, визначено метод *fullPrintOn:*. Цей новий метод розширяє поведінку надкласу – класу *Morph*: він викликає метод надкласу та виконує додаткове виведення в потік (див. лістинг 10.8).

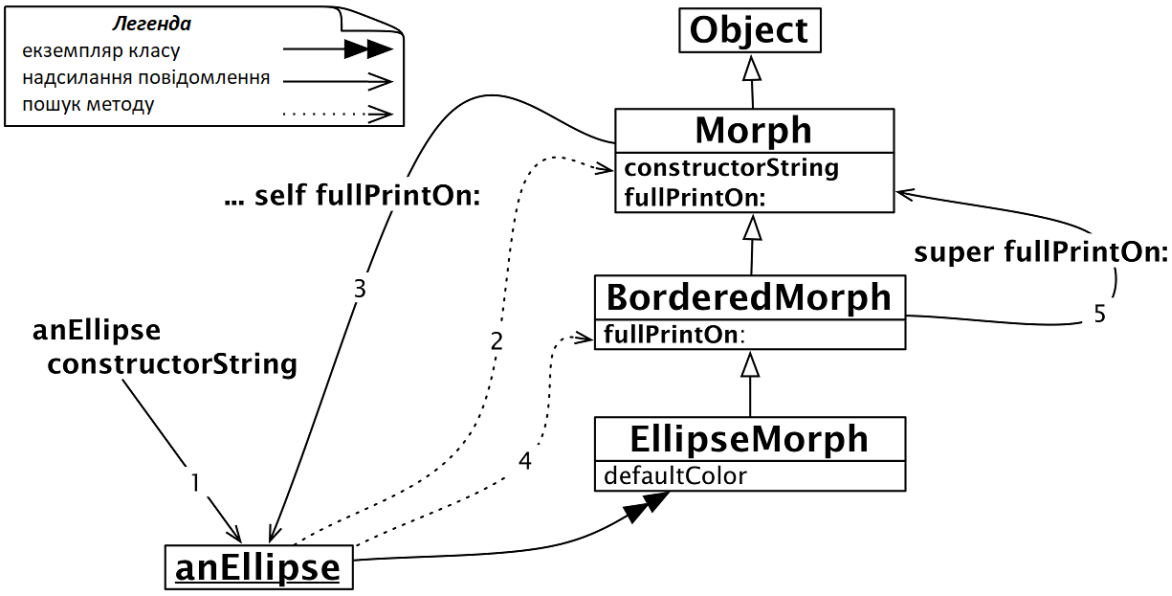

Рис. 10.3. Надсилання повідомлень до *self* і *super*

Розглянемо повідомлення *constructorString*, надіслане екземплярові класу *EllipseMorph*.

```
EllipseMorph new constructorString
>>> '(EllipseMorph new) setBorderWidth: 1 borderColor: Color black'
```

Як саме отримано такий результат за допомогою комбінації повідомлень до *self* і *super*? Спочатку *EllipseMorph new* створить новий екземпляр – назвемо його *anEllipse*. Повідомлення *anEllipse constructorString* (стрілка 1 на рис. 10.3) запустить пошук відповідного методу, який буде знайдено в класі *Morph* (стрілка 2).

Метод *Morph>>constructorString* надсилає повідомлення *fullPrintOn:* об’єктові *self* (стрілка 3). Пошук методу *fullPrintOn:* розпочинається з класу *EllipseMorph* і закінчується в

BorderedMorph: буде знайдено *BorderedMorph>>fullPrintOn:* (див. рис. 10.3, стрілка 4). Важливо зауважити, що надсилання повідомлення до *self* розпочинає пошук методу з класу отримувача повідомлення, в нашому випадку з класу об'єкта *anEllipse*.

Далі *BorderedMorph>>fullPrintOn:* надсилає повідомлення до *super*, щоб розширити функціональність методу, наслідуваного від надкласу.

Оскільки повідомлення надіслано до *super*, то пошук методу розпочинається в надкласі класу, в методі якого зазначено це надсилання, тобто в *Morph*. Тому зразу знайдеться і виконається метод *Morph>>fullPrintOn:* (стрілка 5).

10.14. Крок назад

Надсилання повідомлення до *self* динамічне в сенсі того, що, подивившись на метод, який його містить, ми не зможемо з'ясувати, який метод буде виконано. Справді, отримати повідомлення, яке містить вираз з *self*, може екземпляр підкласу, в якому перевизначено відповідний метод. Тут *EllipseMorph* міг би перевизначити метод *fullPrintOn:*, тоді його було б виконано методом *constructorString*. Зауважте, що, дивлячись тільки на метод *constructorString*, ми не можемо передбачити, який метод *fullPrintOn:* буде виконано (визначений в класі *EllipseMorph*, чи в *BorderedMorph*, чи *Morph*) під час виконання *Morph>>constructorString*, оскільки це залежить від отримувача повідомлення *constructorString*.

> **Важливо** Повідомлення до *self* запускає *пошук методу в класі отримувача*. Надсилання до *self* динамічне в сенсі того, що, дивлячись на метод, який його містить, ми не можемо перебачити, котрий метод буде виконано.

Зауважте, що пошук методу при повідомленні до *super* не починається в надкласі отримувача. Інакше б для опрацювання «*super fullPrintOn: aStream*» пошук розпочався б з класу *BorderedMorph*, що спричинило б нескінченний цикл.

Якщо добре поміркувати над повідомленням до *super* і рис. 10.3, то стане зрозуміло, що прив'язки *super* статичні. Все, що має значення, – це клас, в методі якого надсилають повідомлення до *super*. На противагу цьому значення *self* динамічне. Воно завжди представляє отримувача повідомлення, яке виконується. Це означає, що всі повідомлення, надіслані до *self*, запускають пошук з класу отримувача.

> **Важливо** Повідомлення до *super* запускає пошук методу, починаючи з *надкласу того класу, в якому є метод, що надсилає повідомлення до super.* Кажуть, що надсилання повідомлення до *super* статичне, бо достатньо поглянути на метод з повідомленням, щоб з'ясувати, в якому класі почнеться пошук – в надкласі класу, який містить цей метод.

10.15. Сторона екземпляра та сторона класу

Оскільки класи є об'єктами, то вони можуть мати свої змінні і свої методи. Ми називаємо їх *змінними екземпляра класу* і *методами класу*, але вони нічим не відрізняються від звичайних змінних екземпляра і методів. Вони лише оперують з іншими об'єктами – з класами в цьому випадку.

Змінна екземпляра описує стан екземпляра, а метод – його поведінку.

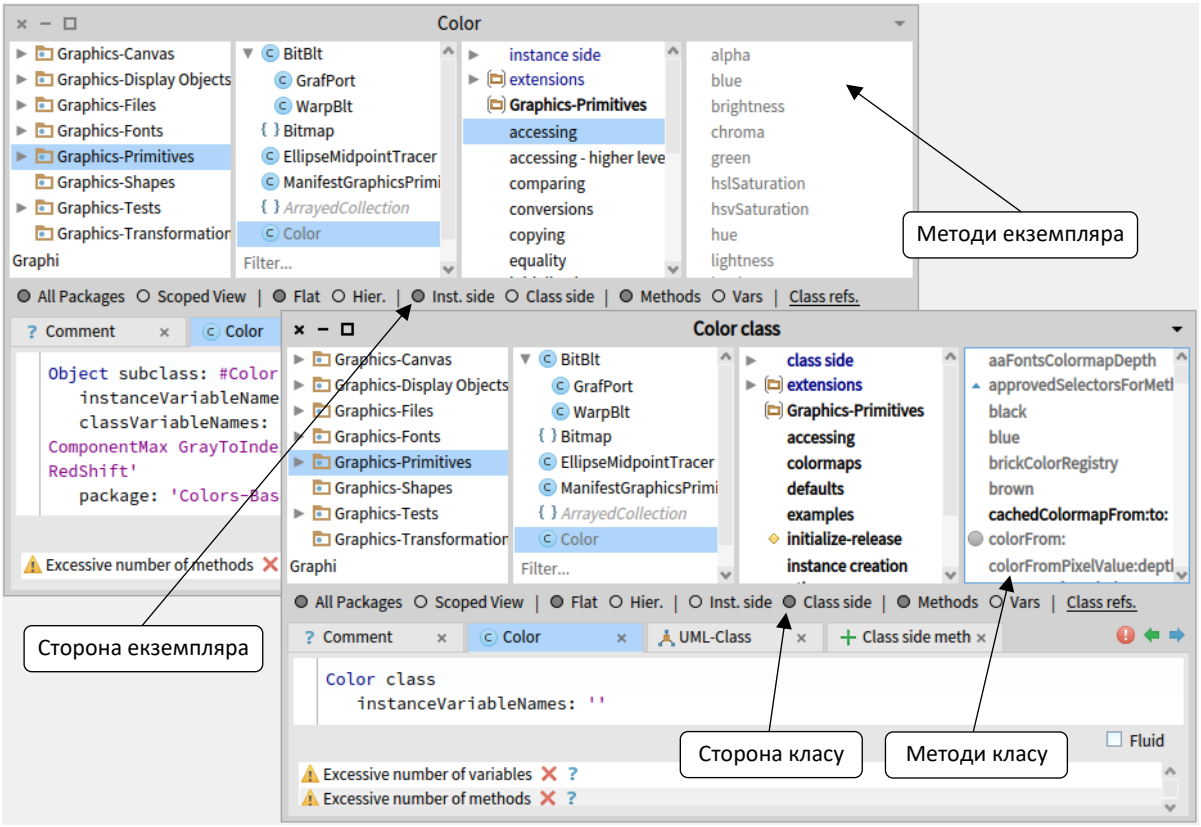

Рис. 10.4. Перегляд класу та його метакласу

Так само змінні екземпляра класу є звичайними змінними екземпляра, що оголошені в метакласі (метаклас – це клас, чиїм екземпляром є клас).

- *Змінні екземпляра класу* описують стан класу. Наприклад, змінна екземпляра класу *superclass*, що задає його надклас.
- *Методи класу* є методами оголошеними метакласом (які будуть виконані над класом). Наприклад, класові *Date* можна надіслати повідомлення *today*, відповідний метод визначено в метакласі *Date class*. Його буде виконано з класом *Date* як з отримувачем повідомлення.

Клас і його метаклас – це два окремих класи, незважаючи на те, що перший є екземпляром другого. Проте це здебільшого не має значення для пересічного розробника, бо він зосереджується на визначенні поведінки своїх об’єктів і класів, що їх створюють.

Клас є екземпляром свого метакласу, тому Оглядач допомагає переглядати і клас, і метаклас так, ніби вони дві сторони одного цілого: *сторона екземпляра* та *сторона класу*, як зображено на рис. 10.4.

- Якщо вибрати в Оглядачі клас *Color*, то за замовчуванням відкриється *сторона екземпляра*. Будуть відображені методи, які виконуються, коли повідомлення надіслано до *екземпляра* класу *Color*.
- Клацання на перемикачі ***Class side*** перемкне вікно на *сторону класу*: до методів, які будуть виконуватися, коли повідомлення надіслано самому класові *Color*.

Наприклад, вираз *Color paleBlue* надсилає повідомлення *paleBlue* класові *Color*. Тому оголошення методу *paleBlue* можна знайти на *стороні класу Color*, а не на стороні екземпляра.

Сторона екземпляра і сторона класу можуть містити однойменні методи, але це будуть різні методи з різним призначенням. Перегляньте приклади нижче.

```
"Метод класу blue - зручний спосіб створення об'єкта"
Color blue
>>> Color blue

"Метод читання red визначено на стороні екземпляра, повертає канал Red
   моделі кольору RGB"
Color blue red
>>> 0.0

"Однойменний метод читання blue визначено на стороні екземпляра,
   повертає канал Blue моделі кольору RGB"
Color blue blue
>>> 1.0
```

Створення метакласу

Новий клас оголошують в Оглядачі, заповнюючи шаблон, який він надає на стороні екземпляра. Після компіляції система створює не тільки оголошений клас, а й відповідний метаклас, який можна потім редагувати, перемкнувши Оглядача на сторону класу. Єдина частина шаблону створення метакласу, яку є сенс безпосередньо редагувати, це список імен змінних екземпляра метакласу.

Як тільки клас було створено, на стороні екземпляра Оглядача можна оголошувати, редагувати і переглядати методи, які будуть оброблені екземпляром цього класу, або його підкласів.

10.16. Методи класу

Методи класу можуть бути вельми корисними. Перегляньте як хороший приклад *Color class*. Легко бачити, що є методи класу двох видів: *методи створення екземплярів*, наприклад, *Color class>>blue*, і сервісні методи, наприклад, *Color class>>wheel:*. Це типові приклади, хоча можна натрапити на методи класу іншого призначення.

Сервісні методи зручно оголошувати на стороні класу, бо їх можна виконати без попереднього створення будь-яких додаткових об'єктів. Більшість таких методів містять коментарі, які пояснюють їхнє використання. З коментарями можна навіть експериментувати. Відкрийте метод *Color class>>wheel:*, двічі клацніть на початку коментаря *"(Color wheel: 12) inspect"* і натисніть [*Cmd* + *D*]. Ви побачите у вікні Інспектора результат виконання методу – масив створених кольорів.

Обізнаним з мовами Java і C++ методи класів можуть видатися подібними до статичних методів. Проте однорідність об'єктної моделі Pharo, де класи – це звичайні об'єкти, означає, що вони дещо відрізняються: тоді, коли виклики статичних методів у Java можна прив'язати на етапі компіляції як виклики звичайних процедур, методи класу Pharo зв'язуються динамічно на етапі виконання. Це означає, що наслідування, перевантаження і надсилання повідомлень до *super* працюють для методів класу в Pharo, але вони не працюють для статичних методів у Java.

10.17. Змінні екземпляра класу

Пригадаймо звичайні змінні *екземпляра*, оголошені в класі: всі екземпляри класу матимуть однаковий перелік змінних, але у кожного екземпляра він буде свій з окремим набором значень. Екземпляри підкласів також міститимуть ці змінні, бо успадкують їх.

Розповідь про змінні екземпляра *класу* буде така сама, бо клас є екземпляром іншого класу – метакласу. Тому змінні екземпляра класу оголошують в метакласі, і кожен клас має власний приватний набір значень у цих змінних.[1]

Змінні екземпляра класу працюють так само, як екземпляра: підкласи їх успадковують. Підклас успадкує змінні екземпляра класу, але *кожен підклас матиме власні приватні копії цих змінних*. Об’єкти не поділяють змінних з іншими екземплярами, так само і класи та їхні підкласи не поділяють змінних екземпляра класу.

Можна було б, наприклад, використовувати змінну *count* екземпляра класу, щоб стежити за кількістю створених його екземплярів. Тоді будь-який підклас мав би власну змінну *count*, а екземпляри підкласів рахувалися б окремо. У наступному підрозділі описано відповідний приклад.

Лістинг 10.9. Оголошення класів *Dog* і *Hyena*

```
Object subclass: #Dog
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Example'

Dog subclass: #Hyena
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Example'
```

Лістинг 10.10. Оголошення методів і змінної екземпляра класу

```
"Оголошення змінної екземпляра класу"
Dog class
   instanceVariableNames: 'count'

"Ініціалізація лічильника"
Dog class >> initialize
  count := 0.

"Відстеження кількості нових екземплярів"
Dog class >> new
  count := count + 1.
  ^ super new

"Метод-селектор"
Dog class >> count
  ^ count
```

[1] Відмінність тільки в тому, що в метакласу існує єдиний екземпляр об’єкта – його клас, тому набір змінних класу також єдиний. Він може тиражуватися тільки в підкласах (*прим. – Ярошко С.*).

10.18. Приклад. Змінні екземпляра класу та підкласи

Припустимо, що оголошено клас *Dog* і його підклас *Hyena* (див. лістинг 10.9). Припустимо також, що до класу *Dog* додали змінну екземпляра класу *count*, тобто оголосили її в метакласі *Dog class* (див. лістинг 10.10). Тоді *Hyena* автоматично успадкує цю змінну з класу *Dog*.

Далі припустимо, що визначили метод класу *Dog*, щоб ініціалізувати *count* нулем, і збільшувати його на одиницю під час створення нового екземпляра (див. лістинг 10.10).

Тепер, коли створюють новий екземпляр класу *Dog*, значення його поля *count* збільшується, а класу *Hyena* залишається незмінним. Легко переконатися, що кількість екземплярів класу *Hyena* обчислюється окремо.

```
Dog initialize.
Hyena initialize.
Dog count
>>> 0

Hyena count
>>> 0

| aDog |
aDog := Dog new.
Dog count
>>> 1 "Лічильник збільшився"

Hyena count
>>> 0 "Залишився незмінним"
```

Про ініціалізацію класу

Коли створюють об'єкт якогось класу, наприклад, *Dog new*, то *initialize* викликається автоматично, як частина виконання повідомлення *new* (у цьому можна переконатися, переглянувши визначення методу *new* в класі *Behavior*). Але у випадку з класами це дещо не так. Звичайне оголошення класу не викликає автоматично *initialize*, бо система не може здогадатися, що для одного з цілком робочих класів потрібно це зробити. Тому ми повинні явно викликати *initialize* в прикладі.

За замовчуванням методи ініціалізації автоматично виконуються тільки тоді, коли класи звантажуються – під час запуску системи. Згодом поговоримо про ліниву ініціалізацію.

10.19. Крок назад

Змінні екземпляра класу приватні для класу в такій самій мірі, як змінні екземпляра приватні для екземпляра. Оскільки класи та їхні екземпляри є різними об'єктами, то це має такі наслідки.

1. Клас не має доступу до полів його власних екземплярів. Тому, наприклад, клас *Color* не має доступу до змінних створеного ним об'єкта *aColorRed*. Іншими словами, незважаючи на те, що *клас* використовується для створення екземпляра (за допомогою *new* або інших допоміжних методів створення екземпляра як *Color red*), він не має якого-

небудь прямого доступу до змінних своїх екземплярів. Замість цього клас має використовувати методи доступу (відкритий інтерфейс), як будь-які інші об’єкти.

2. Обернене також істинне: *екземпляр* класу не має доступу до змінних екземпляра свого класу. У попередньому прикладі окремий екземпляр *aDog*, не має прямого доступу до змінної *count* класу *Dog* – тільки через метод-селектор.

> **Важливо** Клас не має доступу до змінних своїх власних екземплярів. Так само екземпляр класу не має доступу до змінних свого класу.

Через це методи ініціалізації завжди визначають на стороні екземпляра, сторона класу не має доступу до змінних екземпляра, тому не може ініціалізувати їх! Все, що клас може зробити, це відправити повідомлення ініціалізації новоствореному екземплярові, або використати методи доступу.

Java не має нічого подібного до змінних екземпляра класу. Статичні поля класу в Java і C++ більше схожі на змінні класу Pharo (про них йдеться в підрозділі 10.23), бо в усіх мовах всі підкласи і всі їхні екземпляри розділяють ті самі статичні змінні.

10.20. Приклад. Оголошення Одинака

Шаблон проєктування Одинак (англ. *Singleton*) часто трактують неправильно, а його неправильне застосування призводить до побудови доступу в процедурному стилі до єдиного глобального об’єкта. Проте Одинак надає типовий приклад використання змінних екземпляра класу і методів класу.

Уявимо, що потрібно реалізувати клас *WebServer* і використати шаблон Одинак, щоб гарантувати існування тільки одного екземпляра цього класу.

Клас *WebServer* оголосимо так.

```
Object subclass: #WebServer
   instanceVariableNames: 'sessions'
   classVariableNames: ''
   package: 'Web'
```

Далі на стороні класу додамо змінну екземпляра (класу) *uniqueInstance*.

```
WebServer class
   instanceVariableNames: 'uniqueInstance'
```

Так клас *WebServer* отримає нову змінну екземпляра на додачу до успадкованих від *Behavior* змінних *superclass*, *methodDict* тощо. Це означає, що значення додаткової змінної буде описувати екземпляр класу *WebServer class*, тобто клас *WebServer*.

Наявність змінних екземпляра легко перевірити за допомогою повідомлення *allInstVarNames*.

```
Object class allInstVarNames
>>> #(#superclass #methodDict #format #layout #organization #subclasses
   #name #classPool #sharedPools #environment #category)

WebServer class allInstVarNames
>>> #(#superclass #methodDict #format #layout #organization #subclasses
   #name #classPool #sharedPools #environment #category #uniqueInstance)"
```

Тепер можемо оголосити метод класу *uniqueInstance*, як зображено в лістингу 10.11.

Лістинг 10.11. Метод-селектор класу

```
WebServer class >> uniqueInstance
  uniqueInstance ifNil: [ uniqueInstance := self new ].
  ^ uniqueInstance
```

Цей метод спочатку перевірить, чи *uniqueInstance* було ініціалізовано. Якщо ні, то метод створить екземпляр класу і присвоїть його змінній *uniqueInstance*. Наприкінці метод поверне значення *uniqueInstance*. Оскільки *uniqueInstance* змінна екземпляра класу, то метод класу може безпосередньо доступатись до неї.

Коли вираз *WebServer uniqueInstance* виконуватиметься вперше, буде створено екземпляр класу *WebServer* і збережено його в змінній *uniqueInstance*. Наступного разу замість створення нового екземпляра повернеться створений раніше. Такий підхід до визначення методу читання – перевіряти, чи змінна ініціалізована, та створювати об'єкт, якщо її значення *nil* – називається «*лінивою ініціалізацією*».

Зауважте, що код створення екземпляра написаний в лістингу як «*self new*», а не як «*WebServer new*». У чому відмінність? Метод *uniqueInstance* визначено в класі *WebServer class*, тому можна подумати, що різниці немає. Справді, доки хтось не створить підклас *WebServer*, вони однакові. Але припустимо, що *ReliableWebServer* оголошено підкласом *WebServer*, і він наслідує метод *uniqueInstance*. Очевидно, ми мали б сподіватися, що *ReliableWebServer uniqueInstance* поверне екземпляр *ReliableWebServer*.

Використання *self* гарантує нам це, бо *self* буде прив'язно до відповідного отримувача, до одного з класів *WebServer* чи *ReliableWebServer*. Зауважте також, що *WebServer* і *ReliableWebServer* матимуть різні значення своїх змінних *uniqueInstance*.

Від перекладача. Уважний читач міг зауважити, що в одному з попередніх розділів вже було використано Одинака! Пригадайте, щоб дати змогу запускати Lights Out Game командою головного меню, в класі *LOGame* оголосили змінну класу *TheGame*, а методи *LOGame class>>open* і *LOGame class>>close* визначили так, щоб можна було відкрити тільки один екземпляр гри. Той одинак зроблено інакше, ніж описано тут. Дочитайте розділ до кінця. Можливо, ви захочете переробити *LOGame*.

10.21. Зауваження щодо лінивої ініціалізації

Ініціалізація початкових даних екземпляра загалом є прерогативою методу *initialize*. Помістивши всі виклики методів для обчислення початкових значень у метод *initialize*, покращують читабельність коду: згодом не доведеться полювати на всі методи-селектори, щоб дізнатися, які початкові значення вони задають. Хоча ініціалізація змінних у відповідних методах доступу (з перевіркою на *nil*) може видатися привабливою, її варто уникати, хіба що є вагомі причини для використання.

Не використовуйте надмірно шаблон лінивої ініціалізації.

У наведеному в лістингу 10.11 методі *uniqueInstance* використано лініву ініціалізацію, бо зазвичай користувачі не сподіваються викликати *WebServer initialize*. Натомість вони очікують, що клас готовий повернути новий єдиний екземпляр. Тому лінива ініціалізація тут виправдана. Також, якщо ініціалізувати змінну дуже дорого (наприклад,

відкрити зв’язок з базою даних чи мережевий сокет), то можна вибрати відтермінування ініціалізації, доки справді не буде потрібен такий об’єкт.

10.22. Спільні змінні

Розглянемо один з аспектів Pharo, не охоплений шістьма правилами базової моделі – спільні змінні.

Pharo підтримує спільні змінні трьох різних видів.

1. *Глобально* спільні змінні.

2. *Змінні класу*: змінні, спільні для класу і його екземплярів та підкласів (не плутайте зі змінними екземпляра класу, про які йшлося раніше).

3. *Змінні пулу*: змінні, спільні для групи класів.

Імена таких змінних прийнято починати з великої літери, щоб попередити, що вони спільні для кількох об’єктів.

Глобальні змінні

У Pharo всі глобальні змінні зберігаються в просторі імен *Smalltalk globals*, який є екземпляром класу *SystemDictionary*. Глобальні змінні доступні звідусіль. Кожен клас ідентифікується глобальною змінною. Також окремі глобальні змінні використовують, щоб називати спеціальні або часто використовувані об’єкти.

Змінна *Processor* надає доступ до екземпляра класу *ProcessorScheduler*, який є основним планувальником процесів у Pharo.

```
Processor class
>>> ProcessorScheduler
```

Інші корисні глобальні змінні

Smalltalk є екземпляром класу *SmalltalkImage*. Він містить доволі функціональності для керування системою. Зокрема, він містить посилання на основний простір імен *Smalltalk globals*, реалізований як системний словник. Він містить сам *Smalltalk*, бо це глобальна змінна. Ключі цього простору імен – це символи, що називають глобальні об’єкти в Pharo. Наприклад,

```
Smalltalk globals at: #Boolean
>>> Boolean
```

Smalltalk також є глобальною змінною, що підтверджують приклади.

```
Smalltalk globals at: #Smalltalk
>>> Smalltalk

(Smalltalk globals at: #Smalltalk) == Smalltalk
>>> true
```

World є екземпляром класу *PasteUpMorph*, що представляє екран. *World bounds* повертає прямокутник, який визначає простір цілого екрана. Всі морфи на екрані є підморфами *World*.

Undeclared – це інший словник, який містить усі невизначені змінні. Якщо в тексті методу посилаються на невизначену змінну, то Оглядач зазвичай запрошує визначити її, наприклад, як глобальну змінну або змінну класу. Якщо ж потім видалити визначення змінної, то код посилатиметься на невизначену змінну. Інспектування *Undeclared* може часом допомогти пояснити незрозумілу поведінку.

Використання глобальних змінних у коді

Рекомендованою практикою є строге обмеження використання глобальних змінних. Зазвичай краще використати змінну екземпляра класу або змінну класу і надати методи доступу до неї. Справді, якщо б реалізацію Pharo переробляли заново, то більшість глобальних змінних, які не є класами, замінили б одинаками чи чимось ще.

Звичний спосіб визначити глобальну змінну – це виконати за допомогою «*Do it*» присвоєння ще невизначеному імені, що починається з великої букви. Синтаксичний аналізатор запропонує визначити глобальну змінну. Щоб визначити глобальну змінну програмним способом, виконують «*Smalltalk globals at: #НазваГлобальноїЗмінної put: nil*». Щоб видалити її, виконують «*Smalltalk globals removeKey: #НазваГлобальноїЗмінної*».

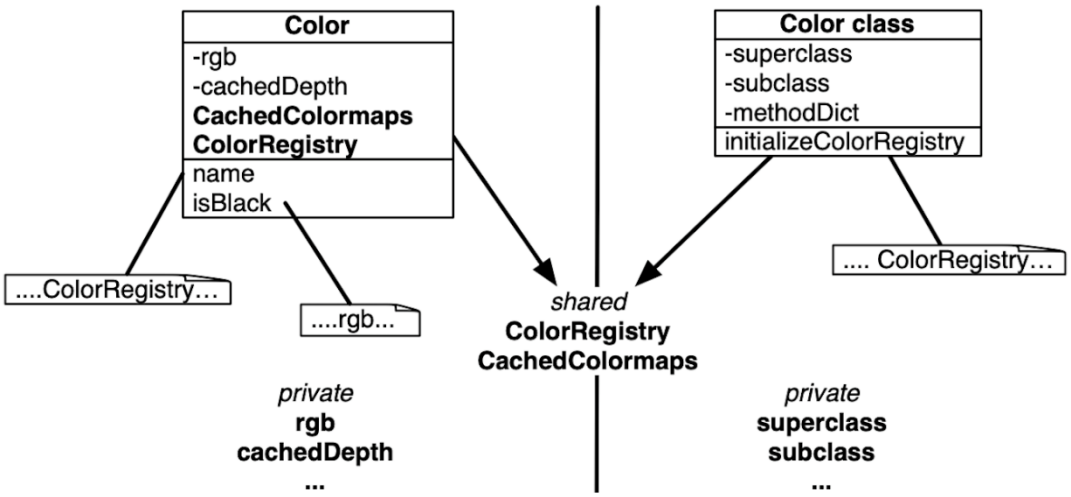

Рис. 10.5. Методи класу та методи екземпляра мають доступ до різних змінних

10.23. Змінні класу

Іноді виникає потреба надати доступ до одних даних всім екземплярам класу і самому класові. Це можливо за допомогою *змінних класу*. Термін *змінна класу* означає, що життєвий цикл змінної такий самий, як у класу. Однак цей термін не виражає того, що змінні класу є спільними для всіх екземплярів класу, а також для самого класу, як показано на рис. 10.5. Насправді кращою назвою була б *спільні змінні*, бо вона зрозуміло виражає їхню роль і попереджає про небезпеку їхнього використання, зокрема модифікації.

На рис. 10.5 видно, що *rgb* і *cachedDepth* поля екземпляра класу *Color*, тому доступні тільки екземплярам класу *Color*. Видно також, що *superclass*, *subclass*, *methodDict* тощо змінні екземпляра класу, доступні тільки класу *Color class*.

А ще видно щось нове: *ColorRegistry* та *CachedColormaps* змінні класу, визначені в *Color*. Великі букви в їхніх іменах підказують, що вони спільні. Насправді, не тільки всі екземпляри класу можуть до них доступатись, а й сам клас *Color* та *всі його підкласи*. До цих спільних змінних можуть доступатись і методи класу, і методи екземпляра.

Змінну класу визначають у шаблоні оголошення класу. Наприклад, клас *Color* визначає багато змінних класу, щоб пришвидшити створення кольорів. Його оголошення зображено в лістингу 10.12.

Лістинг 10.12. Клас *Color* і його змінні

```
Object subclass: #Color
   instanceVariableNames: 'rgb cachedDepth cachedBitPattern alpha'
   classVariableNames: 'BlueShift CachedColormaps ColorRegistry
      ComponentMask ComponentMax GrayToIndexMap GreenShift
      HalfComponentMask IndexedColors MaskingMap RedShift'
   package: 'Colors-Base'
```

Змінна класу *ColorRegistry* екземпляр класу *IdentityDictionary*, що містить часто вживані кольори, доступні за іменами. Цей словник спільно використовують всі екземпляри *Color* і сам клас. Він доступний для всіх методів екземпляра і класу.

Лістинг 10.13. Використання лінивої ініціалізації

```
Color class >> ColorRegistry
  ColorRegistry ifNil: [ self initializeColorRegistry ].
  ^ ColorRegistry
```

Лістинг 10.14. Ініціалізація класу *Color*

```
Color class >> initialize
  ...
  self initializeColorRegistry.
  ...
```

Ініціалізація класу

Наявність полів класу порушує питання, як їх ініціалізувати?

Один зі способів – лінива ініціалізація, описана раніше. Це можна зробити додаванням методу-селектора для доступу до змінної, який під час виконання ініціалізує її, якщо вона ще не була ініціалізована (див. лістинг 10.13). Це означає, що завжди потрібно буде використовувати селектор для доступу і ніколи не звертатися до змінної безпосередньо. Доведеться також затрачати час на надсилання повідомлення і на перевірку ініціалізації.

Інший спосіб – перевантажити метод класу *initialize*, як у прикладі з класом *Dog* (див. лістинг 10.14).

Якщо зупинили вибір на цьому варіанті, то треба буде пам’ятати про виклик методу *initialize* відразу після того, як його визначили (достатньо виконати *Color initialize*). Хоча методи ініціалізації класу виконуються автоматично, коли код класу завантажується в пам’ять, наприклад, зі сховища Monticello, проте вони *не виконуються* самі, коли вперше написані та скомпільовані в Оглядачі, або відредаговані та перекомпільовані.

10.24. Змінні пулу

Змінні пулу – це змінні, спільні для кількох класів, які можуть бути не пов’язані наслідуванням. Змінні пулу мають бути визначені як змінні спеціально призначеного класу, підкласу *SharedPool*. Наша порада – уникати використовувати їх у своїх класах. Вони

можуть стати в нагоді тільки у рідкісних і специфічних випадках. Мета цього параграфа – розповісти про них достатньо, щоб розуміти їхнє використання в коді.

Щоб отримати доступ до змінних пулу, клас має згадати його назву у своєму визначенні. Наприклад, клас *Text* зазначає, що використовує пул *TextConstants* (див. лістинг 10.15), який містить усі текстові константи, наприклад, *CR* і *LF*. Клас *TextConstants* визначає змінну *CR*, зв'язану зі значенням *Character cr*, тобто з символом переводу каретки.

Лістинг 10.15. Спільний словник у класі *Text*

```
ArrayedCollection subclass: #Text
   instanceVariableNames: 'string runs'
   classVariableNames: ''
   poolDictionaries: 'TextConstants'
   package: 'Collections-Text'
```

Це дає змогу методам класу *Text* доступатись до змінних спільного пулу в тілі методу *безпосередньо*. Наприклад, можемо написати такий метод.

```
Text >> testCR
  ^ CR == Character cr
```

Незважаючи на те, що клас *Text* не визначає змінну *CR*, він може отримати прямий доступ до неї в спільному пулі *TextConstants*, оскільки оголосив, що використовує пул.

Нижче наведено визначення класу *TextConstants*. Це спеціальний клас, підклас *SharedPool*, який містить змінні класу.

```
SharedPool subclass: #TextConstants
   instanceVariableNames: ''
   classVariableNames: 'BS BS2 Basal Bold CR Centered Clear CrossedX
      CtrlA CtrlB CtrlC CtrlD CtrlDigits CtrlE CtrlF CtrlG CtrlH CtrlI
      CtrlJ CtrlK CtrlL CtrlM CtrlN CtrlO CtrlOpenBrackets CtrlP CtrlQ
      CtrlR CtrlS CtrlT CtrlU CtrlV CtrlW CtrlX CtrlY CtrlZ Ctrla Ctrlb
      Ctrlc Ctrld Ctrle Ctrlf Ctrlg Ctrlh Ctrli Ctrlj Ctrlk Ctrll Ctrlm
      Ctrln Ctrlo Ctrlp Ctrlq Ctrlr Ctrls Ctrlt Ctrlu Ctrlv Ctrlw Ctrlx
      Ctrly Ctrlz DefaultBaseline DefaultFontFamilySize DefaultLineGrid
      DefaultMarginTabsArray DefaultMask DefaultRule DefaultSpace
      DefaultTab DefaultTabsArray ESC EndOfRun Enter Italic Justified
      LeftFlush LeftMarginTab RightFlush RightMarginTab Space Tab
      TextSharedInformation'
   package: 'Text-Core-Base'
```

І знову ж таки, ми рекомендуємо уникати використання змінних пулу та спільних словників.

10.25. Абстрактні методи і абстрактні класи

Абстрактний клас – це клас, створений швидше для того, щоб від нього наслідували, аніж створювали його екземпляри. Абстрактний клас зазвичай незавершений у тому сенсі, що він оголошує не всі методи, які використовує. Абстрактними називають методи-заповнювачі, щодо яких припускають, що вони будуть перевизначені в підкласах.

Pharo не має спеціального синтаксису, щоб зазначити, що клас чи метод абстрактні. Замість того, за домовленістю, тіло абстрактного методу складається з виразу «*self subclassResponsibility*». Це означає, що підклас відповідальний за визначення конкретної реалізації цього методу. Метод з виразом «*self subclassResponsibility*» обов’язково має бути перевантаженим, а отже ніколи не повинен виконуватись. Якщо ви забули про це, і викликали метод, то буде запущено виняток.

Клас вважається абстрактним, якщо хоч один з його методів абстрактний. Ніщо насправді не забороняє створити екземпляр абстрактного класу. Все працюватиме, доки не виконається абстрактний метод.

Лістинг 10.16. Абстрактний метод < класу *Magnitude*

```
Magnitude >> < aMagnitude
  "Відповідає, чи отримувач менший за аргумент."

  ^ self subclassResponsibility
```

Лістинг 10.17. Виклик абстрактного методу в класі *Magnitude*

```
Magnitude >> >= aMagnitude
"Відповідає, чи отримувач більший або рівний аргументу."

^ (self < aMagnitude) not
```

Лістинг 10.18. Перевизначення абстрактного методу в підкласі *Character*

```
Character >> < aCharacter
"Повертає true, якщо код отримувача < код aCharacter'a."

^ self asciiValue < aCharacter asciiValue
```

10.26. Приклад. Абстрактний клас *Magnitude*

Magnitude – абстрактний клас, який допомагає визначити об’єкти, які можна порівнювати один з одним. Підкласи *Magnitude* повинні реалізувати методи <, = і *hash*. За допомогою цих повідомлень *Magnitude* визначає інші методи: >, >=, <=, *max:*, *min:*, *between:and:* й інші для порівнювання об’єктів. Підкласи наслідують такі методи. Метод *Magnitude* >> < абстрактний і визначений, як показано в лістингу 10.6.

На противагу йому, метод >= конкретний і визначений в термінах методу < (див. лістинг 10.7). Те саме стосується й інших методів порівняння: всі вони визначенні в термінах методу <.

Character – підклас *Magnitude*. Він перевизначає метод < (якщо пригадуєте, той позначений як абстрактний у класі *Magnitude* використанням виразу «*self subclassResponsibility*») і заміняє його своєю версією (див. визначення методу в лістингу 10.18). *Character* також явно визначає методи = і *hash* та наслідує з *Magnitude* методи >=, <=, ~= й інші.

10.27. Підсумки розділу

Об’єктна модель Pharo одночасно проста й однорідна. Все є об’єктом, і майже все відбувається через надсилання повідомлень.

- Все є об’єктом. Примітивні сутності, наприклад, цілі числа є об’єктами, так само й класи є об’єктами першого класу.
- Кожен об’єкт є екземпляром класу. Класи визначають структуру своїх екземплярів через *закриті* змінні екземпляра і їхню поведінку через *відкриті* методи. Кожен клас є єдиним екземпляром свого метакласу. Змінні класу приватні, доступні всім екземплярам класу і самому класу. Класи не можуть напряму доступатись до змінних екземпляра, екземпляри не можуть доступатись до змінних екземпляра своїх класів. Для доступу визначають методи, якщо це потрібно.
- Кожен клас має один надклас. Коренем ієрархії простого наслідування є *ProtoObject*. Визначені програмістом класи зазвичай наслідують клас *Object* або його підкласи. Немає синтаксису, щоб оголосити клас абстрактним. Абстрактний – це звичайний клас з одним чи більше абстрактним методом (метод, чиїм тілом є вираз «*self subclassResponsibility*»).
- Все відбувається через надсилання повідомлень. Ми не *викликаємо методи*, а *надсилаємо повідомлення*. Отримувач повідомлення вибирає власний метод, щоб відповісти на нього.
- Пошук методу перебирає ланцюжок наслідування. Надсилання повідомлень до *self* динамічне і розпочинає пошук методу з класу отримувача, а повідомлення до *super* розпочинають пошук методу в надкласі класу, який містить метод з цим повідомленням. З цього погляду повідомлення до *super* більш статичні, ніж до *self*.
- Є три види спільних змінних. Глобальні змінні доступні будь-де в системі. Змінні класу спільні для класу, його підкласів і екземплярів. Змінні пулу спільні для декількох вибраних класів. Потрібно уникати використання спільних змінних завжди, коли це можливо.