Pharo 9 на прикладах

С. Дюкас
Г. Ракіч
та
С. Каплар
К. Дюкас

переклад С. Ярошко

Pharo 9 на прикладах

Пориньте у світ живих об’єктів

Pharo 9 by Example

Stéphane Ducasse and Gordana Rakic with Sebastijan Kaplar and Quentin Ducasse

September 1, 2022

BoD – Books on Demand

Pharo 9 на прикладах

Стефан Дюкас і Джордана Ракіч разом з Себастьян Каплар і Квентін Дюкас

Переклад українською з доповненнями

Сергій Ярошко

ЛНУ ім. Івана Франка
2022

УДК 004.432.2
Д 95

Перекладено за оригіналом:

Stéphane Ducasse and Gordana Rakic with Sebastijan Kaplar and Quentin Ducasse.
Pharo 9 by Example

Переклад:

Ярошко С. А. – канд. фіз.-мат. наук, доцент
(Львівський національний університет імені Івана Франка)

Рецензенти:

Іванчук Я. В. – д-р техн. наук, професор
(Вінницький національний технічний університет);

Міца О. В. – д-р техн. наук, доцент
(Ужгородський національний університет);

Яремко Г. В. – канд. пед. наук, доцент
(Національний університет «Львівська політехніка»)

Рекомендовано до друку
Вченою радою Львівського національного університету імені Івана Франка.
Протокол № 37/10 від 26.10.2022 р.

Стефан Дюкас

Д 95 Pharo 9 на прикладах / С. Дюкас, Дж. Ракіч [та ін.] ; пер. з англ. С. Ярошка. – Львів : ЛНУ ім. Івана Франка, 2022. – 270 с.

ISBN 978-617-10-0757-4

Pharo – це сучасна динамічно типізована об’єктно-орієнтована мова програмування з відкритим кодом і середовище інтерактивного програмування, сучасне втілення класичної системи Smalltalk. У книзі розглянуто синтаксис мови та типові способи використання середовища програмування, наведено велику кількість прикладів використання наявних класів і оголошення нових. Описано об’єктну модель Pharo, базові класи, колекції, взаємодію з потоками, систему модульного тестування й основи застосування технології розробки, керованої тестами, засоби рефлексії, інструменти та способи взаємодії з системами контролю версій.

Для студентів факультету прикладної математики та інформатики і всіх, хто цікавиться сучасними технологіями програмування.

УДК 004.432.2

ISBN 978-617-10-0757-4

Зміст

Глосарій

Debugger	*Налагоджувач*	Програма, інструмент для дослідження коду, що виконується. Можна використовувати для написання програм на льоту
File Browser	*Оглядач файлів*	Інструмент для доступу до файлової системи комп'ютера, має можливості завантаження, швидкого перегляду файлів
Finder	*Шукач*	Програма, інструмент для відшукання методів, класів, програмного коду, анотацій, прикладів. Уміє шукати за зразком перетворень
Iceberg	*Менеджер пакетів*	Система контролю версій для взаємодії з Git
Image	*Образ системи*	Колекція всіх об'єктів системи, завантажується у віртуальну машину
Inspector	*Інспектор об'єктів*	Програма, інструмент, який дає змогу заглянути всередину об'єкта, вивчити та змінити його стан
Morph	*Морфа*	Видимий елемент інтерфейсу користувача
Pharo Launcher	*Пускова програма*	Спеціальна програма, написана на Pharo, для відстежування та запуску версій Pharo
Playground	*Пісочниця*	Вікно для безпечного виконання фрагментів коду, інтегроване з інспектором об'єктів
Spotter	*Навідник*	Вікно діалогу для швидкого відшукання будь-чого в системі
System Browser	*Оглядач класів, Системний оглядач*	Головний інструмент системи Pharo, який надає доступ до всіх пакетів класів, їхніх методів, дає змогу оголошувати нові класи та редагувати наявні
Test Runner	*Менеджер тестів*	Програма керування модульними тестами
Transcript	*Консоль системи*	Текстове вікно, яке можна використовувати для виведення тестових повідомлень від об'єктів, публікує повідомлення про помилки під час інсталяції пакетів
Virtual machine, VM	*Віртуальна машина*	Частина Pharo, залежна від платформи, виконує байт-код програм
Workspace	*Робоче вікно*	Редактор текстів, можна використовувати для завантаження файлів, випробування фрагментів коду, зберігання до файлу
World Menu	*Головне меню*	Pharo – цілий світ на комп'ютері, тому його головне меню називається «Світ»

Розділ 1

Про цю книгу

Це видання книги засноване на попередньому виданні *Pharo на прикладах*, авторами якого були Андрю П. Блек, Стефан Дюкас, Оскар Нірстраз, Демієн Полле, Демієн Кассу та Маркус Денкер. Воно також опирається на випущене для Pharo 5.0 видання *Оновлене Pharo на прикладах*, авторства Стефана Дюкаса за редакцією Дмитра Загідуліна, Ніколая Гесса та Дімітріса Хлоупіса.

Багато аспектів використання інструментів Pharo змінилися з часу останнього видання, і ми наполегливо працювали, щоб описати ці зміни.

- Ми перебудували початок книги, щоб скоротити розділи.
- Ми додали новий «стартовий» розділ із простим прикладом оголошення лічильника. Він дасть змогу читачеві побачити найважливіші кроки для визначення класу, тестування та збереження його коду.
- Ми стисло описали новий системний оглядач Calypso та застосунок Pharo Launcher.
- Ми подали розділ, присвячений системі контролю версій Iceberg та управлінню пакетами. Тему доповнює нова книга *Manage Your Code with Git and Iceberg*, доступна за адресою http://books.pharo.org.
- Ми додали новий розділ про *Traits*.
- Ми спростили розділ з описом *SUnit*, оскільки на http://books.pharo.org тепер доступна супутня книга *Learning Object-Oriented Programming, Design and TDD with Pharo*.
- Ми виправили ті частини книги, які містили помилки в попередніх виданнях, наприклад, розділ *Morphic*. Поточна версія набагато краща порівняно з *Pharo на прикладах*.

Книга *Pharo на прикладах* зазнала чимало змін, модифікацій та оновлень, щоб відповідати поточній версії Pharo. Зміни не відбуваються самі собою, покращення є результатом наполегливої роботи багатьох людей, адже не можна просто змінити «5.0» на «9». Саме тому Стефан Дюкас, один з постійних і головних авторів книги протягом багатьох років, запропонував зазначити Себастьяна, Джордана та Квентіна серед авторів *Pharo 9 на прикладах*.

1.1. Що таке Pharo

Pharo (*Фаро́*) – це сучасна динамічно типізована мова програмування з відкритим кодом і середовище інтерактивного програмування, натхненником яких став Smalltalk. Pharo та його екосистема складаються з шести основних елементів:

- динамічна мова програмування з синтаксисом, схожим на запис звичайною англійською мовою, простим на стільки, що його можна вмістити на поштівці, і, водночас, зрозумілим навіть за першого знайомства;
- середовище «живого» програмування, що дає змогу програмістові змінювати код на льоту під час його виконання;

- потужне інтегроване середовище розробки з вичерпним комплектом інструментів, що допомагають керувати складним кодом і спонукають до проєктування правильної архітектури;
- багата бібліотека класів, яка робить середовище таким потужним, що його можна сприймати як віртуальну операційну систему, встановлену на віртуальну машину з дуже швидким JIT-компілятором та повним доступом до всіх засобів і бібліотек «рідної» операційної системи;
- культура розробки, в якій зміни та вдосконалення заохочуються та високо цінуються;
- спільнота, яка вітає програмістів з усіх кінців світу з будь-якими вміннями та знанням довільних мов програмування.

Pharo прагне надати вільно поширювану платформу для професійного створення програмного забезпечення та потужну надійну основу для досліджень і розробки динамічних мов і середовищ. Pharo також слугує середовищем реалізації для платформи Seaside розробки веб-додатків, яка доступна на сайті http://www.seaside.st.

Ядро Pharo містить лише той код, що розповсюджується за ліцензією MIT (група ліцензій, розроблених Масачусетським технологічним інститутом для розповсюдження вільного програмного забезпечення). Проєкт Pharo розпочався у березні 2008 року як відгалуження Squeak, крос-платформного втілення класичної системи програмування Smalltalk-80, а перша бета-версія 1.0 побачила світ 31 липня 2009 року. Поточною версією Pharo є 9.0[1]. Її було випущено в липні 2021 року.

Pharo – дуже мобільне, оскільки може працювати в багатьох операційних системах: OS X, Windows, Linux, Android, iOS та Raspberry Pi. Його віртуальна машина цілком написана на підмножині Pharo. Це робить його також легким для налагодження, розуміння та внесення змін і доповнень. Pharo є рушієм широкого спектра інноваційних проєктів від мультимедійних застосунків і освітніх платформ до комерційних середовищ розробки веб-застосувань.

Однією з головних рис Pharo є те, що це насправді нове втілення Smalltalk, а не звичайна копія колишніх його реалізацій. Ми усвідомлюємо, що проєкти в стилі Великого вибуху для створення з нуля всього й одразу рідко коли досягають успіху, тому Pharo пропагує еволюційні та поступові зміни. Еволюція означає, що Pharo допускає помилки і не прагне досягти досконалих рішень за один великий крок, навіть, якщо б нам цього дуже хотілося. Натомість Pharo підтримує невеликі послідовні зміни, але зроблені багато-багато разів. Такий підхід зберігає в стабільному стані навіть найостаннішу розроблену версію, що дає змогу експериментувати з важливими новими засобами чи бібліотеками. Це сприяє внеску та швидкому зворотному зв'язку від спільноти, на яку Pharo покладається задля свого успіху. Врешті, Pharo не має атрибуту «лише для читання»: воно щодня інтегрує зміни, зроблені спільнотою. Pharo отримало доповнення від сотні авторів з цілого світу, і ви теж можете вплинути на Pharo! Просто перегляньте http://github.com/pharo-project/pharo.

1.2. Для кого ця книга

Ця книга не вчитиме вас програмувати. Читач мав би знати якісь мови програмування. Допоможе також знання основ об'єктно-орієнтованого програмування.

[1] На час виходу перекладу випущено Pharo 10, і триває робота над Pharo 11 (*прим. – Ярошко С.*).

Книга ознайомить вас із середовищем програмування Pharo, мовою та відповідними інструментами. Ми продемонструємо загальні ідіоми та практичні підходи, але головний наголос зробимо на технології, а не на об'єктно-орієнтованому проєктуванні. Ми наводитимемо хороші наочні приклади так часто, як тільки це буде можливо.

Онлайновий курс Pharo

На сайті http://mooc.pharo.org доступний чудовий безкоштовний масовий відкритий онлайн курс по Pharo. Він надає хороший вступ до Pharo й об'єктно-орієнтованого програмування і добре доповнює цю книгу.

Подальше читання

Ця книга не навчить вас усьому, чого ви потребуєте, чи хотіли б дізнатися про Pharo. Знадобляться перелічені нижче книжки, доступні на http://books.pharo.org.

- *Learning Object-Oriented Programming, Design and TDD with Pharo* розповість про головні аспекти об'єктно-орієнтованого проєктування та розробки, керованої тестами. Це хороша книжка для вивчення об'єктно-орієнтованого програмування.
- *Pharo with Style*. Це книга з розряду «мусите прочитати». Вона пояснює, як написати якісний і читабельний код на Pharo. Всього за одну годину зможете суттєво підвищити стандарт свого коду.
- *The Spec UI framework* покаже, як розробляти на Pharo додатки зі стандартним інтерфейсом користувача.

Книги технічного спрямування.

- *Manage Your Code with Git and Iceberg* детально пояснює, як керувати вашим кодом на Git.
- *Enterprise Pharo*. Містить різні розділи, що стосуються інтернету, конвертерів, звітності та документації, потрібних для розгортання програм.
- *Deep into Pharo* розкриває глибші теми ніж *Pharo на прикладах*.

А також є книги для розширення власного кругозору

- *A simple reflective object kernel* переглядає всі фундаментальні моменти принципу «об'єкти є основою всього», відправляючи вас у невелику подорож, щоб створити рефлексивне ядро мови. Це справді надзвичайно.

Окрім того, є багато інших книг про Smalltalk у вільному доступі на http://stephane.ducasse.free.fr/FreeBooks.html.

1.3. Порада

Не переймайтеся, якщо ви не зрозумієте відразу якусь частину коду на Smalltalk. Ви не повинні знати геть усе! Алан Найт виразив цей принцип так[2]:

Намагайтесь не хвилюватися. Програмісти-початківці на Smalltalk часто стикаються з труднощами, бо думають, що вони мусять докладно розуміти, як все працює, перш ніж його використовувати. Це призводить до того, що вони затрачають чимало часу, перш

[2] http://alanknightsblog.blogspot.com/2011/10/principles-of-oo-design-or-everything-i.html

ніж запрограмують *Transcript show: 'Hello World'*. Та однією з величезних переваг ООП є можливість на запитання «Як це працює?» відповісти «Мені байдуже».

Не вагайтеся ні миті запитати нас, якщо ви чогось не зрозуміли, складного чи простого. Ви можете скористатися нашими списками розсилання (pharo-users@lists.pharo.org, pharo-dev@lists.pharo.org) або IRC чи Discord. Ми любимо запитання і радо вітаємо зацікавлених з будь-якими навиками програмування.

1.4. Відкрита книга

Ця книга відкрита в такому розумінні:

- книгу опубліковано під ліцензією Creative Commons із зазначенням авторства і розповсюдження на таких самих умовах (скорочено: CC BY-SA). Це означає, що Ви можете вільно поширювати й адаптувати цю книгу, якщо тільки дотримуєтесь вимог ліцензії, доступних за таким URL: http://creativecommons.org/licenses/by-sa/3.0/;
- ця книга описує лише ядро Pharo. Ми б хотіли заохотити й інших авторів додавати до неї параграфи, присвячені тим частинам Pharo, які ми не описали. Якщо Ви хотіли б долучитися до такої праці, будь ласка, контактуйте з нами. Ми б дуже хотіли побачити більше книг, присвячених Pharo!
- ви можете також зробити свій внесок безпосередньо до цієї книги через GitHub. Просто додайте свою пропозицію до списку розсилки, IRC або Discord, дотримуючись розташованих там інструкцій. Репозиторій GitHub можна знайти тут https://github.com/SquareBracketAssociates/UpdatedPharoByExample.

1.5. Спільнота Pharo

Спільнота Pharo дружелюбна й активна. Нижче подано короткий перелік ресурсів, які можуть бути корисними для вас:

- http://www.pharo.org – головний сайт Pharo;
- http://www.github.com/pharo-project/pharo є головним обліковим записом Pharo на GitHub;
- активні обговорення Pharo відбуваються на сервері Discord – на IRC платформі для чатів. Щоб стати учасником, достатньо попросити про запрошення на сайті Pharo http://pharo.org/community у розділі Discord. Запрошуємо усіх;
- користувачі Pharo запустили вікі на https://github.com/pharo-open-documentation/pharo-wiki;
- підтримується каталог з проєктами Awesome: https://github.com/pharo-open-documentation/awesome-pharo;
- можливо, ви чули про SmalltalkHub, http://www.smalltalkhub.com/, що був аналогом SourceForge/GitHub для проєктів на Pharo впродовж десяти років. Тут розташовано багато додаткових пакетів для Pharo і завершених проєктів. Нині спільнота використовує здебільшого git репозиторії: GitHub, GitLab та Bitbucket.

1.6. Приклади і вправи

Ми намагаємося надати стільки прикладів, скільки можливо. Зокрема, є багато прикладів, що демонструють фрагменти коду, придатні для безпосереднього виконання. Ми використовуємо довгу стрілку >>>, щоб вказати на результат, який ви отримаєте, якщо виокремите певний вираз і виберете команду *Print it* з його контекстного меню.

```
3 + 4
>>>7   "якщо ви позначите 3+4 і виберете 'print it', то отримаєте 7"
```

Якщо ви хочете випробувати у Pharo такі фрагменти коду і не маєте часу набирати їх власноруч, то можете завантажити текстовий файл з текстами всіх прикладів із сайту книги http://books.pharo.org/pharo-by-example/ з бічної панелі *Resources*.

1.7. Друкарські домовленості

Ми завжди зазначаємо перед текстом методу префікс з іменем класу, в якому він визначений. Завдяки цьому ви завжди знатимете, до якого класу належить метод.

Наприклад, якщо ви побачите в книзі код

```
MyExampleSetTest >> testIncludes
  | full empty |
  full := Set with: 5 with: 6.
  empty := Set new.
  self assert: (full includes: 5).
  self assert: (full includes: 6).
  self assert: (empty includes: 5) not
```

і захочете перенести його у Pharo, то вам потрібно буде в класі *MyExampleSetTest* визначити метод

```
testIncludes
| full empty |
full := Set with: 5 with: 6.
empty := Set new.
self assert: (full includes: 5).
self assert: (full includes: 6).
self assert: (empty includes: 5) not
```

Під час перекладу книги всі приклади були випробувані у Pharo 9.0 в ОС Windows 10. Здобутий досвід описано у важливих на думку перекладача коментарях і доповненнях, виділених окремим форматуванням.

Від перекладача. У книзі використано різні способи форматування тексту: фрагменти програм і синтаксичні елементи, розташовані безпосередньо в тексті, виділено *курсивом*; згадані в тексті частини графічного інтерфейсу користувача виділено *товстим курсивом*; «*команди меню*» виділено курсивом і лапками, а комбінації клавіш – квадратними дужками, як у прикладі: [*Alt* + *Shift*].

1.8. Подяки

Найперше хочемо подякувати Алану Кею (Alan Kay), Дену Інґальсу (Dan Ingalls) та їхній команді за те, що зробили Squeak, це дивовижне Smalltalk середовище розробки, доступним як проєкт з відкритим кодом, адже Pharo бере свій початок саме від нього. Існування Pharo також було б неможливе без неймовірної праці розробників мовою Squeak.

Також хочемо подякувати Гіларі Фернандес (Hilaire Fernandes) і Сержу Стінквічу (Serge Stinckwich) за дозвіл перекласти їхні статті про Smalltalk та Дам'єну Касо (Damien Cassou) за написання розділу, присвяченого потокам. Ми особливо дякуємо Олександру Бергелю (Alexandre Bergel), Орлі Гріві (Orla Greevy), Фабріціо Перін (Fabrizio Perin), Лукасу Ренглі (Lukas Renggli), Джорджу Рессіа (Jorge Ressia) та Ервану Вернлі (Erwann Wernli) за їхні детальні рецензії. Дякуємо Університету Берна (Швейцарія) за люб'язну підтримку цього проєкту з відкритим вихідним кодом і за розміщення веб-сайту цієї книги протягом кількох років.

Ми також дякуємо спільноті Pharo за їхній ентузіазм у підтримці проєкту створення книги і за переклади першого видання *Pharo by Example*.

1.9. Спеціальні подяки

Хочемо подякувати першим авторам цієї книги! Без їхньої початкової версії було б дуже важко зробити цю. Книга «Pharo на прикладах» (Pharo by Example) є основною для залучення новачків, тому має дуже важливе значення.

Особлива подяка Девіду Віксу (David Wickes) за його значний вклад у редагування та Сергію Ярошку за переклад українською та всі зауваження, які покращили зміст книги.

Дякуємо Адріану Семпеліну (Adrian Sampaleanu), Манфреду Крюнерту (Manfred Kröhnert), Маркусу Шлагеру (Markus Schlager), Вернеру Касенсу (Werner Kassens), Майклу Окіфу (Michael OKeefe), Арьї Хофману (Aryeh Hoffman), Полю Макінтошу (Paul MacIntosh), Ґауравy Сінху (Gaurav Singh), Джіґясу Ґруверу (Jigyasa Grover), Крейґу Алену (Craig Allen), Сержу Стінквічу (Serge Stinckwich), avh-on1, Юрію Тимчуку, zio-pietro, Вів'єн Моро (Vivien Moreau), Лівей Чу (Liwei Chou) за виправлення помилок і відгуки. Особлива подяка Демієну Кассу (Damien Cassou) та Сирілю Ферліко (Cyril Ferlicot) за їхню велику допомогу в оновленні книги.

Нарешті хочемо подякувати інституту Inria за постійну значну фінансову підтримку і членам команди RMoD за невтомне просування Pharo. Хочемо також подякувати членам консорціуму Pharo.

Особлива подяка Демієну Полле (Damien Pollet) за цей чудовий шаблон книги та Джозефу Гегану (Joseph Guégan) за надання нам права використовувати його чудове зображення сходів маяка Екмюль.

S. Ducasse, S. Kaplar, Gordana Rakic, and Q. Ducasse

Розділ 2
Початок роботи з Pharo

У цьому розділі ви навчитеся запускати Pharo, дізнаєтеся, які файли входять до системи Pharo, та яке їхнє призначення, вивчите різні способи взаємодії з середовищем програмування, ознайомитеся з деякими основними інструментами. У розділі описано, по-перше, як встановити Pharo; по-друге, як запустити Pharo. Ви також матимете нагоду запустити інтерактивний посібник *ProfSteph*, який ознайомить вас з основами синтаксису, допоможе визначити новий метод, створити об'єкт і надіслати йому повідомлення.

2.1. Встановлення Pharo

Застосунок Pharo створено для автономного використання. Він чудово працює без підтримки додаткових бібліотек, тому для запуску Pharo не потрібно нічого інсталювати у вашій системі. Як буде пояснено пізніше, Pharo складається з віртуальної машини (ВМ), образу, опису колекції змін і колекції фраґментів програмного коду. Встановити і налаштувати їх на вашому комп'ютері можна кількома різними способами.

Pharo Launcher

Pharo Launcher – це багатоплатформний застосунок, який полегшує управління кількома образами та віртуальними машинами Pharo. Він доступний для безкоштовного завантаження з http://pharo.org/download. Оберіть на сайті кнопку вашої операційної системи, щоб завантажити відповідний Pharo Launcher. Він містить все необхідне для запуску Pharo. Докладний опис його використання наведено в параграфі 2.3.

Скрипти автоматичного встановлення

Замість Pharo Launcher для завантаження конкретних версій Pharo можна використовувати скрипти, розміщені на https://get.pharo.org. Це справді зручно для автоматизації процесу встановлення Pharo.

Для завантаження найновішої версії системи Pharo 9.0 використовуйте скрипт

```
wget -O- get.pharo.org/90+vm | bash
```

Щоб запустити її на виконання, введіть команду

```
./pharo-ui Pharo.image
```

Станом на нині наведені скрипти працюють в операційних системах macOS і Linux.

2.2. Файли компонентів Pharo

Pharo складається з чотирьох головних файлів. Під час читання цієї книги ви не будете взаємодіяти з ними безпосередньо, проте корисно розуміти їхнє призначення.

1. *Віртуальна машина* (ВМ) – єдиний компонент Pharo, що відрізняється для кожної операційної системи. ВМ є виконавчою системою, подібною до віртуальної

машини Java. Кожного разу, коли користувач компілює фраґмент тексту програми, компілятор генерує байт-код Smalltalk. ВМ отримує такий байт-код, перетворює його на команди процесора і виконує їх. Pharo постачається з Cog VM, з дуже швидкою JIT-машиною (JIT – just in time, машина, що компілює на льоту).

Виконуваний файл віртуальної машини називають:

- *Pharo.exe* для Windows;
- *pharo* для Linux;
- *Pharo* для OS X (всередині пакета його також називають *Pharo.app*).

Інші перелічені компоненти можна переносити між операційними системами. Їх можна копіювати і запускати на відповідній ВМ.

2. *Образ системи* є миттєвим знімком стану працюючого Pharo. Він міститься у *.image* файлі, що має крос-платформний формат: образ Pharo з однієї операційної системи можна використовувати в будь-якій іншій з відповідною ВМ. Файл образу зберігає стан усіх об'єктів системи, актуальний на певний момент часу, включно з класами та методами, оскільки вони також об'єкти. Образ є контейнером віртуальних об'єктів.

   Файл образу називають відповідно до версії системи, наприклад, *Pharo9.0.image* містить образ Pharo 9.0. Він синхронізований з файлом змін *Pharo9.0.changes*.

3. *Файл змін* зберігає протокол усіх модифікацій вихідного тексту оголошення класів, зокрема, всіх зроблених вами змін під час сеансу програмування у Pharo. Кожна версія середовища постачається з майже порожнім файлом змін, що називається відповідно до версії, наприклад, *Pharo9.0.changes*. Цей файл містить історію змін кожного методу, яку можна використати для відстеження відмінностей або скасування внесених змін. Це означає, що ви зможете відновити всі запрограмовані вами методи з *.changes* файлу навіть тоді, коли образ системи не вдалося зберегти через зависання чи через власну забудькуватість. Файл змін пов'язаний з файлом образу, вони завжди працюють разом. Краще не чіпайте їх. Навіть при тому, що Pharo може працювати без них, ви ризикуєте втратити всю свою роботу, якщо не збережете її за допомогою Git.

4. У *.source* файлі зберігається вихідний код усіх частин системи, що змінюється нечасто. Цей файл важливий, оскільки файл образу зберігає лише об'єкти з компільованими методами та їхніми байт-кодами і не містить вихідних текстів. Зазвичай source-файл генерують один раз під час випуску нової версії Pharo. Для Pharo 9.0 він називається *Pharo9.0.sources*.

Пара Image/Change

Файли образу та змін (*.image* та *.changes* файли) з інсталяційного архіву є початковою точкою живого середовища програмування, яке ви пристосовуватимете для власних потреб. Коли будете працювати з Pharo, ці файли зазнаватимуть змін, тому переконайтеся, що вони розташовані в каталозі, в якому ваша операційна система не забороняє вам записувати. Було б добре також вилучити їх зі списку перевірки своєї антивірусної програми. Завжди зберігайте ці два файли разом, ніколи не перейменовуйте їх і не редагуйте за допомогою редактора текстів, бо *.image* файл зберігає стан пам'яті етапу виконання об'єктів вашої системи, пов'язаний з відповідними місцями *.changes* файлу задля доступу до вихідного коду. Було б добре зберігати запасну копію завантажених

.image та *.changes* файлів. Тоді ви завжди зможете розпочати роботу з самого початку і повторно завантажити створений вами код. Проте найефективнішим способом зберігання коду є використання системи контролю версій Iceberg разом з Git, яка забезпечує легке зберігання коду в сховищі та відстежування змін.

Загальне налаштування

Ви можете помістити описані чотири головні компоненти Pharo до одного каталогу, проте частіше віртуальну машину та *.source* файл розташовують в окремому каталозі і надають до нього доступ тільки для читання усім користувачам.

Оберіть будь-який зручний для вас та ефективний для вашої операційної системи варіант. Якщо ви тільки починаєте працювати з Pharo, то ліпше використовувати Pharo Launcher, оскільки він все налаштує сам.

2.3. Запуск Pharo за допомогою Pharo Launcher

Pharo Launcher – це інструмент, який допомагає завантажувати та впорядковувати образи Pharo. Він дуже зручний для отримання нових версій Pharo та для оновлення наявних (для виправлення виявлених помилок). Також він надає вам доступ до попередньо сконфігурованих образів зі спеціальними бібліотеками, що звільняє від ручного встановлення та налаштування таких бібліотек.

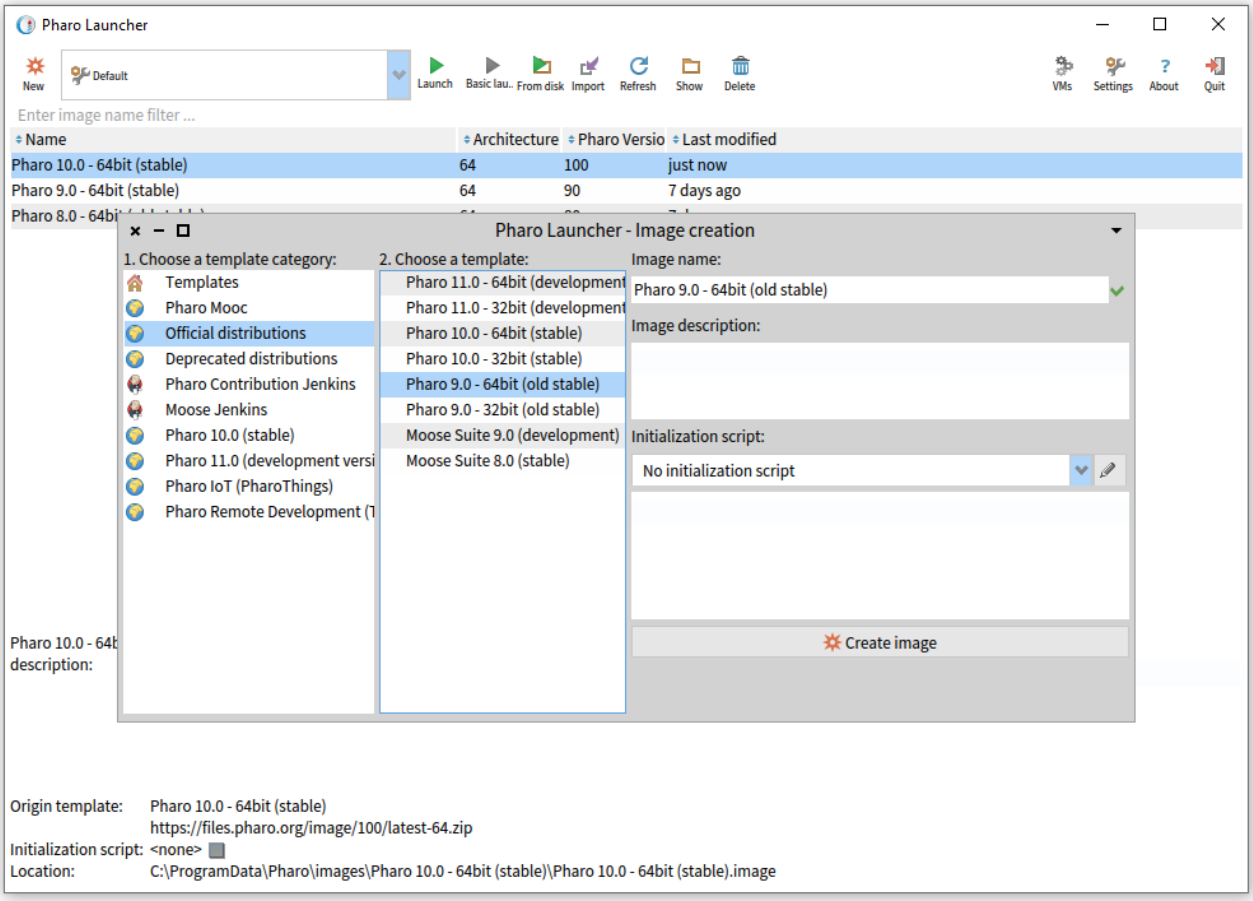

Рис. 2.1. Pharo Launcher – найлегший спосіб керувати образами Pharo

Pharo Launcher можна завантажити з https://pharo.org/download. Тут розміщено стислі пояснення щодо завантаження і посилання на версії додатку для різних платформ.

Докладні інструкції щодо завантаження, налаштування та використання Pharo Launcher можна знайти на https://pharo-project.github.io/pharo-launcher/.

Після встановлення та запуску Pharo Launcher матиме вигляд, подібний до зображеного на рис. 2.1. Щоб завантажити з сервера новий образ Pharo, потрібно клацнути кнопку ***New*** у лівому верхньому куті вікна. Саме вона відкриває діалог, зображений поверх вікна додатку на рис. 2.1.

Від перекладача. Pharo Launcher завантажує файли образу, змін та вихідного коду і віртуальну машину в папки, вказані в його налаштуваннях. За замовчуванням, вони прив'язані до імені користувача ОС. Наприклад, образ буде завантажено до папки *\Users\ActualUserName\Documents\Pharo\images*, де *ActualUserName* – ім'я користувача. Якщо ваше ім'я задано кирилицею, то Pharo Launcher може мати проблеми з розпізнаванням шляху до завантажених файлів, тому налаштування за замовчуванням варто змінити.

Вгорі праворуч у Pharo Launcher є кнопка *Settings*, що відкриває вікно налаштувань. Місце розташування папок задають у вкладці «*Pharo Launcher*», у пунктах «*Location of template sources file*», «*Location of your images*» тощо. Зазначте тут повний шлях до зручної для вас папки так, щоб він не містив кириличних літер. Налаштування за замовчуванням варто змінити також тоді, коли ви вважаєте за недоречне зберігати Pharo в папці Documents.

Під час завантаження Pharo Launcher будьте готові прийняти архів розміром майже 56 Мб, який можна вилучити згодом після встановлення застосунку. Розгорнутий застосунок займе близько 140 Мб. Це досить відчутні затрати дискового простору, але воно того варте, адже Launcher вміє завантажувати образи та віртуальні машини, стежить за оновленнями, повідомляє вас про всі новинки.

У вікні діалогу *Image creation* оберіть потрібний вам образ (стабільну версію) та натисніть кнопку *Create image* – Launcher завантажить і розпакує відповідні файли. Для них потрібно приблизно 100 Мб дискового простору. Перед першим запуском завантаженого образу Launcher завантажить відповідну віртуальну машину. Вона займе ще майже 30 Мб. Архів віртуальної машини застосунок не видаляє, можна зробити це власноруч.

Вікно Pharo Launcher відображає список образів Pharo, що зберігаються локально на вашій машині (зазвичай у спільній системній папці). Ви одразу можете запустити будь-який з них: двічі клацніть на ньому, або позначте і натисніть кнопку ***Launch***. Контекстне меню, яке викликають правою кнопкою мишки, містить кілька корисних команд: копіювання та перейменування образів, відшукання їхнього розташування у файловій системі тощо.

Окрім образів, завантажених із сервера, ви можете за допомогою Pharo Launcher запускати власні образи системи, збережені чи перейменовані під час використання Pharo. Для цього імпортуйте в Launcher відповідну пару *.image* та *.change* файлів.

2.4. Запуск Pharo з командного рядка

Якщо ви використовуєте автономну версію Pharo, то запускайте його так, як це зазвичай роблять у вашій операційній системі: перетягніть файл *.image* на значок віртуальної машини, або двічі клацніть по значку *.image* файлу, або надрукуйте в командному рядку ім'я віртуальної машини і слідом – повне ім'я *.image* файлу.

- У системі macOS двічі клацніть пакунок (bandle) *Pharo9.0.app* у розархівованому завантаженні.

- У системі Linux двічі клацніть (або запустіть з командного рядка) виконуваний bash-скрипт *pharo* з розархівованої папки Pharo.
- У системі Windows зайдіть у розархівовану папку Pharo і двічі клацніть *Pharo.exe*.

Загалом Pharo намагається «робити правильні речі». Якщо ви двічі клацнете на ВМ, то віртуальна машина шукатиме у своїй папці файл образу, щоб завантажити його. Pharo дає змогу зберігати образ у файлі з довільним іменем. Це зручно, коли ви провадите різні розробки: кожному проєкту відповідатиме свій образ. Отож запущена ВМ може знайти кілька *.image* файлів. У цьому випадку машина розпочинає діалог вибору файлу, щоб користувач міг вказати потрібний йому образ. Якщо ви двічі клацнете на *.image* файлі, то операційна система спробує запустити відповідну ВМ, щоб відкрити його. Можливо, вам доведеться допомогти їй у цьому.

Якщо на вашому комп'ютері інстальовано кілька віртуальних машин, то операційна система може вибрати не ту. У цьому випадку надійніше перетягти файл образу на значок потрібної віртуальної машини чи скористатися командним рядком.

Загальна схема команди запуску Pharo з консолі має вигляд:

```
<Pharo executable> <path to Pharo image>
```

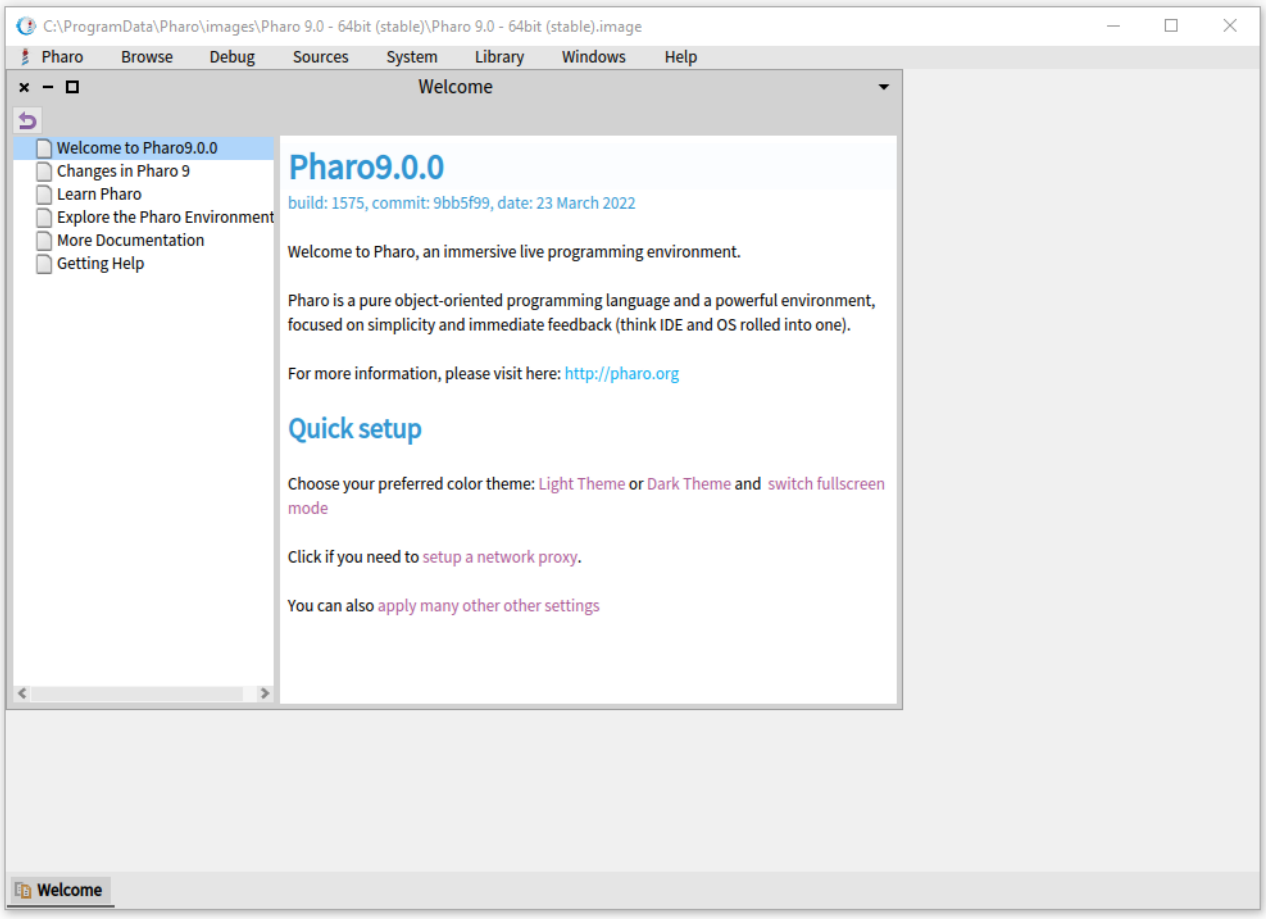

Рис. 2.2. Вікно Pharo після першого запуску

Командний рядок Linux

Припустимо, що ми перебуваємо в папці з файлами Pharo, створеній скриптом автоматичного встановлення.

```
./pharo Pharo.image
```

Командний рядок macOS

Припустимо знову, що Pharo встановили за допомогою скрипту автоматичного встановлення.

```
./pharo Pharo.image
```

Командний рядок Windows

Ми вже зазначали, що скрипти автоматичного встановлення не працюють у Windows, тому ви мусите власноруч завантажити відповідні файли ВМ та образу. Проте запуск також є дуже простим.

```
Pharo.exe Pharo.image
```

Після вдалого запуску ви мали б побачити вікно як на рис. 2.2.

2.5. Зберігання, завершення та повторний запуск сесії Pharo

Вийти з Pharo можна будь-коли. Для цього просто закрийте його вікно так само, як закриваєте інші застосунки. Крім того, можна використати одну з команд *Pharo/Save and quit* або *Pharo/Quit* меню застосунку чи Головного меню Pharo (World Menu), яке з'являється після клацання довільною кнопкою на вільній частині вікна застосунку.

Якщо ви обрали звичайне завершення, то Pharo перепитає вас, чи хочете зберегти образ системи. Якщо ви збережете образ Pharo, то після наступного запуску побачите його *точно в тому стані*, в якому завершили роботу: усі запущені програми, відкриті вікна в тих самих координатах. Так відбувається тому, що файл образу зберігає *всі* об'єкти (редагований текст, розташування вікон, додані методи чи класи – адже *всі* вони є об'єктами), які Pharo завантажує у пам'ять віртуальної машини. Так при виході з Pharo *нічого* не втрачається.

Коли ви запускаєте Pharo вперше, ВМ завантажує вказаний файл образу. Він містить миттєвий знімок багатьох об'єктів, включно з величезним обсягом відкомпільованого коду й інструментами програмування (кожен з яких є об'єктом). Під час роботи з Pharo ви надсилатимете повідомлення цим об'єктам, створюватимете нові об'єкти (за допомогою надсилання повідомлень об'єктам!), деякі з них переставатимуть існувати, а їхня пам'ять буде використана повторно після автоматичного збирання сміття.

Закриваючи Pharo, ви зберігатимете знімок усіх своїх об'єктів. У результаті успішного збереження попередній *.image* файл буде перезаписано новим образом системи. Звичайно, ви можете зберегти образ у файлі з новим іменем.

Як ми вже говорили, *.image* файл працює в парі з *.changes* файлом. Коли ви зберігаєте образ системи, *.image* файл оновлюється миттєвим знімком вашого запущеного Pharo, а *.changes* файл доповнюється журналом усіх змін вихідного коду, зроблених вами після збереження попереднього образу. У більшості випадків ви можете взагалі не

турбуватися про цей файл. Проте, як ми скоро побачимо, *.changes* файл може бути дуже корисним для виправлення помилок чи для відновлення втрачених змін. Але про це – згодом!

Може скластися враження, що образ є ключовим механізмом для зберігання програмних проєктів і керування ними, але це не так. Незабаром ми ознайомимося з набагато кращим засобом для керування кодом та організації командної розробки програмного забезпечення. Образи дуже корисні, але ви звикнете не турбуватися особливо про створення і вилучення образів, оскільки засоби контролю версій, наприклад такі, як менеджер пакетів Iceberg, пропонують набагато ліпші способи керування версіями та поширення коду програмного забезпечення серед команди розробників. Крім того, якщо вам потрібно зберігати об'єкти, можете використати кілька спеціальних систем: *Fuel* (швидкий бінарний серіалізатор об'єктів), або *STON* (текстовий серіалізатор об'єктів), або, навіть, базу даних.

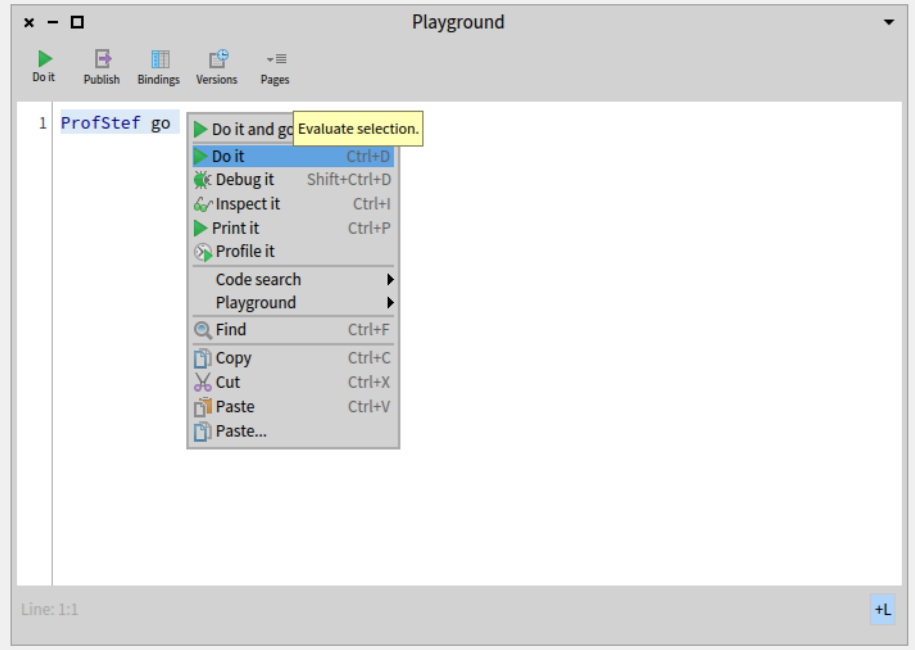

Рис. 2.3. Виконати вираз можна командою «*Do it*» контекстного меню

2.6. Для швидких і нетерплячих

Іноді може видатися нудним читання довгих розділів про систему, наділену настільки багатьма інструментами, як Pharo. Якщо на вашому комп'ютері уже встановлена робоча система Pharo, і ви трохи уявляєте, як все працює, можете захотіти вивчити основи синтаксису за допомогою *ProfStef*, а потім перейти до наступного розділу – посібника для визначення класу простого лічильника і пов'язаних з ним модульних тестів. Ви також можете переглянути відповідне відео на http://mooc.pharo.org.

Щоб запустити *ProfStef*, відкрийте вікно *Playground* відповідною командою розділу *Browse* головного меню, надрукуйте в ньому вираз

```
ProfStef go
```

і виконайте його командою «*Do it*», як показано на рис. 2.3.

Не хвилюйтеся, якщо все це для вас нічого не означає – скоро зрозумієте. Просто читайте далі...

Цей вираз запустить навчальну програму *ProfStef* (рис. 2.4) – чудовий посібник, щоб почати вивчення синтаксису Pharo.

Вітаємо, ви щойно надіслали ваше перше повідомлення! Pharo опирається на концепцію надсилання повідомлень об'єктам. Об'єкти Pharo схожі на солдатів, готових одразу виконати ваш наказ, якщо тільки вони його зрозуміють. Трохи згодом побачимо, як саме об'єкти розпізнають повідомлення.

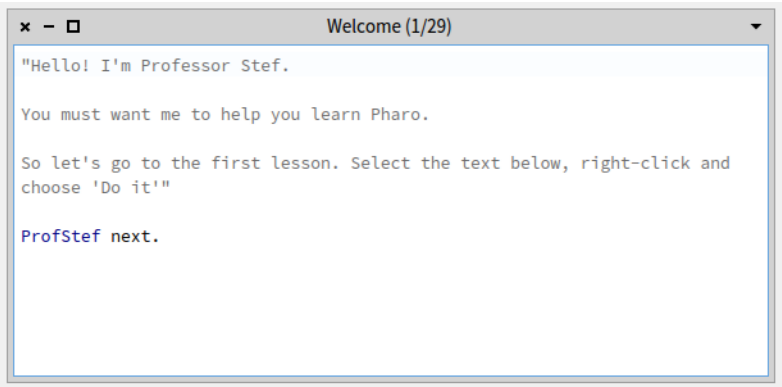

Рис. 2.4. *ProfStef* – простий інтерактивний посібник для вивчення синтаксису Pharo

2.7. Підсумки розділу

- Керувати системами Pharo на вашому комп'ютері можна за допомогою застосунку Pharo Launcher. Завантажити Pharo можна також за допомогою скриптів автоматичного встановлення.
- Робоча система Pharo складається з віртуальної машини, файлу вихідного коду (*.sources*), файлу образу (*.image*) і файлу змін (*.changes*). Останні два з них змінюються, коли ви зберігаєте образ – миттєвий знімок запущеної системи.
- Коли ви відкриваєте образ Pharo, отримуєте середовище в тому ж стані (тобто з тими ж працюючими об'єктами), у якому ви востаннє зберігали образ системи.

Розділ 3
Швидкий огляд Pharo

У цьому розділі ми оглянемо Фаро без заглиблення в деталі, щоб допомогти вам освоїти середовище програмування. Також буде досить багато нагод випробувати написане, тому було б дуже добре під час читання розділу мати під рукою комп'ютер.

Зокрема, ви вивчите різноманітні способи взаємодії з системою і ознайомитеся з деякими основними інструментами. Ви навчитеся також визначати новий метод, створювати об'єкт і надсилати йому повідомлення.

Не забувайте, що це лише швидкий огляд Pharo для випробування середовища програмування. Не зациклюйтеся, коли чогось не зрозумієте. Ви напевне знайдете пояснення в кількох наступних розділах. Вам не обов'язково знати все, або, принаймні, не обов'язково знати все відразу. Занотуйте те, що вас бентежить або інтригує, і продовжуйте читати: все ставатиме дедалі зрозумілішим.

Зауваження. Більшість прикладів вступного матеріалу працюватимуть у Pharo довільної версії, тому можете продовжувати використовувати встановлене середовище Pharo, якщо воно у вас уже є. Проте, оскільки книга написана для Pharo 9.0, то можете помітити відмінності між тим, що написано, і тим, як виглядає чи як поводить себе ваша система.

3.1. Головне меню

Після запуску Pharo ви мали б побачити одне велике вікно, що, можливо, містить декілька менших: вікно консолі, пісочницю тощо (рис. 3.1). Ви можете помітити рядок меню застосунку, але Pharo зазвичай використовує контекстно залежні спадні меню.

Клацання в довільному місці тла головного вікна Pharo відкриває *головне меню*, або *World-меню*. Воно містить низку інструментів, утиліт і налаштувань системи. Як і більшість меню Pharo, головне меню не модальне. Ви можете залишати його на екрані, скільки забажаєте, пришпиливши клацанням на піктограмі канцелярської кнопки, що зображена у правому верхньому кутку вікна меню.

Приділіть трохи уваги дослідженню складу World-меню. Тут ви побачите перелік багатьох головних інструментів Фаро, серед яких є Оглядач класів (System Browser), Пісочниця (Playground), Менеджер пакетів Iceberg та інші. Ми розповімо про них докладніше в наступних розділах.

3.2. Взаємодія з Pharo

Середовище Pharo пропонує користувачеві три способи взаємодії за допомогою мишки чи іншого вказівного пристрою – це різні способи клацання.

Клацнути (click, left-click): натиснути найчастіше вживану кнопку мишки (зазвичай це означає клацнути лівою кнопкою), що для мишки з однією кнопкою означає просте клацання без натискання жодних модифікуючих клавіш. Клацніть, наприклад, у вікні системи, щоб розгорнути головне меню (рис. 3.1).

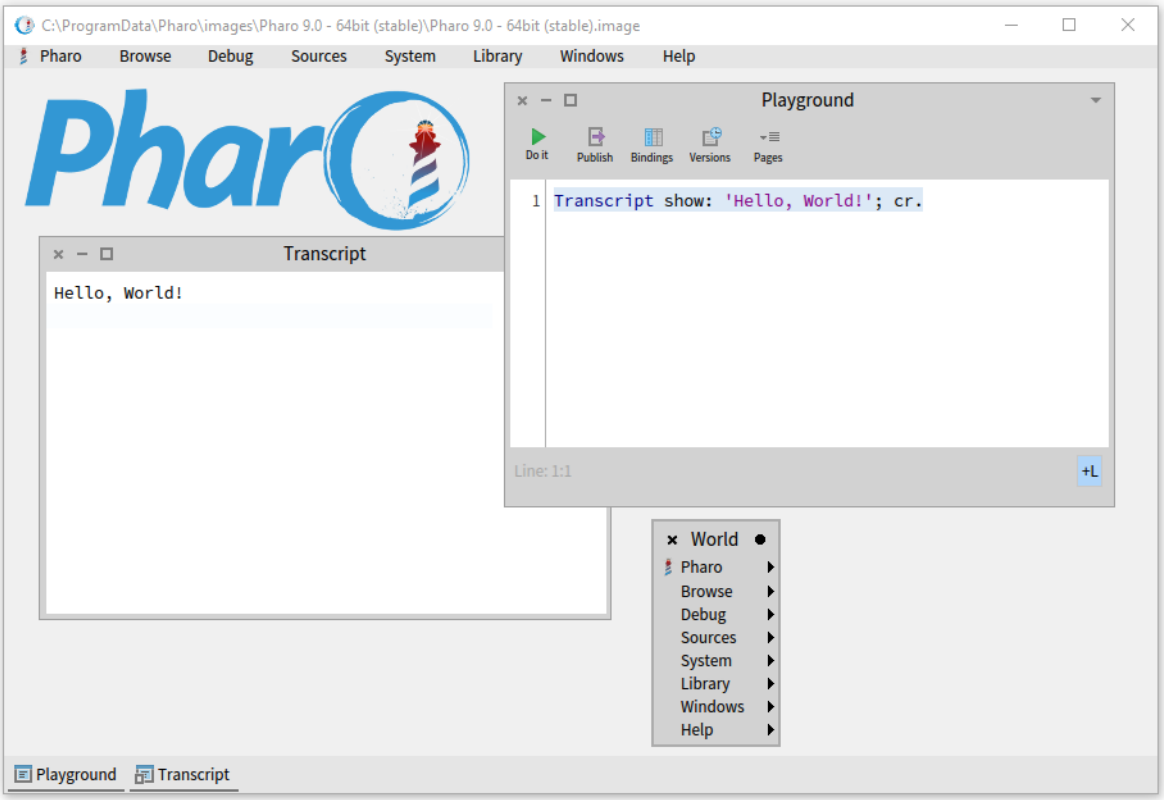

Рис. 3.1. Клацання мишкою відкриває *World-меню*

Контекстно клацнути (action-click, right-click): натиснути іншу часто вживану кнопку мишки, якою зазвичай розгортають контекстне меню (праву кнопку). Таке меню може містити різний перелік команд залежно від місця, на яке вказувала мишка в момент клацання (див. рис. 3.2). Якщо у вашої мишки тільки одна кнопка, вам потрібно буде задати конфігурацію модифікуючих клавіш [*Ctrl*] так, щоб клацання мишкою одночасно з їх натисканням розгортало контекстне меню.

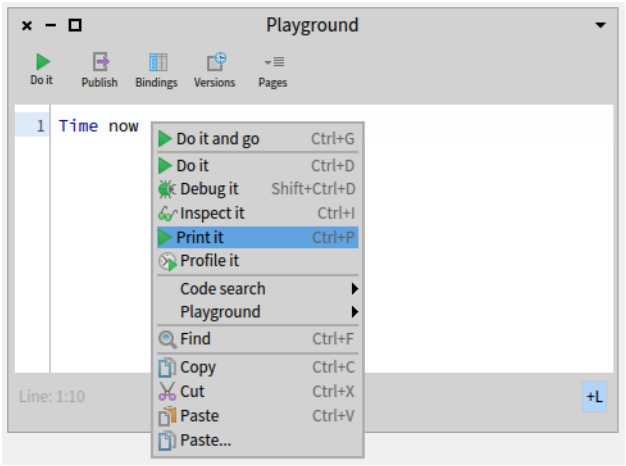

Рис. 3.2. Клацання правою кнопкою розгортає контекстне меню

Метаклацнути (meta-click) – особливий спосіб клацнути у Pharo. Графічний інтерфейс користувача середовища Pharo збудований з об'єктів бібліотеки Morphic. Всі вікна, написи, меню, які ми бачимо на екрані Pharo, є *морфами*. Метаклацанням на будь-якому об'єкті, зображеному у вікні образу, викликають його «морфовий ореол» або меню-ореол – набір маніпуляторів, розташованих навколо об'єкта, які використовують для виконання дій із самим екранним об'єктом. Наприклад, для обертання, зміни розміру,

кольору тощо (див. рис. 3.3). Якщо ви затримаєте вказівник мишки над маніпулятором, то з'явиться спливаюча підказка з поясненням про призначення цього маніпулятора. Спосіб, яким можна зробити meta-click, залежить від вашої операційної системи. Потрібно клацнути мишкою у поєднанні з натисканням модифікуючих клавіш [*Shift* + *Alt*] чи [*Shift* + *Ctrl*] у Windows та Linux, або з [*Shift* + *Option*] у macOS.

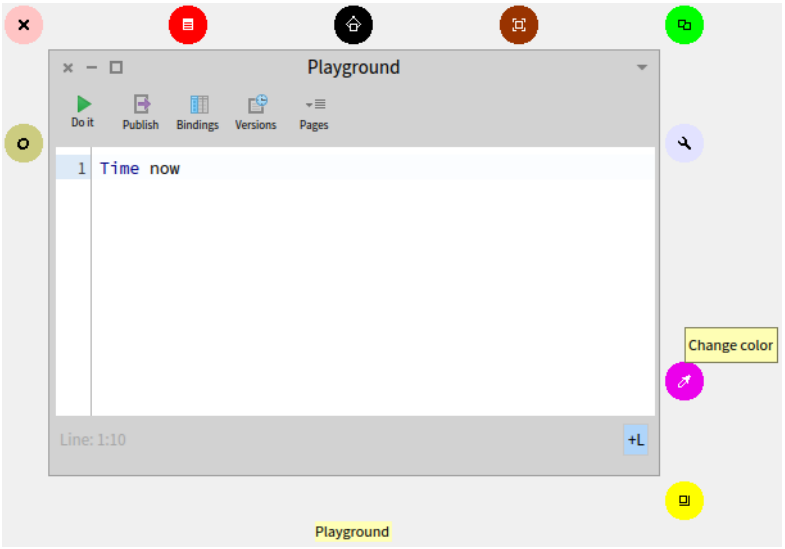

Рис. 3.3. Метаклацання відкриває меню-ореол

Будемо вважати, що зазвичай мишку сконфігуровано так, що *клацнути* можна лівою кнопкою мишки, *контекстно клацнути* – правою, а *метаклацнути* – лівою у поєднанні з модифікуючими клавішами.

Від перекладача. Якщо Pharo працює у вас в операційній системі Windows, то помітите, що звичайне і контекстне клацання можуть діяти однаково. Наприклад, обома кнопками мишки можна вибрати команду головного меню Pharo, закрити вікно. Метаклацання можна виконати лівою кнопкою у поєднанні з [*Alt* + *Shift*].

Про лексикон

Спілкуючись з програмістами на Pharo, ви швидко помітите, що вони ніколи не вживають висловів «*виконати операцію*» чи «*викликати метод*», як це буває в інших мовах програмування. Натомість вони говорять «*надіслати повідомлення*». Це відображає ту ідею, що об'єкти самі відповідають за власну поведінку, а пов'язані з повідомленням методи вибираються динамічно. Коли надсилають повідомлення, то не адресант, а сам об'єкт вибирає метод, виконанням якого потрібно реагувати на отримане повідомлення. Зазвичай цей метод називається так само, як повідомлення.

Як користувачеві, вам не потрібно розуміти, як працює кожне повідомлення, єдине, що вам треба знати, це те, які повідомлення доступні для об'єктів, які вас цікавлять. Так об'єкт може приховати свою складність, а написання програми може залишатися якнайпростішим без втрати гнучкості.

Незабаром ми покажемо, як знайти доступні повідомлення для кожного об'єкта.

3.3. Вікна Playground і Transcript

Давайте виконаємо кілька простих вправ, щоб освоїтися в новому середовищі.

1. Закрийте всі відкриті у Pharo вікна.
2. Знайдіть у меню і відкрийте вікно Transcript – консоль системи – і робоче вікно, або пісочницю Playground. Обидва можна відкрити за допомогою підменю *World* > *Browse* > ... .
3. Розташуйте вікна так, щоб Пісочниця розташувалася поверх Консолі, але видно було обох (див. рис. 3.4).

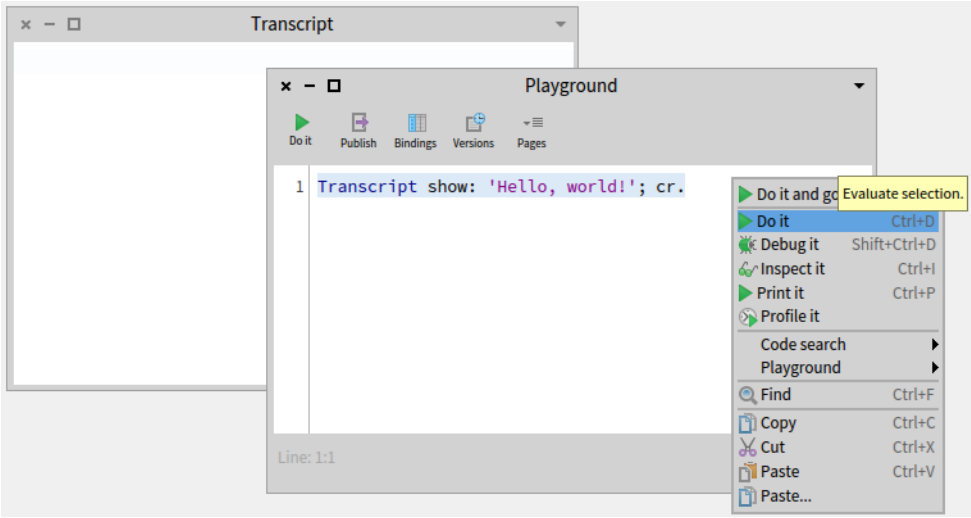

Рис. 3.4. Виконання виразу: виведення рядка тексту в Transcript

Ви можете змінювати розмір вікна за допомогою перетягування одного з його кутів. З усіх відкритих вікон лише одне активне, воно розташоване поверх усіх інших, а рамка клієнтської частини виділена кольором.

Transcript – це об'єкт, який часто використовують для зберігання журналу системних повідомлень. Це така консоль системи.

Пісочниця (*Playground*) зручна для введення і випробування фрагментів коду. Ви також можете використовувати її для звичайного введення тексту, який би хотіли запам'ятати. Наприклад, для створення списків «до виконання» або інструкцій для будь-кого, хто використовуватиме ваш образ системи.

Надрукуйте у Пісочниці такий текст:

```
Transcript show: 'hello world'; cr.
```

Випробуйте подвійне клацання в різних місцях щойно надрукованого тексту. Спостерігайте, що буде позначено: слово, цілий рядок або весь текст залежно від позиції клацання – в межах слова, в кінці рядка, чи в кінці виразу. Зокрема, якщо ви встановите курсор перед першою літерою виразу або після останньої і двічі клацнете, то позначено буде весь абзац.

Позначте весь текст у Пісочниці, контекстно клацніть і виберіть «*Do it*» (як на рис. 3.4). Зверніть увагу, як привітання *hello world* з'явиться у Консолі. Повторіть виконання кілька разів.

3.4. Гарячі клавіші

Щоб виконати вираз, ви не мусите щоразу контекстно клацати. Натомість можете використати комбінації гарячих клавіш, зазначені в пунктах меню. Попри те, що Pharo схоже на середовище, кероване мишею, воно пам'ятає більше двохсот комбінацій клавіш для взаємодії з різноманітними інструментами та надає користувачеві можливість призначити нову комбінацію для будь-якого з 143 000 методів, записаних у образі Pharo.

Залежно від вашої операційної системи, до складу комбінації клавіш входить одна з модифікуючих клавіш: [*Control*], [*Alt*], або [*Command*]. Надалі в книзі будемо позначати її [*Cmd*], тому, коли ви прочитаєте щось на зразок [*Cmd* + *D*], просто замініть наше позначення відповідною до вашої операційної системи клавішею. Наприклад, у Windows це означатиме [*Ctrl* + *D*], а в Linux – або [*Ctrl* + *D*], або [*Alt* + *D*]. Зауважимо також, що у Windows комбінація діятиме незалежно від регістра клавіатури.

Ви мали б помітити в контекстному меню, крім команди «*Do it*», також «*Do it and go*», «*Print it*», «*Inspect it*» та кілька інших. Поглянемо на кожну з них.

3.5. Виконання проти виведення

Надрукуйте вираз *3* + *4* у Пісочниці і застосуйте до нього «*Do it*» за допомогою комбінації клавіш [*Cmd* + *D*].

Не дивуйтеся, що нічого не відбулося! Ви просто надіслали об'єкту *3* повідомлення «+» з аргументом *4*. У результаті виконання відповідного методу було отримано результат, число *7*, яке повернулося до Playground, але вікно не знало, що з ним робити, тому нічого і не показало. Якщо ви хочете бачити результат, то використовуйте замість «*Do it*» команду «*Print it*». Під час її виконання вираз компілюється, компільований код виконується, отриманому результату надсилається повідомлення *printString*, а вікно відображає отриманий рядок.

Позначте *3* + *4* і виберіть «*Print it*» [*Cmd* + *P*]. Цього разу ми побачимо результат, на який сподівалися:

```
3 + 4
>>> 7
```

Ми будемо використовувати в цій книзі позначення «>>>», щоби вказати на результат, отриманий для певного виразу Pharo за допомогою «*Print it*».

3.6. Інспектування

Позначте вираз *3* + *4* або помістіть курсор у рядок з ним і виберіть команду «*Inspect it*» [*Cmd* + *I*].

Мало би відкритися нове вікно з заголовком «*Inspector on 7*», як показано на рис. 3.5. Інспектор є надзвичайно корисним інструментом, що дає вам змогу переглядати будь-який об'єкт у системі та взаємодіяти з ним. Підзаголовок вікна «*a SmallInteger (7)*» повідомляє нам, що *7* є екземпляром класу *SmallInteger*. Верхня панель відображає змінні екземпляра та їхні значення (у числа це єдина змінна *self*). Нижню панель можна використати для надсилання повідомлень екземплярові. Надрукуйте в ній «*self*

squared» і виберіть «*Print it*». Результат виконання виразу з'явиться одразу в нижній панелі інспектора.

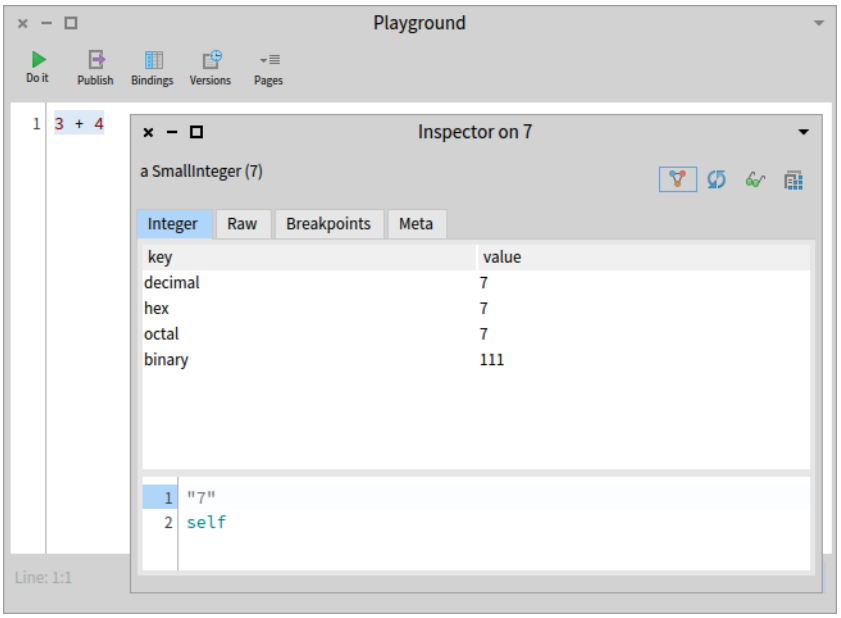

Рис. 3.5. Інспектування звичайного числа

Інспектор містить специфічні сторінки для відображення різноманітної інформації про екземпляр і різних його виглядів, залежно від типу об'єкта, який ви інспектуєте. Спробуйте викликати інспектора для фрагмента «*Morph new openInWorld*». Ви мали б отримати щось схоже до зображеного на рис. 3.6.

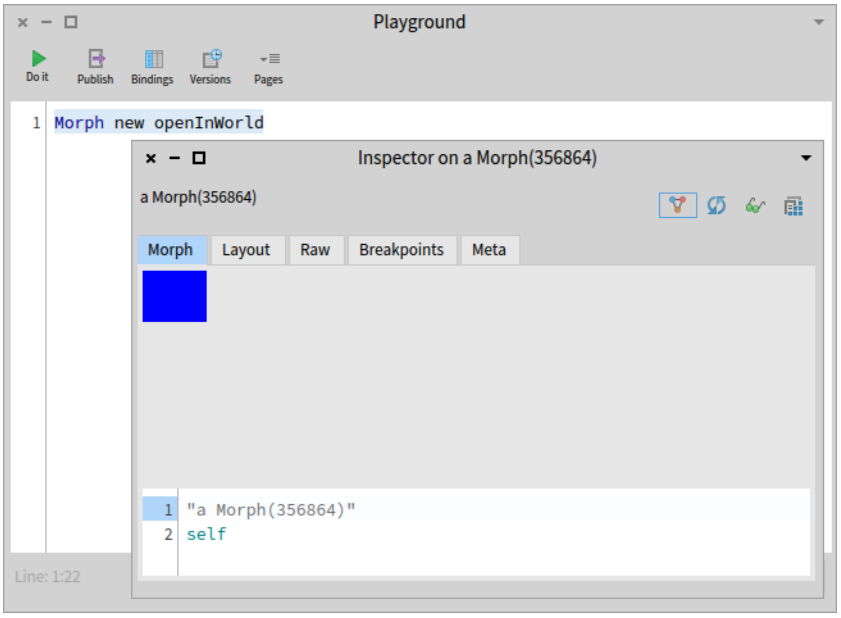

Рис. 3.6. Інспектування екземпляра Morph

3.7. Інші дії

Інші команди контекстного меню Пісочниці.

- «*Do it and go*» додатково відкриває у правій частині Пісочниці сторінку інспектора. Ви можете використовувати її для дослідження структури об'єкта. Випробуйте з цією командою попередній вираз «*Morph new openInWorld*».

- «*Debug it*» відкриває налагоджувач коду.
- «*Profile it*» будує часовий профіль коду за допомогою відповідного інструмента Pharo. Він показує скільки часу займає надсилання кожного повідомлення.
- «*Code search*» надає доступ до кількох реалізованих в Оглядачі класів засобів пошуку програмного коду. Серед них пошук певного виразу у тексті всіх методів, пошук відправників повідомлення та об'єктів, здатних відповісти на нього тощо.

Від перекладача. Пісочниця (Playground) з'явилася порівняно недавно. Замість неї у попередніх версіях Pharo для випробовування фраґментів коду використовували Робоче вікно (Workspace). Воно не має засобів інспектора, але має одну суттєву перевагу: у меню Робочого вікна є команди для роботи з файлами *Open* і *Save as...*, за допомогою яких легко можна зберегти чи завантажити довільний текст, у тому числі й оголошення методів чи класів. Щоб відкрити Робоче вікно, виконайте вираз «*Workspace open*» або «*Workspace openLabel: 'My Great Work'*».

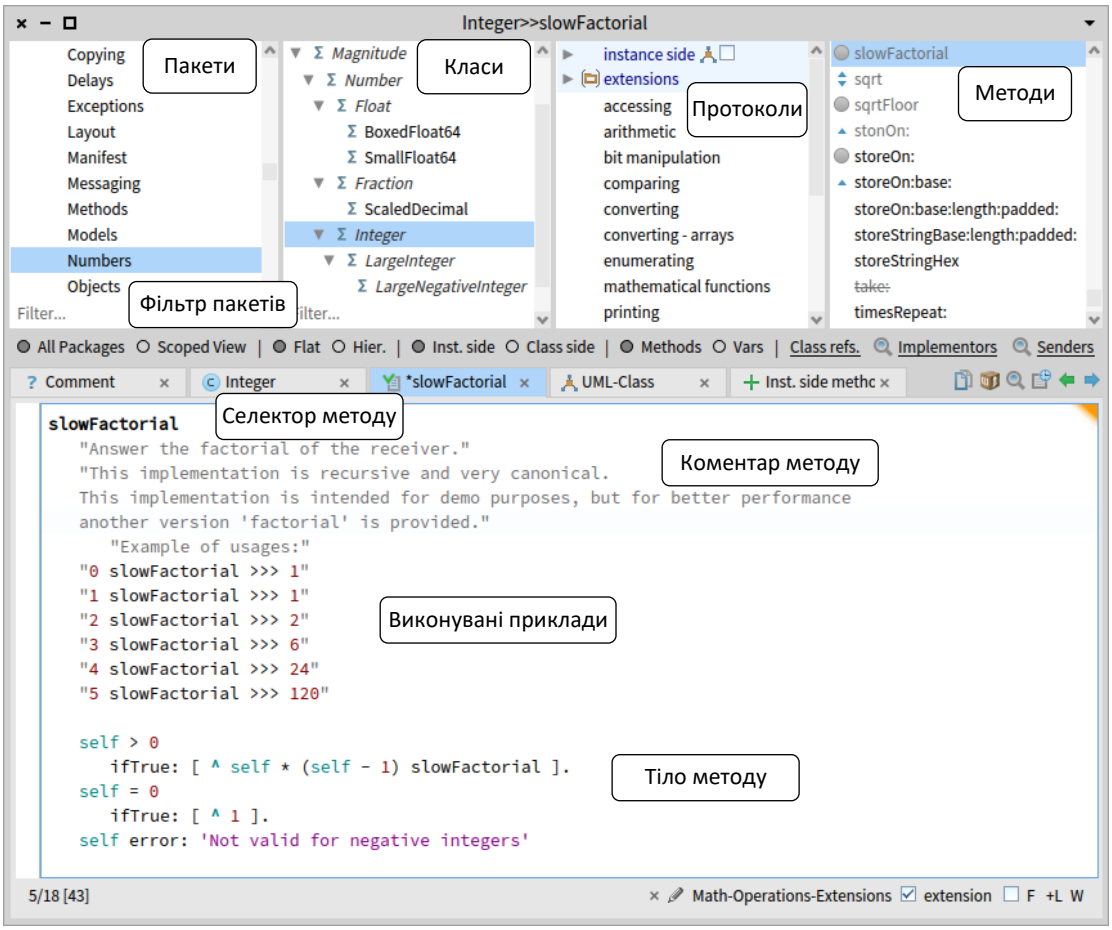

Рис. 3.7. Системний Оглядач відображає метод *slowFactorial* класу *Integer*

3.8. Оглядач класів Calypso

Системний оглядач (*System Browser*), відомий також як Оглядач класів, є одним з головних інструментів для програмування. Ми побачимо згодом, що у Pharo функціонує кілька цікавих оглядачів, проте *System Browser* є основним, його ви знайдете в кожному образі системи. Поточна реалізація Оглядача класів називається *Calypso*. Попередня версія називалася *Nautilus*.

Оглядача легко викликати за допомогою меню *World* > *Browse* > *System Browser*, або комбінацією клавіш [*Cmd* + *O,B*], або програмно, виконавши вираз «*ClyFullBrowserMorph open*». У вікні Оглядача ви побачите спочатку лише перелік пакетів класів. Відкрити можна будь-який з них простим клацанням, але пошук потрібного класу чи методу серед тисяч інших справа невдячна, тому поговоримо про швидші способи.

Відкривання Оглядача на заданому класі чи методі

Ви можете переглянути код будь-якого класу, якщо відкриєте його в Оглядачі класів за допомогою виразу «*ClyFullBrowserMorph openOnClass: ClassName*», де *ClassName* – ім'я класу, що вас цікавить. Ще конкретніше завдання – це відкривання в Оглядачі класів певного методу певного класу.

Відкривання Оглядача на конкретному методі не є повсякденною практикою. Для відшукання та перегляду методів можемо використати потужніші засоби ніж оглядач класів, але зараз для завершення вправи виконайте, будь ласка, такий фрагмент:

```
ClyFullBrowserMorph openOnMethod: Integer>>#slowFactorial
```

Він відкриє Оглядач класів на методі *slowFactorial*. Ми мали б отримати вікно як на рис. 3.7. Заголовок вікна «*Integer>>#slowFactorial*» вказує, що ми переглядаємо клас *Integer* і його метод *slowFactorial*. На рисунку показано різні сутності, які відображає Оглядач: пакети, класи, протоколи, методи та визначення методу.

За замовчуванням у Pharo системним Оглядачем є Calypso. Проте, як ми вже зазначали, в середовищі можна інсталювати й інші оглядачі. Кожен системний оглядач може мати свій графічний інтерфейс, що відрізнятиметься від інтерфейсу Calypso. Надалі ми вживатимемо терміни *Оглядач (класів)*, *Системний оглядач* та *Calypso* як синоніми.

3.9. Підсумки розділу

У цьому розділі ми пробіглися деякими з основних інструментів середовища Pharo, які ви будете використовувати для програмування в ньому. Також ознайомилися трохи з синтаксисом Pharo, хоча, можливо, ви ще не розумієте його повністю. Ось невеликий підсумок того, чого ми навчилися.

- Головне меню – *World menu* – системи викликають клацанням на тлі вікна Pharo. Через нього запускають різноманітні засоби системи.
- Робоче вікно *Playground* – це місце для написання і випробування фрагментів коду. Його можна використовувати також для зберігання довільного тексту.
- Для взаємодії з середовищем зручно використовувати гарячі клавіші, наприклад, для виконання коду в Playground чи іншому засобі. Найважливіші комбінації клавіш: *Do it* [*Cmd* + *D*], *Print it* [*Cmd* + *P*], *Inspect it* [*Cmd* + *I*] та *Browse it* [*Cmd* + *B*].
- Оглядач класів є головним засобом перегляду класів Pharo та розробки нового коду.

Розділ 4
Пошук інформації у Pharo

У цьому короткому розділі опишемо кілька способів відшукання інформації у Pharo.

4.1. Мандрівка системою за допомогою Оглядача

У системі Pharo можна швидко дістатися до оголошення потрібного вам класу чи методу, справді, дуже швидко. Згодом ми розповімо про спеціальний інструмент – *Spotter* – мабуть, найшвидший спосіб відшукати будь-який об'єкт у Pharo. Але доки вчимося, давайте підемо довшим шляхом і використаємо лише Системний оглядач, щоб відшукати метод *printString*, визначений в класі *Object*. Наприкінці пошуку Оглядач виглядатиме, як зображено на рис. 4.1. Виконайте таку послідовність кроків.

- **Відкрийте Оглядач класів** або за допомогою World-меню, або комбінацією клавіш [*Cmd* + *O,B*]. Коли нове вікно Оглядача відкриється, всі панелі, крім крайньої лівої, будуть порожніми. Вона відображає перелік усіх відомих *пакетів*, які містять групи пов'язаних класів.

- **Відфільтруйте пакети**: надрукуйте частину імені пакета в рядку фільтра унизу лівої панелі. Він відбирає для відображення у ній пакети, чиї імена містять введений рядок. Надрукуйте, наприклад, «*Kern*».

- **Розгорніть пакет *Kernel* і виберіть елемент *Objects*.** Якщо вибрати пакет, то друга панель відобразить список усіх *класів*, що входять до цього пакета. Ви мали б побачити ієрархію класу *ProtoObject*.

- **Виберіть клас *Object*.** Якщо вибрати клас, то дві панелі, що залишилися, заповняться даними. Третя панель відображає *протоколи* вибраного класу. Вони зручно групують пов'язані між собою методи. Про протоколи йтиметься згодом у цій книзі. Якщо жодного протоколу не вибрано, то четверта панель відображає список усіх методів.

- **Виберіть протокол *printing*.** Можливо, вам доведеться прокрутити список протоколів, щоб знайти його. Ви також можете клацнути на панелі протоколів і почати друкувати «*pr*» для автоматичного пошуку пунктів списку, що починаються цими літерами. Урешті виберіть *printing*, і ви побачите в четвертій панелі лише ті методи, які стосуються цього протоколу.

- **Виберіть метод *printString*.** Тепер ми бачимо в нижній панелі вихідний код методу *printString*. Його поділяють усі об'єкти системи (крім тих, що його перевизначають).

Існує набагато кращий спосіб знайти метод, наприклад, просто надрукуйте його назву в робочому вікні, контекстно клацніть на ній і виберіть з меню пункт *Code search > Implementors of it*, або просто використайте комбінацію клавіш [*Cmd* + *M*]. Ви отримаєте повний перелік класів, у яких реалізовано метод. Кожен з них можна відкрити в Оглядачі класів, знову скориставшись контекстним меню.

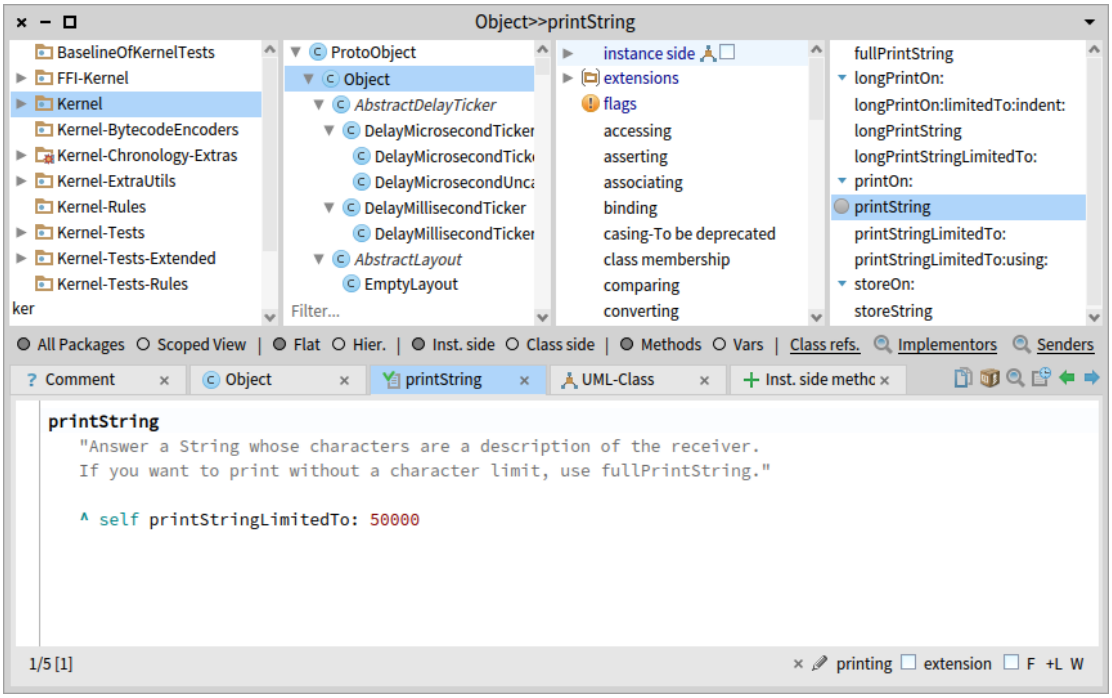

Рис. 4.1. Системний оглядач демонструє текст методу *printString* класу *Object*

4.2. Відшукання класів

Є кілька способів відшукання класів у Pharo. Перший з них, як ми щойно побачили, використовує Системний оглядач для переходу до визначення класу. Для цього потрібно знати (або вгадати), до якого пакета належить клас.

Другий спосіб полягає в тому, щоб надіслати класові повідомлення *browse* – попросити його відкрити себе в Оглядачі. Припустимо, що ми хочемо переглянути клас *Point*.

- **Використання повідомлення *browse***. Надрукуйте «*Point browse*» у Пісочниці і застосуйте «*Do it*». Оглядач відкриється на класі *Point*. Так само ви можете виконати «*(10@20) browse*», оскільки *10@20* є екземпляром класу *Point*.
- **Використання [*Cmd + B*] для перегляду**. У будь-якому текстовому вікні можна використовувати комбінацію [*Cmd* + *B*] (browse), щоб викликати Оглядач. Для цього просто позначають слово і натискають [*Cmd* + *B*]. Випробуйте дію комбінації на слові «*Point*».

Зверніть увагу на те, що коли в Оглядачі класів позначено клас *Point*, але не вибрано ні протоколу, ні методу, то в нижній панелі замість визначення методу бачимо визначення класу (рис. 4.2). Це не що інше, як звичайне повідомлення до батьківського класу з проханням створити підклас. Тут ми бачимо, що клас *Object* попросили створити підклас, який називається *Point*, має дві змінні екземпляра, не має змінних класу і належить до пакета *Kernel-BasicObjects*. Вкладка *Comment* містить опис класу.

Додатково система підтримує такі комбінації клавіш з мишкою:

- [*Cmd*]+*Click* на слові ([*Alt*]+*Right-click* в ОС Windows та Linux). Якщо слово – ім'я класу, то відкриється визначення класу; якщо слово – селектор повідомлення, записаного у виразі, або в тілі методу, то відкриється список класів, що реалізують відповідні методи;

- [*Shift-Cmd*]+*Click* на слові ([*Shift-Alt*]+*Right-click* в ОС Windows та Linux). Якщо слово – ім'я класу, то відкриється список посилань на нього; якщо слово – селектор повідомлення, записаного у виразі, чи в тілі методу, то відкриється список відправників цього повідомлення.

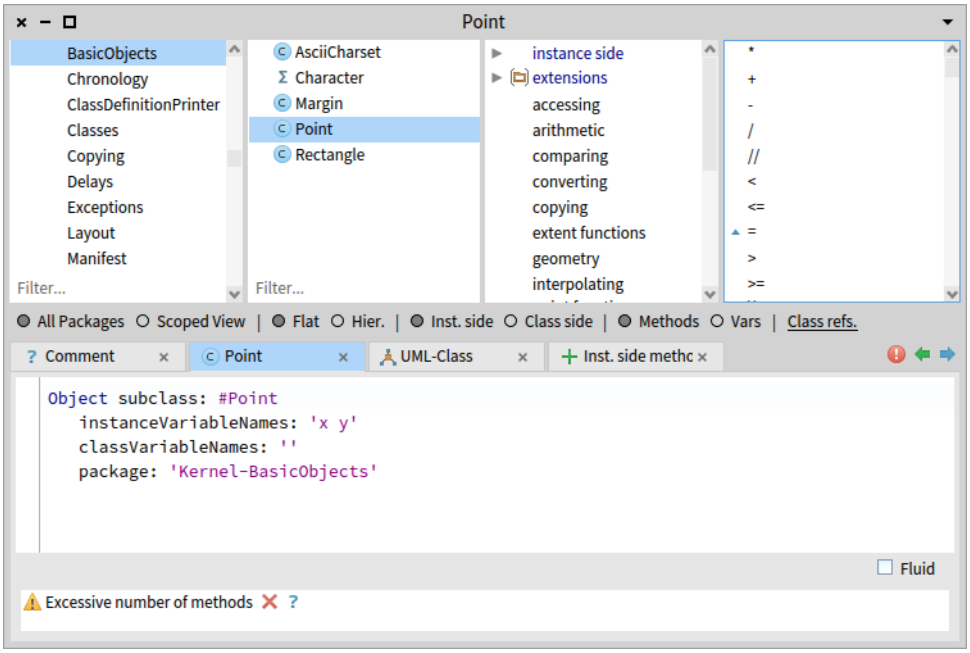

Рис. 4.2. Системний оглядач відкрито на визначенні класу *Point*

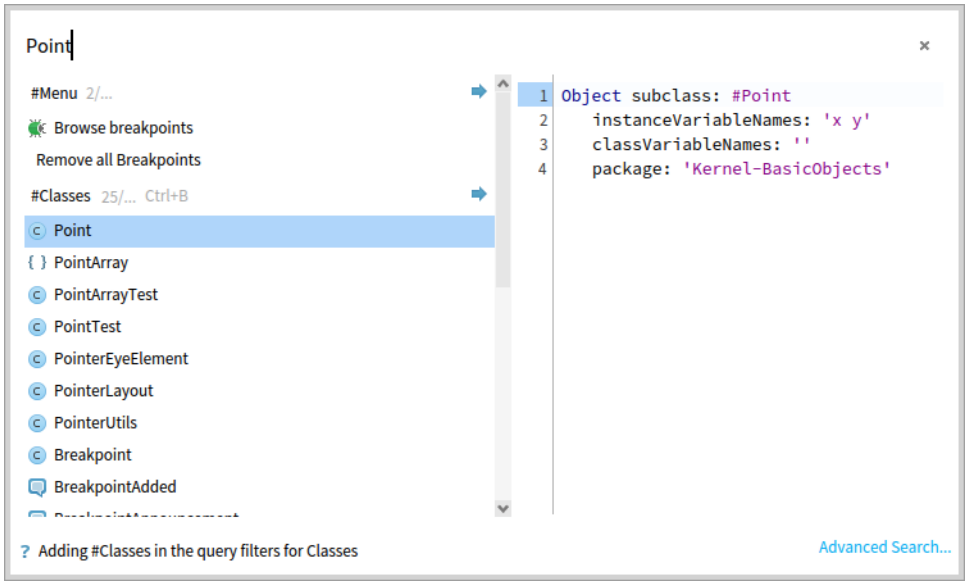

Рис. 4.3. Відшукання класу *Point* за допомогою Навідника

Використання *Spotter*

Найшвидший і, можливо, найкрутіший спосіб відшукати клас – використати *Spotter*. (Spotter можна перекласти як *Навідник* або *Нишпорка*. Ми використовуватимемо назву Навідник або англомовний варіант). Його відкривають натисканням [*Shift-Enter*] (клавіша [*Enter*] основної клавіатури, не додаткової). Spotter дуже потужний інструмент для відшукання класів, методів і виконання багатьох пов'язаних дій. На рис. 4.3 показано процес пошуку слова «*Point*».

Навідник пропонує кілька можливих варіантів пошуку. Ви можете задати *категорію*, яка вас цікавить. Наприклад, для пошуку тільки класів уведіть «*#Classes*» перед

шуканим словом. Якщо не вказувати категорію, Навідник знайде об'єкти всіх категорій (рис. 4.3).

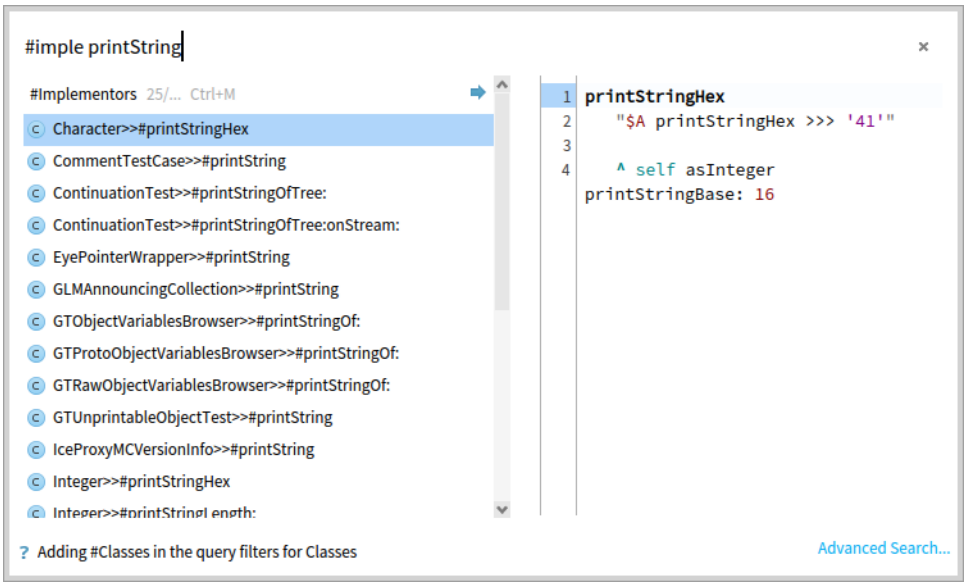

Рис. 4.4. Перегляд реалізації методів, чий селектор містить *printString*

З рис. 4.4 видно, як можна попросити Навідника показати усі реалізації методів, що відповідають заданому повідомленню. Як бачимо, назву категорії можна вводити не повністю. Також пошук не чутливий до регістра введеного тексту. Інші категорії такі:

- *#Menu* – відповідні пункти головного меню;
- *#Packages* – відповідні пакети класів, інстальовані в системі;
- *#Implementors* – реалізація методів, що відповідають уведеному селектору;
- *#Senders* – об'єкти, що надсилають відповідне повідомлення;
- *#Help* – відповідні документи з довідкової системи Pharo.

Щоб задати категорію пошуку, можна ввести лише її перші літери, наприклад, «*#sen printOn:*» знайде всі об'єкти, що надсилають повідомлення *printOn:*.

Результати пошуку в Spotter поміщено в список і поділено за категоріями, наприклад, класи зачислено до категорії *#Classes*, методи – до *#Implementors*, розділи довідки – до *#Help* тощо. Користувач може перебирати пункти списку за допомогою клавіш зі стрілками [*Вгору*]/[*Вниз*], водночас рядок пошуку утримує фокус уведення, тому користувач без зусиль може переходити від вибору пунктів списку до введення нового тексту для пошуку. Натискання клавіші [*Enter*] основної клавіатури на вибраному пункті відкриє Оглядач класів на конкретному вибраному результаті пошуку (а Spotter закриє). Пошук завжди можна завершити клавішею [*Esc*] або клацанням поза межами його вікна.

Від перекладача. Навідник може знайти багато збігів. Справді багато. Серед результатів пошуку він відобразить не більше ніж 25 записів у кожній категорії. Щоб побачити ширший перелік (до 100 записів), клацніть на голубій стрілці, розташованій біля правого краю рядка з назвою категорії. Навідник відкриє розширений список результатів саме в цій категорії. Повернутися назад допоможе зелена стрілка, що з'явиться на місці голубої.

На жаль, у Pharo 9.0 перехід до ширшого переліку може не працювати. Якщо у вашій копії Pharo в тексті методу *OrderedCollection>>spotterItemsFor:* є помилка, то після клацання на голубій стрілці у вас відкриється вікно налагоджувача з текстом «*Instance of OrderedCollection did not understand #collectionSizeThreshold*» у заголовку і підсвіченим відповідним повідомленням у коді методу. Не засмучуйтеся. Навіть добре, що так сталося: маєте нагоду полагодити Pharo! Виправте у вікні налагоджувача «*self collectionSizeThreshold*» на «*self gtCollectionSizeThreshold*», натисніть [*Ctrl+S*], щоб зберегти та відкомпілювати зміни, і клацніть на кнопці *Proceed*, щоб продовжити роботу зі Spotter. Більше вас ця помилка не турбуватиме! Якщо не забудете зберегти образ Pharo перед завершенням роботи.

Використання команди «*Find class*» в Системному оглядачі

У Системному оглядачі ви також можете шукати клас за його іменем. Припустимо, що ви, наприклад, шукаєте невідомий клас, який моделює дату і час.

Відкрийте Оглядача, а тоді – вікно пошуку класів за допомогою комбінації [*Cmd* + *F*] або за допомогою команди «*Find Class*» контекстного меню панелі пакетів. Далі в рядку введення вікна пошуку надрукуйте «*time*». Вікно відобразить список класів, до імен яких входить підрядок «*time*» (без огляду на регістр). Тепер можете вибрати один з них. Для цього мишкою прокрутіть список до потрібного рядка, наприклад, *Time* і клацніть на ньому. Те саме можна зробити за допомогою клавіатури: після введення пошукового слова клавішами зі стрілками знайдіть потрібний рядок. Щоб відкрити вибраний клас, достатньо натиснути клавішу [*Enter*], або клацнути на кнопці ***OK***.

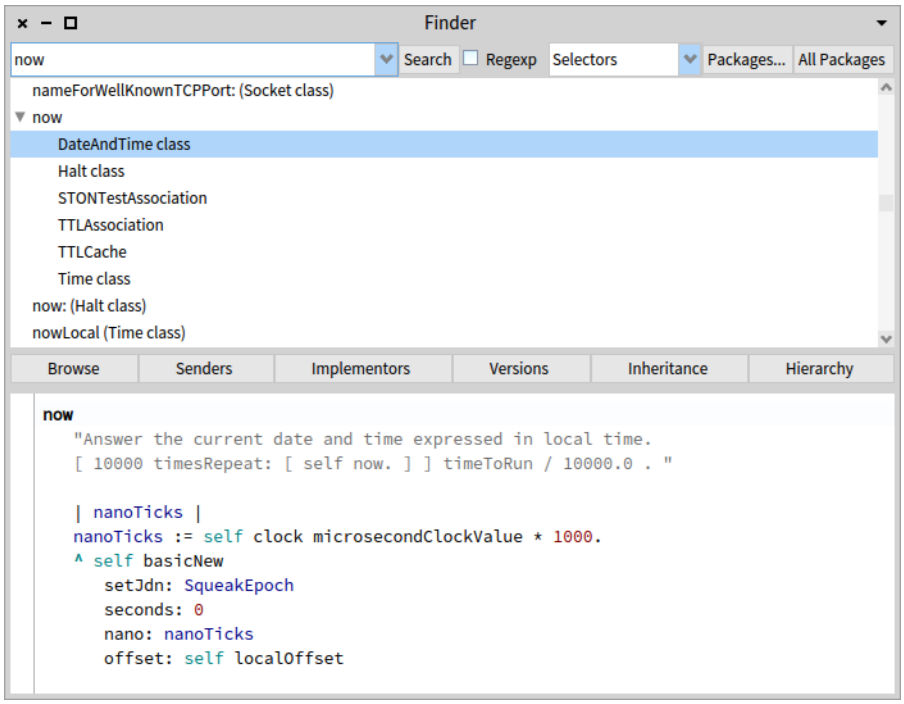

Рис. 4.5. Інструмент Шукач

4.3. Відшукання методів

Іноді ім'я методу чи його частини можна вгадати легше, ніж ім'я класу. Наприклад, якщо вас цікавить поточний час, то ви могли б сподіватися, що метод називатиметься «*now*» або міститиме «*now*» як підрядок. Але де він міг би бути? Spotter і Finder можуть вам допомогти. (Finder – Шукач, ще один інструмент відшукання методів і класів).

За допомогою Навідника

Як ми вже зазначали, Навідник вміє знаходити і методи. Щоб відшукати метод, ви можете використати категорію *#Implementors*, просто надрукувавши «*#Implementors aMethodName*». Навідник відобразить усі реалізовані методи зі схожим чи точно таким іменем. Наприклад, якщо ви надрукуєте «*#imp now*», то побачите перелік методів, чиї селектори починаються на «*now*».

За допомогою Шукача

Відкрийте Finder відповідною командою підменю *Browse* головного меню Pharo (у World-меню чи в рядку меню). Уведіть «*now*» в рядок пошуку і клацніть на кнопці ***Search*** (або натисніть [*Enter*]). Ви мали б побачити список результатів, як на рис. 4.5.

Шукач відобразить список усіх імен методів, що містять підрядок «*now*». Щоб швидко прокрутити його власне до методу *now*, перемістіть фокус уведення до списку і натисніть [*n*]. Такий автоматичний пошук працює у всіх вікнах з прокручуванням. Розгорніть пункт «*now*», і ви побачите перелік усіх класів, що реалізують цей метод. Якщо вибрати котрийсь з них, то панель коду внизу вікна відобразить текст методу.

Щоб знайти реалізацію методу лише за точним збігом імені, потрібно в рядок пошуку ввести потрібне ім'я, обрамлене лапками. Наприклад, «"*now*"».

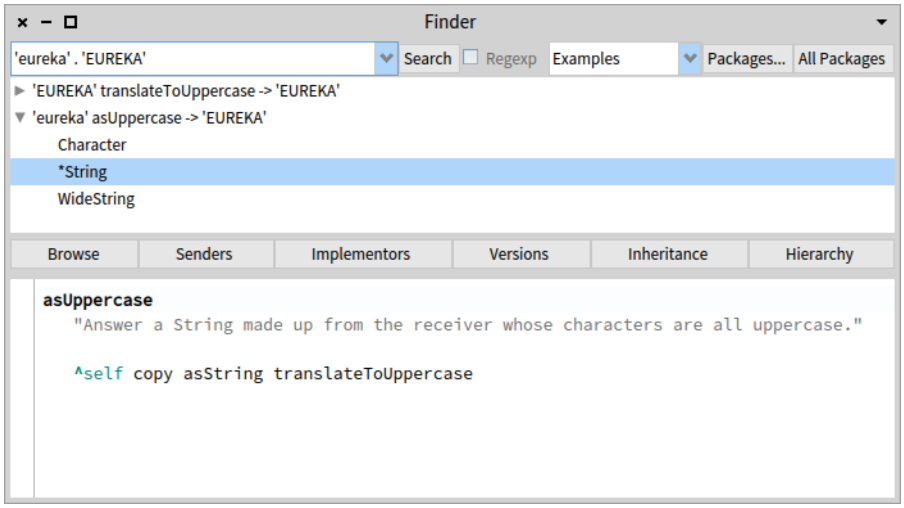

Рис. 4.6. Пошук методу, що перетворює рядок *'eureka'* на рядок *'EUREKA'*

4.4. Пошук методів за зразками

За замовчуванням Шукач налаштовано на відшукання методів: у другому спадному списку, розташованому праворуч вгорі, задано категорію пошуку «*Selectors*». Її можна змінити, наприклад, на «*Classes*» і відшукати класи за фрагментом імені так само, як ми шукали методи. Якщо обрати категорію «*Source*», то Шукач переглядатиме код методів включно з коментарями.

Проте Шукач здатен також на щось справді потужне та унікальне. Часом трапляється так, що ви впевнені в існуванні методу, але гадки не маєте, як би він міг називатися. Ви знаєте, що він мав би робити, проте не знаєте, як його викликати. Шукач і тоді може допомогти! Припустимо, що ви хотіли б знайти метод, який переводить рядок літер до верхнього регістру (наприклад, перетворює *'eureka'* на *'EUREKA'*). Ми можемо описати Шукачеві вхідні дані й очікуваний результат виконання методу, і він спробує знайти для нас такий метод.

Оберіть «*Examples*» у спадному списку категорій пошуку Шукача, надрукуйте «*'eureka' . 'EUREKA'*» в рядку пошуку і натисніть [*Enter*] або клацніть кнопку ***Search*** (не забудьте про апострофи!).

Тоді Шукач запропонує метод, який робить те, що вам потрібно, та покаже список класів, які реалізують метод з таким іменем. У нашому випадку він визначить, що метод «*asUppercase*» є одним з тих, чия дія збігається зі зразком, як показано на рис. 4.6.

Клацніть на трикутнику ліворуч від виразу *'eureka' asUppercase --> 'EUREKA'*, щоб розгорнути список класів, які реалізують цей метод.

Зірочка на початку рядка списку класів позначає той метод, який застосовують для отримання потрібного результату. Тому зірочка перед класом *String* повідомляє нам, що метод *asUppercase*, визначений в класі *String*, після виконання поверне той результат, який нам потрібно. Класи без зірочки також реалізують метод з іменем *asUppercase*, але результат вони повертають інший. Так метод *Character>>asUppercase* не спрацював у нашому прикладі, оскільки *'eureka'* не є екземпляром класу *Character* (це екземпляр класу *String*).

За допомогою Шукача ви можете також знаходити методи з одним чи кількома аргументами. Наприклад, якщо вам потрібно знайти метод, який обчислює найбільший спільний дільник двох натуральних чисел, то можете спробувати пошукати за зразком «*25 . 35 . 5*». Ви можете також надати кілька зразків, щоб звузити межі пошуку. Довідковий текст у нижній панелі вікна пояснює, як це зробити.

4.5. Підсумки розділу

- *Spotter* – Навідник – потужний інструмент для відшукання інформації та переміщення системою.
- *Finder* – Шукач – дає змогу знаходити класи, методи тощо за іменем чи його фрагментом. Окрім того, він дозволяє знаходити методи на підставі зразків об'єктів: отримувача повідомлення, аргументів повідомлення та результату.

Розділ 5

Перший практикум. Розробка простого лічильника

Давайте для початку створимо у Pharo простий лічильник згідно з описаними далі вказівками. Ви навчитеся створювати пакети, класи, методи, екземпляри, модульні тести та ще дещо. Цей практикум охопить більшість важливих дій, які доводиться виконувати під час розробки у Pharo. Ви можете також переглянути відео, що ілюструють цей приклад, доступні в онлайновому курсі на http://mooc.pharo.org. Їхні назви мають формат *yy-Redo-xxx*. Ми покажемо також як за допомогою Iceberg зберегти свій код у хмарному сховищі, наприклад, у GitHub.

Зауважимо, що цей невеликий практикум описує *традиційний* процес розробки. Традиційний у тому сенсі, що спочатку ви створите пакет і оголосите клас, *тоді* оголосите змінні екземпляра, *тоді* визначите його методи і *лише тоді* виконаєте їх. Нині розробники зазвичай використовують у Pharo інший процес, який називається *розробка, керована тестами* (Test-Driven Development, TDD): вони *спочатку* виконують вираз. Відповідні методи ще не оголошені, тому вираз спричиняє помилку. Налагоджувач перехоплює її, і розробник програмує безпосередньо в налагоджувачі, дозволяючи системі визначити змінні екземпляра та методи на льоту, щоб виправити помилку.

Після закінчення практикуму ви ліпше освоїтеся з Pharo, тому ми наполегливо рекомендуватимемо вам виконати його ще раз з використанням TDD: напишіть модульні тести, запустіть їх, напишіть код, щоб тести проходили, знову запустіть тести. У Pharo можна застосовувати екстремальне програмування, кероване тестами, Extreme TDD: напишіть тести, запустіть їх, напишіть код у налагоджувачі, щоб тести пройшли. Екстремальне програмування – надпотужний підхід. Щоб відчути це, потрібно випробувати його власноруч. У тому ж онлайновому курсі є ще одне відео, що демонструє цей потужний спосіб написання програм. Ми справді закликаємо вас переглянути його та попрактикуватися.

5.1. Завдання

Ми хочемо мати змогу створити лічильник, збільшити його двічі (на одиницю), зменшити і переконатися, що він має очікуване значення – дорівнює одиниці. Наступний фраґмент коду описує сказане програмно. Він також стане чудовим модульним тестом, який визначимо трохи згодом.

```
| counter |
counter := Counter new.
counter increment; increment.
counter decrement.
counter count = 1
```

Ми напишемо всі класи і методи, потрібні для реалізації цього прикладу.

5.2. Створення пакета і класу

У цьому параграфі ви створите свій перший клас. У Pharo класи оголошують у пакетах, тому мусимо спочатку створити пакет, щоб помістити в нього клас. Такі дії виконують кожного разу при створенні класу, тому будьте уважні.

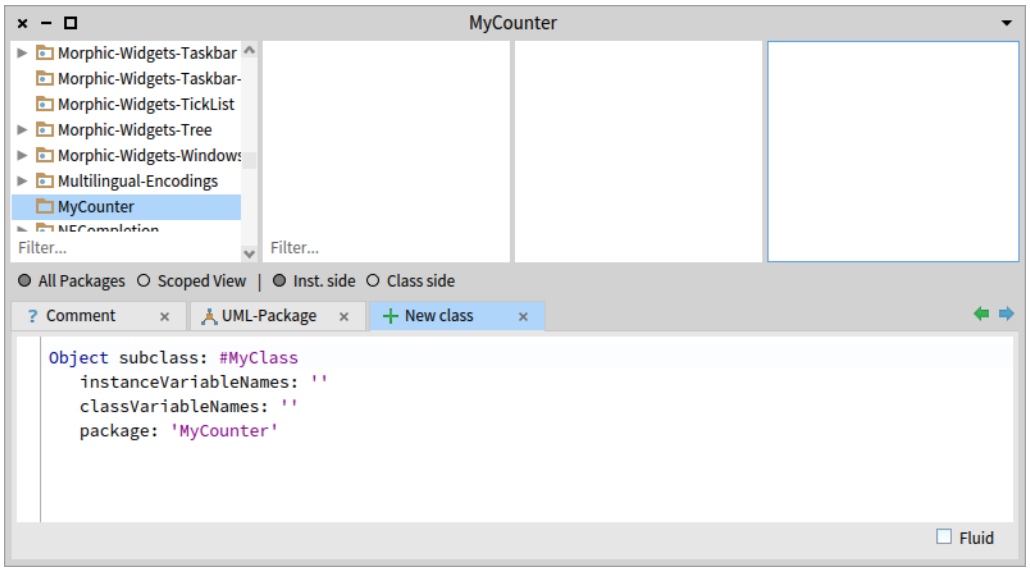

Рис. 5.1. Новостворений пакет і шаблон оголошення класу

Створення пакета

Пакети створюють за допомогою Системного оглядача: контекстно клацніть у його панелі пакетів і виберіть команду «*New Package*». Оглядач запитає у вас ім'я пакета – введіть «*MyCounter*». Одразу після створення новий пакет з'явиться у списку пакетів, його ім'я буде позначено. На рис. 5.1 зображено, що мало б вийти в результаті.

Створення класу

Нижня панель Оглядача мала б містити відкриту вкладку ***New class*** з шаблоном оголошення класу. Щоб створити новий клас, потрібно просто відредагувати шаблон і відкомпілювати написане. У шаблоні є п'ять частин, які можна змінювати.

- **Специфікація надкласу** задає базовий клас для того, який ви створюєте. За замовчуванням тут вказано *Object* – найбільш загальний з усіх класів Pharo, саме той, що потрібен нам для створення класу лічильника. Так буде не завжди: часто ви потребуватимете оголошувати класи на базі більш конкретизованих.

- **Ім'я класу.** Далі ви мали б вказати ім'я класу: замініть *#MyClass* на *#Counter*. Зверніть увагу на те, що ім'я класу починається з великої букви, а перед іменем обов'язково вказують знак *#*. Така форма запису зумовлена тим, що у Pharo клас називають за допомогою символу, екземпляра класу *Symbol*, унікального значення, зображеного рядком. Символи завжди починаються знаком *#*.

- **Специфікація змінних екземпляра.** Тепер вам потрібно заповнити імена змінних екземпляра класу *Counter* після слів *instanceVariableNames:*. Нам потрібно тільки одну змінну, що називається '*count*'. Не забудьте апострофи (одинарні лапки)! Перелік імен змінних задають рядком, екземпляром класу *String*, що містить імена, відокремлені пропуском. Рядки завжди обрамляють апострофами.

- **Специфікація змінних класу.** Їх оголошують після *classVariablesNames:*. Залиште тут порожній рядок (пару апострофів), як вказано у шаблоні, оскільки нам не потрібні змінні класу.
- **Специфікація пакета.** Вона вже містить правильне значення '*MyCounter*', яке ми не будемо зачіпати.

Ви мали б отримати таке оголошення класу:

```
Object subclass: #Counter
   instanceVariableNames: 'count'
   classVariableNames: ''
   package: 'MyCounter'
```

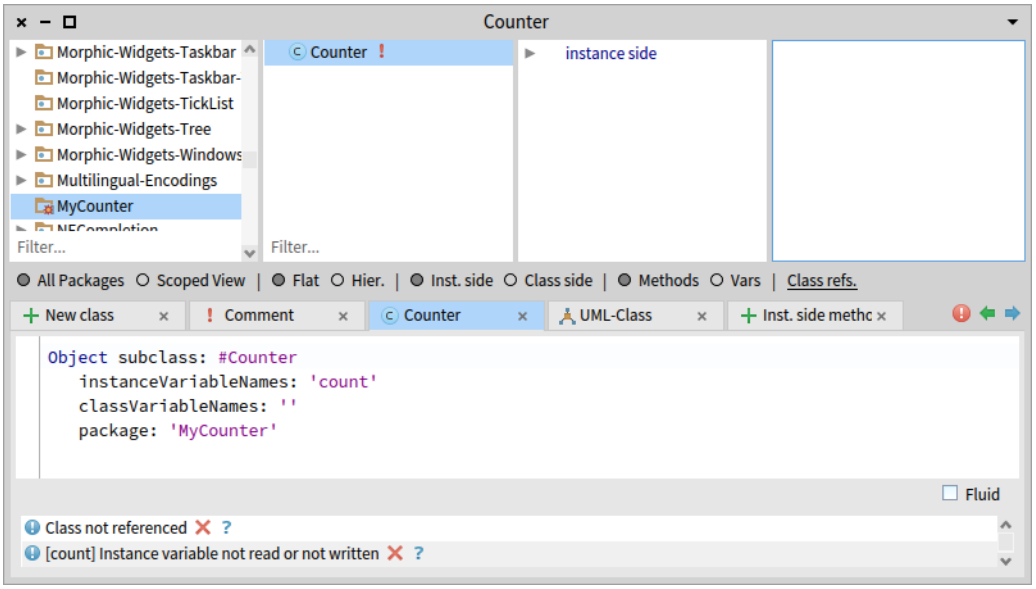

Рис. 5.2. Створено клас *Counter*, що наслідує *Object* і має одну змінну екземпляра *count*

Тепер ми маємо текст оголошення класу *Counter*. Щоб визначити клас у нашій системі, потрібно його *відкомпілювати* або через контекстне меню нижньої панелі командою «*Accept*», або комбінацією клавіш [*Cmd* + *S*]. (Скорочення від «*Save*»: зберігання оголошення класу означає його компіляцію). Після завершення компіляції клас одразу стане частиною системи.

На рис. 5.2 показано, як мав би виглядати Системний оглядач після компіляції.

Вбудований у Pharo інструмент аналізу якості коду запуститься автоматично і виведе кілька попереджень унизу вікна Оглядача. Поки що не турбуйтеся про них. Причина їхньої появи в тому, що наш клас ще ніде не використовується.

Ми з вами дисципліновані розробники, тому маємо додати до класу *Counter* коментар, що пояснює його призначення. Для цього відкрийте вкладку ***Comment*** нижньої панелі Оглядача та клацніть на перемикачі ***Toggle Edit/View comment***. Тепер ви можете замість стандартного тексту ввести, наприклад, такий коментар:

```
`Counter` is a simple concrete class which supports incrementing and
        decrementing.
Its API is
- `decrement` and `increment`
- `count`
Its creation message is `startAt:`
```

Для написання коментарів використовують розмітку *Microdown*, інтуїтивно зрозумілий діалект *Markdown*. Розмічений коментар гарно відображається у вікні оглядача. Знову збережіть зроблені вами зміни командою «*Accept*», або комбінацією [*Cmd* + *S*].

На рис. 5.3 зображено створений коментар до класу *Counter*.

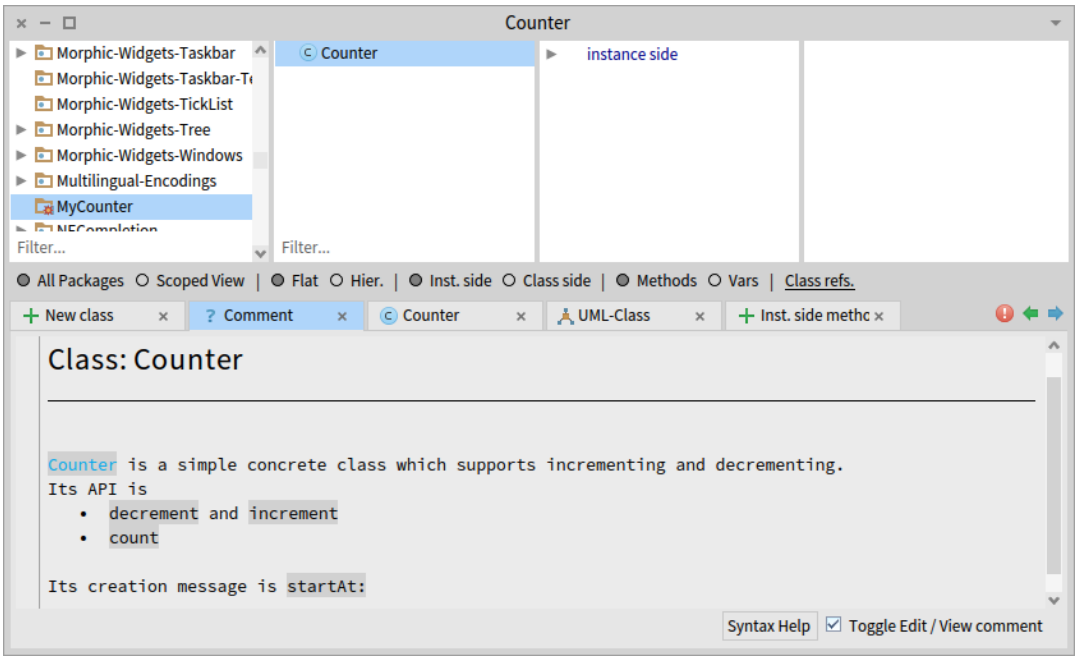

Рис. 5.3. Гарна робота: клас *Counter* тепер має коментар!

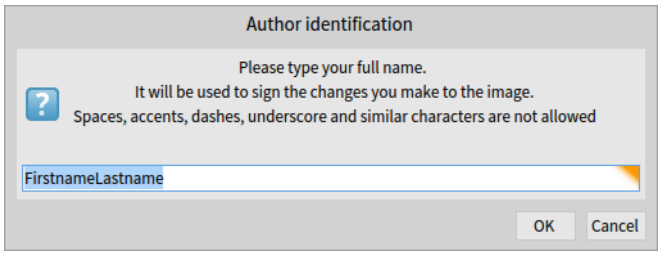

Рис. 5.4. Ідентифікація автора змін

Від перекладача. Pharo автоматично документує всі зміни, зроблені в бібліотеці класів. Під час першого зберігання зроблених вами змін (у тексті методу, в коментарі класу чи в оголошенні класу) система попросить назвати себе. Введіть своє ім'я та прізвище латинкою без пропусків, як у підказці в рядку введення діалогу ідентифікації, зображеному на рис. 5.3. Достатньо зробити це один раз. Збережіть імідж системи, і під час наступних сеансів розробки у Pharo збережений код автоматично підписуватиметься введеним іменем.

5.3. Визначення протоколів і методів

У цьому параграфі ви навчитеся додавати за допомогою Оглядача протоколи й методи.

Ми визначили клас *Counter* з однією змінною екземпляра, яка називається *count* і призначена для зберігання рахунка. Ми хочемо збільшувати, зменшувати та демонструвати її поточне значення. Але у Pharo ми повинні пам'ятати три речі.

1. Усе є об'єктами.
2. Змінні екземпляра є цілком приватними для об'єкта.

3. Єдиний спосіб взаємодіяти з об'єктом – надсилати йому повідомлення.

Саме тому не існує іншого механізму доступу ззовні до змінної екземпляра нашого лічильника, як через надсилання повідомлень об'єктові. Нам потрібно оголосити метод, який повертатиме значення змінної екземпляра. Такі методи називають *методами читання* або *селекторами* (англійською – *getter*). Отже, давайте визначимо метод доступу до змінної екземпляра *count*.

Зазвичай метод поміщають у *протокол*. Протоколи в класі – просто групи методів. Вони не мають синтаксичного значення у Pharo, але надають читачам вашого класу важливу інформацію. Хоча протоколи можна називати довільно, проте розробники у Pharo дотримуються певних домовленостей щодо найменування протоколів. Якщо ви визначаєте метод і не впевнені, до якого протоколу його зачислити, перегляньте спочатку наявний код, чи не знайдеться потрібний протокол серед уже оголошених.

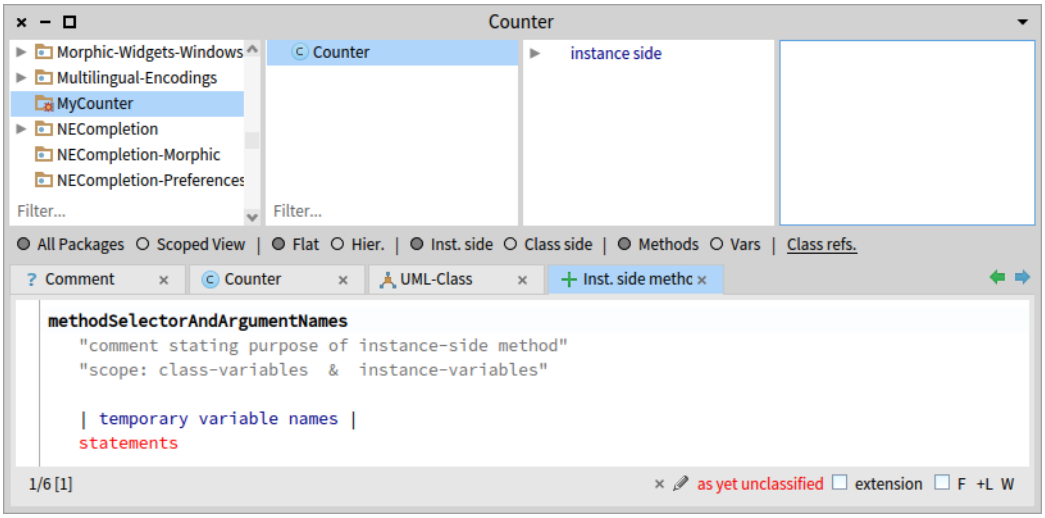

Рис. 5.5. Редактор коду готовий до визначення методу

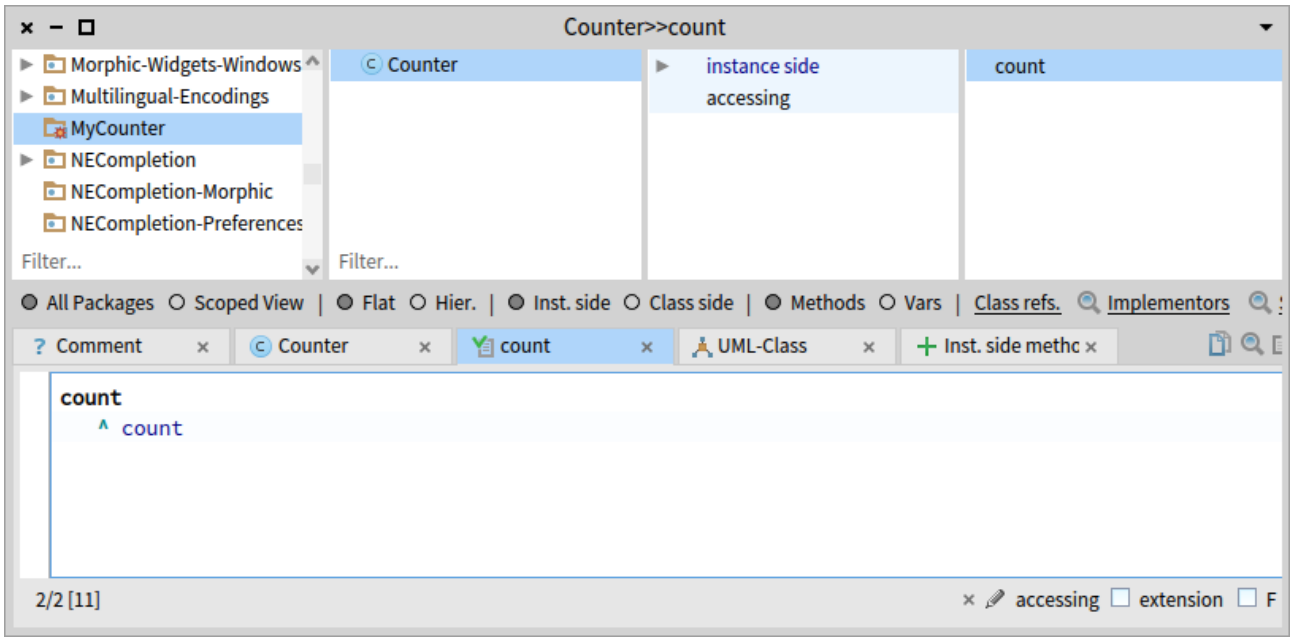

Рис. 5.6. Метод *count* визначено в протоколі *accessing*

5.4. Створення методу

Тепер давайте створимо метод для читання змінної екземпляра *count*. Почнімо з вибору класу *Counter* в Оглядачі та переконаймося, що ми редагуємо сторону екземпляра класу (тобто визначаємо методи для екземплярів нашого класу): посередині вікна Оглядача розташовано перемикач ***Inst. side/Class side***, позначеною має залишатися ліва частина. Далі оберіть вкладку ***Inst. side method*** та визначимо метод.

На рис. 5.5 показано редактор коду, готовий до визначення методу. Він містить шаблон оголошення методу. Нам потрібно замінити його справжнім оголошенням. Щоб позначити весь текст шаблону, двічі клацніть перед його початком або після закінчення. (Так можна позначати весь текст у будь-якому вікні). Почніть набирати текст, і виділення автоматично заміниться на введене.

Надрукуйте таке визначення методу:

```
count
   ^ count
```

Воно визначає метод *count*, що не приймає аргументів і повертає значення змінної екземпляра *count*. Щоб відкомпілювати метод, виберіть «*Accept*» з контекстного меню. Метод буде автоматично віднесено до протоколу *accessing*.

На рис. 5.6 зображено Оглядача після визначення методу.

Тепер ви можете випробувати новий метод, просто надрукуйте в Робочому вікні та виконайте вираз:

```
Counter new count
>>> nil
```

Цей вираз спочатку створить новий екземпляр класу *Counter*, а тоді надішле йому повідомлення «*count*», яке поверне поточне значення лічильника. Воно має повернути *nil* – значення за замовчуванням неініціалізованої змінної екземпляра. Пізніше ми створюватимемо екземпляри з розумнішим початковим значенням.

5.5. Додавання методу запису значення

Доповненням до методу читання є *метод запису* або *модифікатор* (*setter* англійською). Його використовують для того, щоб ззовні об'єкта змінити значення його змінної. Наприклад, вираз «*Counter new count: 7*» спочатку створить новий екземпляр класу *Counter*, а тоді зробить сім його значенням за допомогою повідомлення «*count: 7*». Селектори та модифікатори разом називають *методами доступу* (*accessors*).

Приклад використання модифікатора на практиці:

```
| c |
c := Counter new count: 7.
c count
>>> 7
```

Метод запису поки що не існує, тому як вправу створіть його. Метод *count:* має приймати один аргумент, число, і записувати його у змінну екземпляра. Метод можна випробувати в Пісочниці, виконавши наведений вище приклад.

Підказка: модифікатор може мати такий вигляд:

```
count: anInteger
   count := anInteger
```

Після компіляції модифікатор також мав би потрапити до протоколу *accessing*.

5.6. Визначення класу тестів

У наші дні написання тестів перестало бути необов'язковим заняттям. Ви можете писати їх перед створенням коду чи після, але писати треба обов'язково. Колекція добре продуманих тестів підтримуватиме розвиток вашого застосунку і даватиме впевненість, що ваша програма робить саме те, чого ви від неї очікуєте. Написання тестів є хорошою інвестицією. Один раз написаний код тестів виконується тисячі разів. Наприклад, якщо ми перетворимо на тест наведений раніше приклад, то зможемо автоматично перевірити, чи працює новий модифікатор так, як треба.

Модульні тести пишуть у вигляді методів окремого класу, успадкованого від *TestCase*. Тому ми визначимо клас *CounterTest* так:

```
TestCase subclass: CounterTest
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'MyCounter'
```

Традиційно ім'я класу тестів складається з імені тестованого класу і суфікса *Test*: *Counter* + *Test* дає *CounterTest*.

Тепер можемо написати свій перший тест як метод цього класу. Імена тестових методів мають розпочинатися з «*test*», щоб система правильно розпізнала їх і наділила спеціальними можливостями: тестові методи можна автоматично виконати за допомогою спеціального інструмента – *Test Runner*; Оглядач розташовує перед іменем тестового методу круглу піктограму, яка здатна змінювати колір, а клацання на ній запускає метод на виконання.

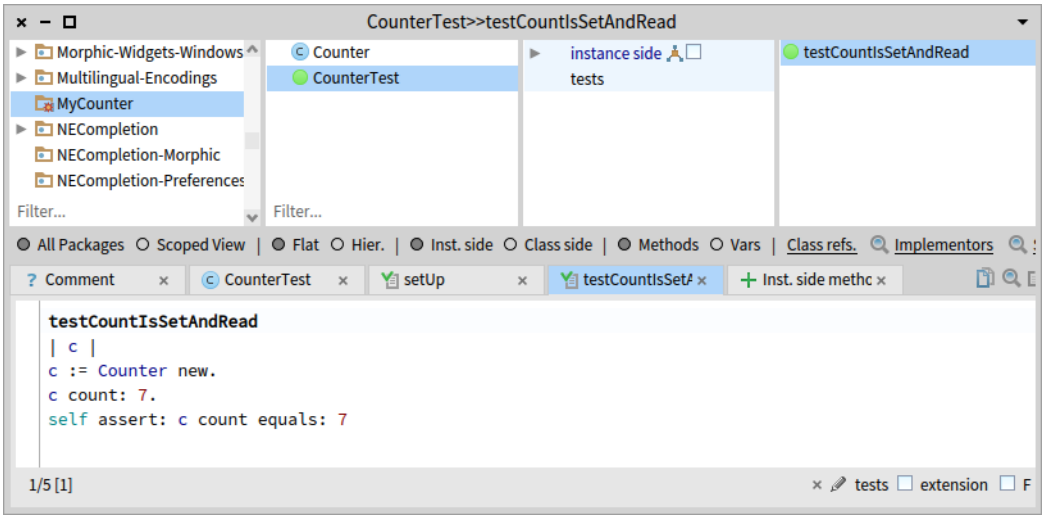

Рис. 5.7. Перший тест визначено і виконано

Визначимо метод для нашого модульного тесту. Найперше він створює екземпляр класу *Counter*, задає його значення, а тоді перевіряє чи містить він задане значення.

Повідомлення «*assert:equals:*» зрозуміле всім нащадкам *TestCase*. Воно перевіряє істинність твердження про те, що два об'єкти рівні між собою. Якщо це не так, то тест завершиться невдачею.

```
CounterTest >> testCountIsSetAndRead
  | c |
  c := Counter new.
  c count: 7.
  self assert: c count equals: 7
```

На рис. 5.7 показано визначення методу *testCountIsSetAndRead* в класі *CounterTest*.

Нагадаємо домовленості щодо позначень методів. Pharo-розробники часто використовують запис «*ClassName >> methodName*», щоб вказати клас, до якого належить метод. Наприклад, раніше ми визначили метод *count* у класі *Counter*. Посилання на нього матиме вигляд «*Counter >> count*». Пам'ятайте, що такий запис *не є* частиною синтаксису Pharo. Просто ми домовилися так записувати те, що «метод екземпляра *count* належить до класу *Counter*».

Надалі будемо записувати імена методів у такій формі щоразу, коли згадуватимемо їх у цій книзі. Зрозуміло, що вам не потрібно друкувати ім'я класу чи знак «>>», коли ви вводитимете текст методу в редакторі коду. Натомість переконайтеся, що на панелі класів Оглядача вибрано належний клас.

Щоб запустити тестовий метод на виконання та переконатися, що тест завершується успіхом, клацніть на іконці ліворуч від імені методу (див. рис. 5.7) або використайте *Test Runner* через меню *World* > *Browse*.

Ви щойно отримали перший зелений тест і добру нагоду зберегти свої напрацювання.

5.7. Зберігання коду в git-репозиторії за допомогою Iceberg

Зроблене можна зберігати в іміджі Pharo. Це хороший спосіб, проте не ідеальний. Він не підходить для налагодження співпраці з іншими розробниками та для поширення коду. Багато сучасних розробників програмного забезпечення взаємодіють через Git, відкриту розподілену систему керування версіями файлів. Збудовані із застосуванням Git сервіси, такі як GitHub, надають розробникам простір для спільного виконання проєктів з відкритим вихідним кодом таких, наприклад, як Pharo.

Pharo взаємодіє з Git за допомогою окремого інструмента – *Iceberg*. Цей параграф продемонструє вам, як створити локальний репозиторій для програмного коду, заносити в нього зміни та переносити їх до віддаленого репозиторію, розташованого, наприклад, на GitHub.

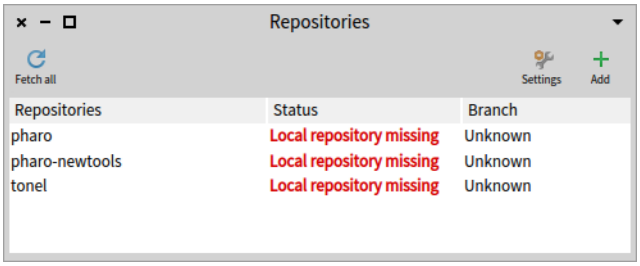

Рис. 5.8. Оглядач репозиторіїв інструмента Iceberg, відкритий на свіжому образі Pharo показує проєкти, для яких не знайдено локальні сховища. Їх треба буде створити, якщо ви захочете змінити версію самого Pharo, але це не зараз

Відкрийте Iceberg

Відкрийте Iceberg через меню *World* > *Sources* або комбінацією клавіш [*Cmd* + *O,I*].

Ви мали б побачити щось схоже на зображене на рис. 5.8 – головне вікно Iceberg. Воно містить проєкт Pharo та деякі інші, отримані разом з образом, та повідомляє рядком «*Local repository missing*», що не може знайти їхні локальні репозиторії. Ви можете не турбуватися про проєкт Pharo та його сховище, якщо не збираєтеся брати участі в його розробці.

Ми плануємо створити свій власний новий проєкт.

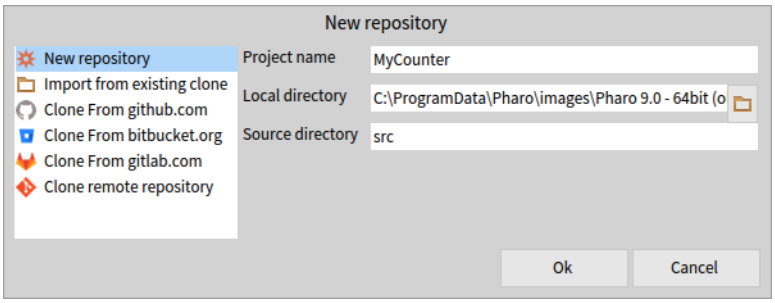

Рис. 5.9. Створення нового проєкту, що називається *MyCounter* та містить підкаталог *src*

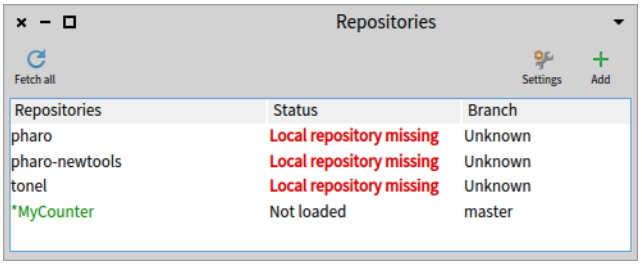

Рис. 5.10. Новостворений проєкт з незбереженими змінами

Додайте та налаштуйте проєкт

Щоб створити новий проєкт, натисніть кнопку ***Add***. Вона відкриє вікно налаштування проєкту (див. рис. 5.9). У списку ліворуч буде обрано рядок «*New repository*» (якщо ні, то оберіть його), а нам залишиться тільки задати ім'я проєкту, уточнити папку для його зберігання та вказати ім'я вкладеної папки для програмного коду. За домовленістю вона називається «*src*».

Натисніть кнопку «*Ok*» і діалог закриється, а в головному вікні Iceberg з'явиться створений проєкт, як на рис. 5.10.

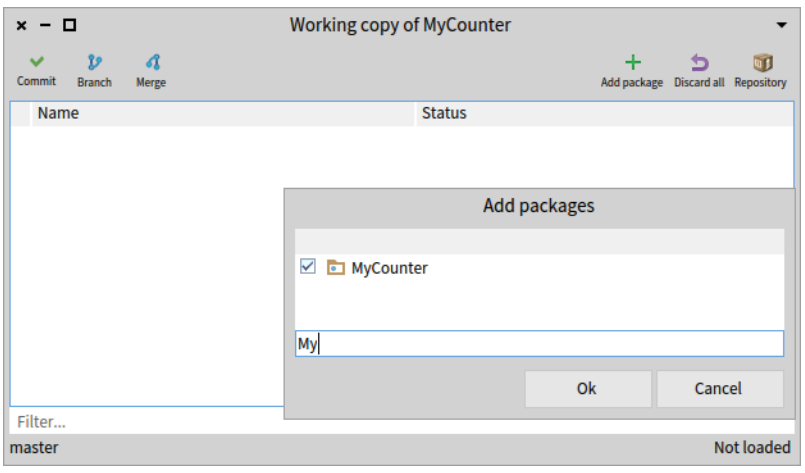

Рис. 5.11. Додавання пакета до проєкту кнопкою ***Add package***

Додайте до проєкту пакет

Вміст проєкту можна переглянути в оглядачі робочої копії репозиторію інструмента Iceberg. Щоб відкрити його, двічі клацніть на рядку з проєктом *MyCounter* у головному вікні Iceberg. Щойно створений проєкт не містить пакетів, тому ви побачите порожнє вікно, як на рис. 5.11. Клацніть на кнопці ***Add package*** та виберіть пакет *MyCounter* на модальній панелі, що відкриється (його легко відшукати серед багатьох пакетів за допомогою фільтра).

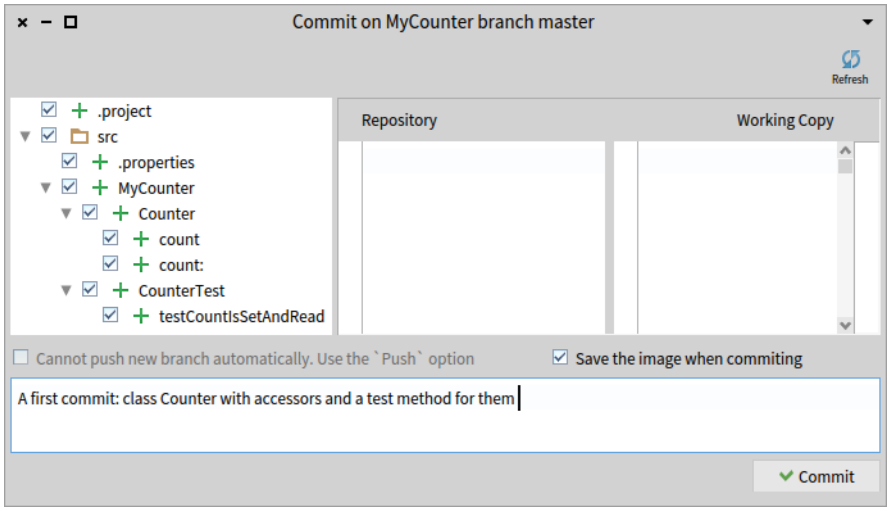

Рис. 5.12. Iceberg показує зміни, які належить перенести до сховища

Збережіть пакет

Одразу після додавання до проєкту пакет матиме статус «*Uncommitted changes*». Так Iceberg повідомляє нам, що код в пакеті відрізняється від того, що є у сховищі. Так і має бути, адже в сховищі ще немає коду! Збережіть його, клацнувши кнопку ***Commit***. Iceberg покаже вам у новому вікні всі зміни, які потрібно зберегти (див. рис. 5.12). Уведіть коротке пояснення щодо змісту змін і перенесіть їх до сховища ще однією кнопкою ***Commit***.

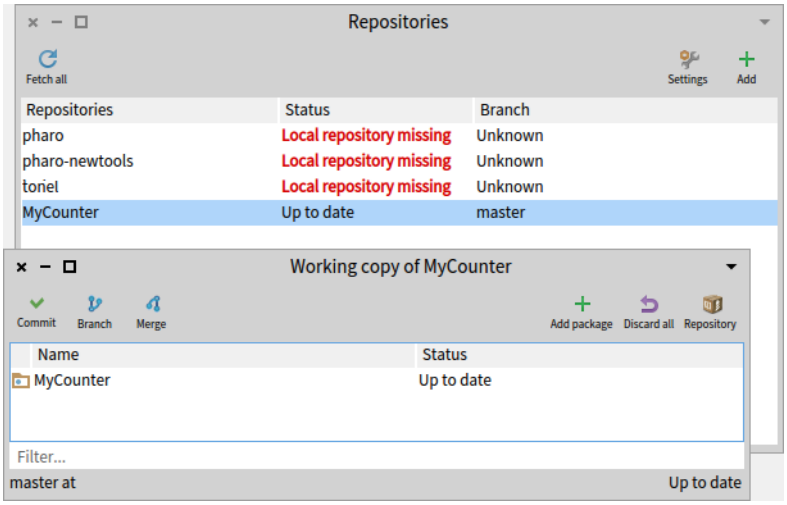

Рис. 5.13. Iceberg підтверджує, що зміни збережено

Код збережено

Як тільки ви завершите зберігання, Iceberg відзначить, що ваша система і локальний репозиторій синхронізовані (рис. 5.13).

Гарна робота! Незабаром ми побачимо, як перенести зміни коду до віддаленого сховища, а зараз повернемося до нашого класу *Counter*.

5.8. Додавання нових методів

Перш ніж додати до класу *Counter* наступні методи, напишемо тести, що їх перевіряють. Почнемо з тесту для повідомлення *increment*.

```
CounterTest >> testIncrement
  | c |
  c := Counter new.
  c count: 0; increment; increment.
  self assert: c count equals: 2
```

Тепер ваша черга! Напишіть визначення методу *increment* так, щоб цей тест завершився успіхом. Коли закінчите, спробуйте написати тест для повідомлення *decrement* і визначте в класі *Counter* метод, що задовольнить його. Нові методи розташуйте в протоколі *operations*.

Розв'язок

```
Counter >> increment
  count := count + 1

Counter >> decrement
  count := count - 1
```

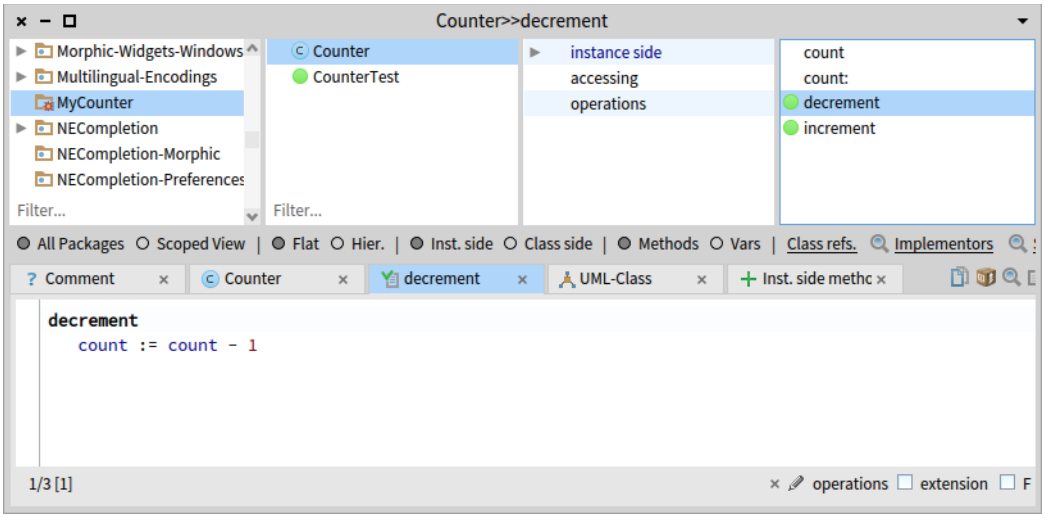

Рис. 5.14. У класу *Counter* стало більше методів і зелених тестів

Чи зауважили ви, що тест для методу *increment* називається *testIncrement*? Це не випадково, адже такий спосіб найменування дає змогу середовищу пов'язувати метод з його модульним тестом. Якщо ви все зробили правильно, то біля імен методів *increment* та *decrement* в Оглядачі класів з'являться такі ж піктограми, як біля методів-тестів (див. рис. 5.14). Клацніть на кожній з них, і ви запустите тести на виконання.

Ваші тести мали б завершитися успіхом. І знову було б доречно зберегти свою роботу. Збереження у ті моменти, коли всі тести зелені, є хорошою практикою. Щоб зберегти зміни, занесіть їх до сховища за допомогою Iceberg.

5.9. Метод ініціалізації екземпляра

На цей момент початкове значення нашого лічильника залишається невизначеним. У цьому легко переконатися, виконавши вираз:

```
Counter new count
>>> nil
```

Давайте напишемо тест, який стверджує, що новий екземпляр класу *Counter* містить значення 0 у змінній *count*:

```
CounterTest >> testInitialize
  self assert: Counter new count equals: 0
```

Клацніть на піктограмі методу, щоб запустити його на виконання, і отримаєте вікно з повідомленням «*Got nil instead of 0*» (отримано *nil* замість *0*). Це вікно можна поки що закрити, але зауважте, що піктограма стала *жовтою*. Це ознака того, що тест завершився невдачею: усе було виконано, проте твердження виявилося хибним. Така ситуація відрізняється від отримання *червоного* тесту, коли тест не виконується через помилку, наприклад, якщо потрібний метод не визначено. Незабаром ми побачимо і такі.

5.10. Визначення методу ініціалізації

Тепер потрібно написати метод ініціалізації, який задає початкове значення змінної *count* екземпляра лічильника. Ми вже знаємо, що екземпляри створюють надсиланням класові повідомлення *new*. Процес влаштовано так, що кожен новостворений об'єкт автоматично отримує повідомлення *initialize*. Це дає нагоду екземплярові налаштувати свій початковий стан. Визначимо метод *initialize*, який задаватиме правильне початкове значення лічильника.

Повідомлення *initialize* надсилають екземплярові, тому відповідний метод потрібно визначити на *стороні екземпляра* так само, як інші методи, що опрацьовують повідомлення до екземпляра (такі як *increment* чи *decrement*). Метод *initialize* відповідає за налаштування значень за замовчуванням змінних екземпляра.

Отже, визначте в класі *Counter* на стороні екземпляра у протоколі *initialization* зображений нижче метод (тіло методу поки що порожнє, заповніть його самостійно).

```
Counter >> initialize
  "Set the initial value of the count to 0"

  "Тут має бути ваш код"
```

Якщо ви все зробили правильно, то тест *testInitialize* тепер завершиться успіхом. Як завжди, збережіть свою роботу перш ніж рухатися далі.

5.11. Визначення методу створення нового екземпляра

Щойно ми зазначили, що метод *initialize* визначають *на стороні екземпляра*, бо його виконує новостворений екземпляр класу *Counter*. Метод *initialize* є методом екземпляра і змінює значення його полів. Давайте тепер розглянемо визначення методу *на стороні класу* – методу класу. Метод класу виконується тоді, коли повідомлення отримує сам клас, а не його екземпляри (таким повідомленням, наприклад, є *new*). Щоб визначити

метод класу, потрібно перемкнути Оглядач класів на сторону класів. Увімкніть для цього перемикач ***Class side***.

Визначте новий метод для створення екземплярів, що називається *startingAt:*. Він отримує ціле число як аргумент і повертає новий екземпляр класу *Counter*, який у змінній *count* містить задане число.

З чого почати? Звичайно ж, з тесту.

```
TestCounter >> testCounterStartingAt5
  self assert: (Counter startingAt: 5) count equals: 5
```

Тут повідомлення *startingAt: 5* надсилають самому класові *Counter*.

Ваша реалізація методу може виглядати якось так:

```
Counter class >> startingAt: anInteger
  ^ self new count: anInteger.
```

Тут у тексті ми використали позначення вигляду «*ClassName class >> methodName*», щоб вказати на *метод класу*. Перший рядок коду означає «*startingAt:* є методом класу у класі *Counter*».

На що вказує *self* у наведеному коді? Як завжди, *self* вказує на об'єкт, що виконує метод, і тому тут він вказує на сам клас *Counter*.

Давайте напишемо ще один тест, щоб просто переконатися, що все працює належно.

```
CounterTest >> testAlternateCreationMethod
  self assert: ((Counter startingAt: 19) increment; count) equals: 20
```

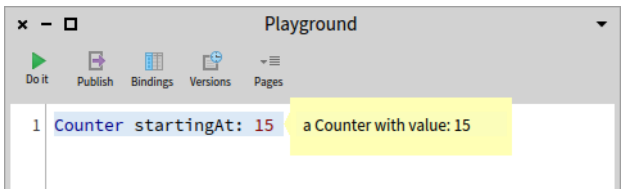

Рис. 5.15. Інформативне представлення лічильника

5.12. Покращення опису об'єкта

Коли ви інспектуватимете екземпляр *Counter* або Налагоджувачем, або Інспектором, відкривши його командою [*Cmd* + *I*] на виразі «*Counter new*», або, навіть, коли ви просто виконаєте команду «*Print it*» на виразі «*Counter new*», отримаєте дуже спрощене зображення свого лічильника: лише '*a Counter*':

```
Counter new
>>> a Counter
```

Ми б хотіли мати більш інформативне зображення, наприклад, таке, що показує значення лічильника. Реалізуйте наступний метод у протоколі *printing* класу *Counter*:

```
Counter >> printOn: aStream
  super printOn: aStream.
  aStream nextPutAll: ' with value: '.
  count printOn: aStream
```

Зауважте, що повідомлення *printOn:* отримує кожен об'єкт, який друкують командою «*Print it*» (див. рис. 5.15), або який відкривають у Інспекторі. За допомогою визначення методу *printOn:* для екземпляру класу *Counter* ми власноруч задаємо спосіб відображення лічильника на екрані. Ми *перевизначаємо* успадкований метод *Object >> printOn:*, що діє за замовчуванням, і який до тепер виконував усю роботу. Пізніше детальніше розглянемо успадкування та перевизначення, а також вивчимо використання потоків і змінної *super*.

Модульний тест для методу *printOn:* визначте самі. Підказка: щоб отримати зображення лічильника у вигляді рядка, таке ж, яке створює метод *printOn:*, надішліть повідомлення «*printString*» до «*Counter new*».

```
Counter new printString
>>> a Counter with value: 0
```

Давайте знову збережемо свою роботу, але цього разу – на віддаленому Git-сервері.

5.13. Зберігання коду на віддаленому сервері

До тепер ви зберігали свій код на локальному диску комп'ютера. Зараз ми покажемо, як його зберегти у віддаленому репозиторії, який ви можете створити на GitHub https://github.com або GitLab.

Створіть проєкт на віддаленому сервері

Спочатку вам треба створити репозиторій на віддаленому Git-сервері. Залишіть його порожнім, щоб не виникало плутанини. Назвіть його якось просто й зрозуміло, наприклад, «*Counter*» або «*Pharo-Counter*». Це те місце, куди ми плануємо відправити наш проєкт з Iceberg.

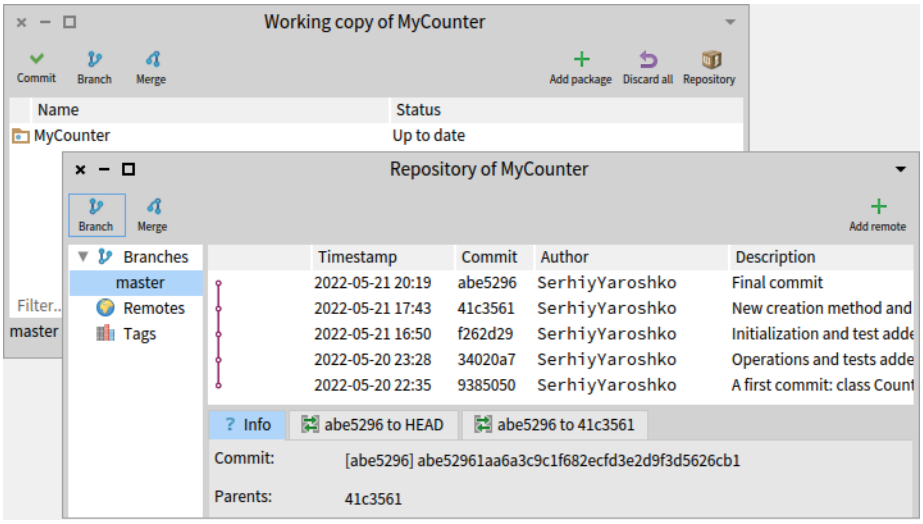

Рис. 5.16. Оглядач репозиторіїв вашого проєкту

Додайте віддалений репозиторій

У головному вікні Iceberg двічі клацніть на рядку з вашим проєктом *MyCounter*, щоб відкрити оглядач робочої копії репозиторію. Тоді клацніть на кнопці ***Repository*** (вона оздоблена піктограмою з зображенням ящика). Вона відкриє оглядач репозиторіїв проєкту *MyCounter*, як показано на рис. 5.16.

Тоді вам просто потрібно додати віддалене сховище для проєкту, що так само просто, як клацнути кнопку ***Add remote***. Вам запропонують вказати ім'я віддаленого сховища і його URL-адресу. Ім'я можна вказати довільне, воно є лише міткою, яку Git використовує локально для ідентифікації. Адреса ж має бути точною і вести до створеного перед цим віддаленого репозиторію. Ви можете використовувати доступ HTTPS (URL-адреса, яка починається з https://github.com для GitHub) або доступ SSH (URL-адреса, яка починається з git@github.com). SSH потребуватиме налаштування агента SSH на вашому комп'ютері з правильними обліковими даними (будь ласка, зверніться до свого постачальника Git, щоб дізнатися, як цього досягти).

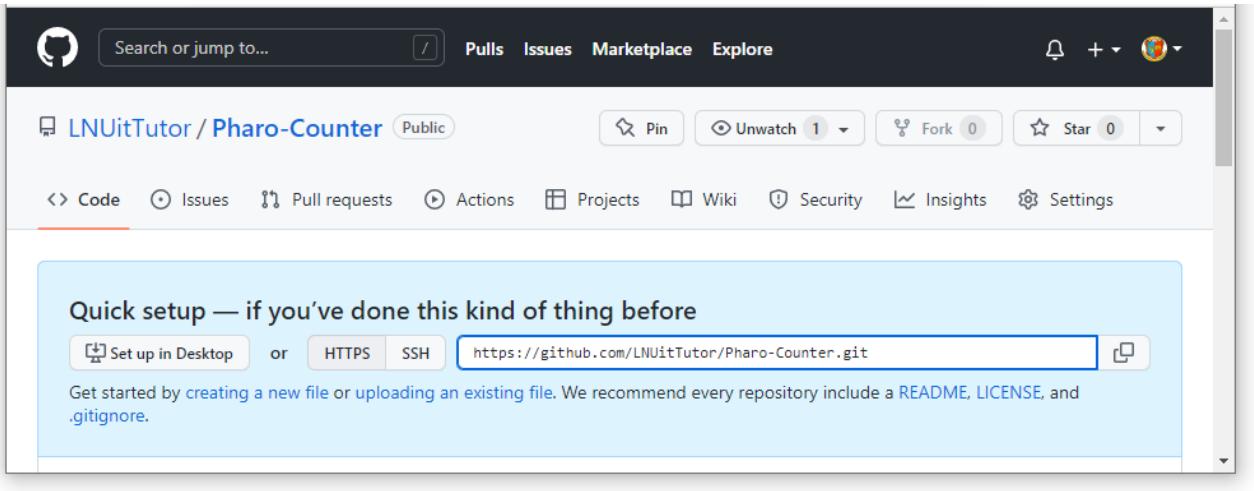

Рис. 5.17. HTTPS адреса нашого репозиторію на GitHub

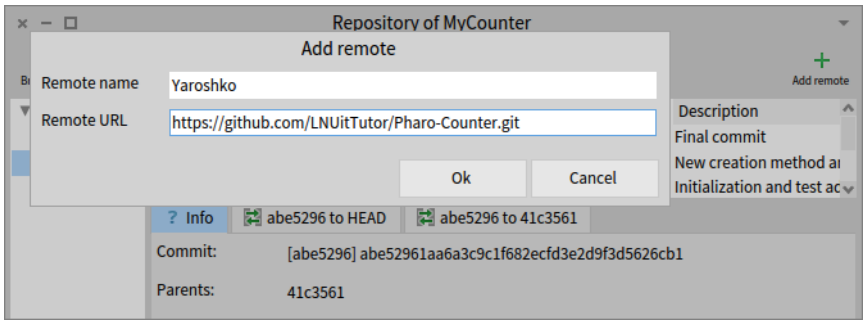

Рис. 5.18. Використання HTTPS адреси з GitHub

Ми використаємо HTTPS доступ. Адресу скопіюємо зі сторінки репозиторію на GitHub (рис. 5.17) і вкажемо її в модальному діалозі *Add remote* (рис. 5.18).

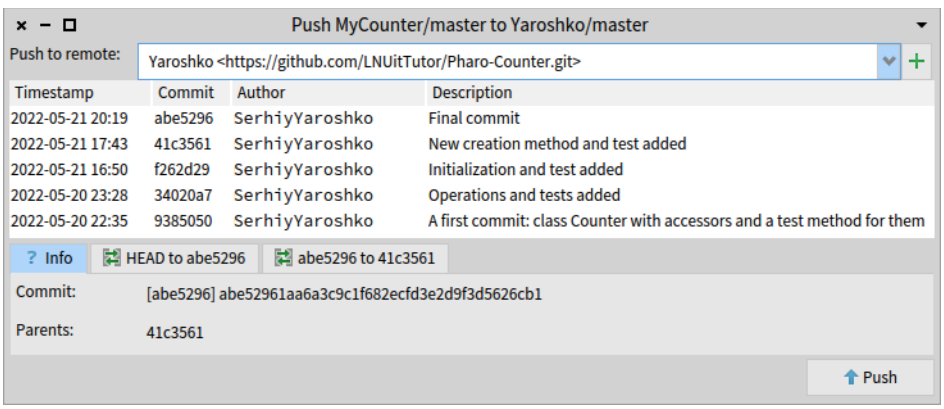

Рис. 5.19. Перелік збережень, які буде перенесено на віддалений сервер

Перенесіть проєкт у віддалене сховище

Як тільки ви вкажете справжню адресу віддаленого сервера, Iceberg відобразить на кнопці ***Push*** помітний червоний індикатор. Він сигналізує, що у вашому локальному сховищі є зміни, ще не перенесені до віддаленого сховища. Все, що потрібно зробити, клацнути на кнопці ***Push***. Iceberg покаже вам список збережених змін (commits), які буде перенесено на сервер (рис. 5.19). Вам треба натиснути ще одну кнопку ***Push***.

Коли ви зробите це вперше, HTTPS зажадає від вас ввести ім'я користувача на GitHub та токен доступу. Pharo збереже їх для вас на майбутнє.

Тепер ви *справді* зберегли свій код і зможете оновити його з іншої машини або місця. Це вміння дасть змогу вам працювати віддалено, а також ділитися програмами та співпрацювати з іншими розробниками.

Від перекладача. Віднедавна GitHub перестав надавати доступ для сторонніх програм через ім'я користувача та пароль, натомість потрібно використовувати ім'я і персональний токен доступу. Перш ніж переносити проєкт на GitHub, згенеруйте собі такий токен: увійдіть у свій обліковий запис, відкрийте сторінку налаштувань (Settings), у самому низу лівої панелі сторінки відшукайте посилання на підрозділ налаштувань розробника (Developer settings) і відкрийте його. Тут ви знайдете підрозділ керування токенами доступу (Personal access tokens), де обліковано всі наявні токени та можна згенерувати нові. Зверніть увагу на те, що згенерований токен не можна побачити двічі. Як тільки ви закриєте сторінку, текстове зображення токена зникне, і ви не зможете відкрити його знову. Тому одразу збережіть його в окремому файлі або перенесіть одразу до Pharo.

Увесь процес генерування докладно описаний у документації, доступній за адресою https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token

---

5.14. Підсумки розділу

У цьому практикумі ви дізналися, як визначати пакети, класи, методи та тести. Він був першим, тому ми використали традиційний для більшості мов програмування процес розробки програм. Однак у Pharo розробники використовують інший робочий процес, розумний і гнучкий: розробку, керовану тестуванням (TDD). Ми пропонуємо вам повторити всю цю вправу в стилі TDD: спочатку визначте тест, запустіть його, отримайте повідомлення про помилку та визначте метод у налагоджувачі, а потім повторіть такі ж дії для наступного методу. Перегляньте друге відео «Counter» у Pharo MOOC, доступне на http://mooc.pharo.org, щоб краще зрозуміти робочий процес.

Розділ 6

Створення невеликої гри

У цьому розділі ми створимо просту гру Lights Out (https://en.wikipedia.org/wiki/Lights_Out_(game)). У процесі роботи ми розширимо наше знайомство з Оглядачем класів, Інспектором об'єктів, Налагоджувачем і системою контролю версій Iceberg. Важливо добре оволодіти цими основними інструментами. У Pharo можна програмувати у звичній манері: визначити клас, потім його поля та методи. Однак у Pharo процес розробки може бути більш продуктивним! Ви можете визначати змінні екземпляра та методи на льоту, писати код у Налагоджувачі, використовуючи точний контекст об'єктів, що існують на поточний момент. Тому ми знову заохочуватимемо вас використовувати розробку, керовану тестуванням, під час написання цієї гри.

Кілька слів попередження: цей розділ містить кілька навмисних помилок, зроблених для того, щоб продемонструвати, як обробляти помилки, що трапилися під час виконання, та знаходити їх в коді. Вибачте, якщо це вас трохи розчарує, але мусимо побачити ці важливі технічні прийоми в дії, тому намагайтеся дотримуватися наших інструкцій.

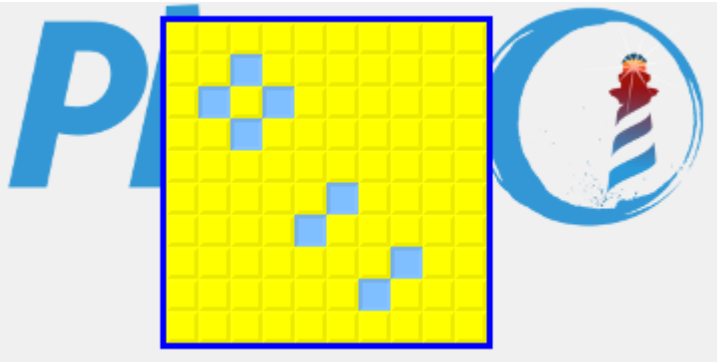

Рис. 6.1. Поле гри Lights Out

6.1. Гра Lights Out

Ігрове поле – це прямокутник, заповнений жовтими клітинками. Натискання на одну з них перемикає колір чотирьох сусідніх, і вони стають синіми (рис. 6.1). Повторне натискання на клітинку знову змінить колір сусідів: вони стануть жовтими. Мета гри – отримати якомога більше синіх клітинок.

«Lights Out» має два типи об'єктів: ігрове поле та 100 окремих клітинок. Програмний код, що реалізує гру, міститиме два класи: один для гри, інший – для клітинок.

6.2. Створення нового пакета класів

Нам потрібно створити новий пакет. Як і раніше, зробимо це в Оглядачі. Якщо не пригадуєте, як створюють пакети, перегляньте ще раз розділ 3 «Швидкий огляд Pharo» та розділ 5 «Розробка простого лічильника».

Ми назвемо пакет *PBE-LightsOut*. Так ви зможете швидко знайти його серед усіх інших: просто надрукуйте «*PBE*» в рядку фільтра, і бачитимете тільки його.

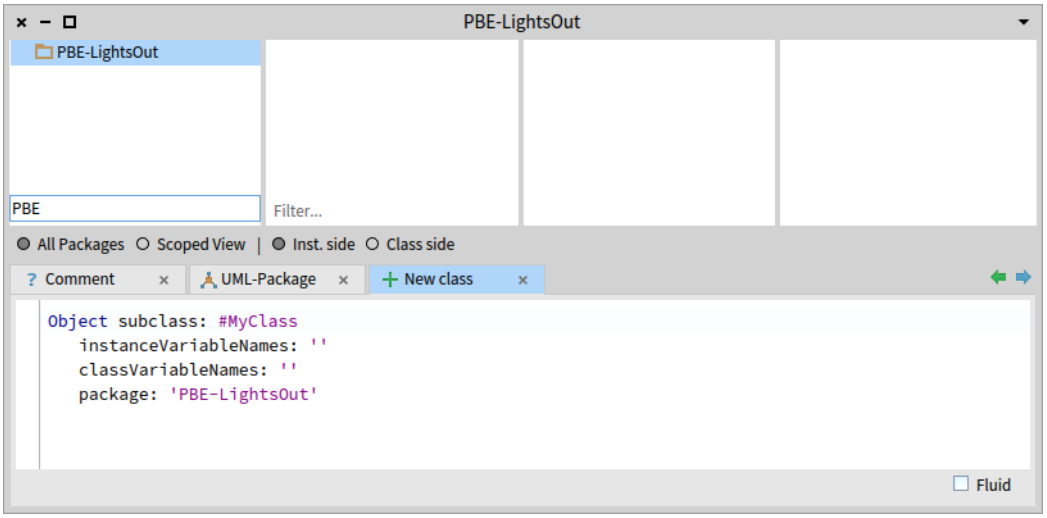

Рис. 6.2. Відфільтровування пакетів для ефективнішої роботи

6.3. Визначення класу *LOCell*

Зараз у новому пакеті, звісно, немає класів. Проте в нижній панелі Оглядача, панелі редагування, відображається шаблон для створення нового класу (див. рис. 6.2). Давайте заповнимо його необхідними даними.

Лістинг 6.1. Визначення класу *LOCell*

```
SimpleSwitchMorph subclass: #LOCell
   instanceVariableNames: 'mouseAction'
   classVariableNames: ''
   package: 'PBE-LightsOut'
```

6.4. Створення нового класу

Лістинг 6.1 містить визначення нового класу, яке ми плануємо використовувати.

Давайте на хвилину замислимося, що ми бачимо у шаблоні визначення класу. Чи це є якась спеціальна форма, яку потрібно заповнити, щоб створити новий клас? Чи це є новий синтаксис? Ні, це просто повідомлення, яке надсилають об'єктові! Визначення класу – це вираз Pharo, який надсилає повідомлення до існуючого класу *SimpleSwitch-Morph* з проханням створити підклас *LOCell*. Саме повідомленням є «*subclass:instance-VariableNames:classVariableNames:package:*». Воно дещо багатослівне. Усі аргументи є рядками, крім імені підкласу, який створюємо – його задано символом #*LOCell*. Ім'я пакета Оглядач вказує автоматично – це наш новий пакет, в якому оголошуємо клас (див. рис. 6.2). І '*mouseAction*' задає ім'я змінної екземпляра, яку ми використаємо, щоб вказати, яка дія відбудеться, коли хтось клацне на клітинці.

То чому ми наслідуємо від *SimpleSwitchMorph*, а не від *Object*? Дуже скоро ми побачимо переваги наслідування спеціалізованих класів і довідаємося, для чого потрібна змінна екземпляра *mouseAction*.

Щоб повідомлення надійшло до класу, підтвердьте його зміни командою контекстного меню або комбінацією [*Cmd* + *S*]. Повідомлення буде надіслано, клас відкомпільовано, і ми отримаємо щось таке, як на рис. 6.3.

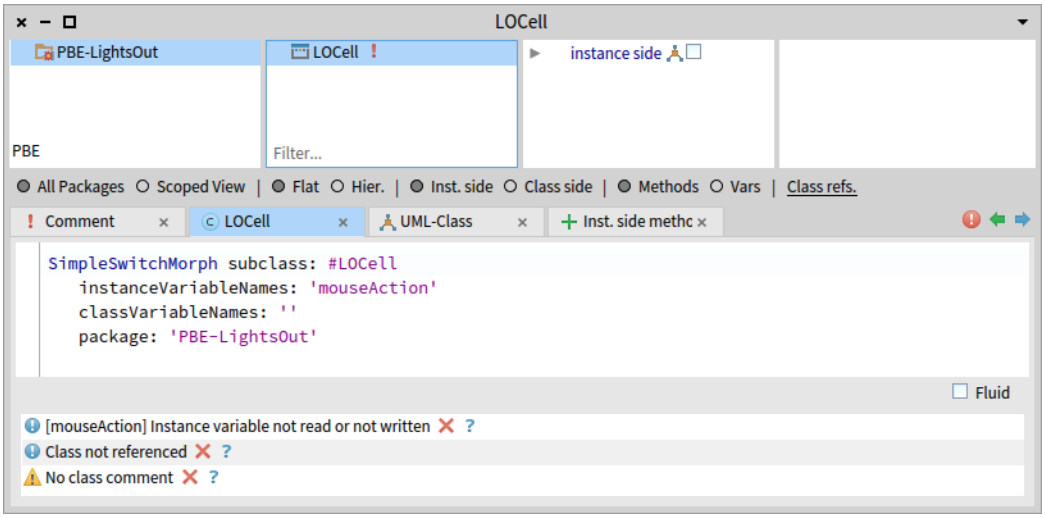

Рис. 6.3. Новостворений клас *LOCell*

Новий клас з'явився в панелі класів Оглядача, а панель редагування тепер показує визначення класу. Унизу вікна видно відгук Помічника з якості: він автоматично запускає перевірку правил якості на вашому коді та звітує про результат. Можете не звертати на нього увагу: трохи забагато для початку.

6.5. Про коментарі

Pharo-розробники високо цінують не тільки читабельність їхнього коду, а також і хороші якісні коментарі.

Коментарі до методів

Людям властиво вірити, що добре написані методи не обов'язково коментувати. Ніби призначення та зміст методу мають бути очевидними після прочитання. Це хибна думка, яка заохочує неохайність. Звісно, погано написаний код, крім коментування, потрібно виправити та перебудувати. Хороший коментар не виправдовує складний для читання код.

Очевидно, що немає змісту коментувати тривіальні методи. Коментар не мав би бути перекладом коду людською мовою, натомість мав би пояснювати, що цей метод робить, контекст його виконання або резон створення. Коментар має розвіяти можливі сумніви читача щодо призначення коду та сприяти його правильному розумінню.

Коментарі до класів

Ви вже знайомі з вкладкою коментарів Оглядача класів. Перейдіть на неї, і побачите шаблон якісного коментаря, наданий розробниками Pharo. Прочитайте його! Зразок побудовано згідно з CRC-дизайном[3], який розробили Кент Бек і Уорд Канінгем, коли працювали над Smalltalk у 80-х роках минулого століття (перегляньте їхню статтю[4], щоб довідатися більше). Коротко кажучи, коментар кількома реченнями описує *відповідальність* класу, позаяк він *взаємодіє* з іншими класами, щоб реалізувати цю відповідальність. Додатково можна зазначити інтерфейс класу (основні повідомлення, які розуміє екземпляр класу), навести приклад використання (зазвичай у Pharo

[3] Class Responsibility Collaborators – «Клас Відповідальність Взаємодія», метод аналізу при проектуванні об'єктно-орієнтованого програмного забезпечення
[4] Kent Beck and Ward Cunningham «A Laboratory For Teaching Object-Oriented Thinking», https://dl.acm.org/doi/pdf/10.1145/74878.74879

визначають приклади як методи класу) та деякі деталі внутрішнього влаштування класу чи обґрунтування реалізації.

6.6. Додавання методів до класу

Давайте додамо кілька методів до нашого класу. Найперше додамо метод екземпляра з лістингу 6.2.

Лістинг 6.2. Ініціалізація екземпляра класу *LOCell*

```
LOCell >> initialize
  super initialize.
  self label: ''.
  self borderWidth: 2.
  bounds := 0 @ 0 corner: 16 @ 16.
  offColor := Color yellow.
  onColor := Color r:0 g:0.4 b:1.
  self useSquareCorners.
  self turnOff
```

Нагадаємо, що ми використовуємо запис «*ClassName* >> *methodName*» лише для того, щоб вказати, в якому класі визначено метод.

Зауважте, що символи '' у третьому рядку – це дві окремі одинарні лапки без розділювача між ними, а не одна подвійна лапка! Так позначають порожній рядок. Інший спосіб створити порожній рядок – *String new*. Не забудьте зберегти визначення методу.

У цьому методі багато чого відбувається, давайте розберемося з усім.

Методи ініціалізації

Зауважимо, що цей метод *initialize* відрізняється від того, який ми бачили у лічильника в попередньому розділі. Нагадаємо, що це спеціальний метод, і за домовленістю його буде викликано одразу після створення об'єкта. Отже, якщо ми виконаємо *LOCell new*, то повідомлення *initialize* буде автоматично відправлене до новоствореного об'єкта. Методи ініціалізації використовують для налаштування стану об'єктів зазвичай, щоб встановити їхні поля – це саме те, що ми тут зробили.

Виклик ініціалізації надкласу

Перше, що робить цей метод (у другому рядку коду) – викликає метод *initialize* свого надкласу *SimpleSwitchMorph*. Можете бути певні, що будь-який успадкований стан буде правильно ініціалізовано методом *initialize* надкласу. Підклас наслідує набір полів надкласу, тому завжди варто надсилати *super initialize* та ініціалізувати успадкований стан, перш ніж виконувати будь-які інші дії. Ми не знаємо точно, що зробить метод *initialize* класу *SimpleSwitchMorph* (і можемо про це не турбуватися), але з великою імовірністю він встановить певні доцільні початкові значення успадкованим полям. Тому нам краще викликати його, щоб не зіткнутись з якимось невідомим станом морфи.

Далі метод налаштовує стан екземпляра. Наприклад, надсилаючи *self label: ''*, він робить порожній рядок написом цього об'єкта.

Про створення точки та прямокутника

Вираз «*0 @ 0 corner: 16 @ 16*», здається, варто пояснити докладніше. *0 @ 0* створює об'єкт класу *Point* з обома координатами, *x* та *y*, які дорівнюють нулю. Точніше, *0 @ 0* надсилає повідомлення @ числу *0* з аргументом *0*. У результаті число *0* просить клас *Point* створити новий екземпляр з координатами (0; 0). Тепер ми відправляємо цій новоствореній точці повідомлення *corner: 16 @ 16*, що змушує її створити екземпляр *Rectangle* з вершинами (0; 0) та (16; 16). Цей новостворений прямокутник буде присвоєно змінній *bounds*, успадкованій від надкласу. Змінна *bounds* визначає, якого розміру буде наша морфа. По суті, ми сказали: «Будь квадратом 16 на 16 пікселів».

Зауважимо, що початком системи координат вікна Pharo є лівий верхній кут, координата *x* зростає зліва направо, координата *y* – згори донизу.

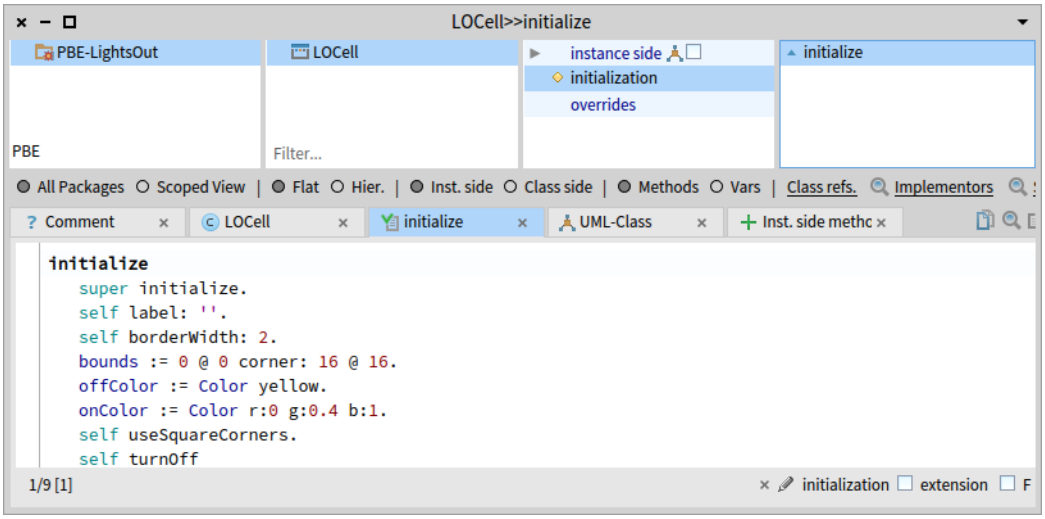

Рис. 6.4. Новостворений метод *initialize*

Про решту

Решта методу мала б «говорити» сама за себе. Частиною мистецтва написання хорошого коду Pharo є вибір хороших імен методів, щоб код можна було читати як ламану англійську. Ви могли б уявити об'єкт, який розмовляє сам з собою і каже: «Наказую собі: використовуй квадратні кути!», «Наказую собі: вимкнися!».

Зверніть увагу на маленьку синю стрілку біля імені методу (див. рис. 6.4). Вона означає, що *initialize* визначений у надкласі і перевизначений у вашому класі. Ще одна дія відбулася без нашої участі: середовище автоматично зачислило наш метод до протоколу *initialization*.

6.7. Інспектування об'єкта

Ви можете невідкладно випробувати дію написаного коду, створивши новий об'єкт класу *LOCell* та проінспектувавши його. Відкрийте Робоче вікно або Пісочницю, введіть вираз «*LOCell new*» і оберіть «*Inspect it*» з контекстного меню (чи натисніть [*Cmd* + *I*]).

Лівий стовпець вкладки *Raw* інспектора показує список полів, а правий – їхні значення (див. рис. 6.5). Інші вкладки демонструють інші сторони *LOCell*. Ми переглянемо їх та поекспериментуємо.

Якщо ви клацнете на одному з полів, то Інспектор відкриє нову панель з деталями обраного поля (див. рис. 6.6). Закрити її можна кнопкою з хрестиком (вгорі праворуч).

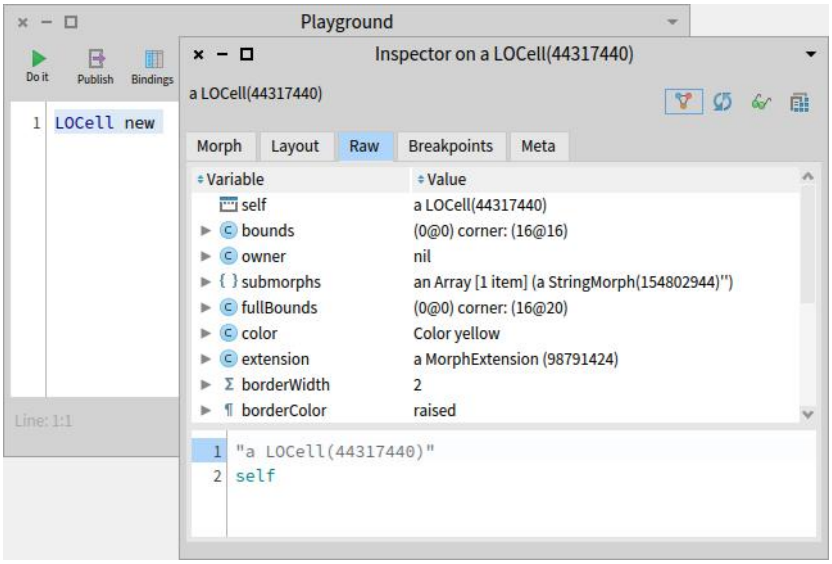

Рис. 6.5. Дослідження екземпляра *LOCell* за допомогою Інспектора

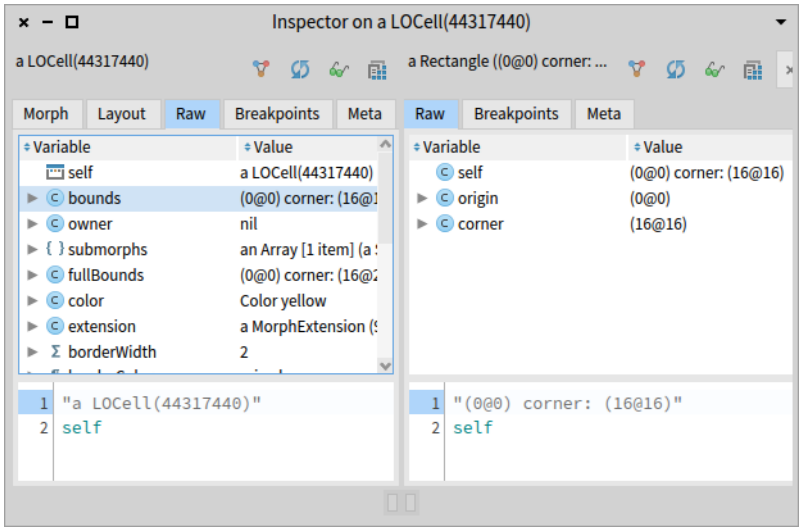

Рис. 6.6. Щоб проінспектувати значення змінної екземпляра (інший об’єкт), достатньо на ній клацнути

Виконання виразів

Нижня панель інспектора працює як маленьке робоче вікно. Вона корисна тим, що у ній псевдозмінна *self* прив’язана до вибраного об’єкта.

Перейдіть до цієї панелі внизу вікна Інспектора, наберіть такий текст: «*self bounds: (100@50 corner: 150@100)*» та виконайте «*Do it*». Значення змінної *bounds* має змінитися в Інспекторі автоматично. Якщо цього не відбулося, то натисніть на кнопку *Refresh* (пара синіх закручених стрілок) у правому верхньому куті вікна, щоб примусово оновити відображення в Інспекторі. Тепер наберіть текст «*self openInWorld*» у робочій панелі та виконайте його.

Меню-ореол

У лівому верхньому куті екрана має з’явитись клітинка, як показано на рис. 6.7. Вона з’явилась саме у тій позиції і того розміру, що вказані у змінній *bounds*: нижче від верхнього краю вікна на 50 пікселів та правіше від лівого на 100, ширина та висота клітинки – по 50 пікселів.

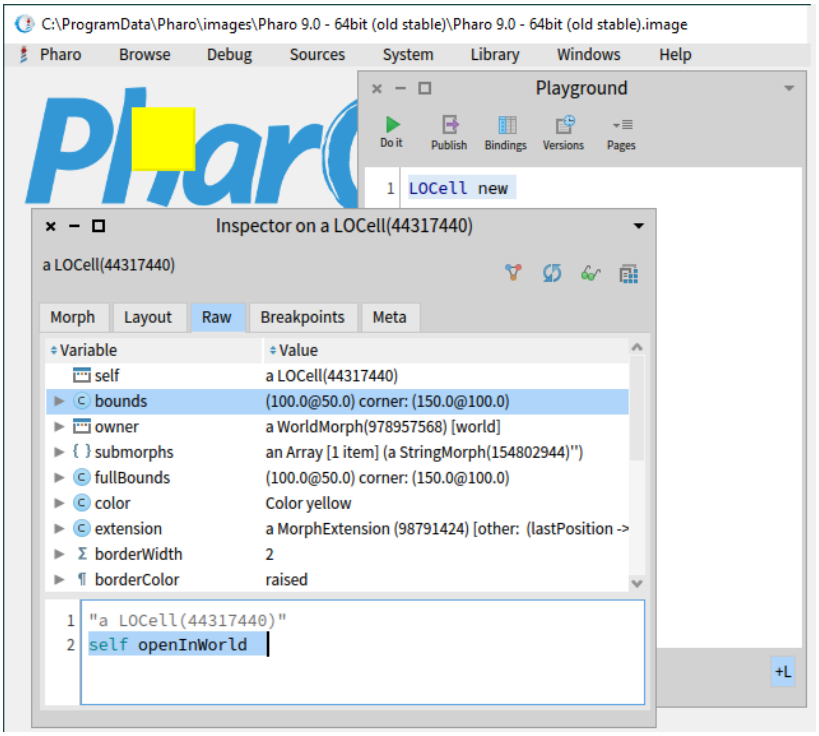

Рис. 6.7. Екземпляр *LOCell* відкрито у *World*

Метаклацніть на клітинці, щоб відкрити її меню-ореол. Це меню надає візуальний спосіб взаємодії з екземпляром класу *Morph* за допомогою маніпуляторів, що зараз оточують морф. Перетягніть клітинку по екрану чорним маніпулятором (вгорі посередині), змініть розмір клітинки жовтим маніпулятором (внизу праворуч). Простежте, як змінюються значення *bounds* в Інспекторі. (Можливо, вам доведеться натиснути кнопку *Refresh*). Зверніть увагу на те, що щойно створений морф уже має поведінку: клацніть на ньому, і колір клітинки зміниться! Закрийте клітинку, клацнувши на рожевому маніпуляторі з хрестиком.

6.8. Визначення класу *LOGame*

Давайте створимо ще один потрібний для гри клас. Назвемо його *LOGame*.

Зробіть видимим шаблон створення класу у вікні редагування коду Оглядача класів. Для цього клацніть на назві пакета (або виберіть команду «*Add Class*» з контекстного меню панелі класів). Відредагуйте код, щоб він виглядав, як показано в лістингу 6.3, і збережіть його.

Лістинг 6.3. Визначення класу *LOGame*

```
BorderedMorph subclass: #LOGame
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-LightsOut'
```

Тут ми наслідуємо клас *BorderedMorph*. Клас *Morph* є базовим для всіх графічних фігур у Pharo. Ми вже бачили *SimpleSwitchMorph*, такий *Morph*, який можна увімкнути та вимкнути. Закономірно, що *BorderedMorph* – це *Morph*, який має межу. Ми також могли б вставити імена змінних екземпляра між лапками в другому рядку, але, поки що, залишимо цей список порожнім.

Лістинг 6.4. Метод ініціалізації гри

```
LOGame >> initialize
  | sampleCell width height n |
  super initialize.
  n := self cellsPerSide.
  sampleCell := LOCell new.
  width := sampleCell width.
  height := sampleCell height.
  self bounds: (5 @ 5 extent: (width * n) @ (height * n) + (2 * self
        borderWidth)).
  cells := Array2D new: n tabulate: [ :i :j | self newCellAt: i at: j ]
```

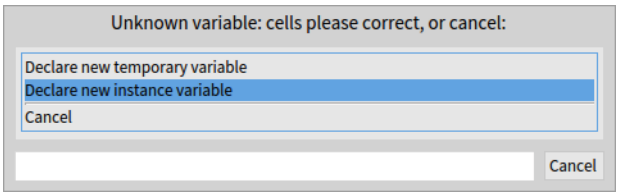

Рис. 6.8. Визначення нової змінної екземпляра «на льоту»

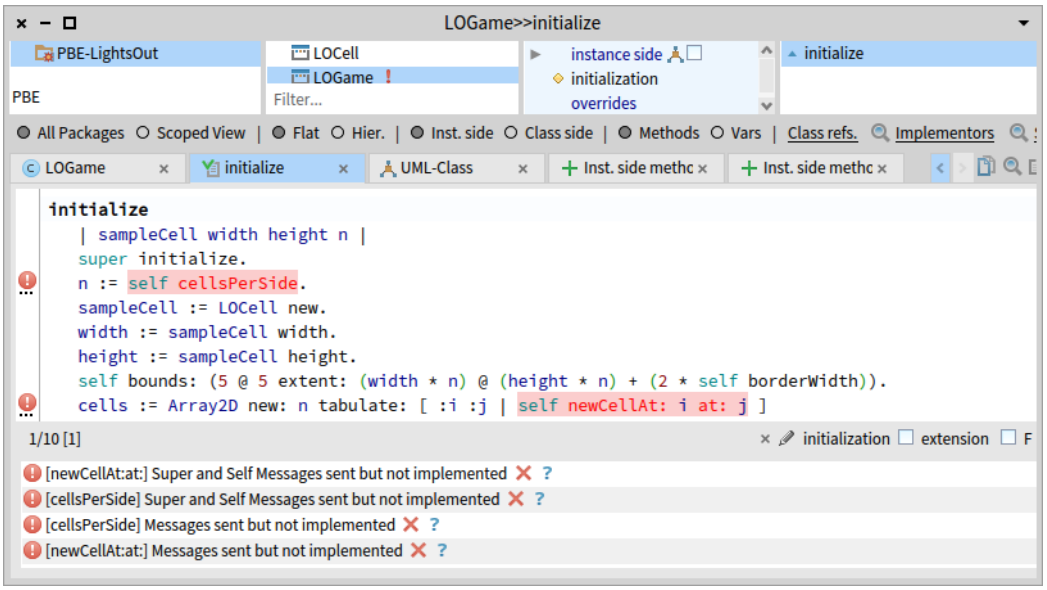

Рис. 6.9. Помилки, знайдені в тексті методу ініціалізації

6.9. Ініціалізація гри

Тепер визначимо метод *initialize* для *LOGame*. Введіть код з лістингу 6.4 в Оглядачі як метод екземпляра *LOGame*, і спробуйте зберегти його. Це досить об'ємний метод, проте не переживайте, ми детально пояснимо кожен його рядок у наступних параграфах.

Pharo поскаржиться, що він не знає, що означає *cells* (див. рис. 6.8), і запропонує вам кілька способів як виправити ситуацію. Оберіть «*Declare new instance variable*», оскільки нам потрібне поле даних екземпляра.

Відкрийте вкладку «*LOGame*» Оглядача і переконайтеся, що в оголошенні класу як за помахом чарівної палички з'явилося оголошення змінної *cells*. Поверніться до вкладки з оголошенням методу *initialize*, і ви побачите, що компілятор виявив ще дві проблеми в тексті методу (рис. 6.9). Наступний параграф розповість, яким незвичним способом можна їх залагодити: ми спробуємо *виконати* проблемний код!

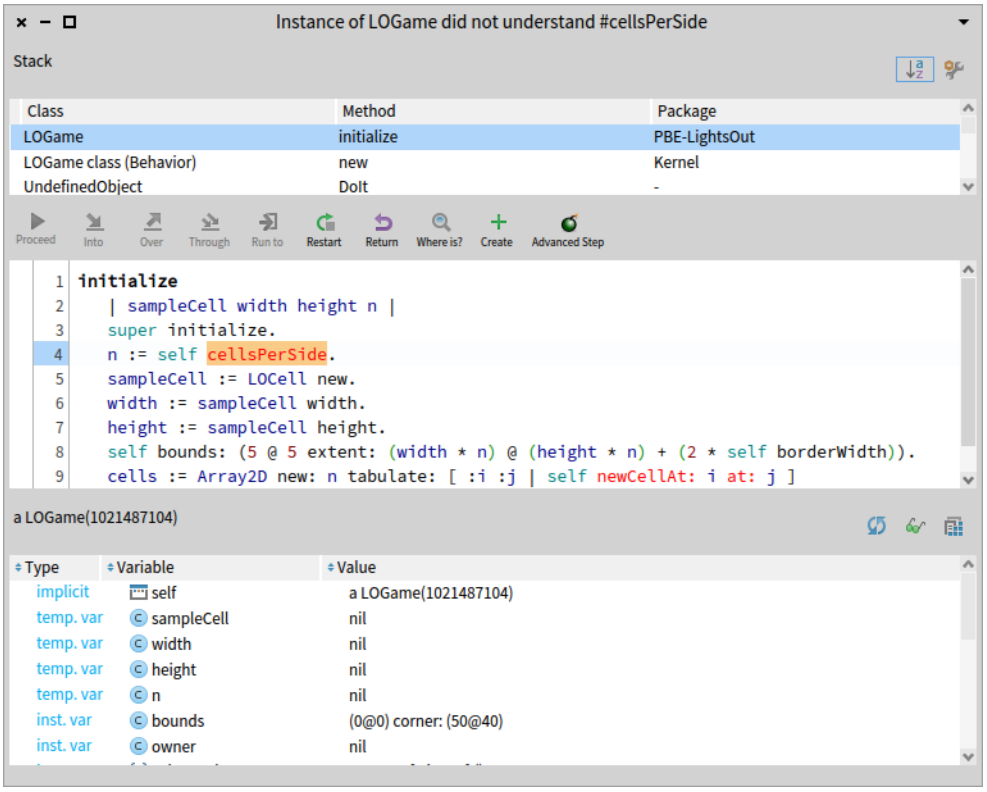

Рис. 6.10. Pharo знайшов невідомий селектор

6.10. Використання переваг Налагоджувача

На цьому етапі, якщо ви відкриєте Робоче вікно, введете фрагмент «*LOGame new*» і виконаєте його, Pharo поскаржиться, що йому невідомий зміст деяких виразів (див. рис. 6.10). Він відкриє Налагоджувач і повідомить, що екземпляр *LOGame* не розуміє повідомлення *cellsPerSize*. Проте *cellsPerSize* не є помилкою: просто це метод, який ще не визначили. Зараз ми це зробимо.

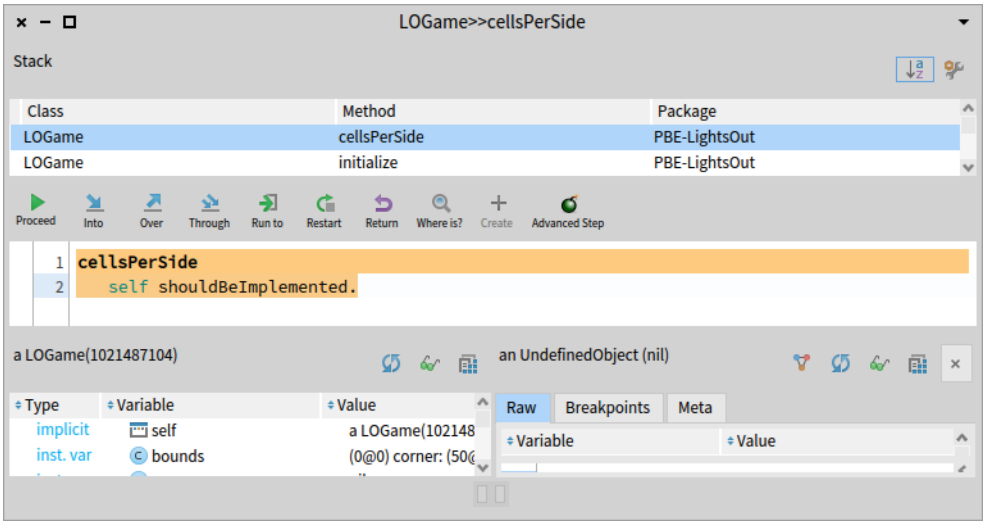

Рис. 6.11. Система створила новий метод, тіло якого потрібно визначити

Не закривайте Налагоджувач, а натисніть на його кнопку ***Create***. У відповідь на запит про клас, який міститиме метод, оберіть *LOGame*. У запиті про протокол методу введіть «*accessing*» і натисніть ***Ok***. Налагоджувач створить метод *cellsPerSize* на льоту й одразу запустить його на виконання. Створена так стандартна реалізація методу помістила в

його тіло повідомлення «*self shouldBeImplemented*». Його виконання запускає виняток і ... знову відкриває Налагоджувач на визначенні методу (рис. 6.11).

Тепер ви можете написати визначення нового методу. Цей метод важко зробити простішим: він завжди повертає константу 10.

```
LOGame >> cellsPerSide
  "The number of cells along each side of the game"
  ^ 10
```

Однією з переваг подання констант за допомогою методів є те, що у випадку розвитку програми, коли значення такої константи стає залежним від інших властивостей, метод можна легко змінити для обчислення цього значення.

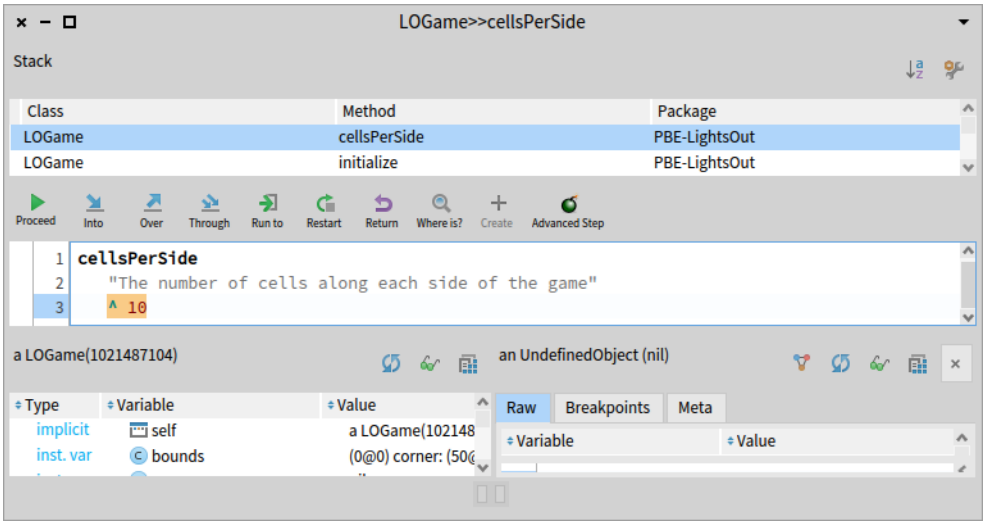

Рис. 6.12. Визначення методу *cellsPerSide* в Налагоджувачі

Не забудьте скомпілювати метод, як тільки напишете його, за допомогою все тієї ж команди «*Accept*». Ви мали б отримати ситуацію, зображену на рис. 6.12. Якщо ви натиснете кнопку ***Proceed***, то програма продовжить своє виконання і зупиниться, бо ми не визначили метод *newCellAt:at:*.

Можемо визначити його, як і попередній, але поки що зупинимось, щоб пояснити детальніше, що вже зробили. Закрийте Налагоджувач і погляньте на визначення класу ще раз (натиснувши на *LOGame* на панелі класів Системного оглядача). Ви побачите, що Оглядач доповнив визначення класу так, що воно тепер містить змінну екземпляра *cells*. Клацніть на методі *initialize* – тут також відбулися певні зміни. Колір повідомлення *cellsPerSide* змінився з червоного на звичайний чорний, і поменшало повідомлень про помилки від Помічника з якості. Червоним залишилося лише повідомлення *newCellAt:at:*.

6.11. Вивчаємо метод *initialize*

Давайте розберемо метод *initialize*. Для зручності пояснення ми перенумерували його рядки.

Рядок 2

У рядку 2 вираз «| *sampleCell width height n* |» оголошує чотири тимчасові змінні. Їх називають тимчасовими, тому що їхня область видимості та тривалість життя

обмежені цим методом. Тимчасовим змінним дають пояснювальні імена, щоб зробити код легшим для сприйняття. Рядки 4–7 задають значення цих змінних.

```
1 initialize
2    | sampleCell width height n |
3    super initialize.
4    n := self cellsPerSide.
5    sampleCell := LOCell new.
6    width := sampleCell width.
7    height := sampleCell height.
8    self bounds: (50 @ 50 extent:
                      (width * n)@(height * n) + (2 * self borderWidth)).
9    cells := Array2D
        new: n
        tabulate: [ :i :j | self newCellAt: i at: j ]
```

Рядок 4

Якого розміру мало б бути поле для гри? Достатнього, щоб вмістити деяку невід'ємну кількість клітинок та намалювати межу навколо них. Скільки клітинок має бути? 5? 10? 100? Наразі ми не знаємо, яке число вибрати, але, якби й знали, то ймовірно могли б змінити свій вибір пізніше. Тому ми делегуємо відповідальність за знання цієї кількості на інший метод, який називаємо *cellsPerSide* (його ще не існувало на момент написання методу ініціалізації). Не лякайтесь цього, насправді це є хорошою практикою писати код, звертаючись до ще не визначених методів. Чому? Бо поки ми не почали писати метод *initialize*, ми й не здогадувались, що нам знадобиться *cellsPerSide*. І в цей момент ми можемо дати йому змістовне ім'я та продовжувати, без переривання процесу. Як ми вже бачили, такий метод легко визначити під час випробування об'єкта, а можливість відкласти реалізацію на потім є суперсилою Pharo.

Отже, рядок 4 відправляє повідомлення *cellsPerSide* до *self*, тобто самому собі. Відповідь, кількість клітинок на одній стороні ігрового поля присвоюється змінній *n*.

Наступні три рядки створюють новий екземпляр *LOCell*, присвоюють його ширину і висоту відповідним тимчасовим змінним. Навіщо все це? Розміри клітинки потрібні, щоб обчислити розміри поля, а в кого ж їх довідатися, як не в самої клітинки? Найкраще з можливих місць зберігання розмірів клітинки – екземпляр класу *LOCell*.

Рядок 8

Рядок 8 задає межі нового об'єкта гри. Не вникаючи у деталі, повірте, що вираз у дужках створює квадрат з лівим верхнім кутом у точці (50; 50) і правим нижнім кутом достатньо далеко, щоб вмістити потрібну кількість клітинок.

Останній рядок

Останній рядок присвоює полю *cells* новостворену матрицю, екземпляр класу *Array2D*, з правильною кількістю рядків і стовпців. Ми створили матрицю, надіславши повідомлення «*new:tabulate:*» до класу *Array2D*. Класи також є об'єктами, тому можемо надсилати їм повідомлення. Ми знаємо, що «*new:tabulate:*» приймає два аргументи, бо він має дві двокрапки (:) в імені. Аргументи вказують зразу після двокрапок. Якщо ви звикли до мов, що передають аргументи всі разом всередині дужок, то це спочатку може видатись дивним. Але не панікуйте, це лише синтаксис. Виявляється, що це дуже навіть хороший синтаксис, оскільки назву методу можна використовувати для пояснення

призначення аргументів. Наприклад, цілком зрозуміло, що «*Array2D rows: 5 columns: 2*» має 5 рядків і 2 стовпці, а не 2 рядки та 5 стовпців.

«*Array2D new: n tabulate: [:i :j | self newCellAt: i at: j]*» створює новий двовимірний масив (матрицю) *n*×*n* та ініціалізує її елементи. Початкове значення кожного елемента залежить від його координат. Елемент на позиції (*i*, *j*) буде ініціалізовано результатом обчислення «*self newCellAt: i at: j*».

6.12. Поділ методів на протоколи

Перш ніж визначати інші методи, глянемо на третю панель угорі Оглядача. Так само, як перша панель Оглядача дає змогу нам класифікувати класи на пакети, панель протоколів допомагає класифікувати методи, щоб не доводилось працювати з дуже довгим списком імен на панелі методів. Такі групи методів називають «протоколами».

За замовчуванням ви матимете віртуальний протокол «*instance side*», який містить усі методи класу.

Якщо ви виконували приклад створення гри, то панель протоколів вашого Оглядача мала б містити протоколи «*initialization*» та «*overrides*». Вони додаються автоматично, коли ви перевизначаєте метод «*initialize*». Системний оглядач автоматично організовує методи та додає їх до відповідного протоколу, коли тільки можливо.

Звідки оглядач знає, як вибрати правильний протокол? Загалом він не знає напевне, але може робити певні припущення. Наприклад, якщо метод *initialize* визначено в надкласі, то він припустить, що наш метод *initialize* потрібно зачислити до того ж протоколу, що й метод, який він перевизначає.

Панель протоколів може також містити протокол «*as yet unclassified*», до якого належать усі некласифіковані методи. Тоді, щоб виправити ситуацію, оберіть команду «*categorize all uncategorized*» з контекстного меню панелі протоколів, або класифікуйте кожен метод вручну.

6.13. Завершення розробки гри

А зараз давайте визначимо інші методи, використані в *LOGame>>initialize*. Ви можете використовувати для цього і Оглядач класів, і Налагоджувач. У будь-якому випадку почнемо з *LOGame>>newCellAt:at:* у протоколі *initialization*.

```
LOGAme >> newCellAt: i at: j

  "Create a cell for position (i,j) and add it to my on-screen
  representation at the appropriate screen position.
  Answer the new cell"

    | c origin |
    c := LOCell new.
    origin := self innerBounds origin.
    self addMorph: c.
    c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
    c mouseAction: [ self toggleNeighboursOfCellAt: i at: j ]
```

Зауваження. Наведений код спричинить виняток, оскільки містить навмисну помилку.

Форматування

Як ви можете бачити, у коді є порожні рядки та відступи. Для того, щоб дотримуватись однакових правил форматування коду, можете покластися на допомогу Pharo: виберіть команду «*Format code*» з контекстного меню панелі редагування методу, або використайте [*Cmd* + *Shift* + *F*]. Це автоматично відформатує текст вашого методу.

Від перекладача. Команду «*Format code*» відшукати не так просто: вона розташована останньою в підменю розділу «*Source code*» контекстного меню панелі редагування методу. Цей розділ – перший розділ меню, але з'являється тільки після того, як збережено текст методу. Отже, для швидкого форматування використовуйте гарячі клавіші.

Перемикання сусідів

Визначений вище метод створює нову *LOCell* та задає її екранне розташування залежно від індексів (*i, j*) у матриці клітинок. Останній рядок робить значенням поля *mouseAction* нової клітинки блок *[self toggleNeightboursOfCellAt: i at: j]*. У ньому визначено поведінку обробника події клацання мишкою на клітинці. Відповідний метод також потрібно визначити.

Лістинг 6.5. Метод опосередкованого виклику

```
LOGAme >> toggleNeighboursOfCellAt: i at: j
  i > 1
        ifTrue: [ (cells at: i - 1 at: j) toggleState ].
  i < self cellsPerSide
        ifTrue: [ (cells at: i + 1 at: j) toggleState ].
  j > 1
        ifTrue: [ (cells at: i at: j - 1) toggleState ].
  j < self cellsPerSide
        ifTrue: [ (cells at: i at: j + 1) toggleState ]
```

Метод *toggleNeighboursOfCellAt:at:* перемикає стан чотирьох сусідніх клітинок угорі, внизу, ліворуч і праворуч від клітинки (*i, j*). Єдине ускладнення полягає в тому, що ігрове поле обмежене, тому перш ніж перемикати стан клітинки, потрібно переконатись, що вона існує.

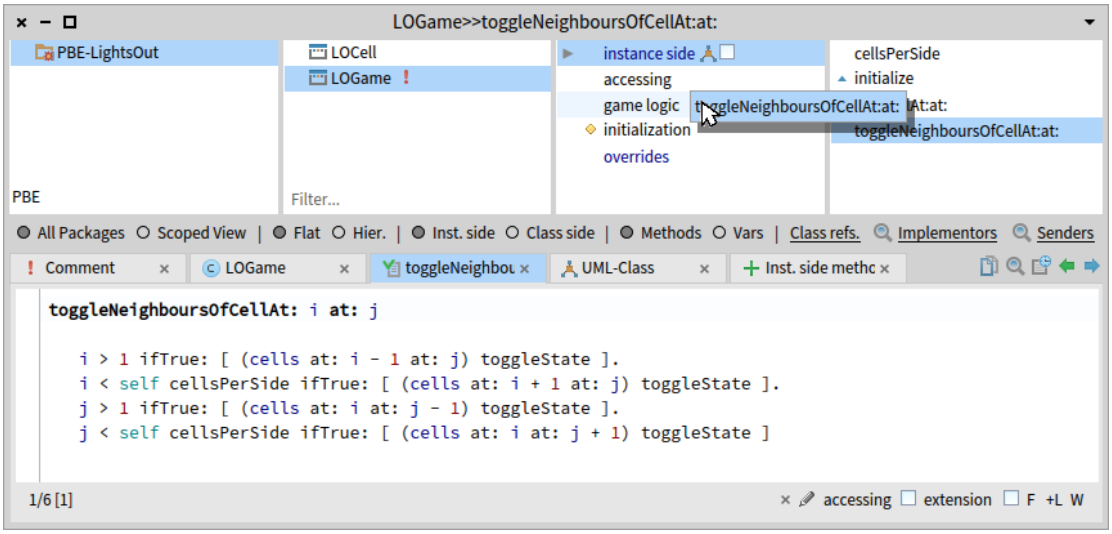

Рис. 6.13 Перетягування методу в протокол

Помістіть цей метод у новий протокол, що називається «*game logic*». Щоб додати новий протокол, використайте відповідну команду контекстного меню панелі протоколів. Тепер, щоб перемістити метод до нового протоколу, просто перетягніть його мишкою (див. рис. 6.13).

Лістинг 6.6. Типовий метод-модифікатор

```
LOCell >> mouseAction: aBlock
  mouseAction := aBlock
```

Лістинг 6.7. Обробник події

```
LOCell >> mouseUp: anEvent
  mouseAction value
```

6.14. Останні методи класу *LOCell*

Для завершення розробки гри Lights Out нам потрібно визначити ще два методи у класі *LOCell*. Цього разу для обробки подій мишки.

Перший з них, зображений у лістингу 6.6, є звичайним методом доступу. Він не робить нічого особливого, лише присвоює змінній *mouseAction* клітинки свій аргумент. Ми мали б оголосити його в протоколі «*accessing*».

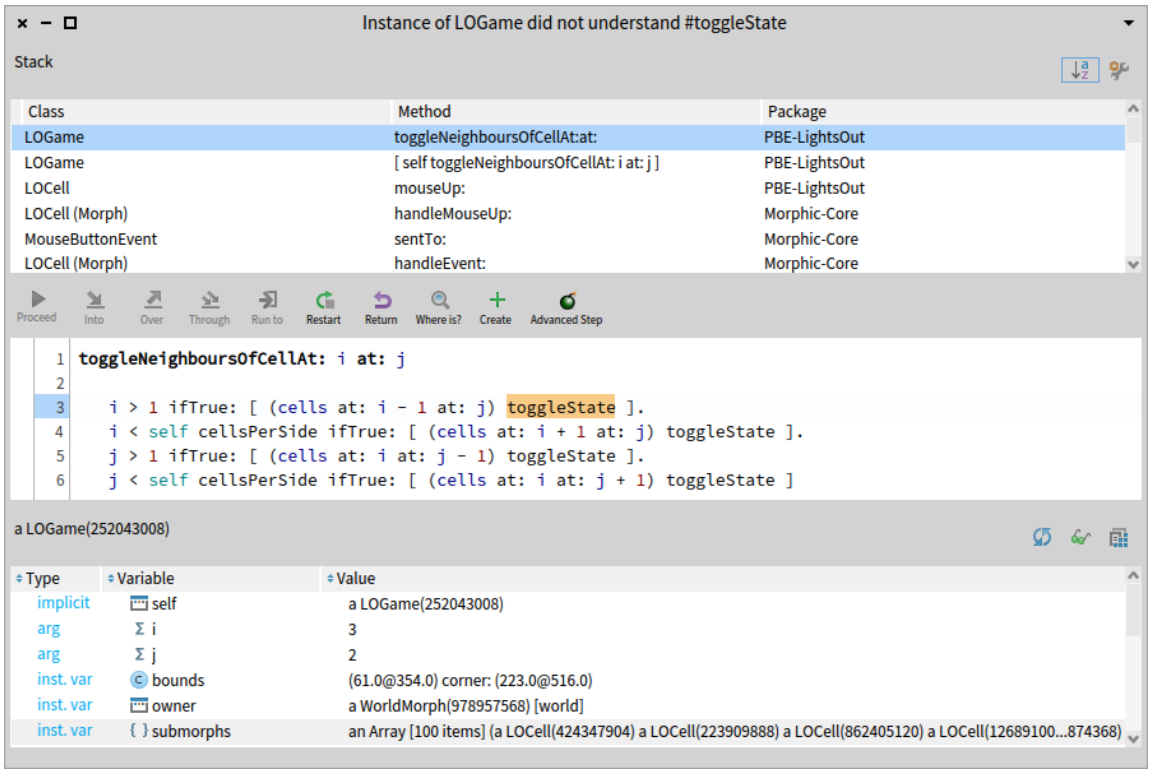

Рис. 6.14. Налагоджувач підсвічує метод, який спричинив помилку

Нам залишилось визначити метод *mouseUp:*. Його автоматично викликатиме графічний інтерфейс користувача в момент відпускання кнопки мишки, коли курсор є над певною клітинкою. Перегляньте лістинг 6.7: метод надсилає повідомлення *value* об'єктові, що зберігається у змінній екземпляра *mouseAction*. В методі *LOGame >> newCellAt:at:* ми створили блок *[self toggleNeighboursOfCellAt: i at: j]*, виконання якого перемикає всіх сусідів клітинки, та присвоїли його змінній *mouseAction* клітинки. Отже, надсилаючи

повідомлення *value* цьому блокові, ми примушуємо його виконатися та, як наслідок, перемкнути стан сусідніх клітинок.

6.15. Випробуємо код

Це все: створення гри Lights Out завершене! Якщо ви виконали усі кроки, то зможете зіграти у гру, що складається лише з двох класів і семи методів. У Робочому вікні або Пісочниці наберіть «*LOGame new openInHand*» та виконайте (командою «*Do it*»).

Гра відкриється причеплена до курсору: ви зможете перемістити її в довільне зручне місце на екрані та зафіксувати там клацанням. Далі ви мали б клацати по клітинках і бачити, як усе працює. Принаймні так в теорії... Але коли ви клацнете по клітинці, з'явиться налагоджувач. У верхній частині його вікна ви побачите стек виконання, що показує усі активні методи. Виберіть будь-який з них, і в середній панелі з'явиться код, що виконується у цьому методі. Частина, що спричинила помилку, буде підсвічена.

Натисніть на рядку з написом «*LOGame toggleNeighboursOfCellAt:at: PBE-LightsOut*» (вгорі списку). Налагоджувач покаже контекст виконання методу, де трапилася помилка (див. рис. 6.14).

Унизу Налагоджувача є ділянка змінних, що належать до контексту виконання. Ви можете інспектувати об'єкт – отримувач повідомлення, яке спричинило виконання методу, позначеного в Налагоджувачі. То ж ви можете переглянути тут значення змінних екземпляра-отримувача. Також можете побачити значення аргументів методу так само, як і проміжні значення, отримані під час виконання.

За допомогою Налагоджувача можете покроково виконувати код, інспектувати параметри методів і локальні змінні, виконувати фрагменти коду так само, як у Робочому вікні, і, що найбільш несподівано для користувачів інших налагоджувачів, змінювати сам код під час його налагодження! Деякі Pharo-програмісти майже постійно працюють у Налагоджувачі – більше, ніж в Оглядачі класів. Перевагою такого підходу є те, що ви бачите, як буде виконано метод, який пишете, зі справжніми параметрами та в актуальному контексті виконання.

У нашому випадку заголовок вікна Налагоджувача (див. рис. 6.14) інформує, що повідомлення *toggleState* надійшло до екземпляра *LOGame*, хоча очевидно, що то мав би бути екземпляр *LOCell*. Найбільш ймовірно, що проблема є з ініціалізацією матриці клітинок. Переглянувши код *LOGame >> initialize*, бачимо, що *cells* заповнено значеннями, які повертає метод *newCellAt:at:*, але, якщо уважніше розглянути його текст, то побачимо, що там немає виразу повернення! За замовчуванням метод у Pharo повертає *self* – отримувача, який у випадку *newCellAt:at:* справді є екземпляром *LOGame*. Як ми уже говорили, для повернення значення з методу у Pharo використовують вираз «^ *значення*».

Лістинг 6.8. Виправлення помилки

```
LOGAme >> newCellAt: i at: j

  "Create a cell for position (i,j) and add it to my on-screen
  representation at the appropriate screen position.
  Answer the new cell"

    | c origin |
    c := LOCell new.
```

```
        origin := self innerBounds origin.
        self addMorph: c.
        c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
        c mouseAction: [ self toggleNeighboursOfCellAt: i at: j ].
        ^ c
```

Лістинг 6.9. Перевизначення події переміщення мишки

```
LOCell >> mouseMove: event
    "Do nothing"
```

Отже, виправте знайдену помилку. Закрийте вікно Налагоджувача. Додайте вираз «^ *c*» у кінці методу *LOGame>>newCellAt:at:*, щоб він повертав *c*. Не забудьте закінчити попередній рядок крапкою (див. лістинг 6.8).

У багатьох випадках ви можете виправити код безпосередньо в налагоджувачі і, натиснувши ***Proceed***, продовжити виконання програми. У багатьох, але не в нашому. Через те, що помилка була в ініціалізації об'єкта, а не в методі, найпростіше буде закрити вікно Налагоджувача, зупинити запущений екземпляр гри і створити новий. Щоправда, відкрити нашу гру простіше, ніж закрити. Вона не має кнопок керування, тому скористаємося меню-ореолом. Чи вдалося вам метаклацнути на межі вікна гри, що має товщину один піксель? Якщо ні, то закрийте спочатку морфу будь-якої клітинки (через меню-ореол), а тоді – морфу поля гри.

Виконайте «*LOGame new openInHand*» знову, бо, якщо використати старий екземпляр гри, то працюватиме блок зі старою логікою.

Зараз гра має працювати правильно... або майже правильно. Якщо ми порухаємо мишку у проміжок часу між натисканням кнопки та відпусканням, тоді клітинка, над якою перебувала мишка, також змінить свій стан. Виявляється, це поведінка, успадкована від *SimpleSwitchMorph*. Ми можемо виправити помилку, просто перевизначивши метод *mouseMove*: так, щоб він нічого не робив (див. лістинг 6.9). Зауважимо також, що методи опрацювання подій мишки варто зачислити до протоколу «*event handling*».

Нарешті ми закінчили!

6.16. Домовленості щодо найменування методів доступу

Якщо ви використовували методи-селектори та методи-модифікатори в інших мовах програмування, то можете очікувати, що методи доступу до змінної екземпляра *mouseAction* називатимуться *getMouseAction* та *setMouseAction*. Але у Pharo діє інша домовленість про іменування. Селектор завжди має таке саме ім'я, як і змінна, яку він повертає, а модифікатор має таке ж ім'я, але з двокрапкою наприкінці: *mouseAction* та *mouseAction:*, відповідно. Селектори і модифікатори разом називають *методами доступу* (accessor methods) і за домовленістю їх зачисляють до протоколу *accessing*. У Pharo всі змінні об'єкта є приватними для нього, тому інші об'єкти можуть читати чи записувати ці змінні виключно через методи доступу, як ті, що описані вище. Звичайно, об'єкт має вільний доступ до всіх успадкованих полів, але ви ніколи не зможете доступитися до змінних іншого об'єкта, навіть, якщо він є екземпляром вашого класу або самим класом.

6.17. Про налагоджувач

Коли трапляється помилка у Pharo, система за замовчуванням відображає налагоджувач. Проте ми можемо повністю контролювати цю поведінку. Наприклад, можемо записати помилку у файл. Можемо навіть серіалізувати стек виконання до файлу, архівувати і відкрити його в іншому образі системи. Під час розробки програм Налагоджувач дає нам змогу рухатись так швидко, як це можливо. Проте у готових системах розробники часто налаштовують налагоджувач так, щоб їхні помилки не заважали занадто сильно роботі їхніх користувачів.

Якщо все пішло не так

Передусім не нервуйте! Цілком нормально творити безлад під час написання програм. Якщо що, то це правило, а не виняток. Напевно, найнеприємніше, що може трапитися, коли ви починаєте експериментувати з графічними елементами в Pharo, – це те, що екран захаращують незнищенні, здавалося б, морфи. Не панікуйте. Спробуйте отримати інспектора на одній з них: метаклацніть на ній, щоб відкрити меню-ореол, виберіть бузковий маніпулятор з гайковим ключем на ньому – це «*debug*», оберіть з його меню команду «*inspect morph*». Як тільки ви відкриєте Інспектора, ви перемогли! Щоб закрити гру, наберіть у панелі редактора в Інспекторі та виконайте:

- якщо ви інспектуєте екземпляр гри, то «*self delete*»;
- якщо ви інспектуєте клітинку гри, то «*self owner delete*».

Від перекладача. Метаклацання – це ще те завдання! Використання ореолу різноколірних маніпуляторів ні на що не схоже. Хотілося б мати якийсь звичний і надійний спосіб контролю запуску та завершення гри, наприклад, за допомогою меню. Наступний параграф описує, як це зробити. Удосконалення гри стане хорошою нагодою продемонструвати нові корисні можливості Pharo.

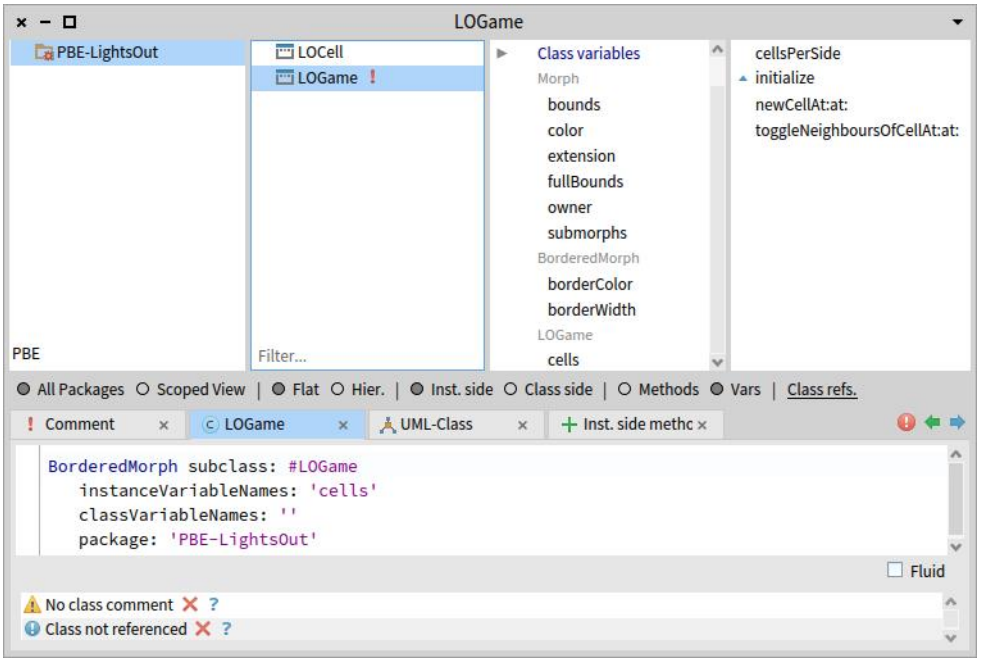

Рис. 6.15. Перелік змінних екземпляра класу *LOGame*

6.18. Удосконалення гри[5]

Ми щойно закінчили, а вже виникли думки щодо удосконалення нашого продукту. Чи дуже зручно вам закривати гру маніпулятором меню-ореола? Нам теж ні. Ніяк не вдається поцілити вказівником мишки в межу поля гри… Здається, варто зробити її ширшою. Але широка чорна рамка виглядатиме сумно, то ж змінимо також і її колір. Щоб досягти бажаного, доведеться провести невелике дослідження ієрархії класів, з якої наслідує клас *LOGame*.

Зміна методу ініціалізації

Значення товщини межі та її кольору мали б десь зберігатися. Давайте з'ясуємо, які поля даних успадкував *LOGame*. Відкрийте Оглядач класів на *LOGame* і оберіть ***Vars*** на перемикачі ***Methods/Vars*** Оглядача (нижче від панелі протоколів). Панель протоколів відобразить перелік усіх змінних екземпляра, структурований за класами, в яких ці поля оголошені, як на рис. 6.15. Бачимо, що надклас *LOGame* містить саме ті поля, які нам потрібні: *borderColor* і *borderWidth*.

Відразу виникає спокуса задати цим змінним бажані значення в методі *LOGame >> initialize*. Наприклад, так:

```
initialize
  | sampleCell width height n |
  super initialize.
  borderColor := Color blue.
  borderWidth := 3.
  n := self cellsPerSide.
  . . .
```

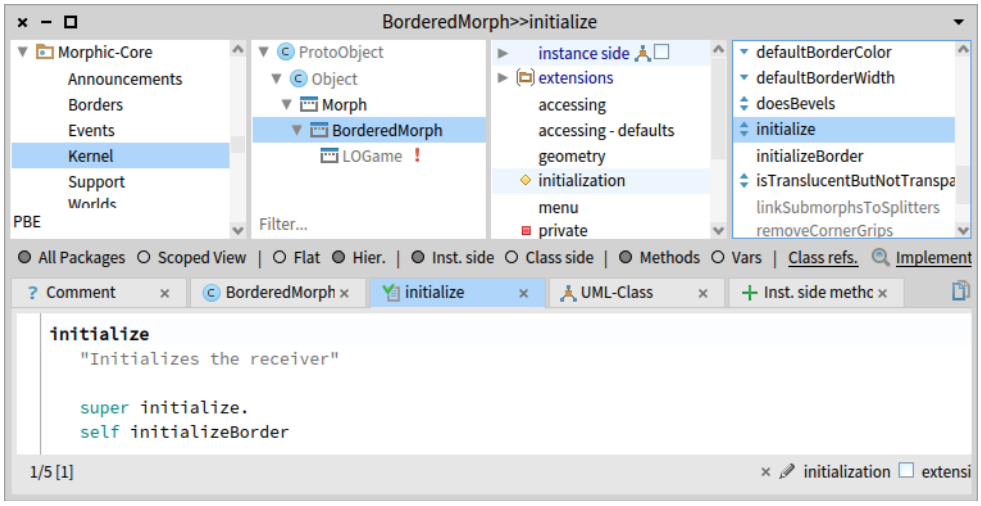

Рис. 6.16. Ієрархія наслідування класу *LOGame*

Якщо зберегти зроблені зміни і запустити гру, то побачимо, що все працює, гра набула саме такого вигляду, як на рис. 6.1. Але не все так добре, як може видатися на перший погляд. Адже поля *borderColor* і *borderWidth* оголошені в класі *BorderedMorph*, і саме він має задавати їхні початкові значення. Схоже, ми запрограмували повторну ініціалізацію змінних *borderColor* і *borderWidth*. Це двічі неправильно: виконується зайва робота, і

[5] Цей параграф написав перекладач книги.

нові значення, задані в підкласі, можуть спричинити несподівані помилки. Давайте продовжимо дослідження коду і придумаємо кращий спосіб, щоб досягти бажаного.

Значення полів *borderColor* і *borderWidth* мав би задавати метод *BorderedMorph >> initialize*. Знайдемо його в Оглядачі класів. Нижче від панелі класів Оглядача є перемикач ***Flat/Hier.***, що приховує або вмикає відображення в панелі ієрархії наслідування класів незалежно від їхньої належності до пакетів. Оберіть ***Hier.***, послідовно клацніть на класі *BorderedMorph*, протоколі *initialization*, методі *initialize*. Ви мали б перейти до методу ініціалізації надкласу, як зображено на рис. 6.16.

З тексту методу легко здогадатися, що змінними *borderColor* і *borderWidth* займається метод *BorderedMorph >> initializeBorder*. А вже в його тілі знайдемо:

```
BorderedMorph >> initializeBorder
      "Initialize the receiver state related to border."

      borderColor:= self defaultBorderColor.
      borderWidth := self defaultBorderWidth
```

Тепер все зрозуміло: значення кольору і товщини межі задають відповідні методи класу *BorderedMorph*. Якщо ми хочемо задавати інші значення в нашому підкласі, то маємо перевизначити методи *defaultBorderColor* і *defaultBorderWidth*. Нагадаємо, що в мові Pharo псевдозмінна *self* вказує на отримувача повідомлення (виконавця методу) і слугує інструментом, за допомогою якого клас може передавати повідомлення своїм підкласам, навіть, ще не написаним.

Отож додамо до протоколу *initialization* класу *LOGame* два нові методи (а присвоєння змінним в методі *LOGame >> initialize* вилучимо).

```
LOGame >> defaultBorderColor
      "answer the default border color for the game board"
      ^ Color blue

LOGame >> defaultBorderWidth
      "answer the default border width for the game board"
      ^ 3
```

Тепер гра працює так, як треба, а ми зайвий раз переконалися, що задавати константи за допомогою методів – це правильний підхід. Такі значення легко модифікувати в підкласах, не порушуючи код інших методів.

Зміна способу запуску

Усі складові частини Pharo можна запустити програмно, або командою меню, або, навіть, комбінацією клавіш. А нашу гру ми запускаємо тільки програмно (з Пісочниці). Давайте виправимо цю несправедливість і додамо відповідну команду до якогось меню! Наприклад, до *World Menu*. Ми ще не вміємо цього робити, але Pharo – відкрита система, тому спробуємо.

Відкрийте Навідника і в рядку пошуку наберіть «*world menu*». Перше ж посилання серед результатів пошуку – *Breakpoint class>>#debugWorldMenuOn:* – показує, як класи системи додають команди до меню. Наприкінці списку результатів є посилання на розділ «World Menu Items» довідкової системи – це саме те, що потрібно. Сам Навідник довідки не відкриє, але нам не буде важко знайти його через команду «*Help > Help browser*» головного меню. У розділі написано, що для того, щоб додати новий пункт до головного

меню, потрібно визначити *метод класу*, який називається *menuCommandOn*:. Довідка надає також приклади оголошення такого методу в деякому вигаданому класі. Не зайвим буде також пошукати в Pharo справжні класи, які реалізують метод *menuCommandOn*:, і подивитися, як воно зроблено.

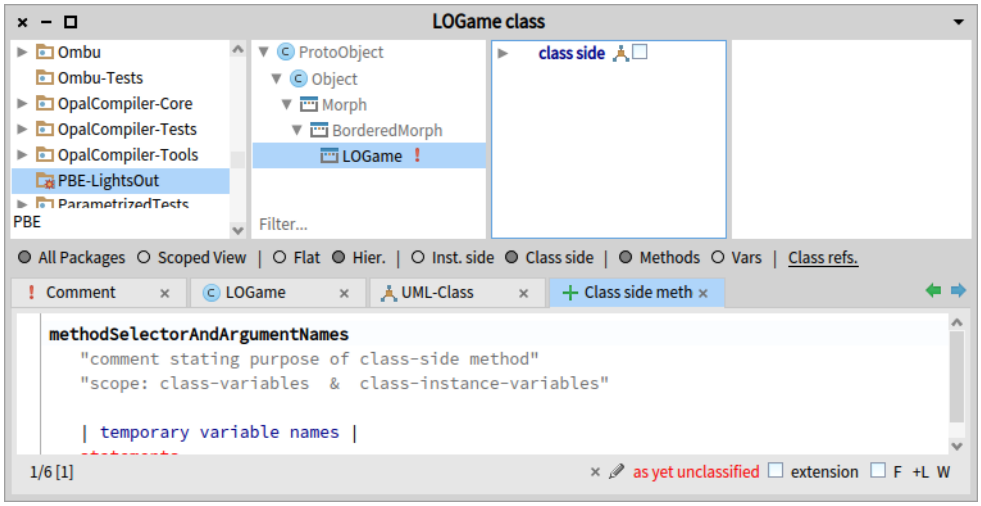

Рис. 6.17. Визначення методу класу

Давайте поміркуємо, чого ми хочемо від команди меню. Вона мала б виконувати той код, який ми набирали в Пісочниці, і запускати гру Lights Out. Але добре було б мати також команду для закривання гри. Займати два рядки головного меню – завелика розкіш, тому об'єднаємо команди відкривання та закривання в одне підменю.

Клацніть в Оглядачі класів на імені *LOGame* і оберіть ***Class side*** в перемикачі ***Inst. side/Class side*** (див. рис. 6.17). Оглядач відобразить перелік методів класу *LOGame* (він поки що порожній). Класи Pharo – також об'єкти, які можуть мати власні методи. Нам потрібно оголосити метод, зазначений в лістингу 6.10.

Лістинг 6.10. Додавання команди до головного меню

```
LOGame class >> menuCommandOn: aBuilder
      <worldMenu>
      (aBuilder item: #LightsOut)
         order: 5.0;
         withSeparatorAfter;
         with: [ (aBuilder item: #Run)
               order: 0;
               action: [ self open ].
            (aBuilder item: #Quit)
               order: 1;
               action: [ self close ] ]
```

Тут прагма *<worldMenu>* вказує на специфіку методу (про прагми ми поговоримо згодом), повідомлення *item:* створює пункт меню з назвою «*LightsOut*», а всі інші повідомлення надходять до цього пункту меню. Зверніть увагу на те, що рядки коду завершуються крапкою з комою. Так у Pharo задають каскад повідомлень, які надходять до того самого отримувача (до пункту меню в нашому випадку). Повідомлення *order:* задає розташування в меню, *withSeparatorAfter* відокремлює пункт від інших частин меню, а *with:* задає структуру вкладеного меню. Пункти підменю описують подібно, тільки замість *with:* використано *action: aBlock* для того, щоб задати реакцію меню на вибір команди.

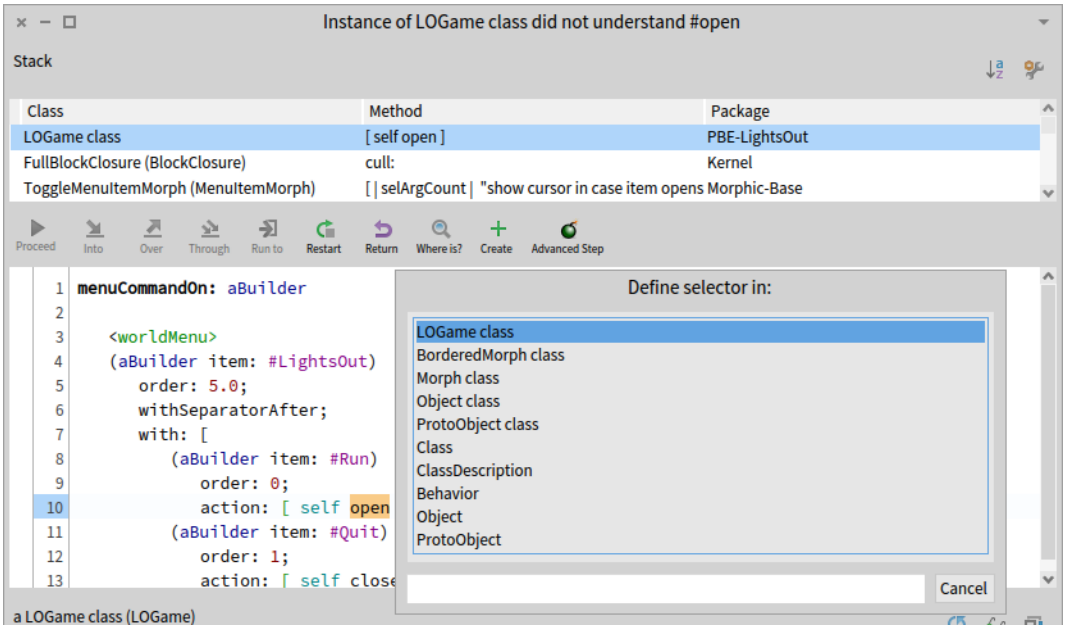

Рис. 6.18. Створення методу класу в Налагоджувачі

Зрозуміло, що команда меню «*Run*» має викликати *LOGame class >> open*, а команда «*Quit*» – *LOGame class >> close*. Ми ще не визначили таких методів класу, але можемо зробити це в Налагоджувачі, як визначали методи екземпляра раніше. То ж викличемо головне меню системи і виберемо «*LightsOut* > *Run*»! Прогнозовано нас «привітає» вікно Налагоджувача повідомленням про помилку. Ми знаємо, де вона трапилася, тому можемо відразу натиснути кнопку «*Create*», вказати клас «*LOGame class*» (див. рис. 6.18), протокол «*instance creation*» і ввести текст методу.

Знайомий нам код «*LOGame new openInHand*» створює безіменний об'єкт, з яким можна взаємодіяти хіба що на екрані за допомогою мишки. Як же його знайде команда закривання? Здається, потрібно зберігати посилання на екземпляр гри, щоб можна було надсилати йому повідомлення програмно. Використаємо для цього змінну класу (у класів Pharo є свої змінні!), яку назвемо «*TheGame*».

У вікні редагування коду Налагоджувача введіть текст методу з лістингу 6.11.

Лістинг 6.11. Створення екземпляра гри

```
LOGame class >> open
  TheGame ifNil: [
     TheGame := self new.
     TheGame openInHand ]
```

Лістинг 6.12. Знищення екземпляра гри

```
LOGame class >> close
  TheGame ifNotNil: [
     TheGame delete.
     TheGame := nil ]
```

Збережіть введений текст, і Налагоджувач перепитає, як оголосити ім'я *TheGame*. Виберіть «*Declare new class variable*». Метод визначено так, що можна буде відкрити лише один екземпляр гри. Посилання на нього зберігатиме змінна класу. До речі, знайдіть у Оглядачі оголошення класу *LOGame* (потрібно повернутися до ***Inst. side***) Воно мало б мати вигляд:

```
BorderedMorph subclass: #LOGame
   instanceVariableNames: 'cells'
   classVariableNames: 'TheGame'
   package: 'PBE-LightsOut'
```

Налагоджувач додав до оголошення у третьому рядку ім'я *TheGame* змінної класу.

Натисніть на кнопку ***Proceed*** Налагоджувача, і він закриється, натомість з'явиться вікно гри! Пограйтеся хвильку, але нам треба визначити ще один метод. Отож виберіть «*World* > *LightsOut* > *Quit*». Старий знайомий Налагоджувач знову повідомить про відсутній метод і допоможе його створити (див. лістинг 6.12). Метод можна зачислити до протоколу *releasing*.

Знову натисніть на ***Proceed*** – закриється і Налагоджувач, і гра. Тепер команди головного меню працюватимуть як треба. Можете випробувати їхню дію (див. рис. 6.19).

Чи завершено роботу над грою? Можливо, що так. Але ви можете захотіти вести статистику гри: рахувати кількість виконаних ходів, кількість увімкнутих клітинок тощо. Було б добре також відображати цю статистику на екрані, але ми залишимо подальші удосконалення гри читачам як вправу.

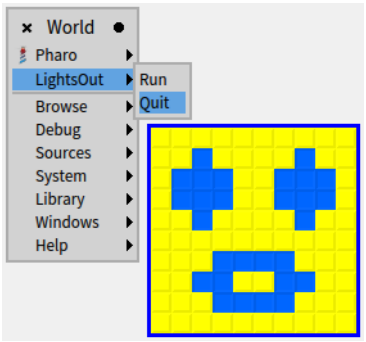

Рис. 6.19 Доповнене головне меню Pharo та гра

Ми успішно доповнили головне меню командою «*LightsOut*», але як тепер повернути його до попереднього вигляду? Через деякий час, награвшись, ви точно захочете зробити це. Виявляється, що достатньо закоментувати прагму <*menuWorld*> у тілі методу *LOGame class* >> *menuCommandOn:*. Не потрібно нічого вилучати, просто приховайте прагму. Захочете знову повернути команду в меню – знімете коментар.

6.19. Зберігання та поширення коду Pharo

Тепер, коли гра Lights Out працює, ви напевно захочете якось її зберегти, щоб мати файл гри та змогу ділитися ним з друзями. Звісно, ви можете зберегти весь образ Pharo і показувати вашу першу програму, запускаючи його. Але ваші друзі, напевно, мають власний код у образах своїх систем і не захочуть втрачати його, використавши ваш образ. Для зберігання коду і відстежування версій уся спільнота Pharo використовує Git і Iceberg – потужний інструмент у складі Pharo, який приховує від користувача низькорівневі деталі.

Використовуйте Iceberg і Git для контролю версій свого коду

Під час розробки лічильника в попередньому розділі ми вже пояснювали вам основи використання Iceberg та Git для зберігання, поширення та контролю версій ваших проєктів. Ви можете використати їх знову для створеної гри. Якщо вам дуже не

терпиться дізнатися більше про Iceberg, то переходьте одразу до розділу 7. Але якщо ви все ще тут, то подивимося, як експортувати програмний код Pharo до файлів.

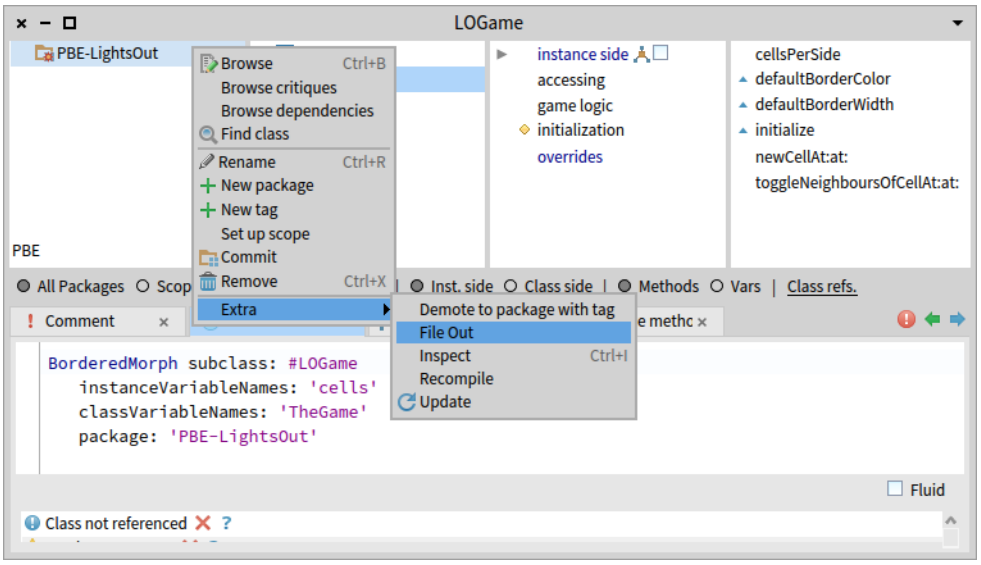

Рис. 6.20. Виведення пакета класів до файлу

Зберігання коду до файлу (додатково)

Найкращий спосіб зберігання коду – використати Git. Найпростіший спосіб отримати текст створених класів – це записати пакет до файлу. Це рудиментарний спосіб, який насправді не відстежує версій, але може виявитися зручним за певних обставин. Спільноти програмістів на Pharo та Squeak називають її «filing out». Позначте в панелі пакетів Системного оглядача *PBE-LightsOut* і виберіть команду «*Extra* > *File Out*» контекстного меню (див. рис. 6.20). Вона виведе весь пакет до файлу «*PBE-LightsOut.st*», створеного в тій папці, де зберігається образ системи. Він більш-менш читабельний, але призначений найперше для комп'ютерів, а не для людей. Ви можете надіслати цей файл друзям і вони зможуть інсталювати його у власний образ Pharo.

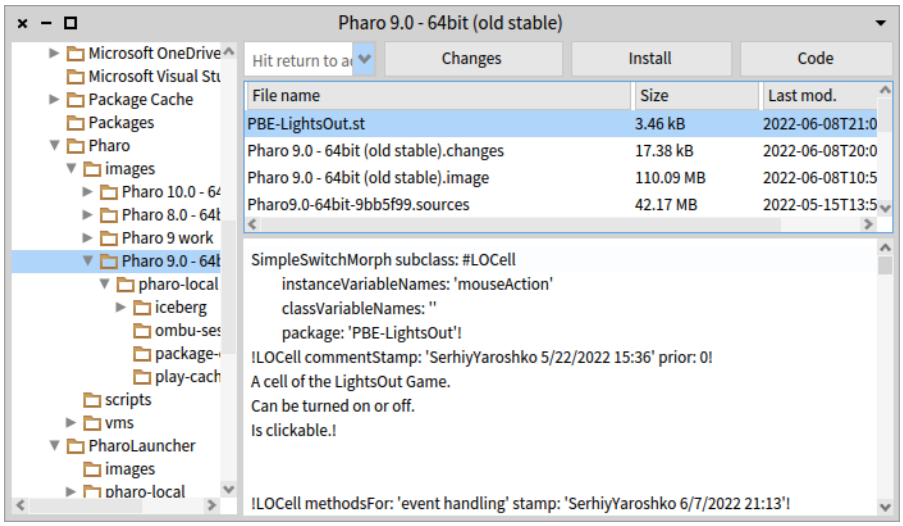

Рис. 6.21. Імпортування коду за допомогою оглядача файлів

Перегляньте отриманий файл за допомогою редактора текстів, щоб зрозуміти, як виглядає текст пакета класів у файлі. Якщо ви хочете переглянути вміст файлу не покидаючи Pharo, то спробуйте надрукувати рядок «*'PBE-LightsOut.st' asFileReference*» в Пісочниці та проінспектувати його.

Відкрийте новий образ Pharo та за допомогою оглядача файлів (*World* > *System* > *File Browser*) знайдіть файл *PBE-LightsOut.st*, відкрийте його контекстне меню та виберіть «*Install into the image*» (див. рис. 6.21). Переконайтеся, що гра працює у новому образі. Надалі використовуйте Git!

Від перекладача. Інспектування «*'PBE-LightsOut.st' asFileReference*» може не спрацювати. Можливо, вам доведеться вказати повне ім'я з маршрутом до нього. За таких умов вміст файлу простіше переглянути в оглядачі файлів Pharo. Подивіться на рис. 6.21: найбільша панель оглядача вже відображає вміст файлу! Перевага інспектора лише в тому, що він виконає синтаксичне підсвічування тексту. До речі, інсталювати файл можна не лише командою контекстного меню, а й кнопкою *Install* оглядача файлів.

6.20. У Pharo ви не можете втратити код[6]

Прикро, але аварія Pharo цілком можлива. Pharo – експериментальна система, яка дає змогу міняти все включно з частинами, які необхідні для функціонування самого Pharo!

Хороші новини в тому, що ви ніколи не загубите вашу роботу, навіть якщо трапиться аварія Pharo, і воно відкотиться до попередньої робочої версії, яка створена, можливо, годину тому назад. Увесь код, виконаний вами в Оглядачі та в Налагоджувачі, зберігається в *.change* файлі, код з Playground зберігається в окремих файлах у папці */pharo-local/playchace* – його можна переглянути звичайним редактором текстів.

Файл змін також можна відкрити звичайним редактором текстів, але зорієнтуватися в ньому буде складно, оскільки розміри файлу іноді вимірюються мегабайтами. Доцільно використати спеціальний оглядач змін, доступний за командою головного меню *World > Sources > Code Changes*. У його лівій панелі ви побачите повний список знімків системи, у правій – перелік подій вибраного знімка, а в нижній – зміст вибраної події. Використовуйте команди контекстного меню для повторного виконання бажаних змін.

6.21. Підсумки розділу

Цей розділ допоміг нам поглибити здобуті раніше знання про Pharo під час розробки простої гри, що використовує бібліотеку графічних елементів (морф). Ми навчилися використовувати Налагоджувач для виправлення помилок і написання коду «на льоту»: так ми оголошували методи екземпляра, методи класу та змінні класу.

Ми відкривали меню-ореол навколо графічного елемента, щоб маніпулювати ним, та, що важливо, щоб позбутися його.

Ми переконалися, що оголошувати константу як результат виконання методу є хорошою практикою, та допомагає легко налаштувати поведінку підкласу.

Невелике дослідження та сміливий експеримент допомогли нам видозмінити недоторкану, здавалося б, частину системи – її головне меню.

Ми також навчилися зберігати пакет класів до текстового файлу та інсталювати його до образу Pharo для швидкого поширення готових програм.

---

[6] Цей параграф перекладач додав з попередньої версії оригіналу.

Розділ 7

Публікація вашого першого проєкту Pharo

У цьому розділі ми детальніше пояснимо, як ви можете за допомогою Iceberg опубліку-вати свій проєкт на GitHub. Iceberg – це інструмент і бібліотека, що дає змогу керувати Git-сховищами. Він робить більше, ніж просто зберігає та публікує ваші файли. Тоді ми коротко покажемо, як переконатися, що і ви можете, й інші розробники можуть завантажити ваш код, не розуміючи його внутрішніх залежностей.

Зауважимо, що цей розділ написано для незалежного читання, тому можуть трапляти-ся повторення матеріалу інших розділів.

Ми не будемо пояснювати таких базових понять як *commit* (запис до сховища), *push* (вивантаження до сховища), *pull* (завантаження зі сховища), чи клонування сховища. За потреби зверніться до посібника з Git. Для читання цього розділу важливо, щоб ви мали можливість з командного рядка публікувати файли на вашому віддаленому сервісі Git. Якщо у вас немає такої можливості, то не сподівайтеся, що Iceberg надасть її. Якщо у вас виникли проблеми з конфігурацією протоколу SSH (який є способом за замовчуванням вивантаження на GitHub), то можете використати замість нього HTTPS, або прочитати книжечку «Manage your code with Iceberg», яку можна знайти на http://books.pharo.org. Давайте почнемо.

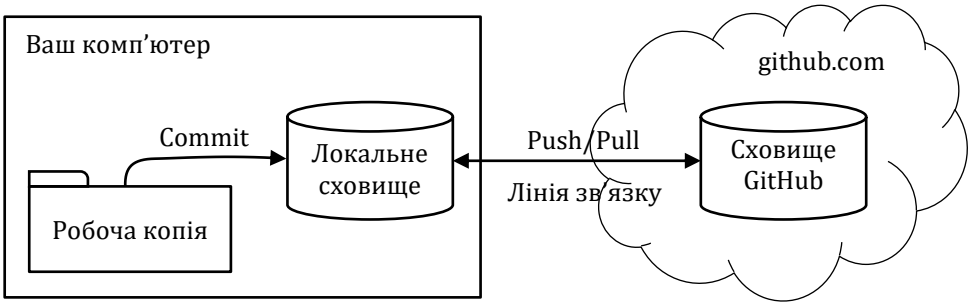

Рис. 7.1. Git – розподілена система контролю версій

7.1. Для нетерплячих

Якщо ви не хочете читати все написане нижче, і відчуваєте себе досить впевнено, ось стислий перелік кроків для того, щоб опублікувати свій код:

- створіть сховище на GitHub або будь-якій іншій Git-платформі;
- [не обов'язково] налаштуйте Iceberg на використання спеціальних ключів SSH;
- додайте проєкт до Iceberg;
- [не обов'язково, але дуже рекомендовано] створіть у вашому локальному сховищі папку з іменем «src» – це правильна загальна домовленість;
- відкрийте свій проєкт у Iceberg і додайте до нього пакети (класів);
- запишіть свій проєкт (виконайте *commit*);
- [не обов'язково] додайте до проєкту базові параметри;

- вивантажте зміни до віддаленого сховища.

І ви закінчили! А тепер давайте пояснимо ці кроки спокійніше.

7.2. Базова архітектура

Оскільки Git є розподіленою системою контролю версій, то вам потрібно мати локальний клон віддаленого сховища та робочу копію проєкту. Локальне сховище та робоча копія зазвичай розташовуються на вашій машині. Зроблені вами зміни в робочій копії будуть записані до локального сховища перед вивантаженням до віддаленого сховища чи сховищ (див. рис. 7.1). Ми використовуємо Iceberg тому, що Pharo трохи ускладнює процес зберігання. Пояснимо у двох словах, чим саме: класи та методи Pharo – це об'єкти, які змінюються на льоту. Коли ви змінюєте вихідний код класу, робоча копія Git не змінюється автоматично. У вас ніби є дві робочі копії: одна – об'єктна в образі вашої системи, інша – на основі Git-файлу. Iceberg створено для того, щоб допомогти синхронізувати їх та керувати ними обома.

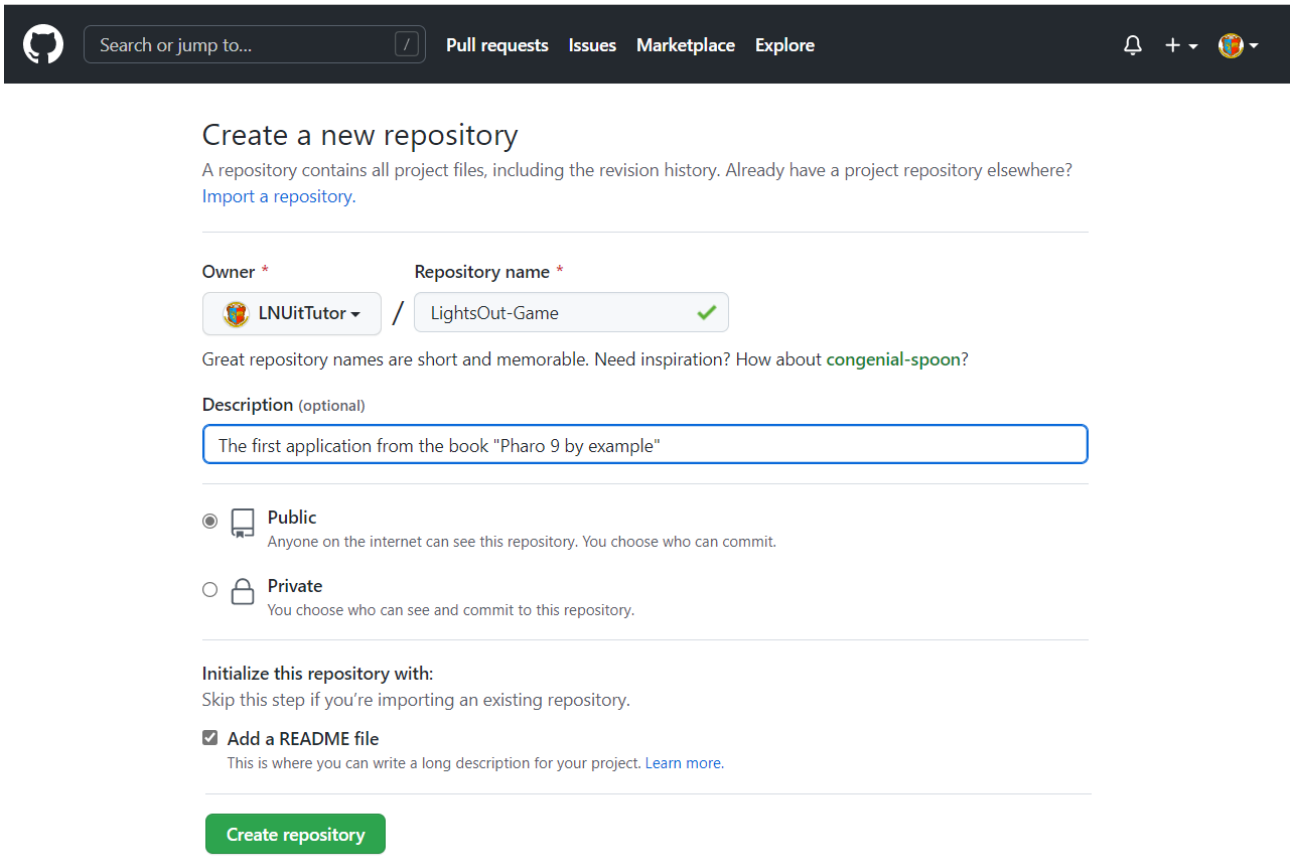

Рис. 7.2. Створення нового сховища на GitHub

Створення нового сховища на GitHub

Вам потрібно створити два сховища: локальне та віддалене. У розділі 5 ми починали з локального, а тепер покажемо, як спершу створити репозиторій на GitHub. Черговість не має особливого значення, але, якщо ви почали з GitHub, то доведеться виконати додаткові кроки у Pharo, про які зараз поговоримо.

Під час створення сховища на GitHub потрібно задати його ім'я, опис і додати файл *README* (див. рис. 7.2).

Налаштування SSH: повідомте Iceberg свої ключі

Щоб мати змогу зберігати код у сховищі на GitHub, ви повинні використати для доступу або HTTPS, або SSH. У першому випадку, як уже було сказано в п. 5.13, вам буде потрібен персональний токен доступу, у другому – потрібно буде налаштувати дійсні облікові дані у вашій системі. Якщо ви використовуєте SSH (спосіб за замовчуванням), то вам потрібно буде переконатися, що ці ключі доступні для вашого облікового запису GitHub, а операційна система додала їх до клієнта SSH для зв'язку з сервером.

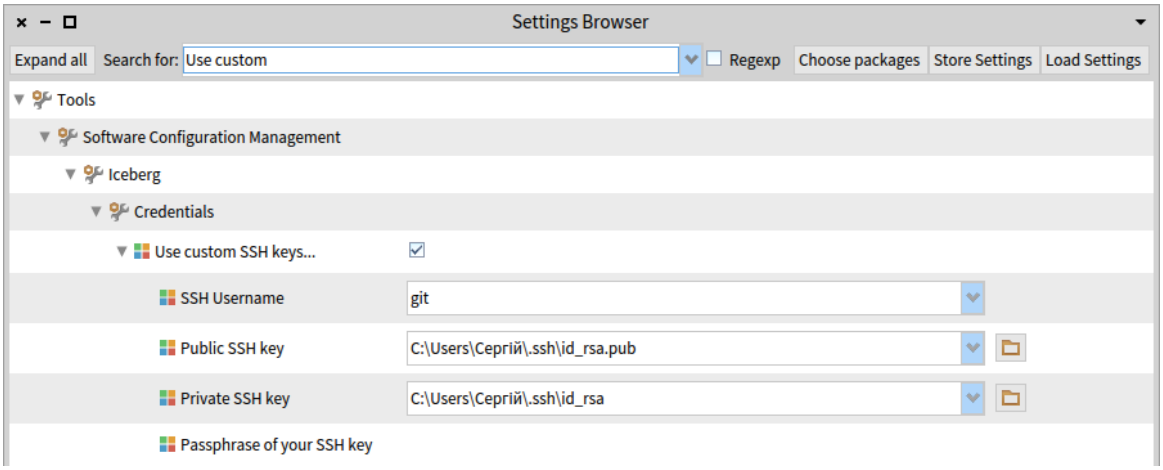

Рис. 7.3 Налаштування ключів SSH

Відкрийте оглядач налаштувань командою меню «*Pharo* > *Settings*» і введіть у рядку пошуку «*Use custom SSH keys*» – так ви швидко перейдете до потрібних налаштувань, що розташовані десь наприкінці всього списку. Введіть посилання на файли з вашими ключами, як показано на рис. 7.3.

Налаштування можна також виконати програмно: виконайте в Робочому вікні такий фрагмент коду

```
IceCredentialsProvider useCustomSsh: true.
IceCredentialsProvider sshCredentials
   publicKey: 'path\to\ssh\id_rsa.pub';
   privateKey: 'path\to\ssh\id_rsa'
```

Зауваження. Для зберігання ключів можна використовувати файли з нестандартними іменами. У цьому випадку потрібно замінити «*id_rsa*» в коді (чи в налаштуваннях) іменем вашого файлу.

Тепер ми готові поглянути на те, як Iceberg керує Git у Pharo.

Від перекладача. Операційна система Windows 10 уже містить готового клієнта SSH. Вам доведеться лише згенерувати ключі та додати їх до клієнта в режимі командного рядка. Докладний опис усіх кроків щодо створення ключів та приєднання їх до облікового запису GitHub можна знайти в посібнику за адресою https://docs.github.com/en/authentication/connecting-to-github-with-ssh, а опис виправлення поширених помилок – https://docs.github.com/en/authentication/troubleshooting-ssh.

Якщо вам не вдалося налаштувати доступ через клієнта SSH, то не переживайте, продовжуйте використовувати HTTPS – він чудово працює.

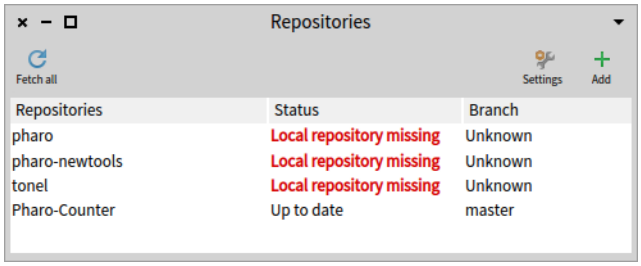

Рис. 7.4. Оглядач репозиторіїв Iceberg, відкритий в новому образі системи

7.3. Про оглядач репозиторіїв Iceberg

На рис. 7.4 зображено головне вікно Iceberg. Воно містить перелік відомих проєктів (сховищ). Серед них є створений нами у розділі 5 *Pharo-Counter* і кілька проєктів, отриманих з образом Pharo під час встановлення системи, наприклад, *pharo*. Напис червоного кольору «*Local repository missing*» означає, що Iceberg не може знайти відповідне локальне сховище.

Передусім не хвилюйтеся про відсутній репозиторій для Pharo: він вам знадобиться, лише якщо захочете зробити внесок у розробку мови Pharo. Звичайно, ми усіх запрошуємо долучатися до проєкту, але, напевне, давайте спочатку закінчимо цей розділ. Ось що відбувається: Iceberg попереджає, що система Pharo не знає, де розміщений відповідний репозиторій Git для її класів. Але ви вільно можете переглядати системні класи й методи, вносити зміни, Pharo чудово працює без локального сховища, бо має вбудовану систему керування версіями коду. Це попередження свідчить лише про те, що якщо ви хочете керувати версіями системного коду Pharo за допомогою Git, то ви мусите повідомити системі, де на вашій локальній машині збережені локальний клон і робоча копія Pharo. Якщо ви не плануєте змінювати системний код Pharo та ділитися ним, то не варто про це турбуватися.

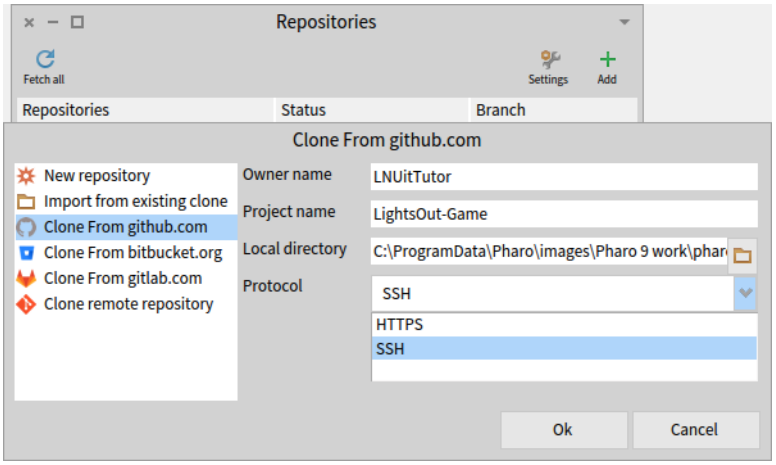

Рис. 7.5. Клонування проєкту з GitHub через SSH або HTTPS

7.4. Додавання нового проєкту до Iceberg

Спочатку додайте проєкт до Iceberg.

- Натисніть кнопку ***Add***, розташовану вгорі праворуч у головному вікні Iceberg.
- Вкажіть розташування репозиторію. Ми копіюємо репозиторій з GitHub, тому потрібно вибрати відповідну опцію, вказати ім’я власника і назву репозиторію.

Нагадуємо, що ви можете використати один з протоколів: SSH або HTTPS (рис. 7.5).

Задані параметри вказують Iceberg клонувати репозиторій, який ми щойно створили на GitHub. Ми задали власника, ім'я сховища, шлях до папки на диску свого комп'ютера, де розташуються локальний клон і робоча копія сховища.

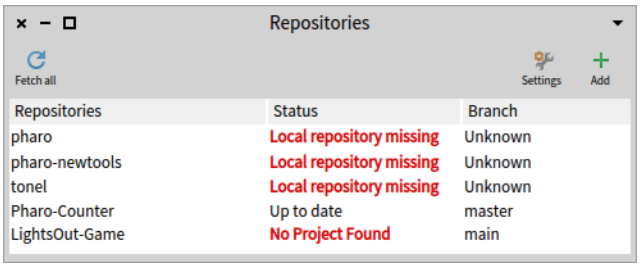

Рис. 7.6. Одразу після клонування порожнього репозиторію Iceberg повідомляє, що в проєкті бракує інформації

Тепер Iceberg додасть заданий репозиторій до списку керованих проєктів і клонує порожній репозиторій на локальний диск. Ви мали б побачити щось схоже на рис. 7.6. Закономірно, що проєкт має статус «*No Project Found*», адже він порожній, і Iceberg не може знайти його метадані. Зараз виправимо цю помилку.

Згодом, коли ви збережете зміни до свого проєкту і захочете завантажити його до іншого образу, клонувавши його ще раз, ви побачите, що Iceberg повідомить інший статус: «*Not loaded*» – проєкт є з усіма даними, проте ще не став частиною образу.

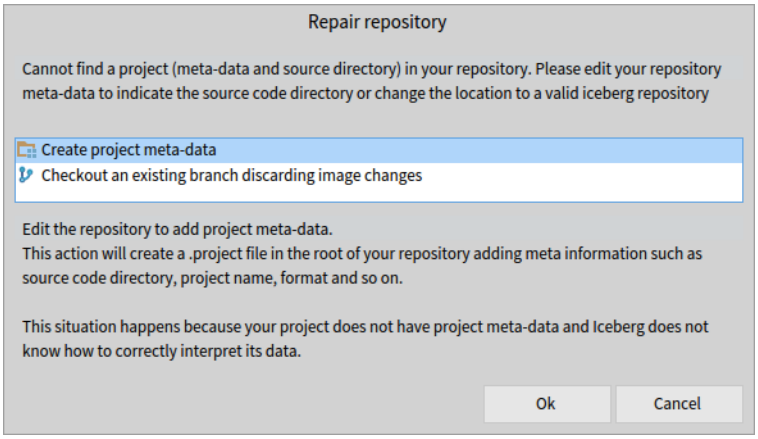

Рис. 7.7. Дія створення метаданих проєкту та її пояснення

Рятівне відновлення

Iceberg – це досить розумний інструмент, здатний допомогти вам вирішити проблеми, які можуть трапитися під час роботи з Git. Загалом, коли ви бачите статус репозиторію, відображений червоним (наприклад, «*No Project Found*», або «*Detached Working Copy*»), попросіть Iceberg виправити ситуацію за допомогою команди «*Repair repository*» контекстного меню проєкту.

Iceberg не може виправити всі помилки автоматично, натомість він запропонує вам можливі шляхи вирішення та пояснить кожен з них. Можливі дії буде впорядковано від імовірно більш правильних до менш. Кожну дію супроводжує пояснення, що вона робить, та як її застосовувати. Завжди варто читати такі пояснення. Якщо правильно налаштувати репозиторій, то дуже важко втратити будь-який фрагмент коду з Iceberg і Pharo, оскільки Pharo зберігає власну копію коду. Дуже важко, проте… можливо. Тому будьте уважні!

Створення метаданих проєкту

Iceberg повідомляє, що він не може знайти проєкт, оскільки немає деяких його метаданих, такі як спосіб кодування файлу та приклад розташування всередині репозиторію. Коли ми запустимо команду відновлення сховища, то побачимо нове вікно, зображене на рис. 7.7. Оберіть у ньому дію «*Create project metadata*» та прочитайте пояснення.

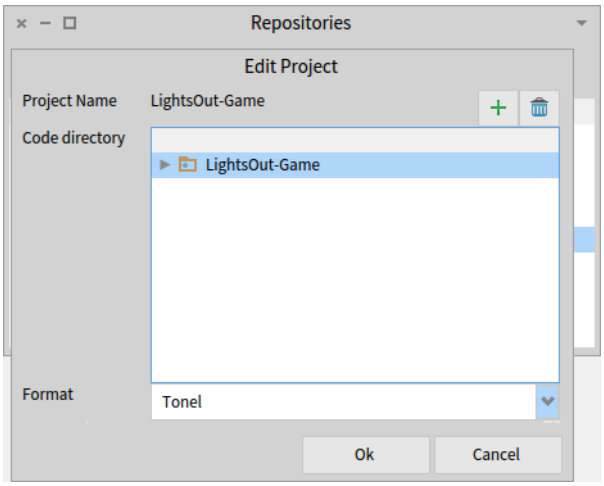

Рис. 7.8. Місце зберігання метаданих і формат проєкту

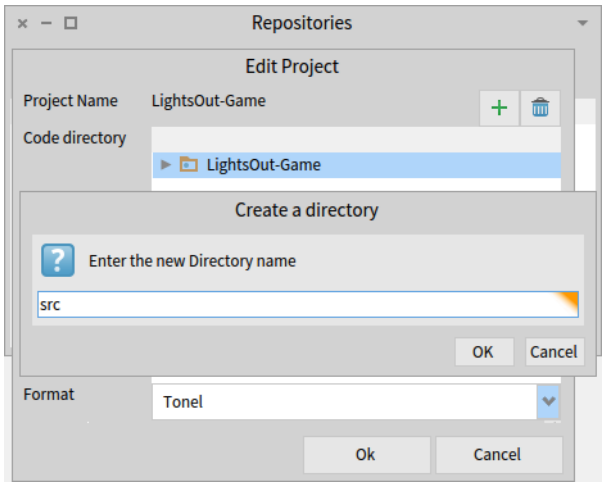

Рис. 7.9. Додавання папки для зберігання коду

Коли ви підтвердите свій вибір, Iceberg покаже структуру папок вашого проєкту та його формат, як на рис. 7.8. Бажаним форматом проєктів Pharo є *Tonel*, бо його розроблено спеціально для використання в різних файлових системах. Формат *Tonel* можна змінити на *Filetree*, проте робіть так лише тоді, коли воно вам справді потрібно.

Ще один не зайвий крок – додати до проєкту папку для зберігання коду. За домовленістю її називають «*src*». Натисніть кнопку з зеленим знаком плюс і введіть ім'я папки у вікні діалогу, як показано на рис. 7.9. Сформовану структуру папок проєкту показано на рис. 7.10. Не забудьте позначити новостворену папку – тільки так ви справді вкажете, що код потрібно зберігати в «*src*». Тисніть ще раз ***Ok***, і відновлення метаданих буде завершено. Назву проєкту позначено зірочкою та зеленим кольором (див. рис. 7.11), бо він містить незбережені зміни. Виконайте команду «*Commit*» контекстного меню проєкту. Iceberg покаже деталі операції (рис. 7.12), а сам проєкт набуде вигляду, як на рис. 7.13.

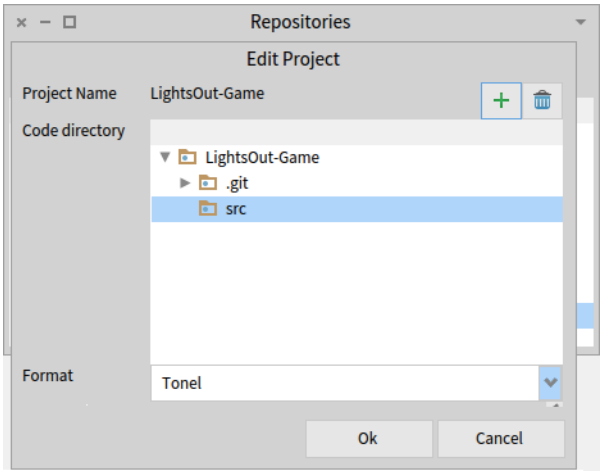

Рис. 7.10. Сформована структура проєкту

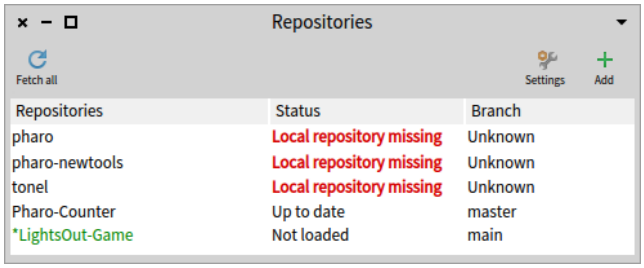

Рис. 7.11. Після відновлення метаданих проєкт містить незбережені зміни

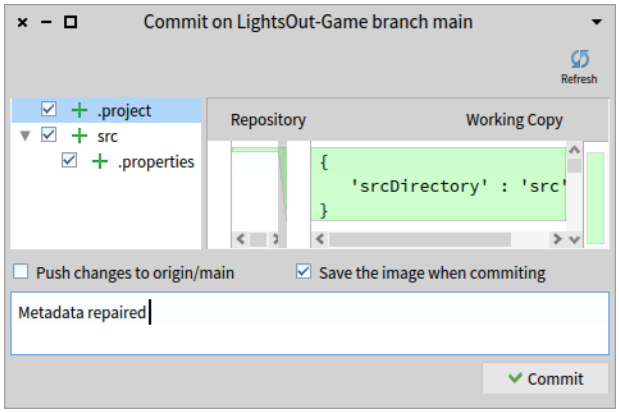

Рис. 7.12. Деталі збереження метаданих

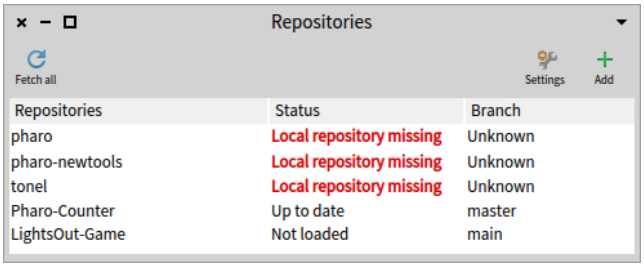

Рис. 7.13. Після відновлення метаданих проєкт містить незбережені зміни

Після того, як ви збережете метадані, Iceberg покаже вам, що проєкт відновлено, але не завантажено, як показано на рис. 7.13. Це нормально, оскільки ми ще не додали до проєкту жодних пакетів. Далі, якщо хочете, можете надіслати збережені зміни до свого віддаленого сховища за допомогою команди «*Push*» контекстного меню проєкту.

Ваше локальне сховище готове, давайте перейдемо до наступної частини.

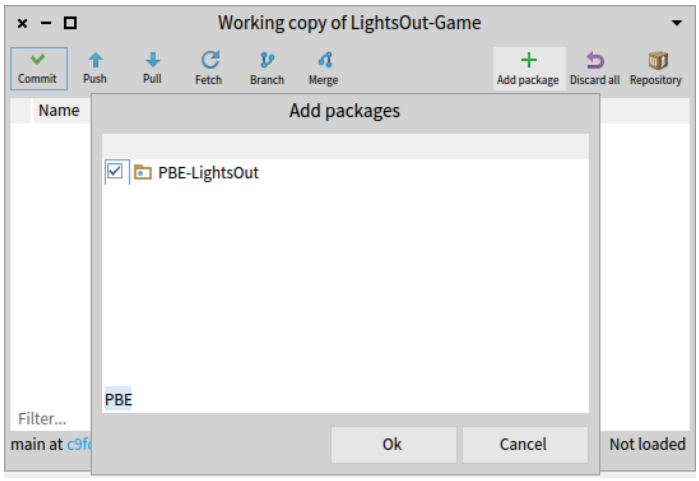

Рис. 7.14. Додавання пакета за допомогою оглядача робочої копії

7.5. Додайте і збережіть пакет за допомогою оглядача робочої копії

Оскільки ваш проєкт уже містить метадані, то тепер Iceberg легко зможе керувати ним. Двічі клацніть на імені проєкту, щоб відкрити його в оглядачі робочої копії. Оглядач відображає перелік пакетів, з яких складається проєкт. У вас він поки що порожній. Додайте пакет «*PBE-LightsOut*» за допомогою кнопки ***Add package***, як показано на рис. 7.14. Iceberg знову просигналізує зеленим кольором і зірочкою перед іменем проєкту про те, що він містить незбережені зміни (див. рис. 7.15).

Збережіть зміни

Збережіть зміни до локального сховища, клацнувши кнопку ***Commit***, як зображено на рис. 7.15. Iceberg демонструє всі частини коду: класи й методи, – які зазнали змін. За замовчуванням усі вони позначені для зберігання. За потреби ви можете змінити ці позначки на власний розсуд: це необов'язкова, проте важлива можливість. Після збереження Iceberg змінить колір імені проєкту на чорний та опис стану на «*1 not published*». Це означає, що образ системи та робочу копію синхронізовано з локальним сховищем,

але воно відрізняється від віддаленого: зміни ще не опубліковано на GitHub. За потреби ви можете зберігати зміни до локального середовища кілька разів підряд.

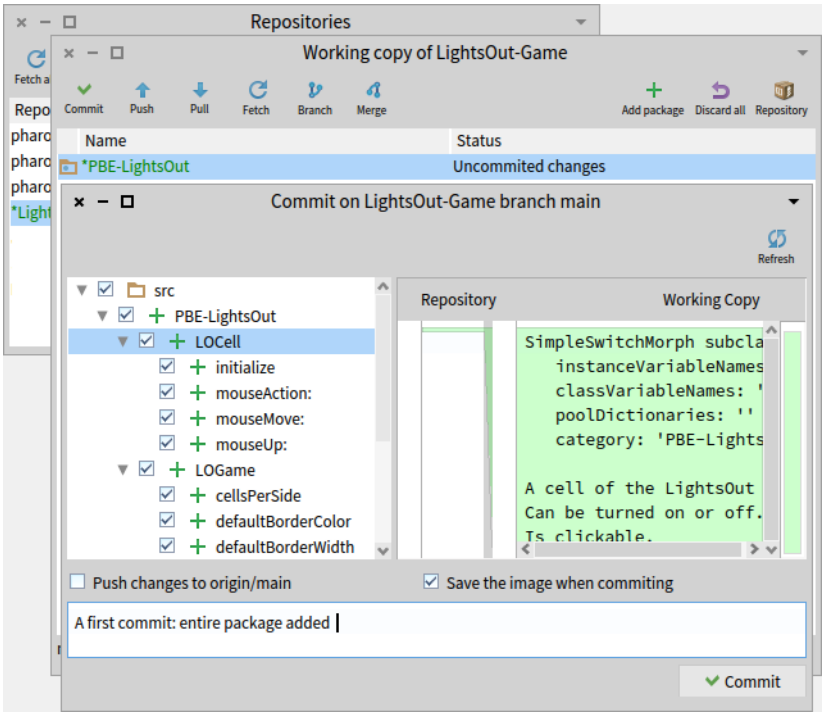

Рис. 7.15. Iceberg та оглядач робочої копії сигналізують, що в проєкті є незбережені зміни – доданий нами пакет. Під час виконання команди *Commit* ви можете вибрати, які саме частини коду зберегти

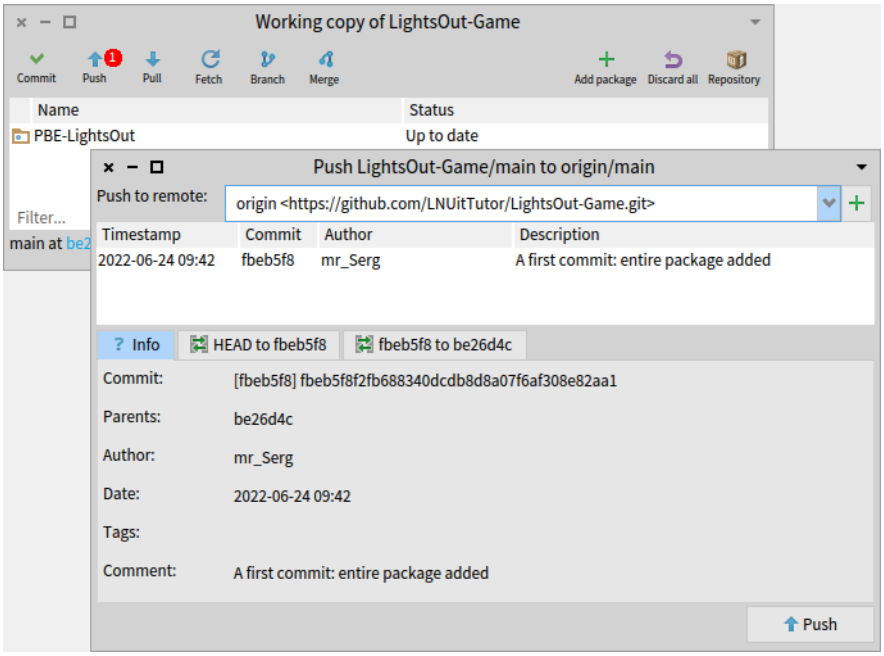

Рис. 7.16. Вивантаження збережених змін

Вивантажте зміни до віддаленого сховища

Натисніть кнопку ***Push*** оглядача робочої копії, щоб перенести збережені зміни з локального сховища до віддаленого. Кнопку легко помітити, оскільки Iceberg прикрасив її червоною позначкою з кількістю змін, які чекають на опублікування. У вас їх може бути дві, якщо ви не виконували «*Push*» для метаданих проєкту. Перед вивантаженням Iceberg покаже всі неопубліковані зміни (див. рис. 7.16), а тоді перенесе їх на віддалений

сервер. Він також може перепитати вас ім'я користувача та токен доступу, якщо ви використовуєте HTTPS і виконуєте «*Push*» вперше.

Тепер ви здебільшого закінчили. Проєкт *LightsOut-Game* набув статусу «*Up to date*», а ви знаєте основні аспекти керування своїм кодом за допомогою GitHub, або будь-якого іншого віддаленого сервісу Git. Iceberg розроблено, щоб допомагати вам, тому, будь ласка, прислухайтеся до нього, якщо не знаєте, що робити. Тепер ви готові використовувати послуги, які пропонує GitHub, для покращення якості та контролю коду!

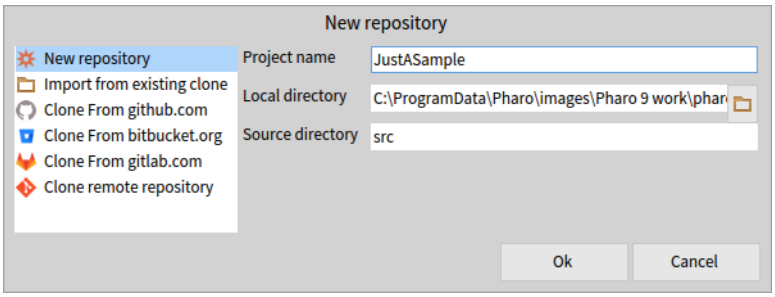

Рис. 7.17. Створення локального сховища у випадку, коли віддаленого ще немає

7.6. Що робити, якщо віддалений репозиторій не створювали?

Ми почали зі створення віддаленого сховища на GitHub, а потім попросили Iceberg додати проєкт, клонувавши його звідти. У результаті отримали віддалене сховище, локальне сховище і робочу копію проєкту. Проте діяти можна і в іншому порядку. Можна створити локальне сховище і зберігати код у ньому. Згодом, коли буде створено сховище на якійсь Git-платформі, код можна буде перенести туди. Створення віддаленого сховища не входить у коло обов'язків Iceberg. Достатньо, що він уміє керувати зберіганням і завантаженням коду.

То як нам досягти такого самого результату, але в оберненому порядку, публікуючи наш локальний проєкт без попереднього віддаленого сховища. Насправді це досить просто, і ми вже так робили в розділі 5, тому давайте пригадаємо.

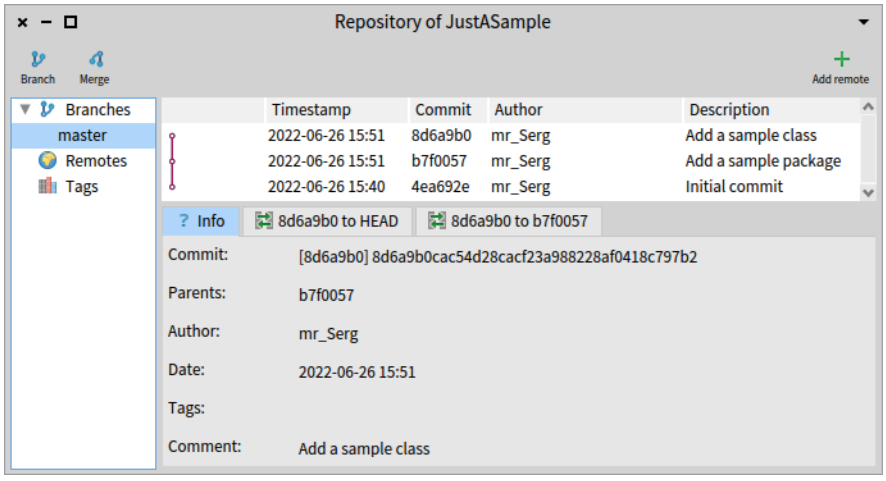

Рис. 7.18. Оглядач репозиторію дає змогу додавати гілки та віддалені сховища, переміщатися між ними

Створіть новий репозиторій

Щоб почати з локального репозиторію, використайте опцію ***New repository*** команди «*Add*» Iceberg, як показано на рис. 7.17 (або 5.9). Ми створимо деякий умовний репози-

торій заради демонстрації. Ви, навіть, можете додати до нього якийсь пакет класів, як ми це вже робили раніше. Після створення збережіть зроблені зміни командою «*Commit*».

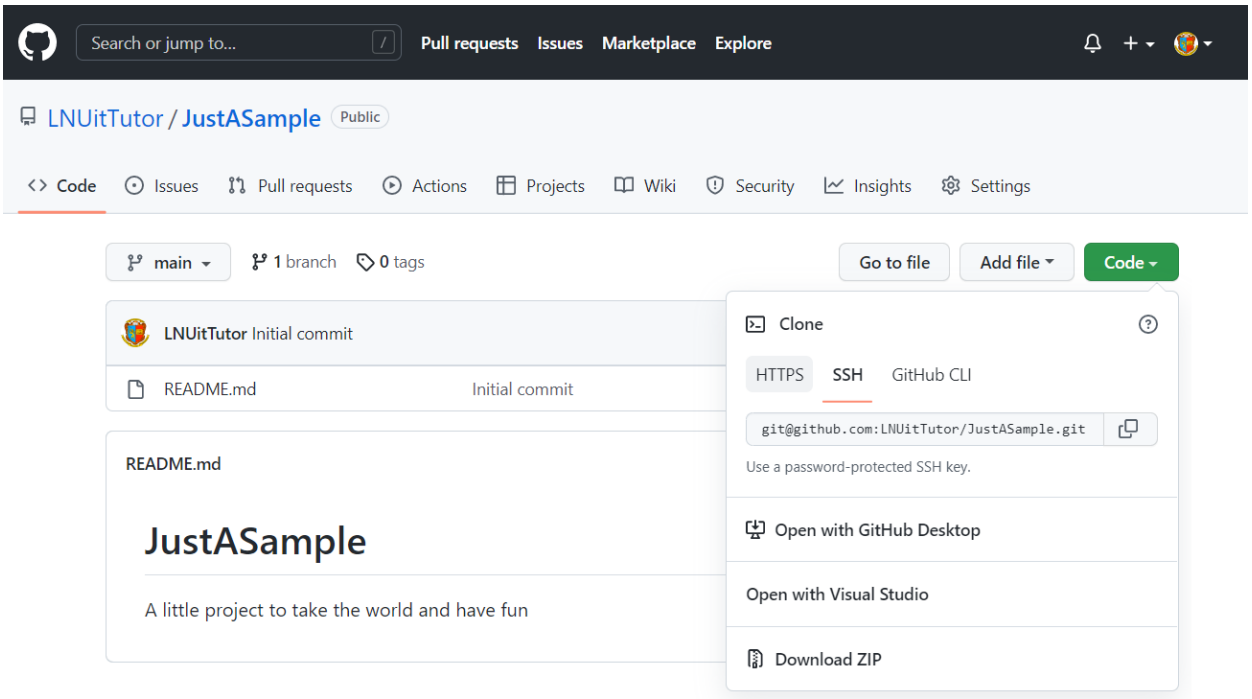

Рис. 7.19. SSH адреса віддаленого сховища на GitHub

Додайте віддалений репозиторій

Якщо ви хочете вивантажувати зміни до віддаленого сховища, то вам доведеться додати його за допомогою оглядача репозиторію. Відкрити його можете командою «*Open Repository*» контекстного меню проєкту або кнопкою ***Repository*** оглядача робочої копії. Оглядач сховища надає вам доступ до пов'язаних з вашим проєктом сховищ Git: ви можете отримати доступ до гілок, керувати ними, а також додавати або вилучати віддалені репозиторії. На рис. 7.18 показано оглядач сховища нашого проєкту.

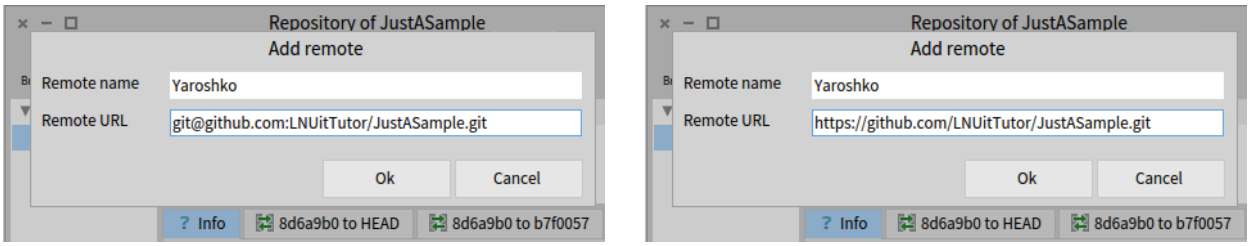

Рис. 7.20. Додавання віддаленого репозиторію в оглядачі сховища
(SSH варіант ліворуч та HTTPS – праворуч)

Віддалене сховище додають кнопкою ***Add remote***. Вона відкриває невеликий діалог, щоб ви могли ввести його координати. Ми починали зі створення локального сховища, тому саме час зайти на GitHub і створити новий порожній репозиторій та довідатися його координати (див. рис. 7.19, або 5.17). Справді, не додавайте до сховища ніяких файлів, навіть *README*, адже ми домовилися починати з Pharo. Натисніть ***Add remote*** і вкажіть координати сховища, як на рис. 7.20.

Вивантажте проєкт у віддалений репозиторій

Тепер ви можете вивантажити усі збережені в локальному сховищі зміни до віддаленого – просто натисніть кнопку ***Push***. Одразу після вивантаження ви побачите в оглядачі сховища, що у вашого проєкту з'явився віддалений репозиторій, як на рис. 7.21.

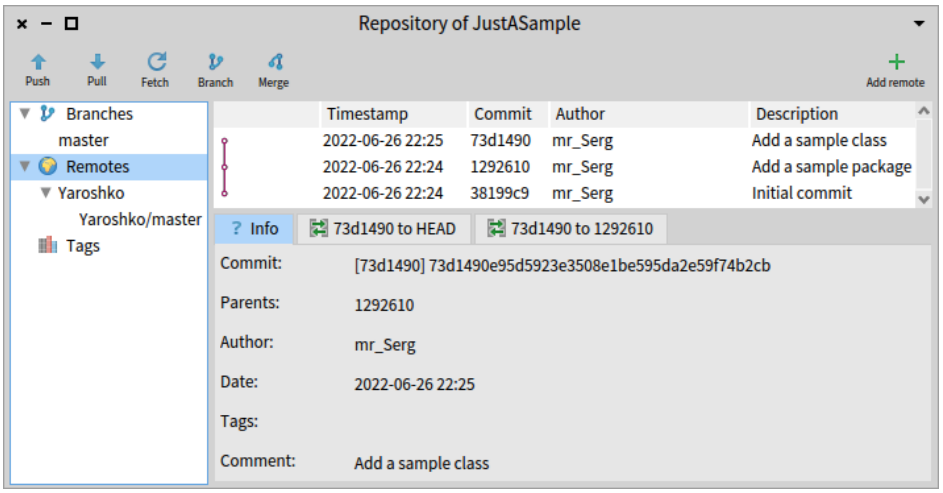

Рис. 7.21 Ви вивантажили проєкт у віддалене сховище

Від перекладача. Одне з правил GitHub – додавати до кожного репозиторію файл *README*. Можливо, ви відгукнетеся на наполегливі пропозиції GitHub і створите такий файл. Тоді віддалене сховище отримає оновлення, а локальне – ні. Щоб їх синхронізувати, потрібно виконати кілька простих кроків: натисніть кнопку *Pull* в оглядачі робочої копії (вона розташована праворуч від *Push*), у вікні діалогу завантаження змін натисніть кнопку *Fetch*, щоб отримати перелік змін з віддаленого сервера, і натисніть кнопку *Pull* діалогу. Тепер сховища знову синхронізовані!

7.7. Конфігурування вашого проєкту

Контроль версій коду – це лише перша частина роботи, щоб ви та інші розробники могли завантажити та використати ваш код. Тепер опишемо, як визначити *Baseline*: карту проєкту, яку ви будете використовувати для визначення залежностей у вашому проєкті та його залежностей від інших проєктів.

Визначення *BaselineOf*

Baseline описує архітектуру проєкту. Виразити залежності між вашими пакетами та іншими проєктами потрібно для того, щоб користувач міг завантажити ваш проєкт без вивчення залежних проєктів чи зв'язків між ними.

Карта проєкту має бути підкласом *BaselineOf*, розміщеним в пакеті «*BaselineOfXXX*», де «*XXX*» – назва вашого проєкту. У попередньому параграфі ми створили простий проєкт *JustASamle* без залежностей. Його карту можна виразити зовсім просто:

```
BaselineOf subclass: #BaselineOfJustASample
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'BaselineOfJustASample'
```

```
BaselineOfJustASample >> baseline: spec
   <baseline>
   spec
      for: #common
      do: [ spec package: 'MySample' ]
```

Як тільки ви визначили карту проєкту, потрібно додати її до проєкту за допомогою оглядача робочої копії, як вже було описано. У підсумку ви мали б отримати щось схоже на зображене на рис. 7.22.

Далі збережіть зміни до локального сховища та вивантажте їх до віддаленого.

Більше інформації про можливості використання *Baseline* можна знайти на Pharo вікі за адресою https://github.com/pharo-open-documentation/pharo-wiki/.

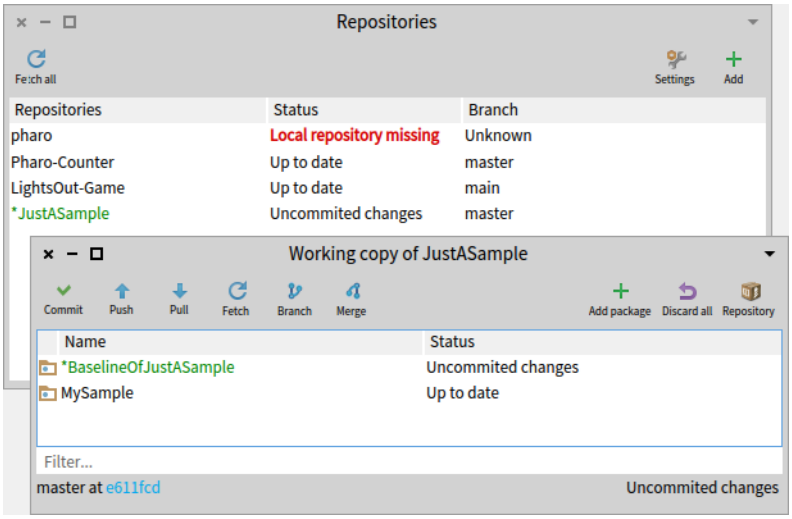

Рис. 7.22. Додавання пакета з картою проєкту за допомогою оглядача робочої копії

7.8. Завантаження з наявного репозиторію

Завантажити опублікований код до нового образу Pharo можна кількома способами.

Завантаження через *Baseline*

Клонуйте в Iceberg віддалене сховище, як ми вже робили в п. 7.4, але створювати метадані не доведеться. Щоб завантажити проєкт в інтерактивному режимі, можете скористатися командою «*Metacello*» контекстного меню (контексто клацніть на назві клонованого сховища в Iceberg, і побачите його). Вона завантажує карту проєкту та виконує її для завантаження пакетів проєкту. Так ви будете впевнені, що всі необхідні підпроєкти також завантажені.

Завантаження вручну

Іноді може виникнути потреба безпосередньо завантажити певний пакет. Як це зробити? Схожою є ситуація з завантаженням проєкту, в якому не визначено *Baseline*. Щоб завантажити конкретний пакет за допомогою Iceberg, потрібно виконати такі кроки:

- додайте проєкт у Iceberg, як ми пояснювали раніше;
- відкрийте оглядач робочої копії подвійним клацанням на імені проєкту;
- позначте потрібний пакет і оберіть команду «*Load*» з його контекстного меню.

Програмне завантаження

Ще один спосіб завантаження полягає в програмному надсиланні повідомлень, як у фрагменті нижче, екземплярові класу *Metacello*:

```
Metacello new
   baseline: 'JustASample';
   repository: 'https://github.com/LNUitTutor/JustASample/src';
   load
```

Це все, що потрібно зробити, щоб завантажити проєкт з визначеною картою (як той, що ми створили в цьому параграфі). Зверніть увагу на те, що в шляху до сховища ми вказали папку, де зберігається код – «*src*» у нашому випадку.

7.9. Оглядаючись назад…

Ви мусите пам’ятати, що коли працюєте у Pharo та змінюєте вміст пакета, керованого Iceberg, то насправді модифікуєте об’єкти, що представляють класи і методи в образі системи, а не змінюєте безпосередньо робочу копію проєкту. Це виглядає так, ніби у вас є дві робочі копії: образ Pharo, завантажений у віртуальну машину, і робоча копія Git на диску.

Коли використовуєте Git для керування своїм проєктом за межами Pharo та Iceberg, наприклад, за допомогою інструмента командного рядка *git*, ви мусите пам’ятати, що код є в образі Pharo *та* в робочій копії (і ще є код у вашому локальному клоні сховища). Щоб оновити образ системи, *спочатку* потрібно оновити робочу копію Git, а *потім* завантажити код з робочої копії в образ. Щоб зберегти свій код, вам *спочатку* треба зберегти код у образі як файли, *потім* додати їх до робочої копії Git, а *потім* зберегти їх у своєму клоні.

Привабливість Iceberg полягає в тому, що він прозоро керує всім цим за вас. Уся виснажлива синхронізація між двома робочими копіями виконується за лаштунками.

Архітектура Iceberg виглядає так:

- ваш код зберігається в образі Pharo;
- Pharo працює як робоча копія (він містить наповнення локального сховища Git);
- Iceberg керує зберіганням вашого коду до робочої копії Git та локального сховища;
- Iceberg керує вивантаженням вашого коду до віддалених сховищ;
- Iceberg керує повторною синхронізацією вашого образу Pharo з локальним сховищем Git, віддаленим сховищем і робочою копією Git.

7.10. Підсумок розділу

У цьому розділі подано найважливіші аспекти того, як правильно розмістити свій код у пакетах та опублікувати його. Це допоможе вам завантажувати його в інші образи та співпрацювати з іншими розробниками Pharo.

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 × 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[7], 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.

Прагми та примітиви. *<primitive: ...>* є анотацією методу. Наведений приклад позначає виклик примітиву[8] 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

[7] *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.*).

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

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 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. Докладніше 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[9], які допомагають писати дуже стислий і виразний код.

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, ' '].
```

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

```
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 виконува-

ти 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.

Розділ 9
Розуміння синтаксису повідомлень

Синтаксис повідомлень Pharo надзвичайно простий, проте він нетрадиційний і може бути потрібно трохи часу, щоб звикнути до нього. У цьому розділі наведено деякі рекомендації, які допоможуть вам освоїтися з синтаксисом надсилання повідомлень. Якщо ви вже не відчуваєте труднощів з синтаксисом, то можете пропустити цей розділ, або повернутися до нього пізніше. Синтаксис Pharo близький до синтаксису Smalltalk, тому програмісти Smalltalk можуть знати синтаксис Pharo.

9.1. Розпізнавання повідомлень

У Pharo все є надсиланням повідомлень за винятком окремих синтаксичних елементів, описаних у попередньому розділі (*:= ^ . ; #() {} [:|]*). Ви можете визначити оператори для власних класів, наприклад +, але всі оператори, наявні та визначені вами, мають однаковий пріоритет. Насправді в Pharo немає операторів! Є просто повідомлення певного виду: *унарні*, *бінарні* або *ключові*. Крім того, ви не можете змінити арність селектора повідомлення. Селектор «–» завжди є селектором бінарного повідомлення, неможливо оголосити унарний мінус як унарне повідомлення з селектором «–».

Порядок надсилання повідомлень у Pharo визначено видом повідомлення. Є лише три види повідомлень: *унарні*, *бінарні* та *ключові*. Спочатку завжди надсилаються унарні повідомлення, потім – бінарні і, нарешті, ключові. Як і в більшості мов програмування, круглі дужки використовують для зміни порядку виконання. Ці правила роблять код Pharo легким для читання, і зазвичай вам не потрібно думати про правила.

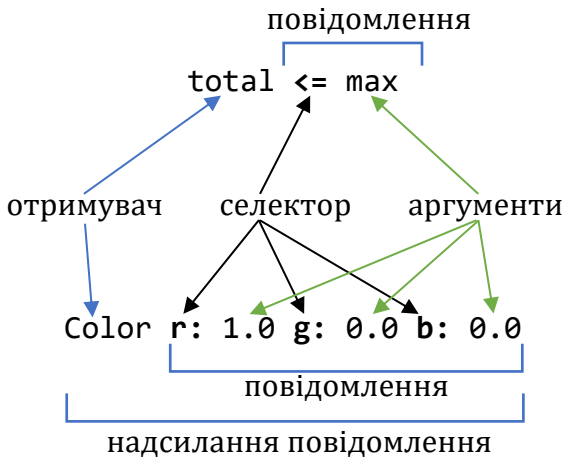

Рис. 9.1. Два приклади надсилання повідомлення, що складаються з отримувача, селектора методу та набору аргументів

Оскільки більшість обчислень у Pharo виконується через надсилання повідомлень, то правильне розпізнавання повідомлень має вирішальне значення. Нам допоможе така термінологія.

- Повідомлення складається з *селектора* та, можливо, аргументів.

- Повідомлення надсилають *отримувачу*.
- Отримувач і повідомлення до нього разом називають *надсилання повідомлення*, як показано на рис. 9.1.

Повідомлення завжди надсилається отримувачу, який може бути окремим літералом, блоком, змінною або результатом виконання іншого повідомлення. На схематичному зображенні підкреслимо отримувача повідомлення, щоб допомогти ідентифікувати його. А також обведемо кожне надсилання повідомлення еліпсом і пронумеруємо їх, починаючи від одиниці, щоб продемонструвати послідовність, у якій надсилаються повідомлення.

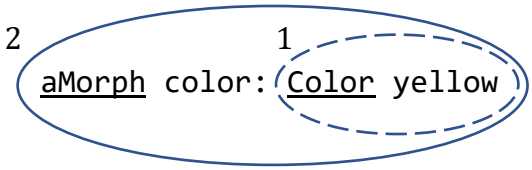

Рис. 9.2. Два надсилання повідомлень: «*Color yellow*» і «*aMorph color: Color yellow*»

На рис. 9.2 зображено два надсилання повідомлень: «*Color yellow*» і «*aMorph color: Color yellow*», тому є два еліпси. Спочатку надсилається повідомлення «*Color yellow*», тому його еліпс має номер 1. Є два отримувачі: *aMorph* отримує повідомлення *color:...*, і *Color* отримує *yellow*. Обидва отримувачі підкреслено.

Отримувачем може бути перший елемент виразу, як *100* у надсиланні повідомлення «*100 + 200*» або *Color* у «*Color yellow*». Проте отримувач може бути також результатом виконання іншого повідомлення. Наприклад, у виразі «*Pen new go: 100*» отримувач повідомлення «*go: 100*» – об'єкт, отриманий у результаті надсилання повідомлення «*Pen new*». Повідомлення завжди надсилається об'єкту, що називається отримувачем, який може бути результатом надсилання іншого повідомлення.

Надсилання повідомлення	**Вид повідомлення**	**Результат**
`Color yellow`	Унарне	Створює жовтий колір
`aPen go: 100`	Ключове	Переміщає ручку на 100 пікселів уперед
`100 + 20`	Бінарне	Додає 20 до 100
`Browser open`	Унарне	Відкриває нового оглядача
`Pen new go: 100`	Унарне та ключове	Створює нову ручку і переміщає її на 100 пікселів уперед
`aPen go: 100 + 20`	Ключове та бінарне	Обчислює 120 і переміщає ручку на 120 пікселів уперед

У таблиці наведено кілька прикладів надсилання повідомлень. Ви мали б помітити, що:

- не у всіх повідомлень є аргументи. Унарні повідомлення, як *open*, їх не мають;
- бінарні повідомлення (*+ 20*) та ключові з одним ключем (*go: 100*) мають по одному аргументу;

- є прості повідомлення та складені. Повідомлення «*Color yellow*» і «*100 + 20*» – прості: одне повідомлення надсилають одному об'єктові. Вираз «*aPen go: 100 + 20*» складається з двох повідомлень: повідомлення *go:* надсилають об'єктові *aPen* з аргументом, який є результатом виконання повідомлення *+ 20*, надісланого об'єктові *100*;
- отримувач може бути виразом (присвоєнням, надсиланням повідомлення або літералом), що повертає об'єкт. У виразі «*Pen new go: 100*» повідомлення *go: 100* надсилають об'єктові, отриманому внаслідок надсилання повідомлення *Pen new*.

9.2. Три види повідомлень

У Pharo визначено кілька простих правил, які задають порядок надсилання повідомлень. Ці правила опираються на відмінності трьох різних видів повідомлень.

- *Унарні повідомлення* – це такі повідомлення, які надсилають до об'єкта без ніякої додаткової інформації. Наприклад, у виразі *15 factorial* повідомлення *factorial* – унарне. Надсилання унарного повідомлення може виконати і базову унарну операцію, і довільний функціонал. Як би там не було, його завжди надсилають без аргументів.
- *Бінарні повідомлення* – це повідомлення, що складаються з операторів (часто арифметичних) і виконують базові бінарні операції. Їх називають бінарними, оскільки вони завжди залучають два об'єкти: отримувача та єдиний аргумент. Наприклад, у виразі «*100 + 20*» *+* бінарне повідомлення, надіслане з аргументом *20* отримувачу *100*.
- *Ключові повідомлення* – це повідомлення, які складаються з одного або кількох ключових слів, кожне з яких закінчується двокрапкою (:) і приймає один аргумент. Наприклад, у виразі «*anArray at: 1 put: 10*» селектор повідомлення – *at:put:*. Ключове слово *at:* приймає аргумент *1*, а *put:* – аргумент *10*.

Важливо зазначити, що:

- не буває надсилання ключових повідомлень без аргументів. Усі повідомлення без аргументів – унарні;
- ключові повідомлення, що мають тільки один аргумент, відрізняються від бінарних повідомлень двокрапкою, яку використовують для вказання кожного аргументу ключового повідомлення.

Унарні повідомлення

Унарні повідомлення – це повідомлення, які не потребують жодних аргументів. Вони відповідають синтаксичному шаблону: *отримувач ім'я_повідомлення*. Селектор складається просто з послідовності літер без двокрапки (:), наприклад, *factorial*, *open*, *class*.

```
89 sin
>>> 0.860069405812453
```

```
3 sqrt
>>> 1.732050807568877
```

```
Float pi
>>> 3.141592653589793
```

```
345 negated
>>> -345
```

```
'blop' size
>>> 4
```

```
true not
>>> false
```

```
Object class
>>> Object class "The class of Object is Object class (BANG)"
```

Важливо Унарні повідомлення відповідають синтаксичному шаблону *отримувач* ***селектор***.

Бінарні повідомлення

Бінарні повідомлення – це повідомлення, які потребують точно один аргумент і чий селектор складається з послідовності одної або кількох літер з набору +, –, *, /, &, =, >, |, <, ~ й @. Запам'ятайте, що селектор –– заборонено через особливості розпізнавання.

```
100@100 >>> 100@100
"creates a Point object"
```

```
3 + 4
>>> 7
```

```
10 – 1
>>> 9
```

```
4 <= 3
>>> false
```

```
(4/3) * 3 == 4
>>> true "equality is just a binary message, and Fractions are exact"
```

```
(3/4) == (3/4)
>>> false "two equal Fractions are not the same object"
```

Важливо Бінарні повідомлення відповідають синтаксичному шаблону *отримувач* ***селектор*** *аргумент*.

Ключові повідомлення

Ключові повідомлення – це повідомлення, що потребують одного чи більше аргументів, і чий селектор складається з одного або кількох ключових слів, кожне з яких закінчується двокрапкою (:).

У прикладі нижче повідомлення складається з двох ключових слів: *between:* і *and:*. Повний селектор повідомлення – *between:and:*, повідомлення надсилають числу *2*.

```
2 between: 0 and: 10
>>> true
```

Кожне ключове слово приймає аргумент. Так «*r:g:b:*» – повідомлення з трьома аргументами, «*at:put:*» – повідомлення з двома аргументами, а «*playFileNamed:*» і «*at:*» – повідомлення з одним аргументом кожне. Щоб створити екземпляр класу *Color*, можна використати повідомлення «*r:g:b:*», як у прикладі нижче. Не забувайте, що двокрапки є частиною селектора.

```
Color r: 1 g: 0 b: 0
>>> Color red "creates a new color"
```

У Java-подібному синтаксисі надсиланню повідомлення «*Color r: 1 g: 0 b: 0*» відповідатиме виклик методу «*Color.rgb(1, 0, 0)*».

```
1 to: 10
>>> (1 to: 10) "creates an interval"
```

```
| nums |
nums := Array newFrom: (1 to: 5).
nums at: 1 put: 6.
nums
>>> #(6 2 3 4 5)
```

Важливо Ключові повідомлення відповідають синтаксичному шаблону *отримувач* ***селекторСловоОдин:*** *аргумент1* ***словоДва:*** *аргумент2 …* ***словоN:*** *аргументN*.

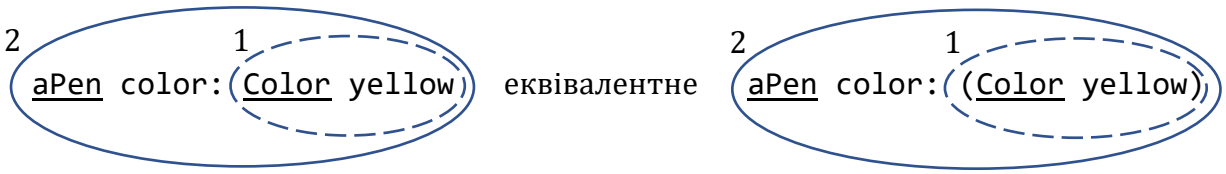

Рис. 9.3. Спочатку надсилаються унарні повідомлення, тому «*Color yellow*» буде першим. Воно поверне об’єкт, який стане аргументом для «*aPen color: Color yellow*»

9.3. Композиція повідомлень

Повідомлення кожного з трьох видів мають інший пріоритет, що дає змогу елегантно комбінувати їх.

- Унарні повідомлення завжди надсилаються першими, тоді – бінарні, і насамкінець – ключові.
- Повідомлення в круглих дужках мають найвищий пріоритет.
- Повідомлення однакового пріоритету опрацьовуються зліва направо.

Ці правила формують дуже природний порядок прочитання. Але, якщо ви хочете бути впевненими, що ваші повідомлення надсилаються в потрібному порядку, то завжди

можете поставити більше дужок, як показано на рис. 9.3. На цій схемі повідомлення *yellow* унарне, а *color:* ключове, тому першим надсилається *Color yellow*. Оскільки повідомлення, надіслані в дужках, надсилаються першими, то взяття *Color yellow* у (непотрібні) дужки лише підкреслює, що воно буде надіслано першим. Далі в розділі проілюстровано кожен із цих моментів.

Унарні > Бінарні > Ключові

Унарні повідомлення надсилаються першими, тоді – бінарні, і насамкінець – ключові. Ми також можемо сказати, що унарні повідомлення мають вищий пріоритет ніж інші.

Важливо *Унарні* > *Бінарні* > *Ключові*

Як видно з прикладів, правила синтаксису Pharo загалом гарантують, що вирази програми можна читати природно, як звичайний текст.

```
1000 factorial / 999 factorial
>>> 1000
```

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

На жаль, правила занадто спрощені для надсилання арифметичних повідомлень, тому доводиться записувати дужки, щоб задати черговість виконання бінарних операторів.

```
1 + 2 * 3
>>> 9
```

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

Арифметичним невідповідностям присвячено окремий параграф.

Наступний приклад трохи складніший (!), проте він добре ілюструє те, що навіть такі вирази можна читати природно.

```
[:aClass | aClass methodDict keys select: [:aMethod |
    (aClass >> aMethod) isAbstract ]] value: Boolean
>>> #(#ifTrue: #| #xor: #asBit #ifFalse:ifTrue: #ifFalse:
      #ifTrue:ifFalse: #or: #& #and: #not)
```

Тут ми хочемо довідатися, які методи класу *Boolean* абстрактні. Запитуємо якийсь клас, аргумент *aClass* блока, про ключі його словника методів і вибираємо ті методи, які є абстрактними. Потім ми прив'язуємо аргумент *aClass* до конкретного значення *Boolean*. Круглі дужки потрібні лише для того, щоб надіслати бінарне повідомлення >>, яке вибирає метод із класу, перед тим, як надіслати цьому методу унарне повідомлення *isAbstract*. З результату зрозуміло, які методи потрібно реалізувати в конкретних підкласах *True* та *False* класу *Boolean*.

Насправді, ми могли б написати простіший еквівалентний вираз «*Boolean methodDict select: [:each | each isAbstract] thenCollect: [:each | each selector].*»

Приклад. У виразі *aPen color: Color yellow* одне унарне повідомлення *yellow* до класу *Color* і одне ключове *color:* до екземпляра *aPen*. Спочатку надсилають унарні повідом-

лення, тому *Color yellow* надсилається першим. Унаслідок його виконання повернеться екземпляр, назвемо його *aColor*, який стане аргументом повідомлення *aPen color: aColor*. На рис. 9.3 графічно зображено почерговість надсилання повідомлень.

```
"Декомпозиція виконання виразу aPen color: Color yellow"
   aPen color: Color yellow
(1)  Color yellow       "спочатку надсилаються унарні повідомлення"
>>> aColor
(2)  aPen color: aColor "потім - ключові"
```

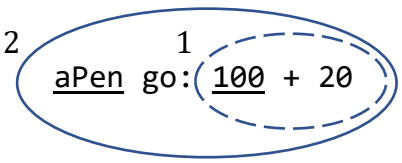

Рис. 9.4. Бінарні повідомлення надсилаються перед ключовими

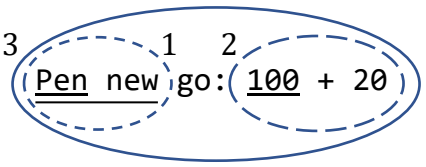

Рис. 9.5. Декомпозиція виразу *Pen new go: 100 + 20*

Приклад. У виразі *aPen go: 100 + 20* два повідомлення: *бінарне* «*+ 20*» і *ключове* «*go:…*». Бінарні повідомлення мають вищий пріоритет ніж ключові, тому спочатку виконається надсилання *100 + 20*: об'єкт *100* отримає повідомлення *+ 20*, повернеться результат – об'єкт *120*. Потім об'єктові *aPen* буде надіслано повідомлення *go: 120*. Послідовність обчислення виразу графічно зображена на рис. 9.4 і текстом програми у коді нижче.

```
"Декомпозиція обчислення виразу aPen go: 100 + 20"
   aPen go: 100 + 20
(1)  100 + 20     "спочатку надсилають бінарні повідомлення"
>>> 120
(2)  aPen go: 120 "потім - ключові"
```

Приклад. Виконайте як вправу декомпозицію виконання виразу *Pen new go: 100 + 2*, складеного з унарного, ключового та бінарного повідомлень (див. рис. 9.5).

Спочатку дужки

Повідомлення в дужках мають вищий пріоритет ніж усі інші.

Важливо (*Вираз*) > *Унарні* > *Бінарні* > *Ключові*

Наведемо кілька прикладів.

З першого прикладу бачимо, що можна обійтися без дужок, якщо порядок обчислення вже такий, як нам треба, тобто, результат обчислення виразу з дужками буде такий самий, як без дужок. Далі обчислюємо тангенс числа *1.5*, заокруглюємо його та перетворюємо на рядок.

```
1.5 tan rounded asString = (((1.5 tan) rounded) asString)
>>> true
```

Другий приклад демонструє, що обчислення факторіала має вищий пріоритет ніж додавання. Якщо потрібно спочатку додати *3* та *4*, то бінарне повідомлення *+* треба взяти в дужки.

```
3 + 4 factorial
>>> 27 "(не 5040)"
```

```
(3 + 4) factorial
>>> 5040
```

Подібно в наступному прикладі дужки потрібні, щоб надсилання повідомлення *lowMajorScaleOn:* відбулося перед *play*.

```
(FMSound lowMajorScaleOn: FMSound clarinet) play
"(1)  надсилання clarinet класові FMSound, щоб створити звук кларнета
 (2)  використання цього звуку в ключовому повідомленні lowMajorScaleOn:
       класові FMSound
 (3)  відтворення отриманого звуку повідомленням play"
```

Приклад. Результат обчислення виразу *(65@325 extent: 134@100) center* – центр прямокутника з лівою верхньою вершиною в точці (65; 325) і розмірами 134×100. Нижче описано декомпозицію виразу, що демонструє порядок надсилання повідомлень у ньому. Спочатку надсилаються повідомлення в дужках: два бінарні повідомлення *@325* і *@100* надсилаються числам *65* і *134*, відповідно, та повертають точки. Перша з них отримує ключове повідомлення *extent:* з аргументом другою точкою, що повертає прямокутник. Насамкінець унарне повідомлення *center* надсилається створеному прямокутнику і повертає точку – його центр. Обчислення виразу без дужок спровокувало б помилку, бо найвищий пріоритет мало б унарне повідомлення *center*, а об'єкт *100* його не розуміє.

```
"Приклад використання дужок"
   (65@325 extent: 134@100) center
(1)  65@325 "бінарне"
>>> aPoint
(2)  134@100 "бінарне"
>>> anotherPoint
(3)  aPoint extent: anotherPoint "ключове"
>>> aRectangle
(4)  aRectangle center "унарне"
>>> 132@375
```

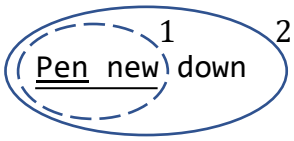

Рис. 9.6. Декомпозиція виразу *Pen new down*

Зліва направо

Тепер ми знаємо, як опрацьовуються повідомлення різних видів або пріоритетів. Залишилося з'ясувати, як надсилаються повідомлення з однаковим пріоритетом. Їх відправляють зліва направо. Зауважте, що ви вже бачили таку поведінку у прикладі *1.5 tan rounded asString*, де всі унарні повідомлення надсилаються зліва направо, що еквівалентно *(((1.5 tan) rounded) asString)*.

Важливо Порядок надсилання повідомлень однакового виду – зліва направо.

Приклад. У виразі *Pen new down* усі повідомлення унарні, тому першим буде надіслане перше зліва: спочатку виконається *Pen new*. Воно поверне нову ручку, яка отримає повідомлення *down* (див. рис. 9.6).

Арифметичні невідповідності

Правила композиції повідомлень прості. Немає поняття математичного пріоритету, тому що арифметичні повідомлення – просто бінарні повідомлення, як і будь-які інші. Тож результат їхнього виконання може видатися неправильним. Далі опишемо типові ситуації, коли потрібні додаткові дужки.

```
3 + 4 * 5
>>> 35 "(не 23) Бінарні повідомлення надсилаються зліва направо"
```

```
3 + (4 * 5)
>>> 23
```

```
1 + 1/3
>>> (2/3) "а не 4/3"
```

```
1 + (1/3)
>>> (4/3)
```

```
1/3 + 2/3
>>> (7/9) "а не 1"
```

```
(1/3) + (2/3)
>>> 1
```

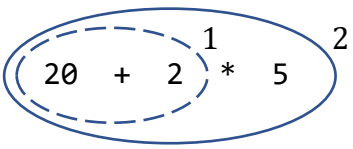

Рис. 9.7. Звичайний порядок виконання

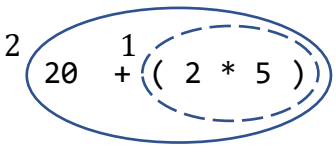

Рис. 9.8. Зміна порядку виконання за допомогою дужок

Приклад. У виразі *20 + 2 * 5* тільки бінарні повідомлення *+* і ***. А у Pharo немає окремих пріоритетів для операторів *+* і ***. Це просто бінарні повідомлення, тому *** не має вищого пріоритету ніж *+*. Першим буде надіслане перше зліва повідомлення *+ 2* об'єктові *20*, потім результат додавання отримає повідомлення ** 5*. Порядок виконання виразу показано на рис. 9.7 і в коді нижче.

```
"Усі бінарні повідомлення мають однаковий пріоритет, тому перше зліва
 повідомлення + 2 буде виконано першим, незважаючи на правила арифметики,
 згідно з якими першим мало б бути * 5."

     20 + 2 * 5
(1)  20 + 2 >>> 22
(2)      22 * 5 >>> 110
```

Як видно з попереднього прикладу, результатом обчислення виразу є 110, а не 30. Це виглядає, мабуть, несподівано, але випливає з правил надсилання повідомлень. Це ціна, яку доводиться платити за простоту моделі. Щоб отримати правильний результат, потрібно використати дужки. Повідомлення в дужках надсилається першим, тому вираз *20 + (2 * 5)* поверне правильний результат, як зображено в коді нижче та на рис. 9.8.

```
"Повідомлення в дужках надсилаються першими, тому * 5 буде надіслано
 перед + (), що дасть правильний результат."
     20 + (2 * 5)
(1)        2 * 5 >>> 10
(2)  20 + 10 >>> 30
```

Важливо Арифметичні оператори у Pharo мають однаковий пріоритет. *+* і *** просто бінарні повідомлення, тому *** не має більшого пріоритету ніж *+*. Використовуйте дужки, щоб отримати очікуваний результат.

Неявний пріоритет (за правилами)	Пріоритет явно заданий дужками
`aPen color: Color yellow`	`aPen color: (Color yellow)`
`aPen go: 100 + 20`	`aPen go: (100 + 20)`
`aPen penSize: aPen penSize + 2`	`aPen penSize: ((aPen penSize) + 2)`
`20 factorial + 4`	`(20 factorial) + 4`

Правила пріоритетності унарних, бінарних і ключових повідомлень у багатьох випадках дають змогу не використовувати дужки. У таблиці в лівому стовпці показані складені повідомлення, записані з урахуванням правил пріоритетності, а в правому – відповідні їм повідомлення, якби таких правил не було. Обидва варіанти надсилання повідомлень діють однаково, або повертають однаковий результат.

9.4. Як розпізнати ключове повідомлення

У початківців часто виникають труднощі з розумінням того, коли потрібно використовувати дужки. Давайте розглянемо, як компілятор розпізнає ключові повідомлення.

Ставити дужки чи ні?

Літери [,], (і) обмежують різні ділянки. У межах такої ділянки ключове повідомлення є найдовшою послідовністю слів, що закінчуються двокрапкою, не перерваною літерами крапка, кома, або крапка з комою.

У наступному прикладі два різних ключових повідомлення: *rotatedBy:magnify:smoothing:* і *at:put:*.

```
aDict
   at: (rotatingForm rotateBy: angle magnify: 2 smoothing: 1)
   put: 3
```

Підказка. Якщо ви відчуваєте труднощі з застосуванням правил пріоритетності, то можете ставити дужки щоразу, коли хочете відокремити два повідомлення з однаковим пріоритетом.

У фраґменті коду нижче дужки не потрібні, бо унарне повідомлення *isNil* має вищий пріоритет ніж ключове повідомлення *ifTrue: [ ]*.

```
(x isNil)
   ifTrue: [ ... ]
```

У наступному фраґменті дужки потрібні обов'язково, бо обидва повідомлення: і *includes:*, і *ifTrue:* – ключові.

```
ord := OrderedCollection new.
(ord includes: $a)
   ifTrue: [ ... ]
```

Якби тут не було дужок, колекція *ord* отримала б невідоме повідомлення *includes:ifTrue:*.

Коли використовувати [ ], а коли ( ) ?

Також можуть виникати труднощі з розумінням того, коли використовувати квадратні дужки, а коли круглі. Головним критерієм є те, скільки разів ви збираєтеся обчислювати вираз. Квадратні дужки перетворять вираз на блокове замикання, яке можна виконати довільну кількість разів, або не виконати жодного, залежно від контексту. Тому квадратні дужки застосовують тоді, коли наперед *невідомо*, скільки разів буде обчислено вираз. Нагадаємо, що вираз може бути надсиланням повідомлення, змінною, літералом, присвоєнням або блоком.

Невідомо, який з аргументів повідомлення *ifTrue:ifFalse:* буде виконано, бо це залежить від результату обчислення умови. Тому його аргументами є блоки. З таких самих міркувань отримувач і аргумент повідомлення *whileTrue:* потребують квадратних дужок. Адже не відомо, скільки разів буде обчислено чи отримувач, чи аргумент.

На противагу цьому, круглі дужки впливають тільки на порядок надсилання повідомлень. Тому кожного разу під час обчислення виразу *(expression)* обчислення *expression* відбудеться точно *один* раз.

```
"І отримувач, і аргумент мусять бути блоками"
[ x isReady ] whileTrue: [ y doSomething ]

"Аргумент буде обчислено кілька разів, тому він мусить бути блоком"
4 timesRepeat: [ Beeper beep ]

"Отримувача обчислюють один раз, тому він не блок.
 Аргумент може не бути обчислений ні разу, тому він блок"
(x isReady) ifTrue: [ y doSomething ]
```

9.5. Послідовність повідомлень

Вирази (наприклад, надсилання повідомлень, присвоєння тощо), відокремлені крапками, виконуються послідовно. Зверніть увагу, між оголошенням локальних змінних і наступним виразом крапка не потрібна. Значенням послідовності виразів є значення, отримане внаслідок обчислення її останнього виразу. Результати обчислення всіх інших виразів послідовності ігноруються. Зазначимо, що крапка є розділювачем, а не термінальним символом, тому після останнього виразу послідовності її можна не ставити.

```
| box |
box := 20@30 corner: 60@90.
box containsPoint: 40@50
>>> true
```

9.6. Каскад повідомлень

Pharo пропонує спосіб надсилання кількох повідомлень тому самому отримувачу без потреби зазначати його щоразу: достатньо відокремити повідомлення крапкою з комою (;). На жаргоні Pharo це називається каскадом.

Схематично синтаксис каскаду можна зобразити так.

```
aReceiverExpression msg1; msg2; msg3
```

Приклади. У Pharo можна програмувати без каскадів. Тоді доведеться зазначати отримувача кожного повідомлення. Наступні два фрагменти коду діють однаково.

```
Transcript show: 'Pharo is '.
Transcript show: 'fun'.
Transcript cr.
```

```
Transcript
   show: 'Pharo is ';
   show: 'fun';
   cr
```

Насправді одержувачем усіх повідомлень каскаду є одержувач першого повідомлення, залученого в каскад. Зверніть увагу, що об'єкт, який отримує каскадні повідомлення, сам може бути результатом надсилання повідомлення. У наступному прикладі *setX:setY:* перше повідомлення каскаду, бо за ним стоїть крапка з комою. Одержувач каскадного повідомлення – це щойно створена точка в результаті виконання *Point new*, а не *Point*. Наступне повідомлення *isZero* надсилається тому самому отримувачу – результатові виконання *Point new*.

```
Point new setX: 25 setY: 35; isZero
>>> false
```

9.7. Підсумки розділу

- Повідомлення завжди надсилають об'єктові. Його називають отримувачем. Він може бути результатом надсилання іншого повідомлення.
- Унарні повідомлення – це повідомлення без аргументів. Вони мають вигляд *receiver **selector***.
- Бінарні повідомлення залучають два об'єкти: отримувача й аргумент. Їхній селектор складається з однієї або більше літери з-поміж +, –, *, /, |, &, =, >, <, ~, @. Вони мають вигляд *receiver **selector** argument*.
- Ключові повідомлення залучають більше ніж один об'єкт і містять у селекторі хоча б одну двокрапку. Вони мають вигляд *receiver **selectorKeywordOne**: argumentOne **keywordTwo**: argumentTwo ... **keywordN**: argumentN*.

- **Правило Один.** Спочатку надсилаються унарні повідомлення, потім – бінарні, а насамкінець – ключові.
- **Правило Два.** Повідомлення в круглих дужках надсилаються перед усіма іншими.
- **Правило Три.** Послідовність надсилання повідомлень одного виду – зліва направо.
- Звичайні арифметичні оператори, як + і *, у Pharo мають однаковий пріоритет. І +, і * є бінарними повідомленнями, тому * не має вищого пріоритету ніж +. Щоб отримати правильний результат у арифметичних обчисленнях, потрібно використовувати круглі дужки.
- Комбінацію літер «––» заборонено використовувати як селектор бінарного повідомлення.

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
```

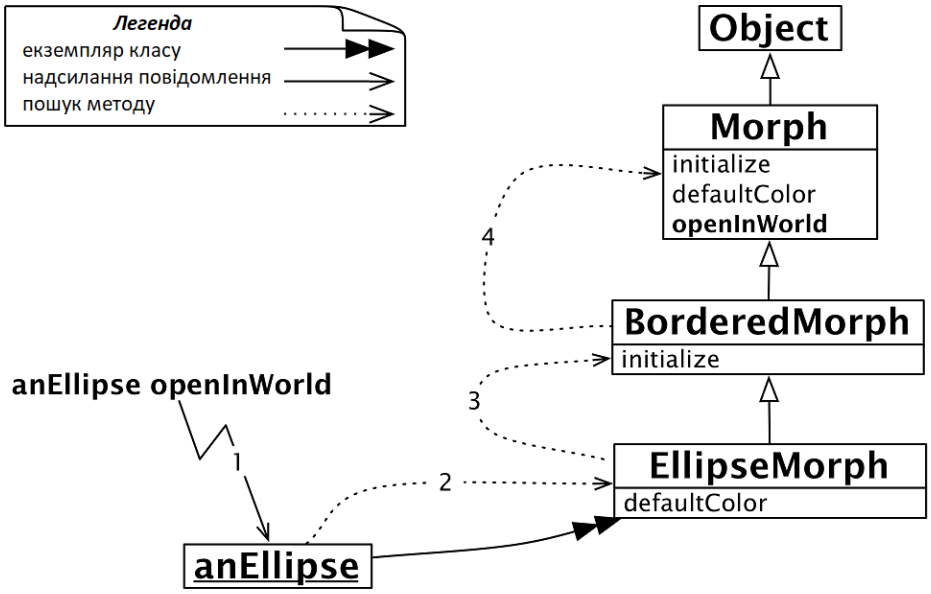

Рис. 10.1. Пошук методу в ланцюжку наслідування

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.

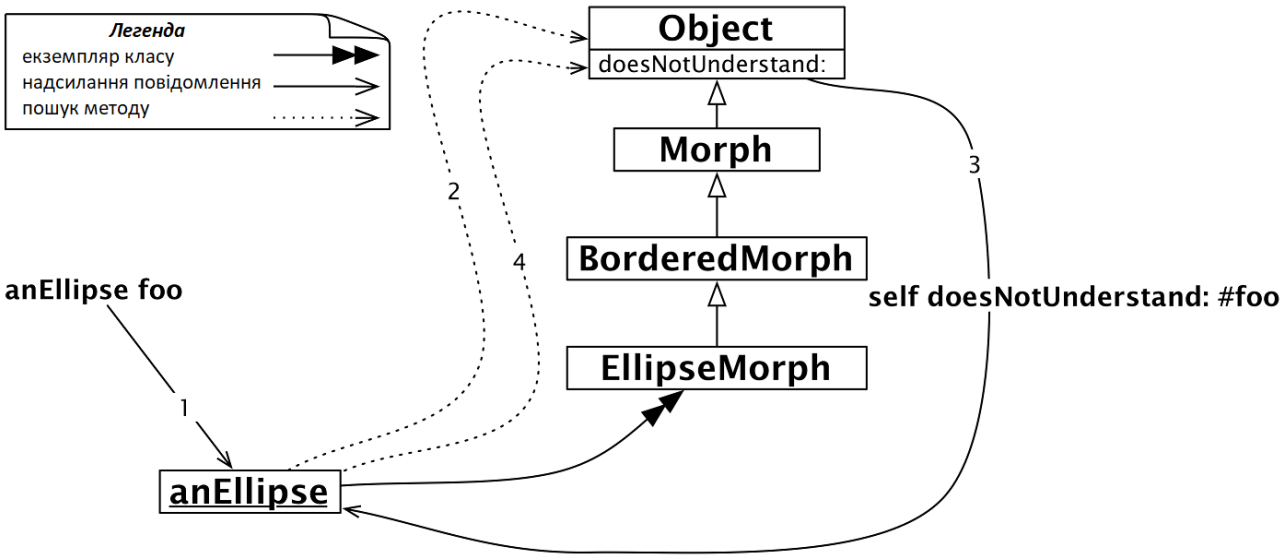

Рис. 10.2. Еліпс не зрозумів повідомлення *foo*

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).

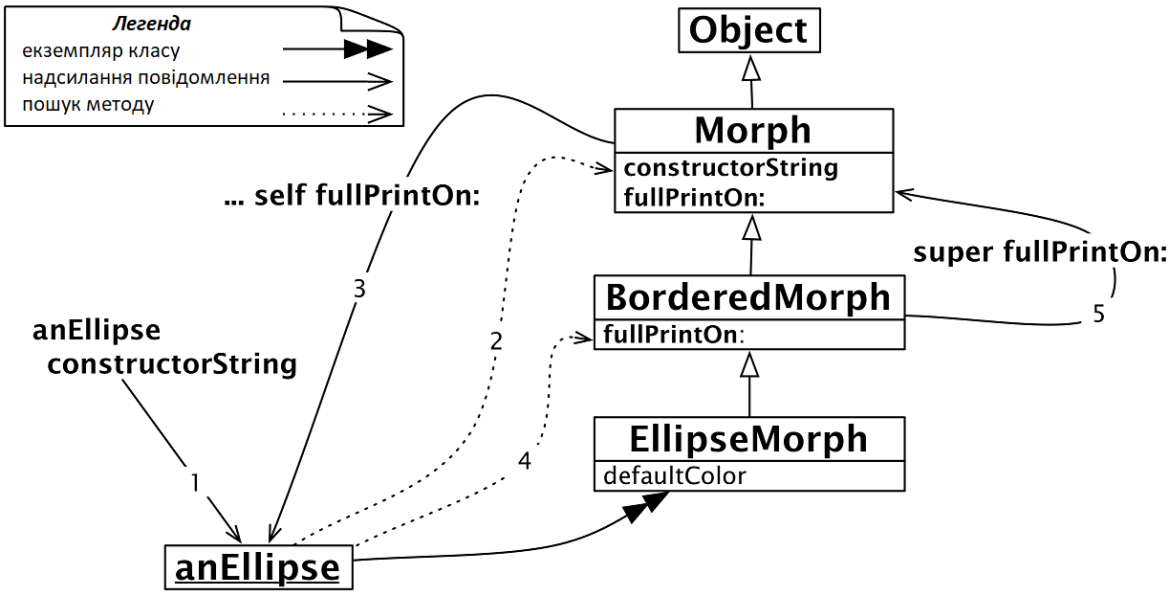

Рис. 10.3. Надсилання повідомлень до *self* і *super*

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.

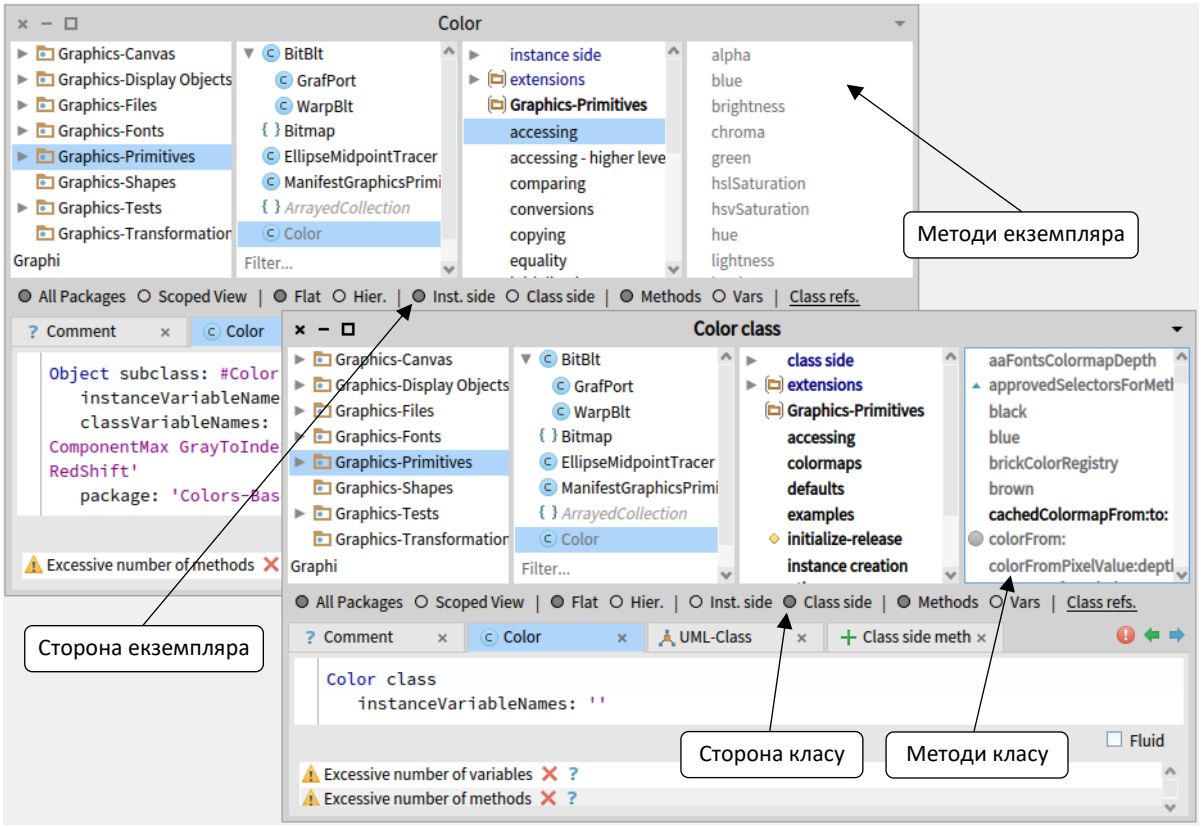

Рис. 10.4. Перегляд класу та його метакласу

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.[10]

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
```

[10] 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*».

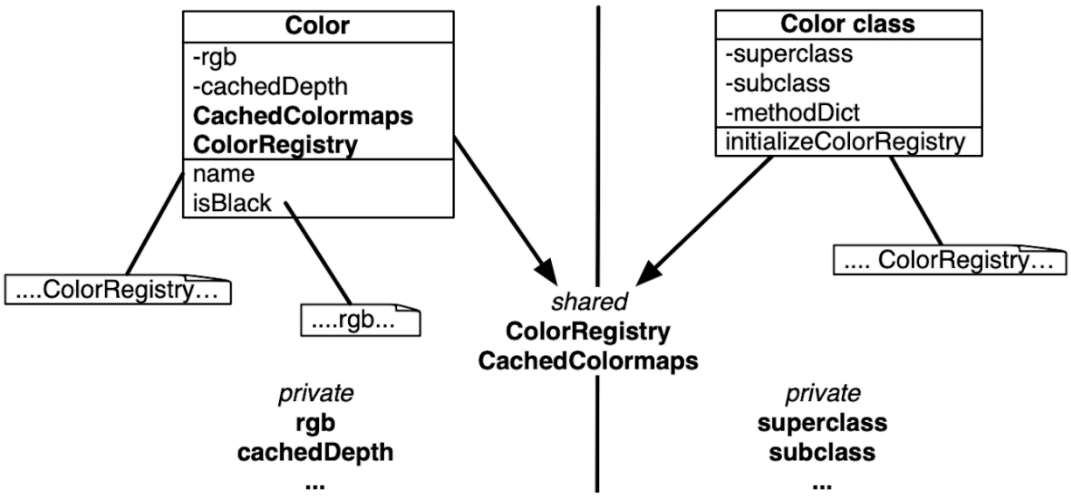

Рис. 10.5. Методи класу та методи екземпляра мають доступ до різних змінних

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.

Розділ 11

Ознаки – код для повторного використання

Хоча Pharo пропонує тільки просте наслідування класів, воно підтримує механізм, який називається *Traits, Ознаки*[11], для спільного використання коду, який описує поведінку і стан, різними непов'язаними між собою класами. Ознака містить набір методів, які можуть працювати в різних класах без обмежень щодо відношення наслідування.

За допомогою ознак можна поширювати код серед різних класів без дублювання. Це полегшує для класів повторне використання певної поведінки, інкапсульованої в одному місці.

Як побачимо, ознаки пропонують спосіб компонування та вирішення конфліктів імен строго визначеним способом. У використанні ознак не обов'язково перемагає останній завантажений метод, як це відбувається в інших мовах. У Pharo чи клас, чи ознака завжди беруть до уваги пріоритетність і вирішують у власному контексті, як усунути конфлікт: методи можна вилучити з компонування, або зробити їх доступними під новим іменем.

11.1. Проста ознака

Наведений нижче код визначає ознаку за допомогою надсилання повідомлення *named:uses:package:* класові *Trait*. Частина *uses:* повідомлення містить порожній масив, що засвідчує те, що ця ознака не містить інших ознак.

```
Trait named: #TFlyingAbility
   uses: {}
   package: 'Traits-Example'
```

Ознаки можуть визначати методи. Ознака *TFlyingAbility* визначає один метод *fly*.

```
TFlyingAbility >> fly
  ^ 'I''m flying!'
```

Ознака не призначена для створення екземплярів – її має використати клас. Цей клас створить екземпляри, які зможуть відповідати на реалізовані в ній повідомлення.

Тепер визначимо клас *Bird*. Він використовує ознаку *TFlyingAbility*. Завдяки цьому клас містить метод *fly*.

```
Object subclass: #Bird
   uses: TFlyingAbility
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Traits-Example'
```

[11] Дослівний переклад слова *traits* – типаж, характерні риси – не дуже доречний у контексті можливостей цього механізму, тому в книзі використано близький варіант *ознака*. Traits (у Pharo) – колекція методів, які можна залучати для побудови класу. Уміння виконувати певні методи – ознака класу, тому, на думку перекладача, термін *ознака* підходить найкраще.

Екземпляри класу *Bird* уміють відповідати на повідомлення *fly*.

```
Bird new fly
>>> 'I''m flying!'
```

11.2. Виклик необхідного методу

Методи не ознаки не мусять визначати повністю всю поведінку. Метод ознаки може викликати методи, які стануть доступними тільки в класі, який використає ознаку.

У прикладі метод *greeting* ознаки *TGreetable* викликає метод *name*, не визначений в ознаці. В такому випадку клас, який використає ознаку, має реалізувати такий *необхідний* метод.

```
Trait named: #TGreetable
   uses: {}
   package: 'Traits-Example'
```

```
TGreetable >> greeting
  ^ 'Hello ', self name
```

Зверніть увагу, що *self* в методі ознаки представляє отримувача повідомлення. Тут діють такі самі правила, як у методах класу.

```
Object subclass: #Person
   uses: TGreetable
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Traits-Example'
```

Визначено клас *Person*, який використовує ознаку *TGreetable*. Потрібно визначити метод *name* в класі *Person*. Його викликатиме метод *TGreetable >> greeting*.

```
Person >> name
  ^ 'Bob'
```

```
Person new greeting
>>> 'Hello Bob'
```

11.3. У методі ознаки *self* вказує на отримувача

Можна засумніватися, на що вказує *self* у методі ознаки. Проте немає різниці між використанням *self* у методі, визначеному в класі, та в методі, визначеному в ознаці: *self* завжди представляє отримувача повідомлення. Місце визначення методу: чи це клас, чи ознака – ніяк не впливає на *self*.

На підтвердження визначено невелику ознаку, її метод *whoAmI* лише повертає *self*.

```
Trait named: #TInspector
   uses: {}
   package: 'Traits-Example'
```

```
TInspector >> whoAmI
  ^ self
```

```
Object subclass: #Foo
   uses: TInspector
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Traits-Example'
```

Наведений нижче фрагмент коду демонструє, що *self* – це отримувач, навіть, якщо повертається з методу ознаки.

```
| foo |
foo := Foo new.
foo whoAmI == foo
>>> true
```

11.4. Стан ознаки

Починаючи з Pharo 7.0, в ознаках можна оголошувати змінні екземпляра. У новому прикладі ознака *TCounting* визначає змінну екземпляра, що називається *count*.

```
Trait named: #TCounting
   instanceVariableNames: 'count'
   package: 'Traits-Example'
```

Ознака може ініціалізувати свій стан спеціальним методом, чий селектор, за домовленістю, будують зі слова «*initialize*» та імені ознаки слідом. Далі ознака *TCounting* визначає метод *initializeTCounting* і метод збільшення значення змінної.

```
TCounting >> initializeTCounting
  count := 0

TCounting >> increment
  count := count + 1.
  ^ count
```

Клас *Counter* використовує ознаку *TCounting*, тому його екземпляри матимуть змінну *count*.

```
Object subclass: #Counter
   uses: TCounting
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Traits-Example'.
```

Щоб правильно ініціалізувати екземпляр класу *Counter*, метод *Counter >> initialize* мав би викликати визначений в ознаці метод *initializeTCounting*.

```
Counter >> initialize
  self initializeTCounting
```

У наступному фрагменті створено екземпляр класу *Counter*. Видно, що його змінну екземпляру ініціалізовано правильно.

```
Counter new increment; increment
>>> 2
```

11.5. Клас може використати кілька ознак

Клас не обмежено використанням лише однієї ознаки. Він може використати їх кілька. Раніше було оголошено ознаку *TFlyingAbility*, а тепер оголосимо ще одну – *TSpeaking-Ability*.

```
Trait named: #TSpeakingAbility
   uses: {}
   package: 'Traits-Example'
```

Вона визначає метод *speak*.

```
TSpeakingAbility >> speak
  ^ 'I''m speaking!'
```

Тепер клас *Duck* може використати обидві ознаки: і *TFlyingAbility*, і *TSpeakingAbility*.

```
Object subclass: #Duck
   uses: TFlyingAbility + TSpeakingAbility
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Traits-Example'
```

Екземпляр класу *Duck* отримав поведінку з обох ознак.

```
| d |
d := Duck new.
d speak
>>> 'I''m speaking!'
d fly
>>> 'I''m flying!'
```

11.6. Перевизначений метод має вищий пріоритет, ніж метод ознаки

Метод, родом з ознаки, діє так само, ніби його визначили в класі (чи ознаці), що використовує її. Користувач ознаки (це може бути клас або інша ознака) завжди може перевизначити метод, який походить з неї, і перевизначений метод матиме вищий пріоритет, ніж метод ознаки.

Проілюструємо це. Можна було б перевизначити метод *speak* у класі *Duck* так, щоб він робив щось інше, наприклад, надсилав повідомлення *quack*.

```
Duck >> quack
  ^ 'QUACK'
Duck >> speak
  ^ self quack
```

Це означає, що:

- метод *TSpeakingAbility >> speak* більше не доступний з класу *Duck*, і
- замість нього використовується новий метод, навіть тими методами, які надсилають повідомлення *speak*.

```
Duck new speak
>>> 'QUACK'
```

Додамо до ознаки *TSpeakingAbility* оголошення ще одного методу.

```
TSpeakingAbility >> doubleSpeak
  ^ 'I double: ', self speak, ' ', self speak
```

Легко переконатися, що визначений в класі *Duck* метод *speak* має вищий пріоритет, ніж його однойменний конкурент з ознаки *TSpeakingAbility*.

```
Duck new doublespeak
>>> 'I double: QUACK QUACK'
```

11.7. Доступ до перевантаженого методу ознаки

Іноді виникає потреба перевизначити метод ознаки і водночас зберегти доступ до нього. Це можливо за допомогою створення псевдоніма перевизначеного методу в частині оголошення використання ознаки операторами @ і -> як у прикладі.

```
Object subclass: #Duck
   uses: TFlyingAbility + TSpeakingAbility @ {#originalSpeak -> #speak}
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Traits-Example'
```

Стрілка означає, що новий метод це те саме, що й старий, тільки оголошено нове ім'я. Тут сказано, що *originalSpeak* нове ім'я методу *speak*.

Щоб доступитися до перевантаженого методу, надсилають повідомлення з його новим іменем так, ніби це ім'я звичайного методу. Далі оголошено новий метод *differentSpeak*, який надсилає повідомлення *originalSpeak*.

```
Duck >> differentSpeak
  ^ self originalSpeak, ' ', self speak
```

```
Duck new differentSpeak
>>> 'I''m speaking! QUACK'
```

Будьте уважні, бо оголошення псевдоніма не змінює імені методу. Справді, якщо йдеться про рекурсивний метод, то він не звертатиметься за новим іменем, а за старим. Псевдонім додає нове ім'я наявному методу, але не змінює його визначення: тіло методу залишиться незмінним. Отже, псевдонім оголошують, так би мовити, для зовнішнього використання.

11.8. Опрацювання конфлікту

Може трапитися, що дві ознаки, використані в одному класі, визначають однойменний метод. Така ситуація призводить до конфлікту. Щоб вирішити його, можна використати дві різні стратегії.

1. За допомогою оператора вилучення (–) можна вилучити конфліктуючий метод з однієї ознаки. Тоді в класу залишиться інший.
2. Перевизначити конфліктуючий метод у класі. У цьому випадку клас матиме доступ тільки до нового методу, відповідні методи ознак стануть недоступними, і конфлікт буде вичерпано. Зрозуміло, що доступ до обох методів ознак можна зберегти за допомогою псевдонімів, як було пояснено раніше.

Розглянемо приклад. Визначимо нову ознаку *THighFlyingAbility*.

```
Trait named: #THighFlyingAbility
   instanceVariableNames: ''
   package: 'Traits-Example'
```

Вона також визначає метод *fly*.

```
THighFlyingAbility >> fly
  ^ 'I''m flying high'
```

Якщо тепер оголосити клас *Eagle*, який використовує обидві ознаки, і *THighFlyingAbility*, і *TFlyingAbility*, то отримаємо конфлікт під час опрацювання повідомлення *fly* до екземпляра класу, бо система не знатиме, котрий з методів виконати.

```
Object subclass: #Eagle
   uses: THighFlyingAbility + TFlyingAbility
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Traits-Example'
```

```
Eagle new fly
>>> 'A class or trait does not properly resolve a conflict between
    multiple traits it uses.'
```

11.9. Вирішення конфлікту – вилучити метод

Щоб вирішити конфлікт під час компонування, можна вилучити метод *fly* з ознаки *TFlyingAbility*:

```
Object subclass: #Eagle
   uses: THighFlyingAbility + (TFlyingAbility - #fly)
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Traits-Example'
```

Тепер у класу тільки один метод *fly*, отриманий з *THighFlyingAbility*.

```
Eagle new fly
>>> 'I''m flying high'
```

11.10. Вирішення конфлікту – перевизначити метод

Інший спосіб вирішити конфлікт – перевизначити конфліктуючий метод у класі, який використовує ознаки.

```
Object subclass: #Eagle
   uses: THighFlyingAbility + TFlyingAbility
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Traits-Example'

Eagle >> fly
  ^ 'Flying and flying high'
```

Тепер доступний тільки один метод *fly* – визначений у класі.

```
Eagle new fly
>>> 'Flying and flying high'
```

Щоб доступитися до перевантажених методів *THighFlyingAbility >> fly* і *TFlyingAbility >> fly*, можна оголосити і використати їхні псевдоніми, як описано в параграфі 11.7.

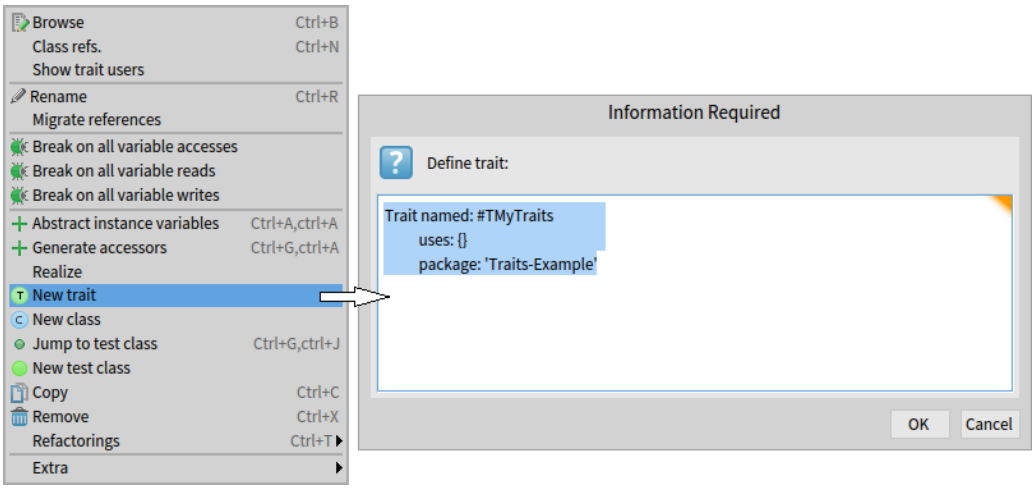

Рис. 11.1. Діалог для створення ознаки

11.11. Ознаки та наслідування

Ознаки визначають методи та змінні екземпляра. Їх треба розглядати як різновид *фрагментів* класу. Тому не можна створити екземпляр ознаки. Ознаку використовують класи, екземпляри яких отримують стан і поведінку ознаки.

Крім того, ознака не призначена для оголошення підкласів. Наслідування ознаки не спрацює. Проте ознаку можуть повторно використати інші ознаки. Кажуть, що її можна скласти з інших ознак.

Отже, ми отримуємо світ, де:

- класи можна розширювати шляхом наслідування. Клас може використовувати ознаки;
- ознаки повторно використовують класи й інші ознаки. Ознаки не наслідують, а тільки компонують.

Якщо два класи, які використовують різні ознаки, що визначають метод з однаковою назвою, як-от *fly* у нашому прикладі, перебувають у відношенні наслідування, пошук методу працює нормально: він працює так, ніби ознаки не існують, а методи визначені безпосередньо в класах.

Уявімо, що *Bird* використовує *TFlyingAbility*, *Eagle* використовує *THighFlyingAbility* та наслідує *Bird*. Екземпляр *Eagle* виконає метод *fly* з *THighFlyingAbility* або локально перевизначений в класі, якщо він там є.

11.12. Про що не було сказано[12]

Окремі важливі моменти ненавмисно випали з пояснень. Поговоримо про них зараз.

Створення ознаки

У розділі було доволі прикладів. Сподіваємося, ви вже здогадалися, як оголошувати ознаки, щоб випробувати їхнє використання. Ви могли б скористатися одним з таких способів.

Оглядач класів можна використовувати для оголошення і нових класів, і нових ознак. За замовчуванням Оглядач пропонує шаблон оголошення класу. Щоб отримати шаблон ознаки, скористайтеся командою «*New trait*» контекстного меню панелі класів Оглядача (див. рис. 11.1). Вона відкриває окреме вікно, в якому можна уточнити деталі нової ознаки. Після натискання на кнопку «*OK*» ознака буде збережена і відкомпільована.

Зверніть увагу на різні піктограми класу та ознаки в контекстному меню. Такі ж Оглядач відображає на панелі класів (імена ознак з'являються тут) і панелі методів.

Якщо ви розпочали роботу зі створення нового пакета класів, то відкрити контекстне меню на порожній панелі класів не вдасться. Тоді можна вручну замінити у вікні редактора коду Оглядача шаблон оголошення класу на оголошення ознаки та зберегти його.

Робоче вікно також може стати в пригоді. Адже це – Pharo! Тут усе можна виконати програмно. Наберіть у вікні, наприклад, «*Trait named: #TMyTraits uses: {} package: 'Traits-Example'*» і виконайте командою «*Do it*» – ви побачите в Оглядачі, що і пакет створено, й ознаку в ньому.

Екземпляри ознаки

Повторимо ще раз: ознаки – не класи, створювати екземпляри ознак не можна. Ви можете спробувати виконати, наприклад, «*TGreetable new*». Тоді у відповідь отримаєте вікно з повідомленням про помилку «*Traits should not be instantiated!*» і роз'яснення, що ознаки не наслідують *Object*, тому не розуміють повідомлень до нього.

Ознаки – дуже специфічні об'єкти. Їх варто трактувати як будівельні блоки для конструювання класів. Перенесіть подумки оголошення методу з ознаки до класу, який її використовує, і все стане на свої місця: і значення *self*, і можливість викликати методи класу, і перевантаження методів. Нагадаємо також, що метод з ознаки можуть запозичити кілька користувачів – класів і інших ознак. Зміна визначення такого методу вплине на кожного користувача.

[12] Цей параграф написав перекладач книги.

Приклад наслідування

Пояснимо співвідношення ознак і наслідування на прикладі. Для цього нагадаємо деякі з оголошених раніше ознак і класів.

```
    Trait named: #TFlyingAbility
       uses: {}
       package: 'Traits-Example'

TFlyingAbility >> fly
    ^ 'I''m flying!'

    Trait named: #THighFlyingAbility
       instanceVariableNames: ''
       package: 'Traits-Example'

THighFlyingAbility >> fly
    ^ 'I''m flying high'

    Object subclass: #Bird
       uses: TFlyingAbility
       instanceVariableNames: ''
       classVariableNames: ''
       package: 'Traits-Example'
```

Потім ми оголошували клас *Duck*, підклас *Object*. Але логічно було б оголосити його підкласом *Bird*! Так і зробимо.

```
    Bird subclass: #Duck
       uses: TSpeakingAbility
       instanceVariableNames: ''
       classVariableNames: ''
       package: 'Traits-Example'
```

Як і раніше, качка вміє і літати, і говорити.

```
    | d |
    d := Duck new.
    d speak
    >>> 'I''m speaking!'
    d fly
    >>> 'I''m flying!'
```

Для польотів клас *Duck* використовує успадкований метод *fly*, запозичений класом *Bird* з ознаки *TFlyingAbility*. «Будівельний блок» ознаки став частиною *Bird*, тому все працює.

Продовжимо експерименти з наслідуванням і змінимо оголошення класу *Eagle*.

```
    Bird subclass: #Eagle
       uses: THighFlyingAbility
       instanceVariableNames: ''
       classVariableNames: ''
       package: 'Traits-Example'
```

Раніше ми отримали конфлікт однойменних методів *fly*, отриманих класом *Eagle* з ознак *THighFlyingAbility* і *TFlyingAbility*. Тепер цього не відбудеться. Клас *Eagle* отримає

метод з *THighFlyingAbility* і так перевантажить метод надкласу. Вибір однозначний – у класі *Eagle* доступний тільки *THighFlyingAbility >> fly*.

Термінологія

Слово *Trait* використовують у C++, коли йдеться про використання шаблонів і характеристики шаблонів. Нехай зовнішня схожість не вводить вас в оману: ознаки Pharo не мають нічого спільного з характеристиками C++. За призначенням ознаки найближчі до підмішаних класів у Python.

11.13. Підсумок розділу

Ознаки – це групи методів і станів, які можна повторно використовувати різним користувачам (класам або ознакам), підтримуючи різновид множинного наслідування в контексті мови з простим наслідуванням. Клас або ознака може використовувати кілька ознак. Ознаки можуть визначати змінні екземпляра та методи. Коли ознаки, які використовуються в класі, визначають метод з однаковою назвою, це призводить до конфлікту.

Конфлікт можна вирішити двома способами:

- користувач (клас або ознака) може перевизначити конфліктуючий метод: метод класу має вищий пріоритет, ніж метод ознаки;
- користувач може вилучити конфліктуючий метод.

Перевантажені методи можна викликати за допомогою псевдонімів, які визначають у частині оголошення використання ознаки.

Не можна створювати екземпляри ознак.

Ознаки, використані для побудови класу, доступні в його підкласах.

Розділ 12
SUnit – модульне тестування у Pharo

SUnit – це невеликий але потужний інструмент, що підтримує створення та виконання тестів. Як можна здогадатися з його назви, SUnit розроблено для *модульного тестування* (для unit-тестів), але, насправді, його можна з успіхом використовувати і для інтеграційного, і для функціонального тестування. Початкову версію SUnit розробив Кент Бек (Kent Beck), її поступово доповнили Йозеф Пелріне (Joseph Pelrine) та багато інших розробників. Платформа SUnit материнська для інших xUnit платформ.

Розділ спеціально написано коротким, щоб переконати читача, що тестування просте. Детальніший опис SUnit і різних підходів до тестування можна прочитати в книзі «*Testing in Pharo*», доступній за адресою http://books.pharo.org.

У цьому розділі почнемо з обговорення того, чому тестуємо код, і що робить тести надійними. Далі продемонструємо низку невеликих прикладів як користуватися SUnit. Зазначимо, що в розділі описано версію SUnit3.3, яку тепер використовують у Pharo.

12.1. Вступ

Інтерес до тестування і розробки програм через тестування (Test Driven Development, TDD) не обмежується середовищем Pharo. Автоматизоване тестування стало невід'ємною ознакою *технології гнучкої розробки програмного забезпечення* (Agile software development), і кожен програміст, який турбується про підвищення якості свого програмного забезпечення, мав би взяти його на озброєння. Справді, розробники багатьма мовами програмування оцінили важливість модульного тестування, і середовища *xUnit* тепер існують для всіх мов програмування.

Ні тестування, ні побудова наборів тестів не є чимось новим – всі знають, що тестування є хорошим способом виявлення помилок. Технологія екстремального програмування (eXtreme Programming) піднесла тестування до рангу однієї з головних практик, наголошуючи на важливості *автоматизованих тестів*, і так перетворила тестування з рутини, яку так не люблять програмісти, на веселе і продуктивне заняття.

SUnit важливий, бо дає змогу писати *виконувані* тести, що містять визначення правильного результату, тому придатні для самоперевірки. Він допомагає також об'єднувати тести в групи, визначати контекст їхнього виконання й автоматично запускати групи тестів. За допомогою SUnit ви зможете за невеликий час створити вичерпний набір тестів для будь-якої мети, тому ми закликаємо вас замість покрокового тестування фрагментів коду в робочому вікні використовувати SUnit та всі його переваги зберігання і автоматичного виконання тестів.

12.2. Чому тестування важливе

Багато програмістів, на превеликий жаль, вважають, що написання тестів – це даремна трата часу. Адже це *інші* програмісти помиляються, а *вони* ніколи не пишуть з помилками. Багато хто з нас говорив: «*Я б писав тести, якби мав більше часу*». Якщо ви ніколи

не робите помилок, і ваша програма ніколи не змінюватиметься у майбутньому, тоді тестування справді є даремним витрачанням вашого часу. Проте, це, найімовірніше, означає, що ваша програма або тривіальна, або її не використовуєте ні ви, ні будь-хто інший. Про тести потрібно думати як про інвестицію в майбутнє: корисний сьогодні набір хороших тестів стане просто *незамінним* у майбутньому, коли зазнає змін ваша програма чи середовище, в якому вона виконується.

Тести виконують кілька завдань одночасно. Найперше вони слугують документацією тієї функціональності програми, яку покривають. Ба більше, така документація активна: проходження тестів без помилок засвідчує її актуальність. По друге, тести допомагають розробникам переконатися, що щойно зроблені зміни певної частини коду не порушують функціонування решти системи, або локалізувати помилки в протилежному випадку. І нарешті, якщо ви пишете тести одночасно з програмою, або навіть перед нею, то починаєте думати про функціональність, яку проєктуєте, і ймовірно про те, *як це виглядатиме для користувача*, ніж як це реалізувати.

Якщо ви спершу пишете тести, а потім – код, то змушені визначати контекст, у якому виконуватиметься проєктована функціональність, спосіб взаємодії з кодом користувача й очікувані результати. Спробуйте так програмувати, і ви побачите, що ваш код стане кращим.

Ми не можемо всесторонньо протестувати жодного справжнього застосунку. Покриття тестами цілої програми просто неможливе, тому не мало б бути метою тестування. Навіть після застосування досконалого набору тестів окремі помилки можуть закрастися до аплікації і «залягти там на дно», очікуючи слушної нагоди, щоб зруйнувати всю систему. Якщо так справді трапиться, то скористайтеся моментом на свою користь! Як тільки знайдено не покриту тестами помилку, напишіть тест, що мав би її виявляти, запустіть його і переконайтеся, що він завершується невдачею. Тепер ви можете перейти до виправлення помилки. Успішне проходження тесту засвідчить, що ви зробили бажане.

12.3. Що робить тест хорошим?

Уміння писати хороші тести найлегше здобути на практиці. Розглянемо умови, за яких тести матимуть найбільше користі.

- *Тести мають бути повторювані*. У вас має бути змога запускати їх так часто, як би ви хотіли. Щоразу тести повинні повертати ті самі результати.
- *Тести повинні виконуватися без втручання людини*. У вас має бути змога запускати їх без особистого нагляду.
- *Тести мають бути інформативні*. Тест повинен «розповідати історію»: мав би діяти як сценарій, прочитавши який, і ви, і будь-хто інший зрозумів би функціональність цієї частини коду.
- *Тест повинен перевіряти один аспект коду*. Коли тест не проходить, він повинен показати, що неправильно працює щось одне. Справді, якщо тест охоплює кілька аспектів, по-перше, він буде падати частіше, а по-друге, це змусить розробника під час виправлення брати до уваги більший набір параметрів.
- *Тести мають бути стабільними* і змінюватися рідше ніж код, який вони перевіряють, адже ніхто не хотів би переписувати весь набір тестів після кожної зміни

в програмі. Єдиний спосіб досягнення такої властивості – писати тести, що взаємодіють з тестованим класом через його відкритий інтерфейс.

Як наслідок кількість тестів мала б бути приблизно пропорційна кількості тестованих функцій, методів. Зміна однієї властивості системи не має «завалювати» всі тести, а лише обмежену кількість з них. Це важливо, бо невиконання сотні тестів набагато серйозніше попередження, ніж невиконання десяти. Проте не завжди вдається досягти такого ідеалу, зокрема, якщо зміни зачіпають ініціалізацію об'єкта або налаштування тестів, то ймовірно всі тести зазнають невдачі.

12.4. SUnit крок за кроком

Написання тестів само собою не складне. Отож напишемо свій перший тест і продемонструємо переваги SUnit. Використаємо приклад тестування класу *Set*.

Виконаємо такі кроки:

- визначимо клас для групування тестів і використання можливостей SUnit;
- визначимо методи-тести;
- використаємо стандартні методи для перевірки очікуваних результатів;
- виконаємо тести.

Пишіть код і виконуйте тести по ходу читання.

12.5. Крок 1. Створіть клас тестів

Найперше потрібно створити новий підклас класу *TestCase*. Назвемо його *MyExampleSetTest* і додамо дві змінні екземпляра класу. Тоді новий клас матиме такий вигляд:

```
TestCase subclass: #MyExampleSetTest
   instanceVariableNames: 'full empty'
   classVariableNames: ''
   category: 'MySetTest'
```

Використаємо клас *MyExampleSetTest*, щоб об'єднати всі тести, які стосуються класу *Set*. Він визначає контекст їхнього виконання. Тут контекст описано двома змінними екземпляра *full* і *empty*, що міститимуть відповідно повну і порожню множини.

Ім'я класу тестів може бути довільним, але за домовленістю воно закінчується на *Test*. Якщо ви визначите клас *Pattern* і назвете відповідний клас тестів *PatternTest*, то в Оглядачі класів вони розташуються поруч в алфавітному порядку (у припущенні, що обидва класи належать до того самого пакета). Усі класи тестів *обов'язково* мають бути підкласами *TestCase*.

12.6. Крок 2. Налаштуйте контекст виконання тестів

Метод *TestCase >> setUp* визначає контекст, у якому працюватимуть тести. Він трохи схожий на метод ініціалізації. Метод *setUp* виконується автоматично перед викликом кожного методу, оголошеного в класі тестів.

Визначимо метод екземпляра *setUp*, як описано нижче, щоб змінна *empty* містила порожню множину, а змінна *full* – множину з двома елементами.

```
MyExampleSetTest >> setUp
  empty := Set new.
  full := Set with: 5 with: 6
```

На жаргоні тестування контекст тесту називають *фікстурою*.

12.7. Крок 3. Напишіть кілька методів тестування

Створимо кілька тестів, визначивши методи в класі *MyExampleSetTest*. Кожен метод представляє один тест. Ім'я методу має розпочинатись словом «*test*», щоб SUnit міг збирати їх в набори тестів. Тестовий метод не приймає аргументів.

Визначимо такі тестові методи. Перший називається *testIncludes* і перевіряє метод *includes:* класу *Set*. Тест говорить, що надсилання повідомлення «*includes: 5*» множині, яка містить *5*, мало б повернути *true*. Зрозуміло, що він покладається на той факт, що метод *setUp* уже виконано.

```
MyExampleSetTest >> testIncludes
  self assert: (full includes: 5).
  self assert: (full includes: 6)
```

Другий тест називається *testOccurrences*. Він стверджує, що кількість входжень елемента *5* в множину *full* дорівнює одиниці навіть після того, як ми додамо ще один елемент *5* до цієї множини.

```
MyExampleSetTest >> testOccurrences
  self assert: (empty occurrencesOf: 5) equals: 0.
  self assert: (full occurrencesOf: 5) equals: 1.
  full add: 5.
  self assert: (full occurrencesOf: 5) equals: 1
```

І нарешті ми переконаємося, що множина *full* більше не містить елемент *5* після того, як ми його вилучили.

```
MyExampleSetTest >> testRemove
  full remove: 5.
  self assert: (full includes: 6).
  self deny: (full includes: 5)
```

Використовуйте метод *TestCase >> deny:*, для підтвердження того, що метод повертає хибу. Вираз «*aTest deny: anExpression*» означає те саме, що й «*aTest assert: anExpression not*», але виглядає набагато зрозуміліше.

У браузері класів ліворуч біля імені класу тестів і біля імені кожного тестового методу можна бачити піктограму – сірий кружечок. Сірий колір свідчить про невизначений статус тестів: поки що результат перевірки невідомий. Після запуску тестів колір піктограми зміниться.

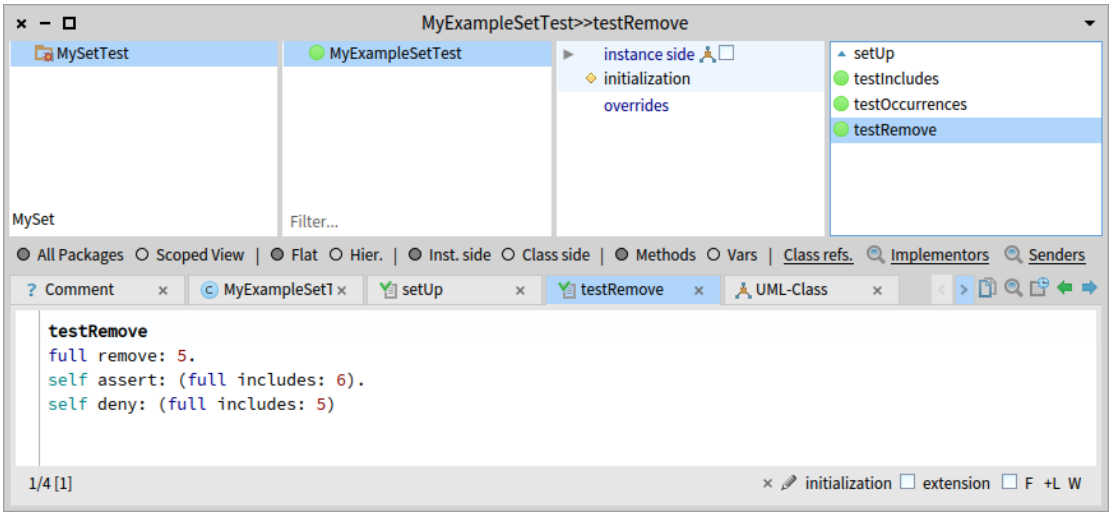

Рис. 12.1. Запуск модульних тестів з Оглядача класів

12.8. Крок 4. Запустіть тести

Найлегше запустити тести на виконання безпосередньо з Оглядача класів. Клацніть контекстно на імені пакета, класу чи окремого методу і виберіть з меню команду «*Run tests (Cmd+T)*», або просто клацніть на піктограмі класу чи методу. Колір піктограми методу зміниться на зелений або жовтий, залежно від того, завершився тест успіхом чи ні. Якщо всі тести успішні, то піктограма класу також стане зеленою, а в протилежному випадку – жовтою (див. рис. 12.1).

Запустити тести можна також за допомогою Менеджера тестів SUnit, який відкривають командою *World > Browse > Test Runner*. Тут ви можете вибрати для запуску кілька наборів тестів і отримати детальний звіт про їхнє виконання.

Відкрийте Менеджер тестів, позначте пакет *MySetTest* і натисніть кнопку ***Run Selected***.

Можна також запустити окремий тест програмно, виконавши такий фрагмент коду «*MyExampleSetTest run: #testRemove*», наприклад, за допомогою команди «*Print it*» у Робочому вікні. Відповіддю буде звичайний звіт про проходження тесту.

Дехто вставляє в тестові методи виконувані коментарі, як показано нижче, щоб можна було запустити тест командою «*Do it*» в Оглядачі.

```
MyExampleSetTest >> testRemove
  "self run: #testRemove"
  full remove: 5.
  self assert: (full includes: 6).
  self deny: (full includes: 5)
```

Зробіть навмисно помилку в методі *MyExampleSetTest >> testRemove*, наприклад, замініть 6 на 7 і запустіть тест ще раз – він мав би завершитись невдачею. Імена тестів, що не пройшли, з'являються в правій частині вікна Менеджера тестів. Щоб перейти до налагодження котрогось з них і побачити, що спричинило помилку, клацніть на імені тесту. Ще один спосіб розпочати налагодження – виконати один з таких фрагментів коду.

```
(MyExampleSetTest selector: #testRemove) debug

MyExampleSetTest debug: #testRemove
```

12.9. Крок 5. Потрактуйте результати тестування

Метод *assert:*, визначений в класі *TestAsserter*. Це надклас *TestCase*, тому *TestCase* та його підкласи відповідальні за всі способи перевірки результатів тестування. Метод *assert:* приймає один аргумент логічного типу. Зазвичай це результат обчислення виразу, який перевіряють. Якщо він дорівнює *true*, то тест завершується успіхом, а в протилежному випадку – невдачею. Вираз з *assert:* називають твердженням тесту.

Насправді є три можливих результати виконання тесту: *завершився успіхом або пройшов*, *завершився невдачею або упав*, *спричинив помилку*.

- **Пройшов**. Ми сподіваємося, що всі твердження в тесті будуть істинні, і він пройде. Якщо всі тести набору проходять, верхня права панель вікна Менеджера тестів забарвлюється зеленим кольором. (Якщо ви запускаєте тести з Оглядача класів, то про їхнє завершення сигналізує поява інформаційного вікна – зеленого, якщо тести пройшли).
- **Упав**. Звичайна річ, коли одне з тверджень тесту виявляється хибним, і тест не проходить. Тести, що не пройшли, забарвлюють верхню праву панель вікна Менеджера у жовтий колір, а їхні імена відображаються в середній правій панелі.
- **Помилка**. В інших випадках трапляються помилки на етапі виконання коду тестового методу, наприклад, «*message not understood*» (об'єкт не знайшов методу опрацювання отриманого повідомлення), або «*index out of bounds*» (індекс вийшов за допустимі межі), або ще якісь. Якщо трапиться помилка, то перевірки в тестовому методі не будуть виконані до кінця, і ми не знатимемо, проходить тест, чи ні, але буде зрозуміло, що щось точно є неправильно. Про тести з помилками в коді сигналізує червоний колір, а їхні імена потрапляють у праву нижню панель вікна Менеджера.

Поекспериментуйте зі створеними раніше тестами так, щоб спричинити виникнення і помилок, і відмов.

12.10. Використання assert:equals:

У випадку відмови тесту повідомлення *assert:equals:* надає кращу діагностику ніж звичайне *assert:*. Наприклад, наведені нижче тести еквівалентні, проте другий з них повідомлятиме очікуване в тесті значення. Це полегшує розуміння причин відмови. У прикладах припускається, що *aDateAndTime* змінна екземпляра тестованого класу.

```
testAsDate
   self assert: aDateAndTime asDate =
      ('February 29, 2004' asDate translateTo: 2 hours).
```

```
testAsDate
   self assert: aDateAndTime asDate
   equals: ('February 29, 2004' asDate translateTo: 2 hours).
```

12.11. Як пропустити тест

Якщо під час розробки програми у вас виникне потреба призупинити перевірку окремого тесту, то замість його вилучення з набору чи перейменування краще пропустити

його. Для цього достатньо надіслати повідомлення *skip* екземплярові класу тестів. У прикладі нижче його використано для визначення умовного тесту.

```
OCCompiledMethodIntegrityTest >> testPragmas
  | newCompiledMethod originalCompiledMethod |
  (Smalltalk globals hasClassNamed: #Compiler) ifFalse: [ ^ self skip ].
   ...
```

Це зручно, автоматично запускається велика кількість тестів, і важливо отримати звіт про їхнє успішне виконання.

12.12. Перевірка виникнення винятків

SUnit надає два додаткові важливі методи *TestAsserter >> should:raise:* та *TestAsserter >> shouldnt:raise:* для тестування виникнення винятків.

Наприклад, ви можете використати «*self should: aBlock raise: anException*», щоб переконатися, чи виконання блока *aBlock* запускає виняток *anException*. Використання методу *should:raise:* демонструє тестовий метод нижче.

```
MyExampleSetTest >> testIllegal
  self should: [ empty at: 5 ] raise: Error.
  self should: [ empty at: 5 put: #zork ] raise: Error
```

Спробуйте запустити цей тест. Пам'ятайте, що першим аргументом повідомлень *should:raise:* та *shouldnt:raise:* мав би бути блок, який містить вираз до виконання.

12.13. Програмний запуск тестів

Зазвичай тести запускають за допомогою Менеджера тестів або Оглядача класів. Програвач можна відкрити командою головного меню або програмно, виконавши «*TestRunner open*».

Запуск одного тесту

Як вже було сказано, окремий тест можна запустити програмно.

```
MyExampleSetTest run: #testRemove
>>> 1 ran, 1 passed, 0 skipped, 0 expected failures, 0 failures,
    0 errors, 0 passed unexpected
```

Запуск усіх тестів тестового класу

Кожен підклас класу *TestCase* у відповідь на повідомлення *suite* повертає набір тестів, який містить усі методи цього підкласу, чиї імена починаються на «*test*». Щоб запустити на виконання цей набір тестів, йому надсилають повідомлення *run*. Наприклад, як показано нижче.

```
MyExampleSetTest suite run
>>> 4 ran, 4 passed, 0 skipped, 0 expected failures, 0 failures,
    0 errors, 0 passed unexpected
```

12.14. Підсумок розділу

Пояснено, чому тести є важливою інвестицією в майбутнє програмного коду. Описано покроково як визначити кілька тестів для класу *Set*.

- Щоб максимізувати свій потенціал, модульні тести мають бути швидкими, повторюваними, незалежними від будь-якої прямої взаємодії з людиною та охоплювати одну функціональну одиницю.
- Тести для класу, що називається *MyClass*, мають належати до класу *MyClassTest*, оголошеного підкласом *TestCase*.
- Контекст виконання тестів налаштовують методом *setUp*.
- Назва кожного тестового методу має розпочинатися словом *test*.
- Для запису тверджень тесту використовують методи класу *TestCase* – *assert:*, *deny:* та інші.
- Запускайте тести!

Кілька методологій розробки програмного забезпечення, такі як екстремальне програмування та розробка, керована тестуванням (TDD), рекомендують писати тести перед написанням коду. Може видатися, що це суперечить глибинним інстинктам розробників програмного забезпечення. Все, що можна сказати: спробуйте! З'ясувалося, що написання тестів перед програмою допомагає дізнатися, що саме потрібно кодувати, допомагає зрозуміти, коли робота закінчена, і допомагає концептуалізувати функціональність класу та розробити його інтерфейс. Крім того, розробка на основі тестування заохочує просуватися швидко, тому що не страшно забути щось важливе.

Розділ 13
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*.

13.1. Object

На всі випадки життя коренем ієрархії наслідування є *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.

13.2. 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.

13.3. 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"
```

13.4. 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
```

13.5. 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
```

13.6. 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.

13.7. 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.

13.8. 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.

13.9. 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*[13]. 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.

[13] Проєктування за контрактом – Вікіпедія (wikipedia.org)

13.10. 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).

13.11. 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 13.1. Загальний *initialize* – метод зачіпка

```
ProtoObject >> initialize
  "Підкласи мали б перевизначити цей метод, щоб ініціалізувати створені
   екземпляри"
```

Лістинг 13.2. 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
```

13.12. 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.

13.13. 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 *Small-Float64*) 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.

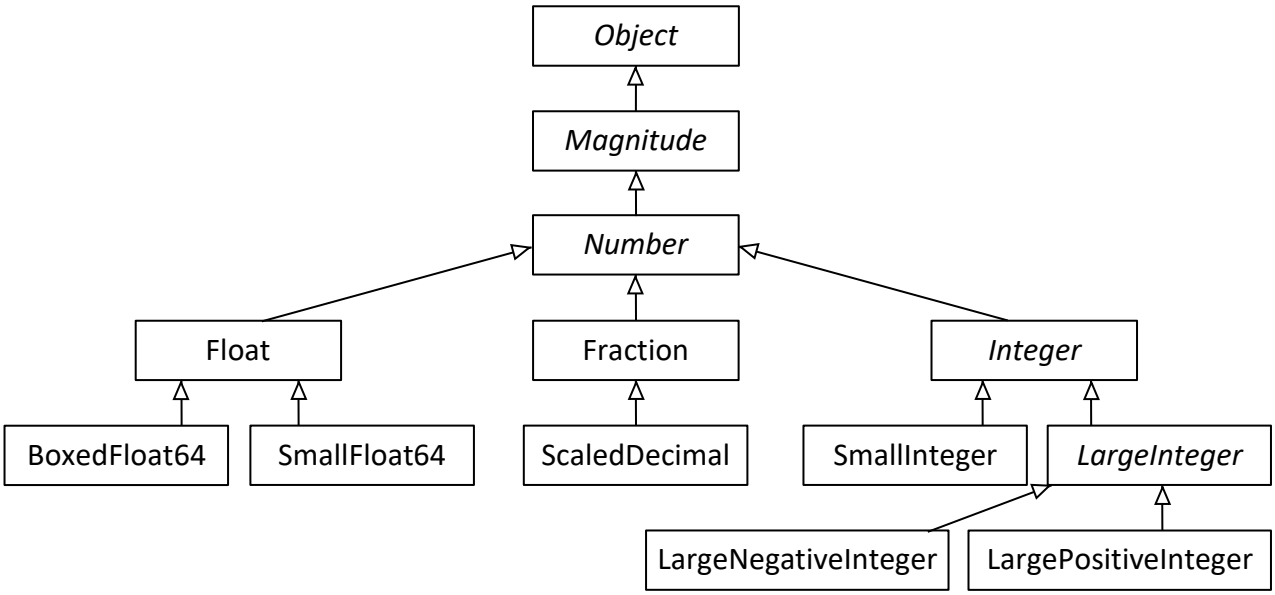

Рис. 13.1. Ієрархія класів чисел

13.14. 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
```

13.15. 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"
```

13.16. 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: *e*, *infinity*, *nan* та *pi*.

```
Float pi
>>> 3.141592653589793

Float infinity
>>> Float infinity

Float infinity isInfinite
>>> true
```

13.17. 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
```

13.18. 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
```

13.19. 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
```

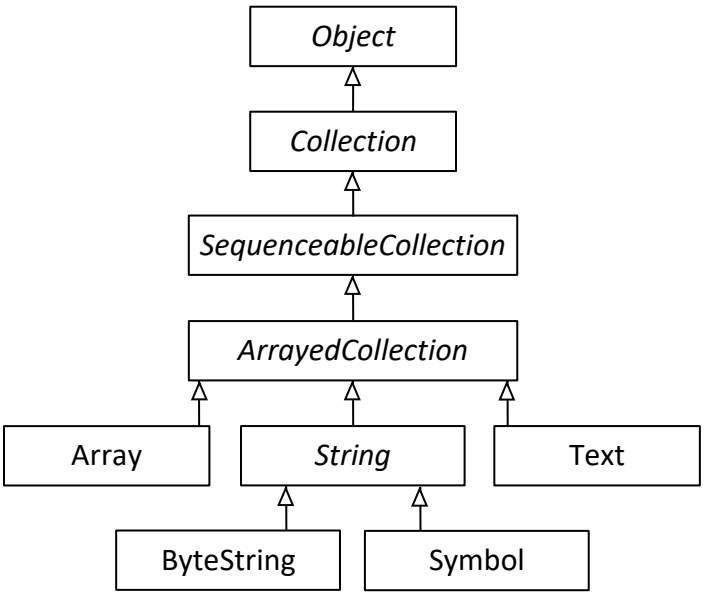

Рис. 13.2. Ієрархія класу *String*

13.20. 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[14].

```
'*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.

[14] glob (programming) – Wikipedia

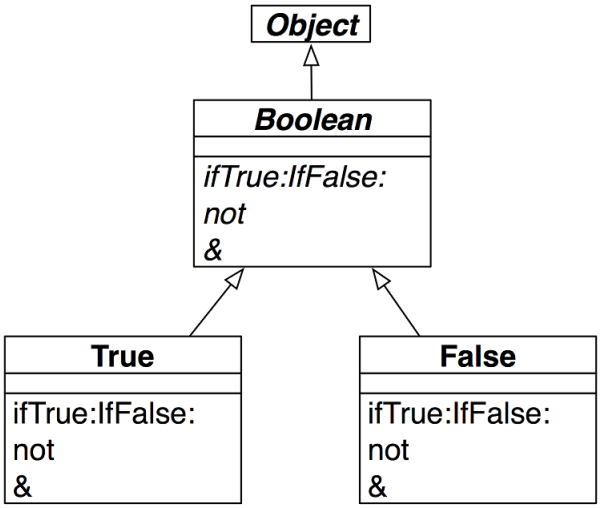

Рис. 13.3. Ієрархія класу *Boolean*

13.21. 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.

13.22. 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.

Розділ 14
Колекції

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

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

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

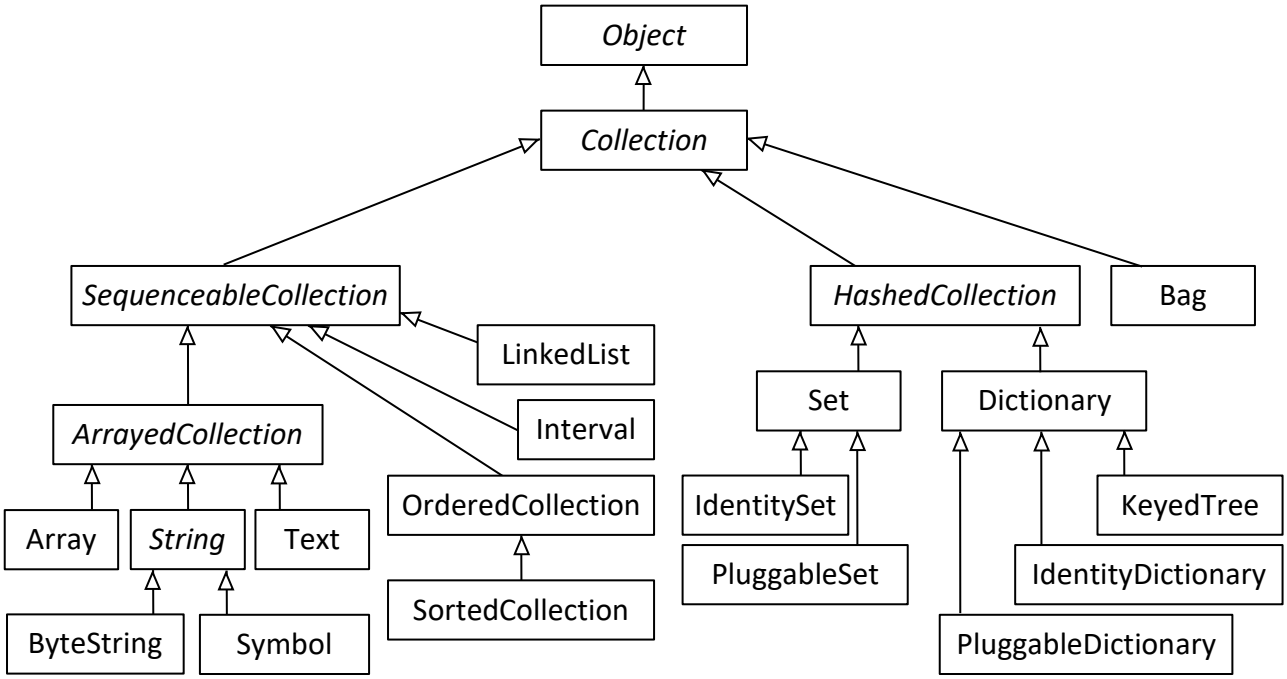

Рис. 14.1. Ієрархія класів чисел

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

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

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

14.1. Функції вищого порядку

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

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

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

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

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

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

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

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

14.2. Різноманіття колекцій

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

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

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

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

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

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

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

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

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

14.3. Реалізація колекцій

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

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

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

14.4. Приклади головних класів

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

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

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

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

14.5. Загальний протокол створення

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

- *new* створює порожню колекцію, підходить для колекцій змінного розміру.

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

Розглянемо приклади.

```
Array with: 1
>>> #(1)

Array with: 1 with: 2
>>> #(1 2)

Array with: 1 with: 2 with: 3
>>> #(1 2 3)

Array with: 1 with: 2 with: 3 with: 4
>>> #(1 2 3 4)

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

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

```
Array withAll: #(7 3 1 3)
>>> #(7 3 1 3)
```

```
OrderedCollection withAll: #(7 3 1 3)
>>> an OrderedCollection(7 3 1 3)
```

```
SortedCollection withAll: #(7 3 1 3)
>>> a SortedCollection(1 3 3 7)
```

```
Set withAll: #(7 3 1 3)
>>> a Set(7 1 3)
```

```
Bag withAll: #(7 3 1 3)
>>> a Bag(7 1 3 3)
```

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

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

```
(1 to: 5) asOrderedCollection addAll: #(6 7 8); yourself
>>> an OrderedCollection(1 2 3 4 5 6 7 8)
```

14.6. Array

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

- *at: anInteger* повертає елемент з індексом *anInteger*.
- *at: anInteger put: anObject* розміщує *anObject* в масиві за індексом *anInteger*.

Масиви мають фіксований розмір, тому не можна додати або видалити елементи з кінця масиву. Наступний код створює масив розміру 5, задає значення першим трьом елементам і повертає перший елемент.

```
| anArray |
anArray := Array new: 5.
anArray at: 1 put: 4.
anArray at: 2 put: 3/2.
anArray at: 3 put: 'Hello'.
anArray at: 1
>>> 4
```

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

Створення за допомогою *new:*

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

Важливо Значенням кожного елемента буде *nil*.

Створення за допомогою *with:*

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

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

Створення літерала #( )

Вираз #( ) задає літерал масиву зі статичними (або *літеральними*) елементами, які мають бути відомі на етапі компіляції ще перед виконанням. Код нижче створить масив

розміру 2, в якому перший елемент задано літералом цілого числа *1*, а другий – літералом рядка *'here'*.

```
#(1 'here') size
>>> 2
```

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

```
#(1+2)
>>> #(1 #+ 2)
```

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

Динамічне створення { . }

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

```
{ 1 + 2 }
>>> #(3)

{(1/2) asFloat} at: 1
>>> 0.5

{10 atRandom . 1/3} at: 2
>>> (1/3)
```

Доступ до елементів

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

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

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

14.7. OrderedCollection

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

```
| ordCol |
ordCol := OrderedCollection new.
ordCol add: 'Seaside'; add: 'SmalltalkHub'; addFirst: 'GitHub'.
ordCol
>>> an OrderedCollection('GitHub' 'Seaside' 'SmalltalkHub')
```

Вилучення елементів

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

```
ordCol add: 'GitHub'.
ordCol remove: 'GitHub'.
ordCol
>>> an OrderedCollection('Seaside' 'SmalltalkHub' 'GitHub')
```

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

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

Перетворення

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

```
#(1 2 3) asOrderedCollection
>>> an OrderedCollection(1 2 3)

'hello' asOrderedCollection
>>> an OrderedCollection($h $e $l $l $o)

(6 to: 18 by: 3) asOrderedCollection
>>> an OrderedCollection(6 9 12 15 18)
```

14.8. Interval

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

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

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

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

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

```
(Interval from: 1 to: 100 by: 0.5) size
>>> 199

(1 to: 100 by: 0.5) at: 198
>>> 99.5

(1/2 to: 54/7 by: 1/3) last
>>> (15/2)
```

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

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

14.9. Dictionary

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

```
| colors |
colors := Dictionary new.
colors at: #yellow put: Color yellow.
colors at: #blue put: Color blue.
colors at: #red put: Color red.
colors at: #yellow
>>> Color yellow

colors keys
>>> #(#red #blue #yellow)

colors values
>>> {Color red. Color blue. Color yellow}
```

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

Спробуйте проінспектувати створений в попередньому прикладі словник. Легко бачити, що екземпляр *Dictionary* реалізовано як колекцію пар (*ключ*; *значення*) – екземплярів класу *Association*, створених за допомогою повідомлення «–>»: *key–>value*. Тому можна легко створити словник з колекції асоціацій, або перетворити словник на масив пар.

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

14.10. IdentityDictionary

Тоді як *Dictionary* використовує результат повідомлень = і *hash* для визначення того, чи два ключі рівні, *IdentityDictionary* використовує перевірку на ідентичність ключів – повідомлення ==. Це означає що, два ключі вважаються рівними *тільки* тоді, коли вони є тим самим об'єктом.

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

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

trouble at: b
>>> 'b'

trouble at: 'foobar'
>>> 'a'
```

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

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

Приклад *IdentityDictionary*

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

```
Smalltalk globals keys collect: [ :each | each class ] as: Set
>>> a Set(ByteSymbol)
```

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

14.11. Set

Клас *Set* – це колекція, яка нагадує математичну множину, тобто це колекція, яка не містить повторюваних елементів, і ці елементи розміщені без жодного порядку. У *Set* елементи можна додавати за допомогою повідомлення *add:*. Доступитися до них за допомогою повідомлення *at:* неможливо. Об'єкти, додані у *Set*, мають реалізувати методи *hash* і =.

```
| s |
s := Set new.
s add: 10/2; add: 4; add: 5.
s size
>>> 2
```

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

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

asSet пропонує зручний спосіб вилучення дублікатів з колекції:

```
{Color black. Color white. (Color red + Color blue + Color green) }
   asSet size
>>> 2
```

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

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

```
{Color black. Color white. (Color red + Color blue + Color green) }
   asBag size
>>> 3
```

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

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

```
'hello' intersection: 'there'
>>> 'eh'
```

```
#Pharo includes: $a
>>> true
```

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

14.12. SortedCollection

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

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

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

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

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

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

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

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

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

Правильна відповідь – використати один з методів *String class >> newFrom:*, *String class >> withAll:* або *Object >> as:*.

```
'hello' asSortedCollection as: String
>>> 'ehllo'
```

```
String newFrom: 'hello' asSortedCollection
>>> 'ehllo'
```

```
String withAll: 'hello' asSortedCollection
>>> 'ehllo'
```

У *SortedCollection* можна зберігати елемент різних типів, якщо тільки їх можна порівнювати між собою. Наприклад, можна поєднати числа різних типів: цілі, дійсні, раціональні.

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

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

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

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

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

14.13. Рядки

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

```
'Hello'
>>> 'Hello'

String with: $A
>>> 'A'

String with: $h with: $i with: $!
>>> 'hi!'

String newFrom: #($h $e $l $l $o)
>>> 'hello'
```

Насправді, клас *String* абстрактний. Під час створення екземпляра *String* отримують або *ByteString* з 8-бітними літерами, або *WideString* з 32-бітними. Щоб не ускладнювати

міркування без потреби, зазвичай не звертають уваги на різницю, і говорять лише про екземпляри *String*.

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

```
'об''єкт' at: 3
>>> $'

'об''єкт' at: 4
>>> $є
```

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

```
| s |
s := 'no', ' ', 'worries'.
s
>>> 'no worries'
```

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

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

```
s at: 4 put: $h; at: 5 put: $u.
s
>>> 'no hurries'
```

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

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

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

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

На відміну від методів, описаних вище, метод *copyReplaceAll: oldSubCollection with: newCollection* створює новий рядок. Цікаво, що аргументами тут є підрядки, а не окремі символи, і їхні розміри можуть не збігатися.

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

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

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

Зіставлення рядків

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

```
'Linux *' match: 'Linux mag'
>>> true

'GNU/Linux #ag' match: 'GNU/Linux tag'
>>> true
```

Ширші засоби зіставлення шаблонів доступні у пакеті Regex.

Підрядки

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

```
'alphabet' first
>>> $a

'alphabet' first: 5
>>> 'alpha'

'alphabet' allButFirst: 4
>>> 'abet'

'alphabet' copyFrom: 5 to: 7
>>> 'abe'

'alphabet' copyFrom: 3 to: 3
>>> 'p' "не $p"
```

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

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

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

Предикати

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

```
'Hello' isEmpty
>>> false

'Hello' includes: $l
>>> true

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

'Joe' anySatisfy: [:c | c isLowercase]
>>> true
```

Форматування рядків

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

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

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

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

```
'look-<t>-here' expandMacros
>>> 'look-    -here'

'<1s> is <2s>' expandMacrosWith: 'Pharo' with: 'cool'
>>> 'Pharo is cool'

'<2s> is <1s>' expandMacrosWith: 'Pharo' with: 'cool'
>>> 'cool is Pharo'

'<1p> or <1s>' expandMacrosWith: 'Pharo' with: 'cool'
>>> '''Pharo'' or Pharo'

'<1?Quentin:Thibaut> plays' expandMacrosWith: true
>>> 'Quentin plays'

'<1?Quentin:Thibaut> plays' expandMacrosWith: false
>>> 'Thibaut plays'
```

Деякі допоміжні методи

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

```
'XYZ' asLowercase
>>> 'xyz'

'xyz' asUppercase
>>> 'XYZ'

'hilaire' capitalized
>>> 'Hilaire'

'1.54' asNumber
>>> 1.54

'this sentence is without a doubt far too long' contractTo: 20
>>> 'this sent...too long'
```

asString проти *printString*

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

```
$A printString
>>> '$A'

$A asString
>>> 'A'
```

```
#ASymbol printString
>>> '#ASymbol'

#ASymbol asString
>>> 'ASymbol'
```

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

14.14. Ітератори колекцій

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

Перебір (*do:*)

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

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

Варіанти перебору

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

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

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

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

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

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

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

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

Перебір словників

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

```
| colors |
colors := Dictionary newFrom: { #yellow-> Color yellow.
   #blue-> Color blue. #red-> Color red }.
colors keysDo: [:key | Transcript show: key; cr]. "друкує ключі"
colors valuesDo: [:value | Transcript show: value; cr]. "друкує значення"
colors associationsDo: [:pair |
   Transcript show: pair; cr]. "виводить у консоль асоціації"

" Текст у Transcript: "
red
blue
yellow
Color red
```

```
Color blue
Color yellow
#red->Color red
#blue->Color blue
#yellow->Color yellow
```

14.15. Збір результатів (*collect:*)

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

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

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

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

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

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

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

А тепер порівняємо його з таким кодом:

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

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

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

```
'abc' collect: [ :each | each asciiValue ]
>>> "Error: Improper store into indexable object"
```

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

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

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

```
(1 to: 5) collect: [ :each | each * 2 ]
>>> #(2 4 6 8 10)
```

14.16. Вибір і відхилення елементів

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

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

Повідомлення *reject:* діє навпаки.

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

Відшукання елемента за допомогою *detect:*

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

```
'through' detect: [ :letter | letter isVowel ]
>>> $o
```

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

```
Smalltalk allClasses
   detect: [ :class | '*cobol*' match: class asString ]
   ifNone: [ nil ]
>>> nil
```

Накопичення результатів з *inject:into:*

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

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

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

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

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

```
| factorial |
factorial := [ :n |
   (1 to: n)
      inject: 1
      into: [ :product :each | product * each ] ].
factorial value: 10
>>> 3628800
```

14.17. Інші повідомлення вищого порядку

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

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

```
Smalltalk allClasses count: [:each | 'Collection*' match: each asString ]
>>> 10
```

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

```
| colors |
colors := {Color white . Color yellow . Color blue . Color orange}.
colors includes: Color blue.
>>> true
```

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

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

14.18. Загальна помилка – використання результату *add:*

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

```
| collection |
collection := OrderedCollection new add: 1; add: 2.
collection
>>> 2
```

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

Код нижче видає бажаний результат.

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

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

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

14.19. Загальна помилка – вилучення елемента під час перебору

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

```
| range |
range := (2 to: 20) asOrderedCollection.
range do: [:aNumber |
   aNumber isPrime ifFalse: [ range remove: aNumber ] ].
Range
>>> "Error: #isPrime was sent to nil"
```

Вирішенням цієї проблеми є створення копії колекції перед ітеруванням.

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

14.20. Загальна помилка – перевизначення = без *hash*

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

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

```
Book >> = aBook
  self class = aBook class ifFalse: [^ false].
  ^ title = aBook title and: [ authors = aBook authors]

Book >> hash
  ^ title hash bitXor: authors hash
```

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

14.21. Підсумки розділу

Ієрархія колекцій забезпечує загальний словник для маніпулювання колекціями різних видів.

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

Розділ 15
Потоки

Потоки використовують для перебору послідовностей елементів: послідовних колекцій, файлів, мережевих потоків. Потоки можуть бути або для читання, або для запису, або для обох дій. Читання або запис завжди пов'язані з вказівником потоку. Потоки легко можна перетворити на колекції та навпаки.

15.1. Дві послідовності елементів

Для розуміння потоку добре підходить метафора «двох послідовностей». Потік можна представити у вигляді двох послідовностей елементів: послідовність минулих елементів і послідовність майбутніх. Точка доступу потоку, або його вказівник, розташована на межі між ними. Розуміння цієї моделі важливе, бо всі потокові операції в Pharo покладаються на неї. Тому більшість потокових класів є підкласами *PositionableStream*. Рисунок 15.1 зображає потік, який містить п'ять літер. Він перебуває в початковому стані, тобто, немає ніякого елемента в минулому, вказівник розташовано на початку потоку. Потік можна повернути до такого стану за допомогою повідомлення *reset*, визначеного в *PositionableStream*.

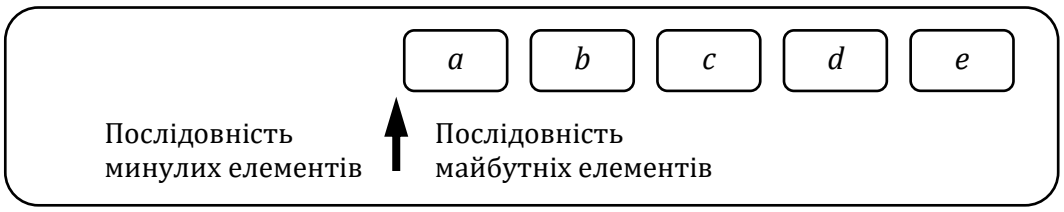

Рис. 15.1. Вказівник потоку розташовано на початку

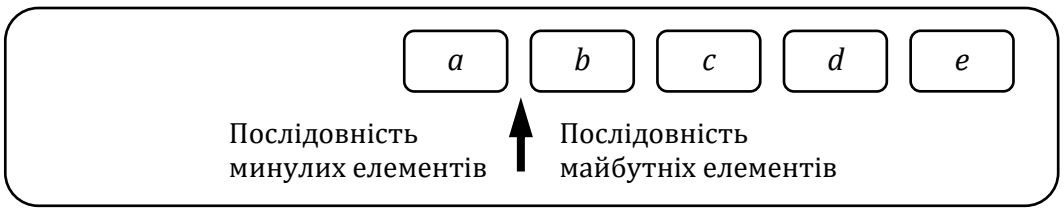

Рис. 15.2. Той самий потік після виконання методу *next*.
Літера *a* тепер у минулому, літери *b*, *c*, *d*, *e* – в майбутньому

«Прочитання елемент потоку» концептуально означає вилучити перший елемент з початку послідовності майбутніх елементів і помістити його в кінець послідовності минулих елементів. Стан потоку після зчитування одного елемента за допомогою повідомлення *next* показано на рис. 15.2. Видно, що вказівник потоку посунувся на один елемент праворуч.

Запис елемента означає заміну першого елемента послідовності майбутніх новим елементом і переміщення його в минуле. На рис. 15.3 зображено стан того ж потоку після запису '*x*' за допомогою повідомлення *nextPut: anElement*. Вказівник знову посунувся на один елемент праворуч.

Рис. 15.3. Той самий потік після запису *x*

15.2. Потоки проти колекцій

Протокол колекції підтримує зберігання, вилучення та перебір елементів колекції, але не дозволяє змішувати ці операції. Наприклад, якщо елементи деякої *OrderedCollection* опрацьовують методом *do:*, то неможливо додати до неї чи вилучити з неї елементи зсередини блоку *do:*. Так само протокол колекцій не дає змоги перебирати дві колекції одночасно, вибирати, по якій колекції просунутися вперед, а по якій ні. Такі алгоритми потребують, щоби індекс обходу або посилання на позицію в колекції підтримувалися поза самою колекцією: саме це і є призначенням *ReadStream*, *WriteStream* та *ReadWriteStream*.

Ці три класи визначені для потокового доступу до колекції. Наприклад, код нижче спочатку накладає потік на інтервал, а опісля зчитує два елементи.

```
| r |
r := ReadStream on: (1 to: 1000).
r next.
>>> 1

r next.
>>> 2

r atEnd.
>>> false
```

Потоки для запису можуть записувати дані в колекцію.

```
| w |
w := WriteStream on: (String new: 5).
w nextPut: $A.
w nextPut: $B; nextPut: $C.
w contents.
>>> 'ABC'
```

Також можна створювати потоки *ReadWriteStream*, які підтримують обидва протоколи: і читання, і запису.

У наступних параграфах ці протоколи описано детальніше.

15.3. Віддавайте перевагу функціям інтерфейсу

Важливо, щоб ви зосередилися на API, наборі повідомлень, які пропонують потокові класи, а не лише на назвах класів. Чому? Тому що Pharo може запропонувати альтернативні реалізації класів або навіть спеціалізовані потоки для опрацювання інших

кодувань, наприклад, двійкових форматів. Такі реалізації, хоча й називатимуться інакше, або навіть не успадковуватимуть базові класи потоків, пропонуватимуть ті самі API. Нарешті, у фрагментах коду цієї глави ми часто посилаємося на класи для створення екземплярів потоку, тоді як у реальному коді ми зазвичай надсилаємо повідомлення (наприклад, *readStream*) екземплярові колекції, щоб накласти на нього потік. Такий підхід робить код більш загальним.

У цьому розділі ми подаємо загальні повідомлення потоку та часто згадуємо кореневий клас, де визначено метод. Однак зауважте, що знати точний клас не так важливо, бо підкласи або інші класи, що надають ті самі API, можуть виконувати власні методи у відповідь на те саме повідомлення.

Потоки справді корисні у роботі з колекціями. Їх можна використовувати для відокремленого в часі читання та запису елементів. Тепер розглянемо можливості потоків для опрацювання колекцій.

15.4. Читання колекцій

Накладання потоку для читання на колекцію по суті надає вказівник на її елемент. Під час читання він автоматично посуватиметься на наступний елемент. Але його можна також перемістити, куди потрібно. Для читання елементів колекції використовують клас *ReadStream*.

У *ReadStream* визначено повідомлення *next* і *next:*. Їх використовують для того, щоб отримати з колекції один або більше елементів.

```
| stream |
stream := ReadStream on: #(1 (a b c) false).
stream next.
>>> 1

stream next.
>>> #(#a #b #c)

stream next.
>>> false
```

```
| stream |
stream := ReadStream on: 'abcdef'.
stream next: 0.
>>> ''

stream next: 1.
>>> 'a'

stream next: 3.
>>> 'bcd'

stream next: 2.
>>> 'ef'
```

15.5. Підглядання

Повідомлення *peek*, визначене у *PositionableStream*, використовують, коли потрібно «підглянути» в потік – довідатися, який елемент наступний в потоці, без переміщення вказівника потоку.

```
| stream negative number |
stream := ReadStream on: '-143'.
"Дивимося на перший елемент потоку, але не використовуємо його."
negative := (stream peek = $-).
negative.
>>> true

"Пропускаємо літеру мінус."
negative ifTrue: [ stream next ].
number := stream upToEnd.
number.
>>> '143'
```

Цей код покладає булевій змінній *negative* значення згідно зі знаком числа в потоці і змінній *number* – абсолютну величину значення цього числа. Визначене в *ReadStream* повідомлення *upToEnd* повертає весь вміст потоку від поточної позиції до кінця і встановлює вказівник потоку на його закінчення. Цей код можна спростити за допомогою повідомлення *PositionableStream >> peekFor:*, яке пересуває вказівник потоку вперед, якщо наступний елемент дорівнює параметру, і залишає його на місці в протилежному випадку.

```
| stream |
stream := '-143' readStream.
(stream peekFor: $-).
>>> true

stream upToEnd
>>> '143'
```

Повідомлення *peekFor:* також повертає булеве значення, яке сигналізує, чи його аргумент дорівнює елементові потоку.

Ви мали б помітити у наведеному прикладі новий спосіб створення потоку: можна просто надіслати повідомлення *readStream* послідовній колекції (як *String* у прикладі), щоб отримати накладений на неї потік читання.

Рис. 15.4. Вказівник потоку розміщено в позиції 2

15.6. Керування вказівником потоку

Існують повідомлення для позиціонування вказівника потоку. За допомогою *PositionableStream >> position:* можна перейти відразу до місця потоку, заданого індексом.

Поточне розташування вказівника можна довідатися за допомогою запиту *position*. Проте треба пам'ятати, що вказівник позиціонується не на елементі, а між елементами. Початкові потоку відповідає індекс 0.

Ви можете отримати потік у стані, зображеному на рис. 15.4, за допомогою наведеного нижче коду.

```
| stream |
stream := 'abcde' readStream.
stream position: 2.
stream peek
>>> $c
```

Щоб помістити вказівник потоку на початок або на закінчення, можна використати повідомлення *reset* або *setToEnd*.

15.7. Пропуск елементів

Повідомлення *skip:* і *skipTo:* використовують, щоб перемістити вказівник потоку вперед від поточної позиції. *skip:* приймає число і пропускає таку кількість елементів потоку від місця розташування. *skipTo:* пропускає всі елементи в потоці доки не знайде елемент, що дорівнює його параметру. Зверніть увагу, що *skipTo:* позиціонує потік після елемента, що збігається.

```
| stream |
stream := 'abcdef' readStream.
stream next
>>> $a
```

Тепер вказівник потоку розташовано відразу після '*a*'.

```
stream skip: 3.
stream position
>>> 4
```

Тепер вказівник після '*d*'.

```
stream skip: -2.
stream position
>>> 2
```

Тепер вказівник після '*b*'.

```
stream reset.
stream position
>>> 0

stream skipTo: $e.
stream next.
```

```
>>> $f
```

Пропуск до заданого елемента позиціонує потік одразу після заданого елемента. Легко бачити, що вказівник потоку було встановлено відразу після літери '*e*'.

```
stream contents
>>> 'abcdef'
```

Повідомлення *contents* завжди повертає копію цілого потоку.

15.8. Предикати

Деякі повідомлення дають змогу перевірити стан потоку: *atEnd* повертає *true* тоді і тільки тоді, коли немає більше елементів, які можна прочитати, тоді як *isEmpty* повертає *true* тоді і тільки тоді, коли в колекції взагалі немає елементів.

Нижче наведено можливу реалізацію алгоритму злиття з використанням *atEnd*, який приймає дві відсортовані колекції та об'єднує їх в нову відсортовану колекцію.

```
| stream1 stream2 result |
stream1 := #(1 4 9 11 12 13) readStream.
stream2 := #(1 2 3 4 5 10 13 14 15) readStream.

"Змінна result міститиме впорядковану колекцію."
result := OrderedCollection new.
[ stream1 atEnd not & stream2 atEnd not ]
   whileTrue: [
      stream1 peek < stream2 peek
         "Менший з елементів вилучаємо з потоку і поміщаємо в результат."
         ifTrue: [ result add: stream1 next ]
         ifFalse: [ result add: stream2 next ] ].

"Один з потоків все ще містить дані. Копіюємо залишок."
result
   addAll: stream1 upToEnd;
   addAll: stream2 upToEnd.

result.
>>> an OrderedCollection(1 1 2 3 4 4 5 9 10 11 12 13 13 14 15)
```

15.9. Запис у колекцію

Ми вже бачили, як читати колекцію, перебираючи кожен її елемент за допомогою *ReadStream*. Тепер навчимося створювати колекції за допомогою *WriteStream*.

Екземпляри *WriteStream* корисні для додавання великої кількості даних у різних місцях колекції. Їх часто використовують для створення рядків, що складаються з постійної та змінної частин, як у наведеному прикладі.

```
| stream |
stream := String new writeStream.
stream
   nextPutAll: 'This image contains: ';
   print: Smalltalk globals allClasses size;
```

```
        nextPutAll: ' classes.';
        cr;
        nextPutAll: 'This is really a lot.'.

    stream contents.
    >>> 'This image contains: 10672 classes.
    This is really a lot.'
```

Таку методику використовують, наприклад, у різних реалізаціях методу *printOn:*. Якщо вас цікавить тільки вміст потоку, то можна використати простіший і ефективніший спосіб створення рядків.

```
    | string |
    string := String streamContents:
        [ :stream |
            stream
                print: #(1 2 3);
                space;
                nextPutAll: 'size';
                space;
                nextPut: $=;
                space;
                print: 3. ].
    string.
    >>> '#(1 2 3) size = 3'
```

Повідомлення *streamContents:*, надіслане класові колекції (тут *String*), створює екземпляр колекції і накладає на неї потік. Потім воно виконує блок, свій аргумент. Після цього *streamContents:* повертає вміст потоку – новостворену колекцію.

Перелічимо методи *WriteStream*, особливо корисні в цьому контексті.

nextPut: повідомлення *nextPut:* додає в потік свій параметр;

nextPutAll: повідомлення *nextPutAll:* приймає колекцію і додає кожен її елемент до потоку;

print: повідомлення *print:* додає до потоку текстове зображення свого параметра.

Є також зручні повідомлення для друку в потік окремих символів, наприклад, *space*, *tab* і *cr* (переведення каретки). Інший корисний метод *ensureASpace* гарантує, що останній символ у потоці пропуск; якщо це не так, то метод додає його.

15.10. Про конкатенацію рядків

Використання *nextPut:* і *nextPutAll:* з *WriteStream* часто найкращий спосіб дописування літер до рядка. Використання оператора конкатенації кома (,) набагато менш ефективне, що демонструють два приклади, які виконують однакове завдання.

```
    [ | temp |
        temp := String new.
        (1 to: 100000)
            do: [:i | temp := temp, i asString, ' ' ] ] timeToRun
    >>> 0:00:01:54.758
```

```
[ | temp |
   temp := WriteStream on: String new.
   (1 to: 100000)
      do: [:i | temp nextPutAll: i asString; space ].
   temp contents ] timeToRun
>>> 0:00:00:00.024
```

Використання потоку може бути набагато ефективнішим ніж оператора конкатенації, бо кома створює новий рядок, який містить конкатенацію приймача і аргументу, то ж вона мусить скопіювати обох. Якщо багаторазово дописувати до того самого приймача, то він щоразу ставатиме довшим і довшим, тому кількість літер, які потрібно копіювати, зростатиме експотенційно. Це також створює багато сміття, яке треба збирати. Використання потоку замість конкатенації є добре відомою оптимізацією.

Насправді, можна використати вже згадане повідомлення *SequenceableCollection >> streamContents:*, щоб зробити це ще простіше.

```
String streamContents: [ :stream |
   (1 to: 100000)
      do: [:i | stream nextPutAll: i asString; space ] ]
```

15.11. Про *printString*

Давайте трохи поговоримо про використання потоку в методах *printOn:*. По суті, метод *Object>>printString* створює потік виведення і передає його як аргумент методу *printOn:*, як подано нижче.

```
Object >> printString
   "Повертає рядок з описом отримувача. За потреби друку без обмежень на
    довжину рядка використовуйте fullPrintString."

   ^ self printStringLimitedTo: 50000
```

```
Object >> printStringLimitedTo: limit
   "Повертає рядок з описом отримувача. За потреби друку без обмежень на
    довжину рядка використовуйте fullPrintString."

   ^self printStringLimitedTo: limit using: [:s | self printOn: s]
```

```
Object >> printStringLimitedTo: limit using: printBlock
   "Повертає рядок з описом отримувача, отриманий у результаті виконання
    printBlock. Переконується, що результат не довший за задане обмеження."

   | limitedString |
   limitedString := String streamContents: printBlock limitedTo: limit.
   limitedString size < limit ifTrue: [^ limitedString].
   ^ limitedString , '...etc...'
```

Видно, що метод *printStringLimitedTo:using:* створює потік і передає його далі.

Якщо в перевизначеному методі *printOn:aStream* у своєму класі надсилати повідомлення *printString* змінним екземпляра свого об'єкта, то створимо ще один потік виведення, а потім скопіюємо його вміст до *aStream*. Нижче наведено приклад.

```
MessageTally >> displayStringOn: aStream
  self displayIdentifierOn: aStream.
  aStream
     nextPutAll: ' (';
     nextPutAll: self tally printString;
     nextPutAll: ')'
```

Тут вираз *self tally printString* запускає такий самий механізм і створює додатковий потік замість того, щоб використати наявний. Це відверто контрпродуктивно. Набагато краще надсилати повідомлення *print:* потокові або *printOn:* змінній екземпляра як у коді нижче.

```
MessageTally >> displayStringOn: aStream
  self displayIdentifierOn: aStream.
  aStream
     nextPutAll: ' (';
     print: self tally;
     nextPutAll: ')'
```

Тепер працює лише потік *aStream*, ніяких додаткових потоків ніхто не створює.

Щоб зрозуміти, як працює метод *print:*, наведемо його оголошення.

```
Stream >> print: anObject
  "Аргумент anObject має надрукувати себе в отримувач."

  anObject printOn: self
```

Інший приклад

Додаткове створення потоку не обмежується методами *printString*. Нижче наведено приклад, знайдений у Pharo (байдуже до класу, в якому знайдено метод), який демонструє таку саму проблему.

```
EpContentStringVisitor >> printProtocol: protocol sourceCode: sourceCode
  ^ String streamContents: [ :stream |
     stream nextPutAll: '"protocol: ';
     nextPutAll: protocol printString;
     nextPut: $"; cr; cr;
     nextPutAll: sourceCode ]
```

Легко бачити, що спочатку створено потік *stream*, а тоді ще один під час виконання виразу «*protocol printString*». Той другий буде знищено після завершення формування рядка, який зображає *protocol*. Кращою реалізацією є зображений нижче варіант.

```
EpContentStringVisitor >> printProtocol: protocol sourceCode: sourceCode
  ^ String streamContents: [ :stream |
     stream nextPutAll: '"protocol: ';
     print: protocol;
     nextPut: $"; cr; cr;
     nextPutAll: sourceCode ]
```

Рис. 15.5. Нова історія порожня. Веб-браузер не показує нічого

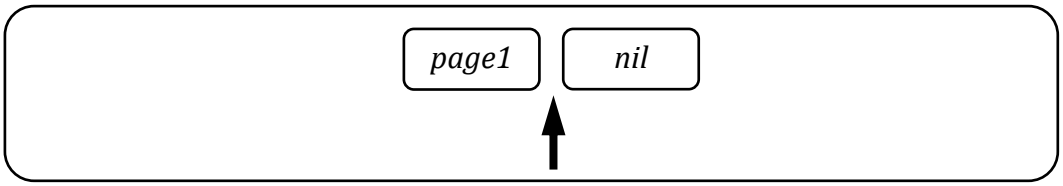

Рис. 15.6. Користувач відкрив першу сторінку

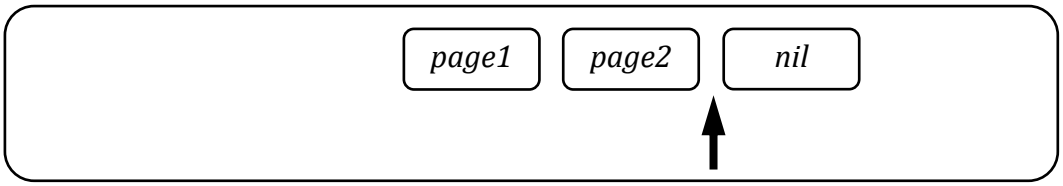

Рис. 15.7. Користувач перейшов за посиланням на другу сторінку

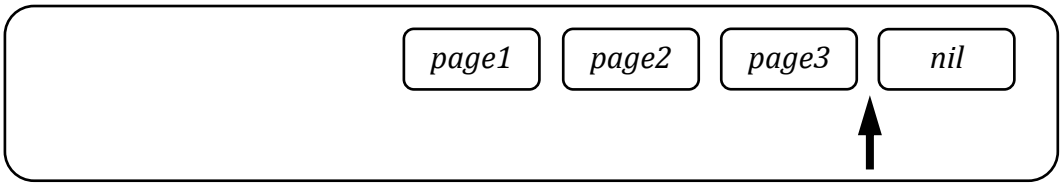

Рис. 15.8. Користувач перейшов за посиланням на третю сторінку

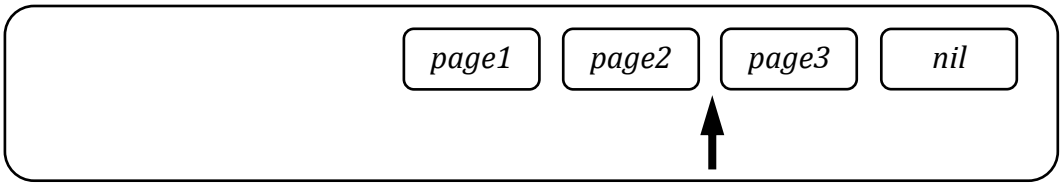

Рис. 15.9. Користувач клацнув кнопку «Назад». Тепер він знову бачить другу сторінку

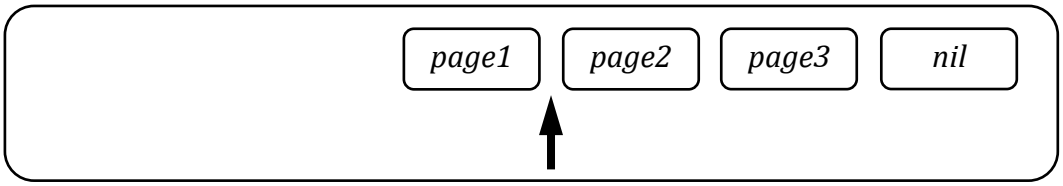

Рис. 15.10. Користувач знову клацнув кнопку «Назад». Відображено першу сторінку

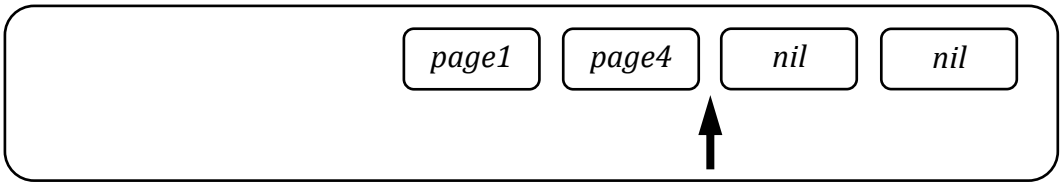

Рис. 15.11. З першої сторінки користувач перейшов за посиланням на четверту. Історія забула другу та третю сторінки

15.12. Одночасні читання та запис

Потік можна використовувати для доступу до колекції для читання і запису одночасно. Припустимо, потрібно створити клас *History*, який керуватиме кнопками «Вперед» і «Назад» у веббраузері. Екземпляр *History* має реагувати на натискання кнопок, як зображено на рис. 15.5 – 15.11.

Таку поведінку можна реалізувати з використанням *ReadWriteStream*.

```
Object subclass: #History
   instanceVariableNames: 'stream'
   classVariableNames: ''
   package: 'PBE-Streams'

History >> initialize
   super initialize.
   stream := ReadWriteStream on: Array new
```

Тут нічого складного: визначили новий клас, який містить потік. Потік створюється під час виконання методу *initialize*.

Потрібні також методи, щоб рухатися вперед і назад.

```
History >> goForward
   self canGoForward
      ifFalse: [ self error: 'Already on the last element' ].
   ^ stream next

History >> goBackward
   self canGoBackward
      ifFalse: [ self error: 'Already on the first element' ].
   stream skip: -2.
   ^ stream next.
```

До цього місця код досить простий. Далі потрібно реалізувати метод *goTo:*, який буде активовано, коли користувач клацне на посилання. Можлива реалізація наведена нижче.

```
History >> goTo: aPage
   stream nextPut: aPage
```

Проте такий варіант незавершений. Коли користувач клацає на посилання, немає ніяких майбутніх сторінок, щоб перейти до них, тобто, кнопку «Вперед» потрібно деактивувати. Щоб зробити це, найпростішим рішенням є записати в потік *nil* відразу після сторінки, щоб зазначити, що браузер перебуває наприкінці історії.

```
History >> goTo: anObject
   stream nextPut: anObject.
   stream nextPut: nil.
   stream back
```

Тепер залишилося реалізувати тільки методи *canGoBackward* і *canGoForward*.

Вказівник потоку завжди розташований між двома елементами. Для того, щоб рухатися в зворотному напрямі, має бути дві сторінки перед поточною позицією: поточна сторінка і та, на яку потрібно потрапити.

```
History >> canGoBackward
  ^ stream position > 1

History >> canGoForward
  ^ stream atEnd not and: [ stream peek notNil ]
```

Корисним буде метод, який дає змогу поглянути на вміст потоку.

```
History >> contents
  ^ stream contents
```

Легко переконатися, що *History* працює належно.

```
History new
   goTo: #page1;
   goTo: #page2;
   goTo: #page3;
   goBackward; goBackward;
   goTo: #page4;
   contents
>>> #(#page1 #page4 nil nil)
```

15.13. Використання потоків для доступу до файлів[15]

Ми бачили, як використовувати потоки для доступу до елементів колекції. Потоки можна також використати для взаємодії з файлами на жорсткому диску комп'ютера. API файлових потоків такий самий, як у щойно вивчених, особливим є тільки спосіб створення потоку.

Запис до файлу

У наведених нижче прикладах ви не побачите явного вказання класів файлових потоків. Усю роботу щодо створення потоку люб'язно виконають для вас об'єкти системи Pharo. Вам потрібно лише попросити їх про допомогу та виконати такі кроки:

- перетворити рядок, що задає ім'я файлу, на посилання на файл;
- отримати від файлу накладений на нього потік;
- виконати виведення в потік;
- не забути закрити потік після завершення роботи.

```
| hello stream |
hello := 'hello.txt' asFileReference.
hello exists
>>> false   "завжди можна перевірити існування файлу"
stream := hello writeStream.
stream nextPutAll: 'Hello World'.
stream close.
```

[15] Цей параграф додав перекладач книги.

Читання з файлу

Послідовність кроків така сама, як під час читання, тільки для отримання потоку використовують повідомлення *readStream* замість *writeStream*, та читають рядки з потоку, а не надсилають їх туди.

```
| hello stream |
hello := 'hello.txt' asFileReference.
hello exists
>>> true

stream := hello readStream.
stream next.
>>> $H

stream upToEnd.
>>> 'ello World'

stream close
```

Автоматичне закривання потоку

Щоб не турбуватися про своєчасне надсилання повідомлення *close*, можна використовувати методи, які зроблять це за вас. Припустимо *hello* – створене раніше посилання на файл. Тоді запис до нього можна виконати коротко:

```
hello writeStreamDo: [ :stream | stream nextPutAll: 'Hello World'].
```

Читання займе не більше місця:

```
hello readStreamDo: [ :stream | stream contents ]
>>> 'Hello World'
```

15.14. Підсумки розділу

Потоки пропонують кращий порівняно з колекціями спосіб для поступового читання і запису послідовності елементів. Є прості способи, щоб перетворювати потоки на колекції і навпаки.

- Потоки можуть бути лише для зчитування, лише для запису, або і для того, і для іншого.
- Щоб перетворити колекцію на потік, потік накладають на колекцію, тобто створюють повідомленням *on: – ReadStream on: (1 to: 1000)*; або надсилають колекції повідомлення *readStream* тощо.
- Щоб перетворити потік на колекцію, йому надсилають повідомлення *contents*.
- Для конкатенації великих колекцій, замість того, щоб використовувати оператор кома, ефективніше створити потік, додати колекції до нього повідомленням *nextPutAll:* і отримати результат за допомогою *contents*.
- Потоки можна використовувати для доступу до файлів. Зручно надсилати повідомлення, які автоматично закривають потік одразу після завершення роботи.

Розділ 16
Морфи

Графічний інтерфейс Pharo називають *Morphic*[16]. Він підтримує два головні аспекти: з одного боку, Morphic визначає всі низькорівневі графічні сутності та відповідну інфраструктуру (події, перемальовування тощо), з іншого – він визначає всі доступні у Pharo графічні елементи (віджети). Morphic написаний на Pharo, тому без обмежень працює в усіх операційних системах. Як наслідок, Pharo виглядає однаково в Unix, MacOS та Windows. На відміну від більшості інших графічних інструментів Morphic не має окремих режимів для *компонування* і *виконання* інтерфейсу: користувач може у будь-який момент зібрати або розібрати кожен графічний елемент. Ми вдячні Ілеру Фернандесу (Hilaire Fernandes) за дозвіл побудувати цей розділ з використанням його оригінальної статті французькою.

16.1. Історія створення

Morphic розробили Джон Мелоні (John Maloney) і Ренді Сміт (Randy Smith) для мови програмування Self десь на початку 1993 року. Пізніше Мелоні написав нову версію Morphic для Squeak, але головні ідеї початкової версії досі живі-здорові та працюють у Morphic для Pharo – це *безпосередність* і *активність*. Безпосередність означає, що фігури на екрані є об'єктами, які можна дослідити та змінити напряму, просто клацнувши на них мишкою. Активність означає, що графічний інтерфейс здатний щомиті реагувати на дії користувача, інформація на екрані оновлюється, щойно зміниться стан, який вона відображає. Простий приклад використання цих ідей: можемо продублювати пункт меню і перетворити його на кнопку, що діятиме так само.

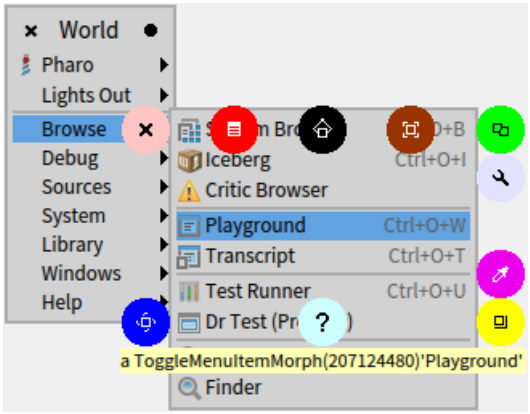

Рис. 16.1. Меню-ореол графічного елемента використовують для маніпуляцій з ним

Відкрийте Головне меню, оберіть якийсь з його пунктів, наприклад, «*Playground*» з розділу «*Browse*», і метаклацніть на ньому, щоб відкрити його меню-ореол, як на рис. 16.1. Клацніть на зеленому маніпуляторі «*Duplicate*», щоб створити копію пункту меню, і перенесіть копію в довільне місце на екрані (рис. 16.2). Ще одне клацання зафіксує кнопку в обраному місці. Тепер можна випробувати її дію і переконатися, що вона працює так само, як початковий пункт меню. Примітно, що створена кнопка завжди

[16] У англійській мові суфікс *-morphic* означає «той, що має форму, вигляд, структуру». Таке значення якнайкраще підходить для вікон графічного інтерфейсу (*прим. – Ярошко С.*).

залишатиметься видимою, виринаючи поверх усіх вікон Pharo. Щоб прибрати її з екрана, знову скористайтеся меню-ореолом.

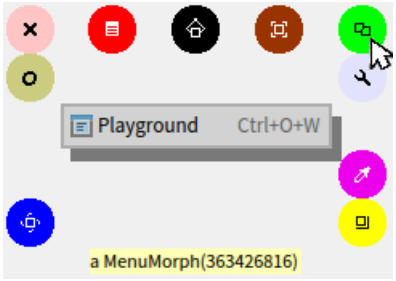

Рис. 16.2. Копію рядка меню можна перетворити на окрему кнопку

Цей приклад демонструє, що ми називаємо *безпосередністю* і *активністю*. Вони надають широке коло можливостей під час розробки альтернативного інтерфейсу користувача та прототипування альтернативних взаємодій.

Morphic потроху старіє, і спільнота Pharo вже кілька років працює над можливою заміною. Заміна Morphic означає розробку нової низькорівневої інфраструктури та нових наборів графічних елементів. Проєкт називається Block, уже виконано кілька ітерацій. Block – це інфраструктура, а Brick – набір побудованих поверх неї графічних елементів. Але давайте отримувати задоволення від Morphic.

16.2. Морфи

Усі об'єкти, які видно на екрані запущеного Pharo, – це "морфи", тобто екземпляри підкласів *Morph*. Сам собою *Morph* – це великий клас з багатьма методами, що дає змогу похідним класам реалізувати цікаву поведінку малою кількістю коду.

Від перекладача. Надалі в тексті використовуватимемо слово *морфа* як термін. Для цього є кілька причин. Видимі елементи інтерфейсу користувача – екземпляри класу *Morph*, морфи. З давньогрецької μορφη – вид, зовнішність, форма, що добре відображає сутність цих об'єктів. Слово «морфа» – анаграма слова «форма», а формами часто називають графічні елементи інтерфейсу користувача в ОС Windows, проте говоритимемо саме «морфа», бо інтерфейс Pharo однаковий для різних операційних систем.

Можна створити морфу, яка відображає довільний об'єкт, хоча отриманий результат залежить від об'єкта! Щоб створити морфу для зображення рядка, виконайте в Робочому вікні такий код.

```
'Morph' asMorph openInWorld
```

Він створить морфу для відображення рядка 'Morph' і відкриє її (тобто відобразить) на екрані, або «*у світі*», бо екран у Pharo називають *world*. Ви мали б отримати графічний елемент (екземпляр *Morph*), яким можна маніпулювати за допомогою метаклацання.

Звісно, можна визначати морфи з цікавішим графічним зображенням, ніж ми щойно бачили. Реалізований за замовчуванням у класі *Object class* метод *asMorph* лише створює *StringMorph*. Наприклад, «*Color tan asMorph*» поверне екземпляр StringMorph надписаний результатом виконання «*Color tan printString*». Щоб отримати кольоровий прямокутник, виконайте інший код.

```
(Morph new color: Color orange) openInWorld
```

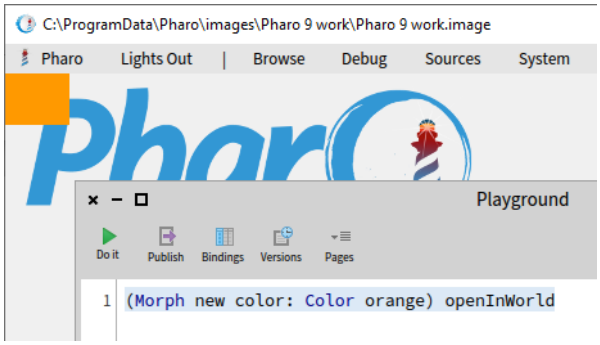

Рис. 16.3. За замовчуванням морфа прямокутна, розташована у лівому верхньому кутку екрана

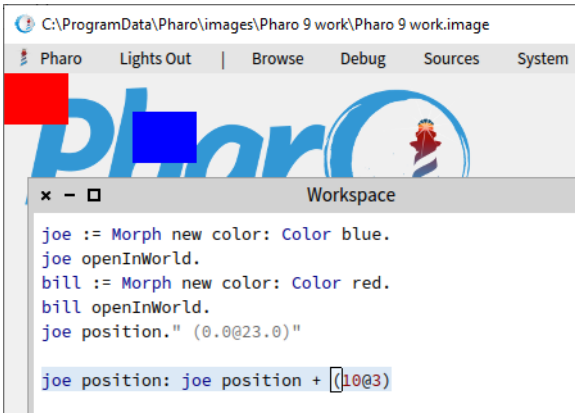

Рис. 16.4. *bill* і *joe* після десяти переміщень

Замість морфи у вигляді рядка отримано оранжевий прямокутник (рис. 16.3).

16.3. Маніпулювання морфами

Морфи – це об'єкти, тому ними можна маніпулювати як усіма іншими об'єктами у Pharo: за допомогою повідомлень можна змінювати їхній стан, створювати нові підкласи *Morph* тощо.

Кожна морфа, навіть невидима в поточний момент, має позицію і розмір. За домовленістю всі морфи займають прямокутну область екрана. Якщо морфа неправильної форми, то її позицію і розмір задає найменший описаний прямокутник, так званий обмежувальний прямокутник, або *межа*.

- Метод *position* повертає екземпляр *Point*, який описує розташування верхнього лівого кута морфи або його обмежувального прямокутника. Початком системи координат є верхній лівий кут головного вікна Pharo, координата *y* зростає донизу, а *x* – праворуч.
- Метод *extent* також повертає точку, але ця точка визначає не координати, а ширину та висоту морфи.

Наберіть наведений нижче код у Робочому вікні і виконайте його командою «*Do it*»[17].

```
joe := Morph new color: Color blue.
joe openInWorld.
bill := Morph new color: Color red.
bill openInWorld.
```

Тоді спробуйте виконати за допомогою «*Print it*» вираз «*joe position*», щоб довідатися поточне розташування першої морфи. Ви мали б отримати щось схоже на «*(0.0@23.0)*».

Тепер змінимо це розташування. Надрукуйте і виконайте кілька разів «*joe position: joe position + (10@3)*». Результат переміщень зображено на рис. 16.4.

Подібно можна взаємодіяти з розміром і іншими властивостями морфи.

```
joe extent. "повідомляє розмір морфи joe, <Print it>"
>>> (50.0@40.0)
joe extent: joe extent * 1.1. "збільшує розмір морфи joe, <Do it>"
```

[17] Локальні змінні у Робочому вікні Pharo 9.0 можна не оголошувати (*прим. – Ярошко С.*).

```
joe color: (Color orange alpha: 0.5). "задає колір морфи joe, <Do it>"

"задає відносне розташування морфи bill, <Do it>"
bill position: joe position + (100@0)
```

Щоб пропорційно збільшити морфу, виконайте за допомогою «*Do it*» вираз «*joe extent: joe extent * 1.1*». Щоб змінити колір морфи, надішліть їй повідомлення *color:* з бажаним екземпляром *Color* як аргумент, наприклад, «*joe color: Color orange*». Щоб додати прозорість, спробуйте «*joe color: (Color orange alpha: 0.5)*».

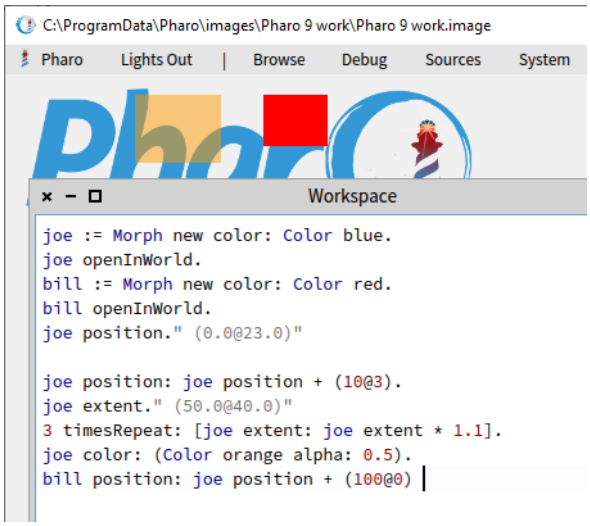

Рис. 16.5. *bill* слідує за *joe*

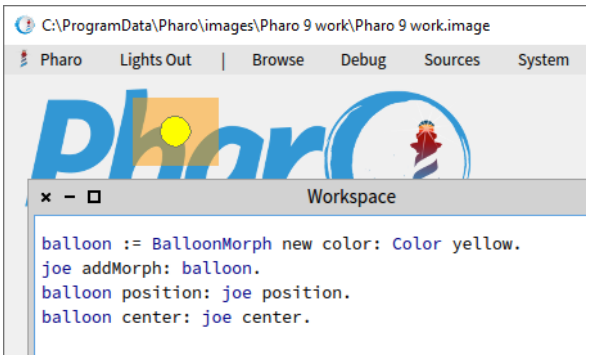

Рис. 16.6. *balloon* вбудовано в *joe* – оранжеву напівпрозору морфу

Для того, щоб задати розташування однієї морфи стосовно іншої, можна використати повідомлення «*bill position: joe position + (100@0)*». Виконуйте його щоразу після переміщення *joe*, і *bill* слідуватиме за нею. Наприклад, перетягніть *joe* мишею, виконайте код і побачите, що *bill* розташується на 100 цяток праворуч від *joe*. Результати виконання всіх повідомлень зображено на рис. 16.5.

Створені морфи можна вилучити:

- надсиланням повідомлення *delete*, наприклад, «*bill delete*»;
- за допомогою маніпулятора з хрестиком меню-ореолу.

16.4. Композиція морфів

Один зі способів створення нового графічного подання полягає у додаванні однієї морфи до структури іншої. Його називають *композицією*. Глибина композиції не обмежена. Щоб додати вкладену морфу до морфи-контейнера, контейнерові надсилають повідомлення *addMorph: anEnclosedMorph*.

Спробуємо додати нову морфу до створеної раніше, як у наведеному коді.

```
balloon := BalloonMorph new color: Color yellow.
joe addMorph: balloon.
balloon position: joe position.
```

Останній рядок розташовує кульку за тими самими координатами, що й *joe*. Зауважте, що координати вкладеної морфи відраховують стосовно екрана, а не стосовно контейнера. Використовувати абсолютні координати для позиціонування морфів,

насправді, не дуже зручно. Ця особливість робить програмування морфів трохи дивним. Але є багато методів для задання позиції морфи, перегляньте протокол *geometry* класу *Morph* і відшукайте деякі з них. Наприклад, щоб розташувати *balloon* посередині *joe*, виконайте «*balloon center: joe center*» (див. рис. 16.6).

Якщо тепер спробувати перетягнути мишкою *balloon*, то виявиться, що мишка захопила і *joe*, і дві морфи перетягаються разом: кульку *вбудовано* всередину прямокутника. У *joe* можна вбудувати більше морфів. Це можна зробити і програмно, і вручну.

16.5. Створення і малювання власних морф

Хоча за допомогою композиції можна створити багато цікавих і корисних графічних представлень, іноді виникає потреба створити щось цілковито інше.

З цією метою визначають підклас класу *Morph* і перевантажують метод *drawOn:*, щоб змінити спосіб відображення морфи.

Рис. 16.7. Екземпляр *CrossMorph* з меню-ореолом. Жовтим маніпулятором можна змінити його розмір

Середовище Morphic надсилає повідомлення *drawOn:* морфі, коли потрібно перемалювати її на екрані. Параметром повідомлення є різновид *Canvas*, очікувана поведінка – морф намалює себе на цьому полотні у його межах. Давайте використаємо ці знання, щоб створити хрестоподібну морфу.

За допомогою Системного оглядача визначимо новий клас *CrossMorph*, що наслідує *Morph*.

```
Morph subclass: #CrossMorph
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-Morphic'
```

Метод *drawOn:* можна визначити як записано нижче.

```
CrossMorph >> drawOn: aCanvas
   | crossHeight crossWidth horizontalBar verticalBar |
   crossHeight := self height / 3.
   crossWidth := self width / 3.
   horizontalBar := self bounds insetBy: 0 @ crossHeight.
   verticalBar := self bounds insetBy: crossWidth @ 0.
   aCanvas fillRectangle: horizontalBar color: self color.
   aCanvas fillRectangle: verticalBar color: self color
```

У відповідь на повідомлення *bounds* морф повертає обмежувальний прямокутник, який є екземпляром класу *Rectangle*. Прямокутники розуміють велику кількість повідомлень для створення інших прямокутників схожих розмірів. Тут використано повідомлення *insetBy:* з аргументом екземпляром *Point*, щоб створити спочатку прямокутник зі зменшеною висотою, а потім – прямокутник зі зменшеною шириною, причому виміри змінюються симетрично з обох сторін, змінюючи розташування фігури.

Щоб протестувати новий морф, виконайте *CrossMorph new openInWorld*.

Результат мав би бути схожим на зображений на рис. 16.7. Це те, що й планували, проте можна зауважити невелику помилку, якщо спробувати захопити морфу мишкою: чутливою зоною є весь обмежувальний прямокутник, включно з незафарбованими ділянками, а не лише хрест. Давайте це виправимо.

Коли середовище намагається визначити, які морфи розташовані під курсором, воно надсилає повідомлення *containsPoint:* до усіх морфів, чиї обмежувальні прямокутники перебувають під вказівником мишки. Тому, щоб зменшити чутливу зону нашої морфи до самого лише хреста, потрібно переписати метод *containsPoint:*. Визначимо його в класі *CrossMorph*.

```
CrossMorph >> containsPoint: aPoint
  | crossHeight crossWidth horizontalBar verticalBar |
  crossHeight := self height / 3.
  crossWidth := self width / 3.
  horizontalBar := self bounds insetBy: 0 @ crossHeight.
  verticalBar := self bounds insetBy: crossWidth @ 0.
  ^ (horizontalBar containsPoint: aPoint) or:
     [ verticalBar containsPoint: aPoint]
```

Цей метод використовує таку саму логіку як *drawOn:*, тому можна бути впевненим, що *containsPoint:* повертає *true* тільки для тих точок, які замальовує *drawOn:*. Зверніть увагу на те, як використали метод *containsPoint:* класу *Rectangle*, щоб зробити важку роботу.

У написаних методах приховано дві проблеми.

Найочевиднішим є дублювання коду. Це принципова помилка: якщо виявиться, що потрібно змінити спосіб обчислення *horizontalBar* або *verticalBar*, то легко можна забути змінити один з двох випадків його використання. Розв'язок проблеми – винести ці обчислення у два окремі методи у протоколі *private*.

```
CrossMorph >> horizontalBar
  | crossHeight |
  crossHeight := self height / 3.
  ^ self bounds insetBy: 0 @ crossHeight
```

```
  CrossMorph >> verticalBar
  | crossWidth |
  crossWidth := self width / 3.
  ^ self bounds insetBy: crossWidth @ 0
```

Тоді можна визначити *drawOn:* і *containsPoint:*, що використовують ці методи.

```
CrossMorph >> drawOn: aCanvas
  aCanvas fillRectangle: self horizontalBar color: self color.
  aCanvas fillRectangle: self verticalBar color: self color
```

```
CrossMorph >> containsPoint: aPoint
  ^ (self horizontalBar containsPoint: aPoint) or: [
     self verticalBar containsPoint: aPoint ]
```

Такий код набагато зрозуміліший, значною мірою завдяки осмисленим іменам приватних методів. Вони такі прості, що легко помітити другу проблему: ділянка в центрі

хреста, яка належить обом прямокутникам, замальовується двічі. Це не важливо, коли колір непрозорий, але ця вада стає помітною, коли для заповнення використовують напівпрозорий колір (рис. 16.8.)

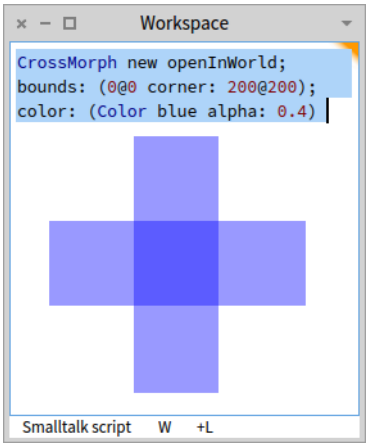

Рис. 16.8. Середину хреста замальовано двічі

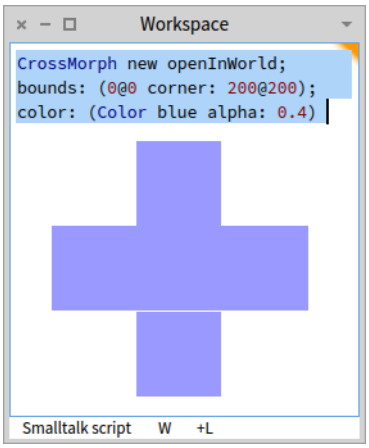

Рис. 16.9. Хрестоподібна морфа містить незамальовану лінію

Щоб переконатися, виконайте такий код у Робочому вікні.

```
CrossMorph new openInWorld;
   bounds: (0@0 corner: 200@200);
   color: (Color blue alpha: 0.4)
```

Виправити помилку можна поділом вертикального прямокутника на три частини з наступним замальовуванням лише двох з них: верхньої та нижньої. І знову можна знайти в класі *Rectangle* метод, який виконає складну роботу: *rect1 areasOutside: rect2* повертає масив прямокутників, які частинами *rect1* поза межами *rect2*. Нижче наведено виправлений код.

```
CrossMorph >> drawOn: aCanvas
     | topAndBottom |
     aCanvas fillRectangle: self horizontalBar color: self color.
     topAndBottom := self verticalBar areasOutside: self horizontalBar.
     topAndBottom do: [ :each |
        aCanvas fillRectangle: each color: self color ]
```

Цей код виглядає робочим, але якщо випробувати його на різних екземплярах, розтягаючи їх, то можна зауважити, що для окремих розмірів на хресті з'являється біла лінія товщиною в цятку (рис. 16.9). Так відбувається через похибки заокруглення, коли розмір прямокутника для замальовування не ціле число, метод *fillRectangle:color:* заокруглює його по-різному, залишаючи один рядок цяток незамальованим. Помилку можна виправити, виконавши заокруглення обчислених розмірів у методах побудови вертикального і горизонтального прямокутників, як подано нижче.

```
CrossMorph >> horizontalBar
   | crossHeight |
   crossHeight := (self height / 3) rounded.
   ^ self bounds insetBy: 0 @ crossHeight
```

```
   CrossMorph >> verticalBar
   | crossWidth |
```

```
    crossWidth := (self width / 3) rounded.
    ^ self bounds insetBy: crossWidth @ 0
```

16.6. Взаємодія через події мишки

Щоб побудувати з морфів живий користувацький інтерфейс, потрібно вміти взаємодіяти з ними за допомогою мишки та клавіатури. Ба більше, морфи мусять уміти реагувати на дії користувача зміною свого вигляду та місця розташування, виконаного за допомогою анімації.

Одразу після натискання кнопки мишки Morphic надсилає кожній морфі, розташованій під вказівником мишки, повідомлення *handlesMouseDown:*. Якщо вона відповідає *true*, то Morphic невідкладно надсилає їй повідомлення *mouseDown:*. Середовище також надішле повідомлення *mouseUp:*, коли користувач відпустить кнопку. Якщо всі морфи відповіли *false*, то Morphic ініціює операцію перетягування (drag-and-drop). Щоб переконатися в цьому, відобразіть на екрані екземпляр *CrossMorph* і клацніть на ньому мишкою – хрестик причепиться до курсора мишки і мандруватиме за ним до наступного клацання.

Згодом побачимо, що повідомлення *mouseDown:* і *mouseUp:* надсилають з аргументом, екземпляром *MouseEvent*, який містить опис події мишки.

Давайте розширимо клас *CrossMorph* так, щоб він зміг обробляти події мишки. Спочатку впевнимося, що усі екземпляри *CrossMorph* відповідають *true* на повідомлення *handlesMouseDown:*. Для цього оголосимо в *CrossMorph* відповідний метод.

```
CrossMorph >> handlesMouseDown: anEvent
  ^ true
```

Припустимо, що під час натискання лівої кнопки мишки потрібно змінити колір хреста на червоний, а після натискання правої – на жовтий. Це можна реалізувати методом *mouseDown:* як показано нижче.

```
CrossMorph >> mouseDown: anEvent
  anEvent redButtonPressed
     ifTrue: [ self color: Color red ]. "клацання мишкою"
  anEvent yellowButtonPressed
     ifTrue: [ self color: Color yellow ]. "контекстне клацання"
  self changed
```

Важливо, що, крім зміни кольору морфи, цей метод також надсилає повідомлення «*self changed*». Завдяки цьому середовище своєчасно надішле морфі повідомлення *drawOn:*.

Потрібно також зауважити, що як тільки морфа опрацьовує події мишки, її вже не вдасться перетягати, як раніше. Натомість потрібно відкрити меню-ореол морфи і перетягнути її за чорний маніпулятор, розташований над нею. Скористайтеся нагодою і змініть колір морфи за допомогою бузкового маніпулятора, розташованого праворуч. Далі можна експериментувати з клацанням на морфі.

Аргумент *anEvent* повідомлення *mouseDown:* є екземпляром класу *MouseEvent*, підкласу *MorphicEvent*. *MouseEvent* визначає методи *redButtonPressed* і *yellowButtonPressed*. Перегляньте цей клас і знайдіть інші методи для опитування події мишки.

16.7. Події клавіатури

Для перехоплення подій клавіатури потрібно зробити три кроки.

1. Передати *фокус клавіатури* певній морфі. Наприклад, можна передати фокус морфі, коли вказівник мишки розташований над нею.
2. Обробити подію клавіатури за допомогою методу *keyDown:*. Середовище надсилає відповідне повідомлення морфі, що має фокус клавіатури, коли користувач натискає клавішу.
3. Зняти фокус клавіатури з морфи, коли курсор більше не розташований над нею.

Давайте розширимо *CrossMorph* так, щоб він реагував на натискання клавіш. Спочатку потрібно організувати інформування морфи про те, що курсор мишки перебуває над нею. Середовище надсилає відповідні повідомлення морфі, яка відповідає *true* на повідомлення *handlesMouseOver:*, тому потрібно оголосити метод, наведений нижче.

```
CrossMorph >> handlesMouseOver: anEvent
  ^ true
```

Це повідомлення подібне до *handlesMouseDown:* для позиції мишки. Коли вказівник мишки заходить на морфу чи покидає її, то морфі надсилаються повідомлення *mouseEnter:* і *mouseLeave:*, відповідно.

Визначимо два методи так, щоб *CrossMorph* захоплював і звільняв фокус клавіатури, третій метод задавав інформування про натискання, а четвертий – обробляв натискання клавіш.

```
CrossMorph >> mouseEnter: anEvent
  anEvent hand newKeyboardFocus: self

CrossMorph >> mouseLeave: anEvent
  anEvent hand releaseKeyboardFocus: self

CrossMorph >> handlesKeyDown: anEvent
  ^ true

CrossMorph >> keyDown: anEvent
  | key |
  key := anEvent key.
  key = KeyboardKey up ifTrue: [ self position: self position - (0 @ 10) ].
  key = KeyboardKey down ifTrue:[self position: self position + (0 @ 10) ].
  key = KeyboardKey right ifTrue:[self position: self position + (10 @ 0)].
  key = KeyboardKey left ifTrue:[self position: self position - (10 @ 0) ]
```

Метод написано так, щоб можна було пересувати морфу за допомогою клавіш зі стрілками. Зауважте, що коли мишка не розташована над морфою, то повідомлення *keyDown:* не надсилається, і морфа не реагує на стрілки. Щоб побачити значення натиснутих клавіш, додайте вираз «*Transcript show: anEvent keyValue*.» до методу *keyDown:* і відкрийте вікно *Transcript*. Тепер спробуйте керувати хрестиком клавішами і спостерігайте за виведенням у консоль.

Аргумент *anEvent* методу *keyDown:* є екземпляром класу *KeyboardEvent*, підкласу *MorphicEvent*. Перегляньте *KeyboardEvent*, щоб більше дізнатися про події клавіатури.

Якщо хочете переміщувати морфу комбінаціями клавіш вигляду [*Ctrl+Key*], то в методі *keyDown:* можна використати розпізнавання, як описано нижче.

```
anEvent controlKeyPressed ifTrue: [
   anEvent keyCharacter == $d ifTrue: [
      self position: self position + (0 @ 10) ] ]
```

Від перекладача. Експерименти з переміщенням *CrossMorph* засвідчують, що перемальовування фігур неправильної форми не найсильніша сторона Morphic. У наведеному вище методі *keyDown:* крок переміщення задано невеликим – 10 цяток. Може виявитися, що під час переміщення хреста клавішами на екрані залишатимуться зафарбовані ділянки в попередньому розташуванні фігури: середовище не завжди витирає прозорі частини морфи. Проблем не виникає, якщо переміщати морфу з розгорнутим меню-ореолом. Можна також збільшити крок переміщення: хоча б 27 по вертикалі і 33 по горизонталі (ці величини пов'язані з розмірами морфи за замовчуванням).

Не зайвим буде додати, що Morphic підтримує три різні події клавіатури: *keystroke*, *keydown* і *keyup*. У цьому параграфі опрацьовували подію *keydown*, а в наступному буде використано *keystroke*.

16.8. Анімація морф

Morphic надає просту систему анімації з двома основними повідомленнями: *step* надсилається морфі через постійні проміжки часу, а *stepTime* визначає інтервал у мілісекундах між двома такими надсиланнями. Саме *stepTime* визначає *мінімальний* час між двома *step*. Якщо ви захочете, щоб *stepTime* задавав одну мілісекунду, то не дивуйтеся, що Pharo буде занадто зайнятий виконанням кроків анімації вашої морфи аж так часто. Додамо, що метод *startStepping* вмикає механізм анімації, а *stopStepping* вимикає його. Щоб довідатися, чи виконується анімація морфи, можна запитати її *isStepping*.

Навчимо *CrossMorph* миготіти, визначивши методи, як зображено нижче.

```
CrossMorph >> stepTime
  ^ 100

CrossMorph >> step
  (self color diff: Color black) < 0.1
     ifTrue: [ self color: Color red ]
     ifFalse: [ self color: self color darker ]
```

Щоб побачити анімацію в дії, відкрийте інспектор морфи маніпулятором налагодження меню-ореолу (маніпулятор лавандового кольору з гайковим ключем), у панелі редактора коду введіть вираз «*self startStepping*» і виконайте його командою «*Do it*». Так само можна зупинити анімацію: «*self stopStepping*».

Зауваження. Якщо створення морфи виконувати кнопкою ***Do it*** у вікні Playground, то вікно інспектора відкриється автоматично.

Для керування анімацією можна використати клавіатуру, наприклад, [+] та [–] для вмикання і вимикання, відповідно.

Зазвичай для опрацювання тексту використовують подію *keystroke*, а для опрацювання гарячих клавіш – *keydown* і *keyup*. Про *keydown* йшлося в попередньому параграфі, тому

використаємо опрацювання *keystroke*. Потрібно оголосити два методи: перший вмикає інформування про виникнення події, а другий – опрацьовує її.

```
CrossMorph >> handlesKeyStroke: anEvent
   ^ true

CrossMorph >> keyStroke: anEvent
   | keyValue |
   keyValue := anEvent keyCharacter.
   keyValue == $+ ifTrue: [ self startStepping ].
   keyValue == $- ifTrue: [ self stopStepping ]
```

Зауважимо, що подія *keystroke* морфи стається тільки тоді, коли вона має фокус уведення клавіатури.

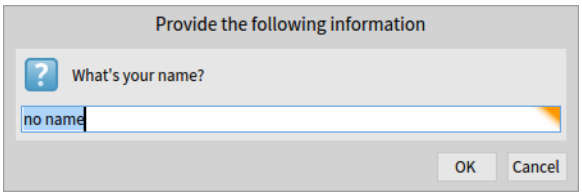

Рис. 16.10. Уведення рядка

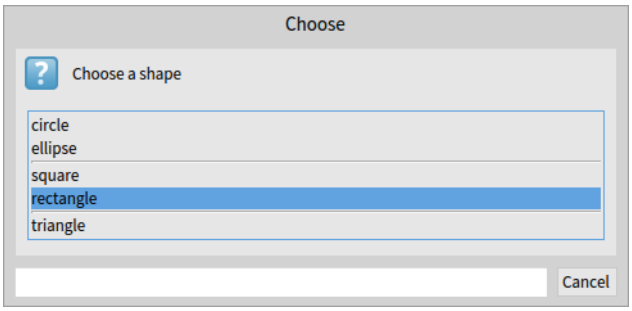

Рис. 16.11. Діалог-меню

16.9. Діалоги

Клас *UIManager* підтримує велику кількість готових панелей діалогу, щоб можна було попросити користувача ввести певні дані. Наприклад, метод *request:initialAnswer:* повертає введений користувачем рядок (див. рис. 16.10). Відкрити діалог досить просто.

```
UIManager default
   request: 'What''s your name?'
   initialAnswer: 'no name'
```

Щоб відкрити діалог, схожий на виринаюче меню, використовують різні варіанти методу *chooseFrom:* (див. рис. 16.11).

```
UIManager default
   chooseFrom: #('circle' 'ellipse' 'square' 'rectangle' 'triangle')
   lines: #(2 4)
   message: 'Choose a shape'
```

Перегляньте клас *UIManager* і випробуйте інші його методи для взаємодії з користувачем. Використайте в діалогах українську. Перевірте, яку відповідь поверне діалог, якщо користувач натисне кнопку ***Cancel***.

16.10. Перетягування

Середовище Morphic підтримує взаємодію морф за допомогою операції перетягування. Дослідимо простий приклад взаємодії двох морф: морфи-приймача і переміщеної морфи. Переміщену морфу тягнуть мишкою і скидають на приймач, приймач може

перевірити, чи задовольняє скинутий об'єкт задану умову, і або прийняти його, або відхилити. Припустимо, що скинута морфа має бути синього кольору. Відхилена морфа сама вирішує, що зробити далі.

Спочатку визначимо морфу-отримувач.

```
Morph subclass: #ReceiverMorph
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-Morphic'
```

Звичайно визначимо її метод ініціалізації.

```
ReceiverMorph >> initialize
  super initialize.
  color := Color red.
  bounds := 0 @ 0 extent: 100 @ 100
```

Хто має вирішити, чи приймач схвалить, чи відхилить скинуту морфу? В загальному випадку, обидві морфи мають бути готовими взаємодіяти. Приймач робить це, відповідаючи на повідомлення *wantsDroppedMorph:event:*. Його перший аргумент – скинута морфа, а другий – подія мишки. Так приймач може, наприклад, побачити, чи натискали якісь командні клавіші в момент скидання. Скинута морфа також має шанс перевірити і подивитися, чи її влаштовує морфа, на яку її скидають, відповідаючи на повідомлення *wantsToBeDroppedInto:*. Реалізація за замовчуванням цього методу (у класі *Morph*) повертає *true*.

```
ReceiverMorph >> wantsDroppedMorph: aMorph event: anEvent
  ^ aMorph color = Color blue
```

Що відбудеться зі скинутою морфою, якщо приймач не хоче її прийняти? Поведінка за замовчуванням – не робити нічого, тобто залишатися поверх нього, але не взаємодіяти з ним. Інтуїтивно зрозумілішою поведінкою було б повернутися на початкову позицію. Цього можна досягти, отримавши від приймача ствердну відповідь на повідомлення *repelsMorph:event:*, коли він не хоче приймати скинуту морфу.

```
ReceiverMorph >> repelsMorph: aMorph event: anEvent
  ^ (self wantsDroppedMorph: aMorph event: anEvent) not
```

Це все, що потрібно зробити в класі приймача.

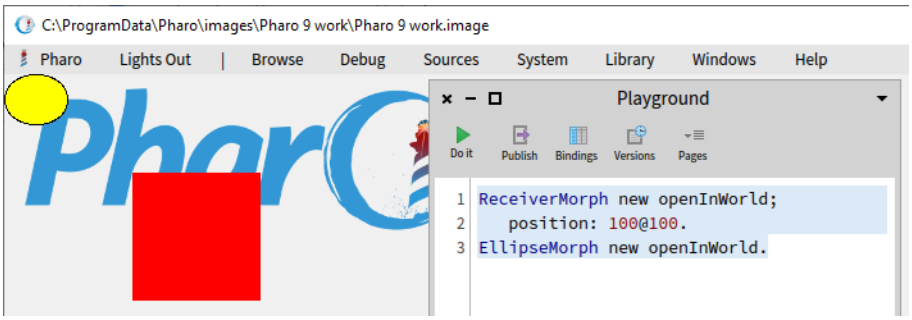

Рис. 16.12. Екземпляри класів *ReceiverMorph* та *EllipseMorph*

Створіть у Робочому вікні екземпляри *ReceiverMorph* та *EllipseMorph* (див. рис. 16.12).

```
ReceiverMorph new openInWorld;
   position: 100@100.
EllipseMorph new openInWorld.
```

Спробуйте перетягнути та скинути жовтий *EllipseMorph* на приймача. Приймач його відхилить і еліпс повернеться на свою початкову позицію.

Щоб побачити іншу поведінку, змініть колір еліпса на синій (надішліть йому повідомлення «*color: Color blue;*» одразу після *new*, перед *openInWorld*). Перетягніть синю морфу на червоний квадрат і відпустіть – скинута морфа стане частиною приймача. Можете створити кілька еліпсів і перетягнути усі на квадрат.

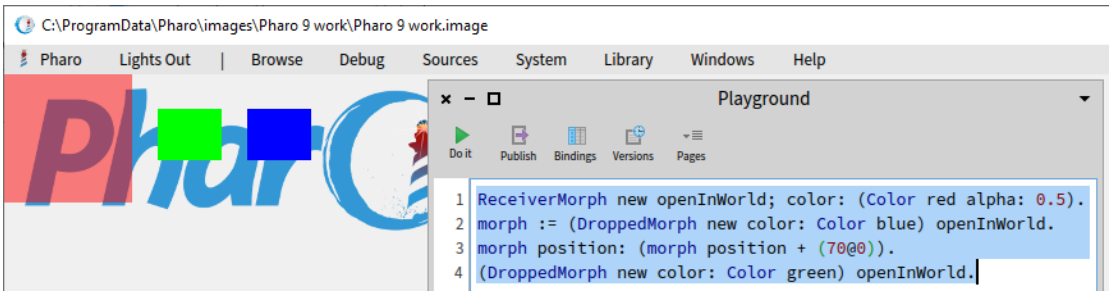

Рис. 16.13. Створення *DroppedMorph* і *ReceiverMorph*

Щоб продовжити експерименти, створимо підклас класу *Morph* і назвемо його *DroppedMorph*.

```
Morph subclass: #DroppedMorph
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-Morphic'

DroppedMorph >> initialize
   super initialize.
   color := Color blue.
   self position: 120 @ 50
```

Тепер можна задати, як поводитиметься скинута морфа, коли приймач відкине її. В описаному нижче випадку вона залишатиметься прикріпленою до вказівника мишки.

```
DroppedMorph >> rejectDropMorphEvent: anEvent
   | h |
   h := anEvent hand.
   WorldState addDeferredUIMessage: [ h grabMorph: self ].
   anEvent wasHandled: true
```

Тут об'єкт *anEvent* – подія мишки, у відповідь на повідомлення *hand* вона повертає «руку», екземпляр класу *HandMorph*, який представляє вказівник мишки та все, що він тримає. Тут сказано об'єктові *World*, що рука має захопити *self* – відкинуту морфу.

Створіть два екземпляри класу *DroppedMorph* різних кольорів (рис. 16.13), перетягніть і скиньте їх на приймача.

```
ReceiverMorph new openInWorld; color: (Color red alpha: 0.5).
morph := (DroppedMorph new color: Color blue) openInWorld.
morph position: (morph position + (70@0)).
```

```
(DroppedMorph new color: Color green) openInWorld.
```

Приймач відкине зелену морфу, і вона залишиться прикріпленою до вказівника мишки.

16.11. Завершений приклад

Давайте розробимо морфу, яка відображає і обертає ігральну кісточку. Клацання на ній запускатиме почергове відображення різних граней, а наступне клацання – зупинятиме анімацію.

Клас кісточки наслідуємо від *BorderedMorph* замість *Morph*, бо використовуватимемо її краї.

```
BorderedMorph subclass: #DieMorph
   instanceVariableNames: 'faces dieValue isStopped'
   classVariableNames: ''
   package: 'PBE-Morphic'
```

Змінна екземпляра *faces* міститиме кількість граней ігральної кісточки. Найбільша кількість граней буде дев'ять! Змінна *dieValue* записуватиме значення видимої в цей момент грані, а *isStopped* відповідатиме за стан анімації: значення *true* означає, що анімацію зупинено. Щоб створити екземпляр ігральної кісточки, визначимо *метод класу*: *DieMorph class >> faces: n* створює кісточку з *n* гранями.

```
DieMorph class >> faces: aNumber
  ^ self new faces: aNumber
```

Визначимо метод *initialize* на стороні об'єкта звичним способом. Пам'ятаємо, що *new* автоматично надсилає повідомлення *initialize* новоствореному об'єкту.

```
DieMorph >> initialize
  super initialize.
  self extent: 50 @ 50.
  self
     useGradientFill;
     borderWidth: 2;
     useRoundedCorners.
  self setBorderStyle: #complexRaised.
  self fillStyle direction: self extent.
  self color: Color green.
  dieValue := 1.
  faces := 6.
  isStopped := false
```

Щоб надати кісточці гарного вигляду, використано кілька методів *BorderedMorph*: задано товстий опуклий край, заокруглені кути і градієнт кольору на видимій грані. Метод екземпляра *faces:* визначено так, щоб перевіряти правильність параметра.

```
DieMorph >> faces: aNumber
  "Задає кількість граней"
  (aNumber isInteger and: [ aNumber > 0 and: [ aNumber <= 9 ] ])
     ifTrue: [ faces := aNumber ]
```

Було б добре переглянути порядок надсилання повідомлень під час створення кісточки. Наприклад, під час виконання «*DieMorph faces: 9*».

- Метод класу *DieMorph class >> faces:* надсилає повідомлення *new* метакласові *DieMorph class*.
- Метод *new* (успадкований *DieMorph class* від *Behavior*) створює новий об'єкт і надсилає йому повідомлення *initialize*.
- Метод *initialize* у *DieMorph* задає *faces* початкове значення 6.
- *DieMorph class >> new* повертає керування методу класу *DieMorph class >> faces:*, який надсилає повідомлення *faces: 9* новоствореному екземпляру.
- Виконується метод екземпляра *DieMorph>>faces:* і задає змінній *faces* значення 9.

Перш ніж визначати *drawOn:*, потрібно оголосити кілька приватних методів, які повідомляють розташування крапок на видимій грані.

```
DieMorph >> face1
  ^ {(0.5 @ 0.5)}

DieMorph >> face2
  ^{0.25@0.25 . 0.75@0.75}

DieMorph >> face3
  ^{0.25@0.25 . 0.75@0.75 . 0.5@0.5}

DieMorph >> face4
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75}

DieMorph >> face5
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.5@0.5}

DieMorph >> face6
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5}

DieMorph >> face7
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 .
     0.5@0.5}

DieMorph >> face8
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 .
     0.5@0.5 . 0.5@0.25}

DieMorph >> face9
  ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 .
     0.5@0.5 . 0.5@0.25 . 0.5@0.75}
```

Ці методи визначають колекції координат для кожної грані. Координати розраховано для квадрата розміром 1×1, достатньо їх масштабувати, щоб розташувати крапки на грані справжнього розміру.

Метод *drawOn:* виконує дві дії: спочатку малює фон грані за допомогою звертання до *super*, а потім малює крапки.

```
DieMorph >> drawOn: aCanvas
   super drawOn: aCanvas.
   (self perform: ('face', dieValue asString) asSymbol)
      do: [ :aPoint | self drawDotOn: aCanvas at: aPoint ]
```

Друга частина цього методу використовує можливості рефлексії Pharo. Малювання крапок на грані – це простий перебір колекції, наданої методом *faceX* для цієї грані, задля надсилання повідомлення *drawDotOn:at:* з кожною координатою. Щоб викликати правильний метод *faceX*, сконструйовано відповідне повідомлення виразом «*('face', dieValue asString) asSymbol*» і використано метод *perform:*, щоб надіслати його.

```
DieMorph >> drawDotOn: aCanvas at: aPoint
   aCanvas
      fillOval: (Rectangle
         center: self position + (self extent * aPoint)
         extent: self extent / 6)
      color: Color black
```

Оскільки координати нормовані до інтервалу [0; 1], то їх масштабують відповідно до розміру грані: *self extent * aPoint*. Тепер можна створити в Робочому вікні екземпляр кісточки (рис. 16.14).

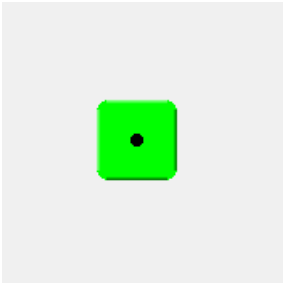

Рис. 16.14. Нова кісточка створена виразом *(DieMorph faces: 6) openInWorld*

Рис. 16.15. Результат виконання *(DieMorph faces: 6) openInWorld; dieValue: 5* і меню-ореол

Щоб мати змогу змінювати видиму грань, створимо відповідний метод доступу. Його можна буде використовувати як ключове повідомлення *myDie dieValue: 5*.

```
DieMorph >> dieValue: aNumber
   (aNumber isInteger and: [ aNumber > 0 and: [ aNumber <= faces ] ])
      ifTrue: [
         dieValue := aNumber.
         self changed ]
```

Щоб швидко змінювати грані, використаємо анімацію.

```
DieMorph >> stepTime
   ^ 100

DieMorph >> step
   isStopped ifFalse: [self dieValue: (1 to: faces) atRandom]
```

Створіть *DieMorph* і побачите, що кісточка обертається! Принаймні, змінює кількість крапок на видимій грані.

Навчимо кісточку запускати або зупиняти анімацію клацанням мишки. Використаємо для цього здобуті знання про події мишки. Спочатку активуємо повідомлення про них.

```
DieMorph >> handlesMouseDown: anEvent
  ^ true
```

Потім оголосимо метод опрацювання клацання: він альтернативно вмикає-вимикає анімацію.

```
DieMorph >> mouseDown: anEvent
  anEvent redButtonPressed ifTrue: [isStopped := isStopped not]
```

Тепер кісточка починатиме або переставатиме обертатися після кожного клацання на ній.

16.12. Більше про полотно малювання

Єдиний аргумент методу *drawOn:* – екземпляр класу *Canvas*, полотно. Це ділянка, на якій морфа відображає себе. За допомогою графічних методів полотна можна створювати такий вигляд морфи, якого забажаєте. Якщо переглянути ієрархію класу *Canvas*, то легко бачити, що він має кілька підкласів. Підкласом за замовчуванням є *FormCanvas*, і більшість ключових графічних методів міститься у *Canvas* та *FormCanvas*. Ці методи можуть малювати точки, лінії, ламані, прямокутники, еліпси, тексти та зображення, а також повертати їх та масштабувати.

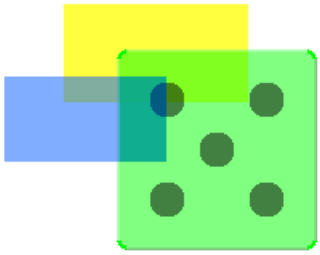

Рис. 16.16. Напівпрозоре відображення грального кісточки

Також можливо використовувати інші види полотна, наприклад, щоб отримати прозорі морфи, використати більше графічних методів, згладжування тощо. Щоб використати ці засоби, потрібний буде *AlphaBlendingCanvas* або *BalloonCanvas*. Але як можна одержати таке полотно у методі *drawOn:*, якщо він отримує своїм аргументом екземпляр класу *FormCanvas*? На щастя, можна перетворити полотно одного типу в інший.

Щоб використати у *DieMorph* полотно з коефіцієнтом прозорості 0.5, перевизначимо *drawOn:*, як показано нижче.

```
DieMorph >> drawOn: aCanvas
| theCanvas |
theCanvas := aCanvas asAlphaBlendingCanvas: 0.5.
super drawOn: theCanvas.
(self perform: ('face', dieValue asString) asSymbol)
   do: [:aPoint | self drawDotOn: theCanvas at: aPoint]
```

Це все, що потрібно зробити! Змінене зображення кісточки видно на рис. 16.16.

16.13. Підсумки до розділу

Morphic – це графічне середовище, в якому можна динамічно компонувати графічні елементи інтерфейсу.

- Будь-який об'єкт можна перетворити на морфу і відобразити його на екрані за допомогою повідомлень *asMorph openInWorld*.
- Морфою можна керувати за допомогою маніпуляторів меню-ореола, яке відкривають метаклацанням на морфі. (Маніпулятори мають виринаючі підказки, які пояснюють їхнє призначення).
- Морфи можна компонувати, вставляючи їх одна в одну перетягуванням або за допомогою повідомлення *addMorph:*.
- Клас морфи можна наслідувати та перевизначити ключові методи, такі як *initialize* та *drawOn:*.
- Можна контролювати взаємодію морфи з мишкою та клавіатурою, перевизначивши такі методи, як *handlesMouseDown:*, *handlesMouseOver:* тощо.
- Морфу можна анімувати, перевизначивши методи *step* (що робити) та *stepTime* (кількість мілісекунд між кадрами).

Розділ 17
Класи і метакласи

У Pharo все є об'єктами, а кожний об'єкт – екземпляр класу. Самі класи теж не становлять виняток – усі класи є об'єктами, причому об'єкти-класи є екземплярами інших класів. Така об'єктна модель компактна, проста, елегантна та однорідна. Вона повністю охоплює суть об'єктно-орієнтованого програмування. Проте новачка наслідки такої побудови можуть збити з пантелику.

Зауважимо, що для того, щоб вільно програмувати на Pharo, не обов'язково повністю розуміти всі тонкощі побудови об'єктної моделі Pharo і наслідки її однорідності, та все ж. Мета цього розділу двояка: (1) заглибитись, на скільки вдасться, у суть питання і (2) з'ясувати, що тут немає ніякої магії чи особливих складнощів: лише прості правила, однаково застосовані у всіх випадках. Послуговуючись цими правилами, можна завжди розуміти, чому ситуація саме така, яка вона є.

17.1. Правила для класів

Об'єктна модель Pharo ґрунтується на обмеженій кількості концепцій, які застосовують завжди однаково. Пригадаємо правила об'єктної моделі, розглянуті в розділі 10 «Об'єктна модель Pharo».

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 самими правилами.

З Правила 1 випливає, що *класи також є об'єктами*, а з Правила 2 – що класи теж мусять бути екземплярами класів. Клас класу називають *метакласом*.

17.2. Метакласи

Метаклас створюється автоматично кожного разу, коли створюється клас. У переважній більшості випадків не потрібно турбуватися чи навіть думати про метакласи. Проте кожного разу, коли ви використовуєте Системний оглядач для пошуку інформації про клас на *стороні класу*, не зайвим буде пригадати, що насправді ви переглядаєте інший клас. Клас і метаклас є окремими класами, попри те, що перший є екземпляром другого. Справді, окрема точка відрізняється від класу *Point*, і так само клас *Point* відрізняється від свого метакласу.

Щоб належно пояснити класи і метакласи, доповнимо правила з розділу 10 «Об'єктна модель Pharo» додатковими.

Правило 7. Кожен клас є екземпляром свого метакласу.

Правило 8. Ієрархія метакласів паралельна ієрархії класів.

Правило 9. Кожен клас наслідує від класів *Class* і *Behavior*.

Правило 10. Кожен метаклас є екземпляром класу *Metaclass*.

Правило 11. Метаклас класу *Metaclass* є екземпляром класу *Metaclass*.

Разом ці одинадцять простих правил становлять об'єктну модель Pharo.

Спершу на невеликому прикладі коротко повторимо п'ять перших правил. Після цього детальніше розглянемо нові правила, використовуючи той самий приклад.

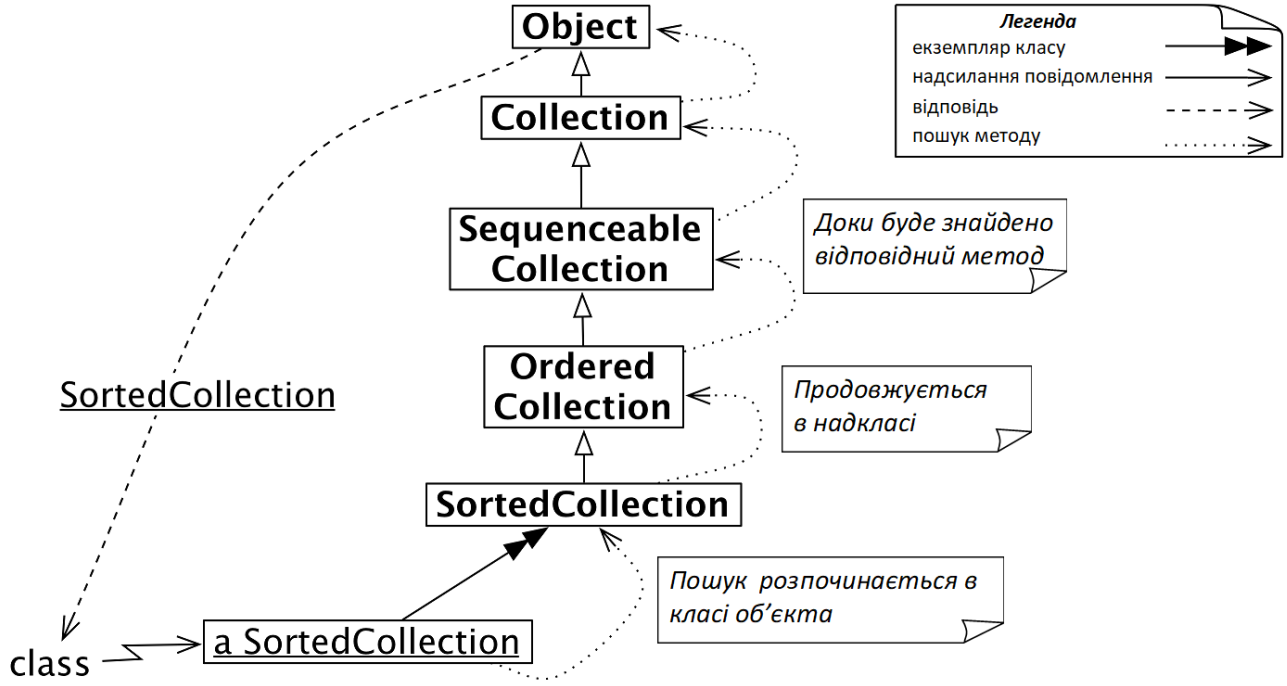

Рис. 17.1. Надсилання повідомлення *class* до впорядкованої колекції

17.3. Ще раз про об'єктну модель Pharo

Правило 1. Оскільки будь-що є об'єктом, то послідовна колекція у Pharo також є об'єктом.

```
OrderedCollection withAll: #(4 5 6 1 2 3)
>>> an OrderedCollection(4 5 6 1 2 3)
```

Правило 2. Кожен об'єкт є екземпляром класу. Послідовна колекція є екземпляром класу *OrderedCollection*.

```
(OrderedCollection withAll: #(4 5 6 1 2 3)) class
>>> OrderedCollection
```

Правило 3. У кожного класу є надклас. Надкласом класу *OrderedCollection* є *SequenceableCollection*, а надкласом *SequenceableCollection* – *Collection*.

```
OrderedCollection superclass
>>> SequenceableCollection
```

```
SequenceableCollection superclass
>>> Collection
```

```
Collection superclass
>>> Object
```

Правило 4. Усе відбувається через надсилання повідомлень, звідки випливає, що *withAll:* – повідомлення, надіслане до *OrderedCollection*, *class* – повідомлення екземпляра послідовної колекції, а *superclass* – повідомлення до класів *OrderedCollection*, *SequenceableCollection* і *Collection*. У кожному випадку отримувачем є об'єкт, бо будь-що є об'єктом, але деякі з них є також класами.

Правило 5. Пошук методу відбувається за ланцюжком наслідування, тому коли надсилають повідомлення *class* результатові обчислення виразу «*(OrderedCollection withAll: #(4 5 6 1 2 3)) asSortedCollection*», воно буде опрацьовано, коли відповідний метод буде знайдено в класі *Object*, як показано на рис.17.1.

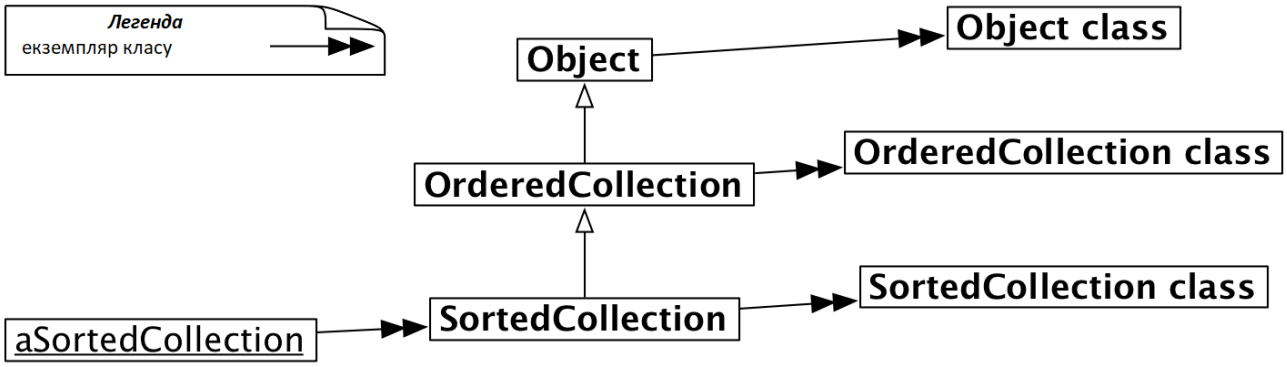

Рис. 17.2. Метакласи та надкласи (вибрані) *SortedCollection*

17.4. Кожен клас є екземпляром свого метакласу

Як вже згадували у параграфі 17.2, класи, чиї екземпляри самі є класами, називають *метакласами*. Інший термін використовують, щоб бути певним у точному розумінні, про що мовиться: про клас *Point*, чи про клас класу *Point*.

Метакласи є неявними

Метакласи створюються автоматично під час визначення класу. Кажуть, що вони *неявні*, оскільки програміст ніколи не повинен хвилюватися про них. Неявний метаклас створюється для *кожного* створеного класу, тому кожен метаклас має єдиний екземпляр.

На відміну від звичайних класів, кожного з яких потрібно назвати, метакласи залишаються анонімними. Але на анонімний метаклас завжди можна послатись через клас, що є його екземпляром. Наприклад, класом класу *SortedCollection* є *SortedCollection class*, а класом класу *Object* є *Object class*.

```
SortedCollection class
>>> SortedCollection class

Object class
>>> Object class
```

Насправді метакласи не повністю анонімні, їхнє ім'я випливає з їхнього єдиного екземпляра.

```
SortedCollection class name
>>> 'SortedCollection class'
```

```
Object class name
>>> 'Object class'
```

З рис. 17.2 видно, що кожен клас є екземпляром свого метакласу. Зазначимо, що на рисунку пропущено класи *SequenceableCollection* і *Collection* тільки для економії місця. Те, що їх не зображено, загалом не змінює суті.

17.5. Запити до метакласів

Той факт, що класи також є об'єктами, дає змогу легко звертатися до них за допомогою надсилання повідомлень.

```
OrderedCollection subclasses
>>> {ObjectFinalizerCollection. SortedCollection. WeakOrderedCollection.
     OCLiteralList. GLMMultiValue. RSGroup}
```

```
SortedCollection subclasses
>>> #()
```

```
SortedCollection allSuperclasses
>>> an OrderedCollection(OrderedCollection SequenceableCollection
     Collection Object ProtoObject)
```

```
SortedCollection instVarNames
>>> #(#sortBlock)
```

```
SortedCollection allInstVarNames
>>> #(#array #firstIndex #lastIndex #sortBlock)
```

```
SortedCollection selectors asSortedCollection
>>> a SortedCollection(#, #= #add: #addAll: #addFirst: #at:put: #collect:
     #copyEmpty #defaultSort:to: #flatCollect: #fromSton: #groupedBy:
     #indexForInserting: #insert:before: #intersection: #join: #median
     #reSort #sort: #sort:to: #sortBlock #sortBlock: #stonOn:)
```

17.6. Ієрархія метакласів паралельна ієрархії класів

Правило 8 стверджує, що надкласом метакласу не може бути звичайний клас: він змушений бути метакласом надкласу єдиного екземпляра цього метакласу. Метаклас класу *SortedCollection* наслідує метаклас *OrderedCollection* (надкласу *SortedCollection*).

```
SortedCollection class superclass
>>> OrderedCollection class
```

```
SortedCollection superclass class
>>> OrderedCollection class
```

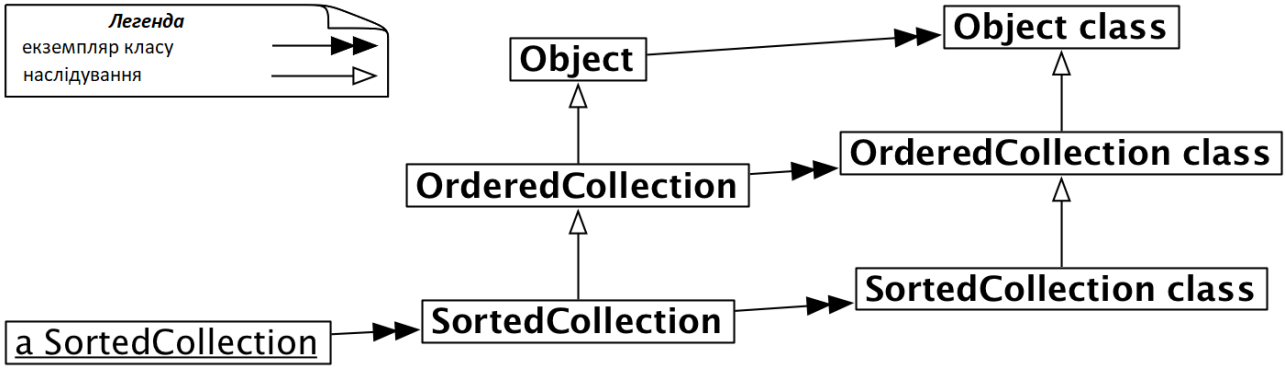

Рис. 17.3. Ієрархія метакласів (вибраних)паралельна ієрархії класів

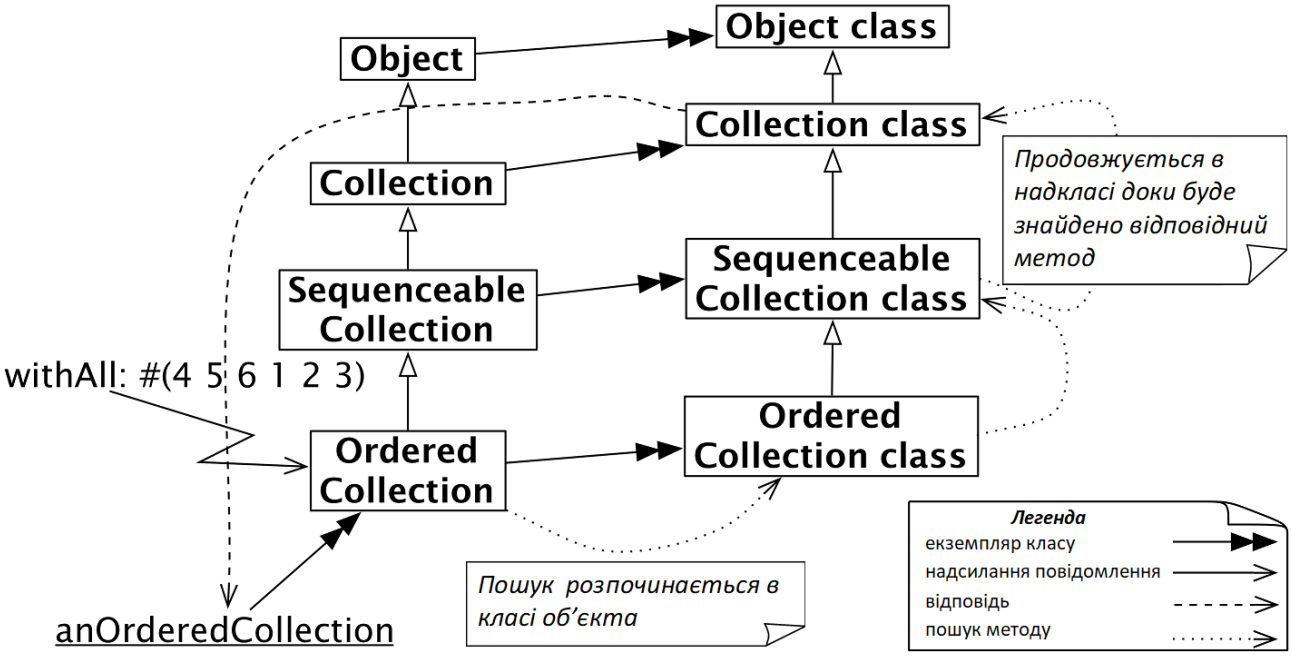

Рис. 17.4. Алгоритм пошуку методів класів такий самий, як методів звичайних об'єктів

Це те, що означає твердження «ієрархія метакласів паралельна ієрархії класів». На рис. 17.3 продемонстровано це на прикладі ієрархії *SortedCollection*.

```
SortedCollection class
>>> SortedCollection class
```

```
SortedCollection class superclass
>>> OrderedCollection class
```

```
SortedCollection class superclass superclass
>>> SequenceableCollection class
```

```
SortedCollection class superclass superclass superclass superclass
>>> Object class
```

17.7. Однорідність класів і об’єктів

Цікаво зупинитися на мить, щоб оглянути загальну картину, й усвідомити, що немає різниці між надсиланням повідомлення об’єкту та класу. В обох випадках пошук відповідного методу *розпочинається в класі отримувача* і *продовжується вгору по ланцюжку наслідування*.

Тому повідомлення, які надсилають класам, прямують ланцюжком наслідування метакласів. Розглянемо, наприклад, метод *withAll:*, реалізований на стороні класу *Collection*. Якщо надіслати повідомлення *withAll:* класові *OrderedCollection*, то пошук методу відбудеться так само, як і для будь-якого іншого повідомлення. Він розпочнеться з класу *OrderedCollection class* (пошук починається з класу отримувача, а отримувачем є *OrderedCollection*) і рухатиметься вгору по ієрархії метакласів, доки метод не буде знайдено у класі *Collection class* (див. рис. 17.4). Метод поверне новий екземпляр класу *OrderedCollection*.

```
OrderedCollection withAll: #(4 5 6 1 2 3)
>>> an OrderedCollection (4 5 6 1 2 3)
```

Єдиний спосіб пошуку методів

У Pharo існує єдиний спосіб пошуку методів. Класи є лише об’єктами і поводяться як інші об’єкти. Класи можуть створювати нові екземпляри лише завдяки тому, що вміють відповідати на повідомлення *new*, а метод опрацювання *new* вміє створювати нові екземпляри.

Зазвичай об’єкти «не класи» не розуміють цього повідомлення, однак ніщо не заважає додати метод *new* у «не метаклас», якщо на це є вагома причина.

17.8. Інспектування об’єктів і класів

Оскільки класи є об’єктами, то їх також можна інспектувати.

Проінспектуйте два об’єкти: *OrderedCollection withAll: #(4 5 6 1 2 3)* і *OrderedCollection*.

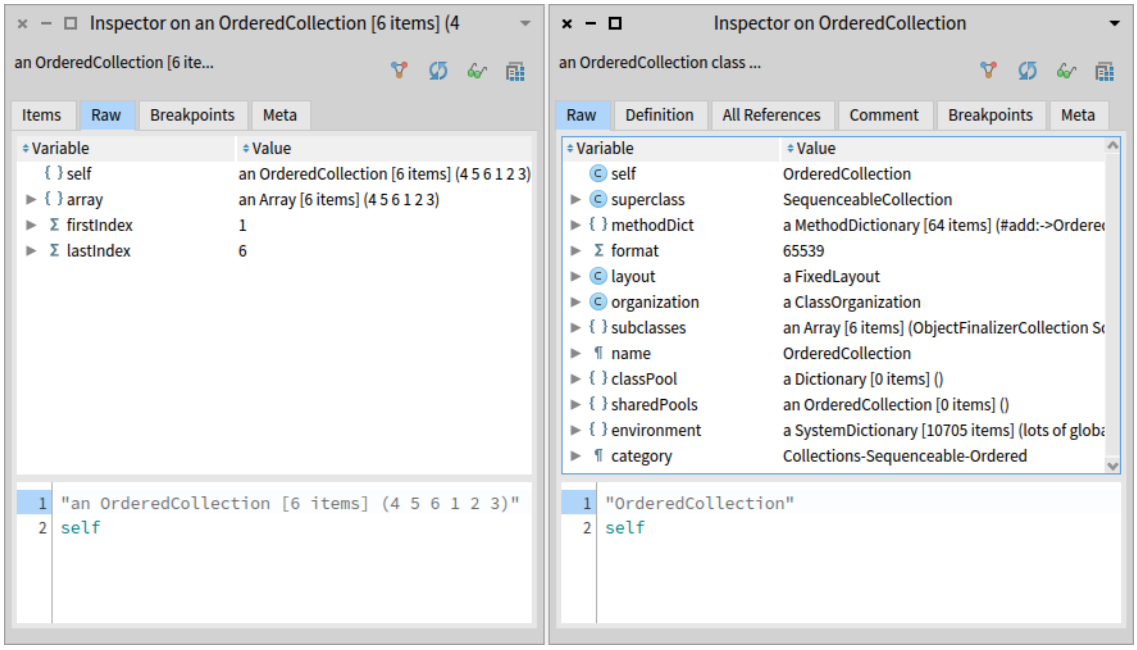

Рис. 17.5. Класи також є об’єктами

Зауважимо, що в першому випадку маємо справу з *екземпляром* класу *OrderedCollection*, а в другому – з самим класом *OrderedCollection*. Різницю можна не помітити відразу, бо заголовок інспектора містить ім'я *класу* інспектованого об'єкта, але у випадку екземпляра наявний також артикль «*an*» або «*a*».

Інспектор з класом *OrderedCollection* дає змогу побачити надклас, змінні екземпляра, словник методів тощо метакласу *OrderedCollection*, як показано на рис. 17.5.

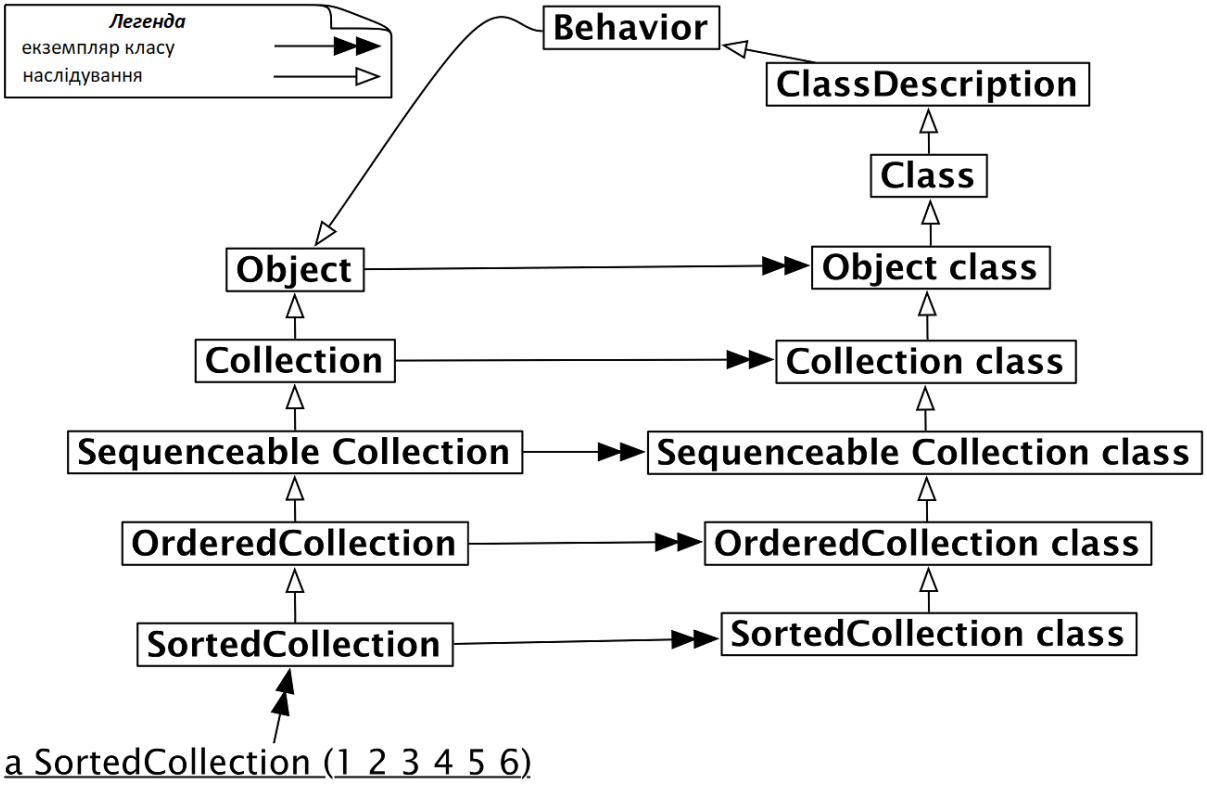

Рис. 17.6. Метакласи наслідують *Class* і *Behavior*

17.9. Кожен метаклас наслідує класи *Class* і *Behavior*

Кожен метаклас є різновидом класу (таким, що має єдиний екземпляр), тому наслідує клас *Class*. Так само *Class* наслідує свої надкласи *ClassDescription* і *Behavior*. Оскільки будь-що у Pharo є об'єктом, то всі ці класи у підсумку наслідують клас Object. Повну картину можна бачити на рис. 17.6.

Де визначено *new*?

Щоб зрозуміти важливість того факту, що метакласи наслідують *Class* і *Behavior*, достатньо подумати, де визначено *new*, і як його знаходить алгоритм пошуку методів.

Коли класові надсилають повідомлення *new*, відповідний метод шукається у ланцюжку метакласів і, зрештою, у їхніх надкласах *Class*, *ClassDescription* і *Behavior*, як зображено на рис. 17.7.

Коли повідомлення *new* надсилають класові *SortedCollection*, пошук розпочинається в метакласі *SortedCollection class* і просувається догори ланцюжком наслідування. Нагадуємо, що алгоритм пошуку такий самий, як для будь-якого об'єкта.

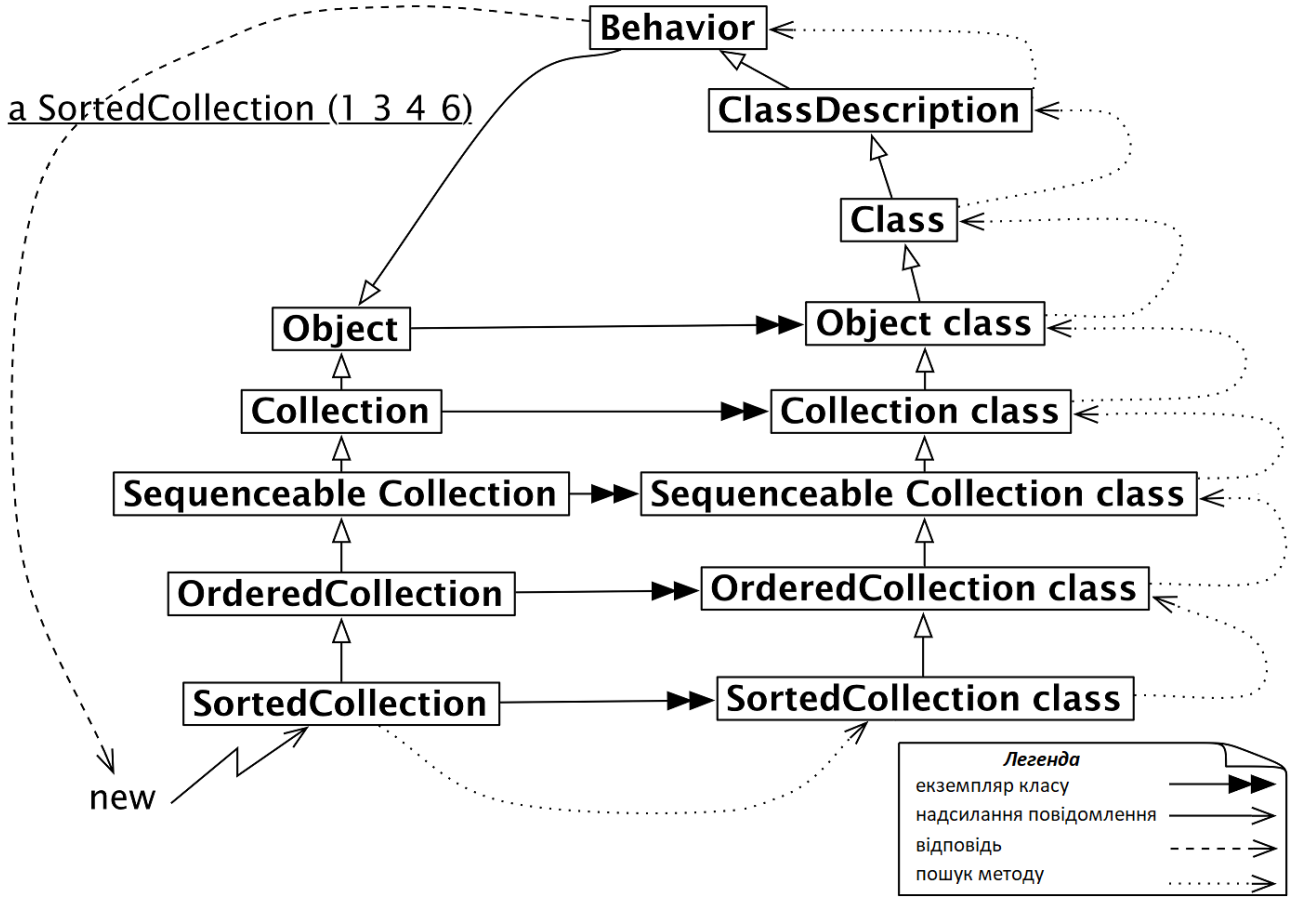

Рис. 17.7. *new* – звичайне повідомлення, відповідний метод шукається в ланцюжку метакласів

Питання *«Де визначено new?»* має вирішальне значення. Вперше *new* визначено в класі *Behavior* і може бути перевизначено у підкласах, включно з довільними метакласами визначених класів, якщо є така потреба.

Тоді, коли повідомлення *new* надіслано класові, пошук стартує, як годиться, з метакласу цього класу і продовжується по ланцюжку надкласів до класу *Behavior*, якщо метод не було перевизначено десь на цьому шляху.

Зазначимо, що результат надсилання *SortedCollection new* є екземпляром *SortedCollection*, а *не Behavior*, хоча метод і міститься в класі *Behavior*!

Метод *new* завжди повертає екземпляр *self*, тобто класу, який отримав повідомлення, навіть якщо воно реалізоване в іншому класі.

```
SortedCollection new class
>>> SortedCollection "не Behavior!"
```

Поширена помилка

Початківці часто допускають таку помилку: шукають *new* у надкласі отримувача. Те саме стосується і *new:*, стандартного повідомлення для створення об'єкта заданого розміру. Наприклад, *Array new: 4* створює масив з 4 елементів. Ви не знайдете цього методу ні в класі *Array*, ні в жодному з його надкласів. Замість цього вам потрібно шукати його в класі *Array class* і його надкласах, оскільки пошук розпочинається саме з нього (див. рис. 17.7).

Методи *new* і *new:* визначені в метакласах, бо їх виконують у відповідь на повідомлення, надіслане класові.

Додамо також, що клас є об'єктом, тому може бути отримувачем повідомлень, методи для яких визначені в класі *Object*. Коли класові *Point* надсилають повідомлення *class* або *error:*, алгоритм пошуку приведе ланцюжком наслідування метакласів (через *Point class*, *Object class*, …) аж до *Object*.

17.10. Призначення класів *Behavior*, *ClassDescription* і *Class*

Behavior

Behavior підтримує мінімальний набір змінних, необхідний для об'єктів, які мають екземпляри; сюди входять: посилання на надклас, словник методів і формат класу. Формат класу – це ціле число, що позначає відмінності у способі виділення пам'яті для екземплярів: вказівник чи не вказівник, компактне розміщення чи розгорнуте, а також базовий розмір екземплярів. Клас *Behavior* наслідує *Object*, тому і він, і всі його підкласи можуть поводитись як об'єкти.

Behavior також є базовим інтерфейсом для компілятора. Він містить методи, потрібні для створення словника методів, компілювання методів, створення екземплярів (наприклад, *new*, *basicNew*, *new*:, *basicNew*:), управління ієрархією класів (наприклад, *superclass:*, *addSubclass:*), доступу до методів (наприклад, *selectors*, *allSelectors*, *compiledMethodAt:*), доступу до екземплярів і змінних (наприклад, *allInstances*, *instVarNames*, …), доступу до ієрархії класів (наприклад, *superclass*, *subclasses*) і запитів (наприклад, *hasMethods*, *includesSelector:*, *canUnderstand*:, *inheritsFrom*:, *isVariable*).

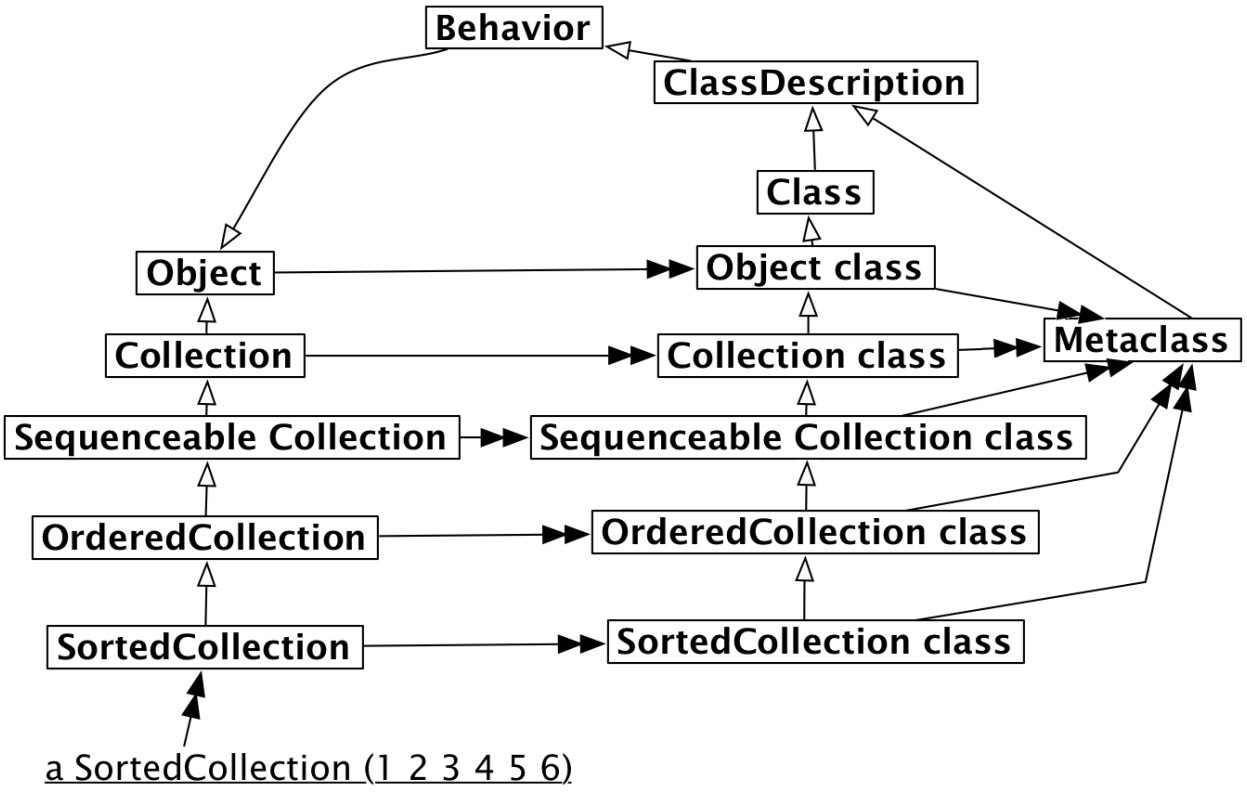

Рис. 17.8. Кожен метаклас – екземпляр класу *Metaclass*

ClassDescription

ClassDescription – абстрактний клас, що надає засоби, потрібні двом його безпосереднім підкласам, *Class* і *Metaclass*. *ClassDescription* розширює базову функціональність *Behavior*: іменовані змінні екземпляра, категоризація методів у протоколи, підтримка наборів змін та їхнє протоколювання, більшість механізмів, потрібних для внесення змін.

Class

Class представляє загальну поведінку всіх класів. Він підтримує ім'я класу, методи компіляції, сховище методів та змінні екземпляра. Він надає конкретне представлення іменам змінних класів і змінних спільного пулу (*addClassVarName*:, *addSharedPool*:, *initialize*). Оскільки метаклас є класом для свого єдиного екземпляра (не метакласу), то всі метакласи, зрештою, наслідують від класу *Class* (див. рис. 17.6–17.9).

17.11. Кожен метаклас є екземпляром класу *Metaclass*

Наступне питання стосується метакласів. Оскільки вони теж об'єкти, то мусять бути екземплярами якогось класу, але якого? Метакласи – також об'єкти, екземпляри класу *Metaclass*, як зображено на рис. 17.8. Екземпляри класу *Metaclass* – анонімні класи, кожен з яких має точно один екземпляр, який є класом.

Metaclass визначає загальну поведінку метакласів. Він містить методи для створення екземплярів (*subclassOf:*), створення ініціалізованого екземпляра – єдиного екземпляра метакласу, ініціалізації змінних класу, екземплярів метакласу, компіляції методів та отримання інформації про клас (зв'язки наслідування, змінні екземплярів тощо).

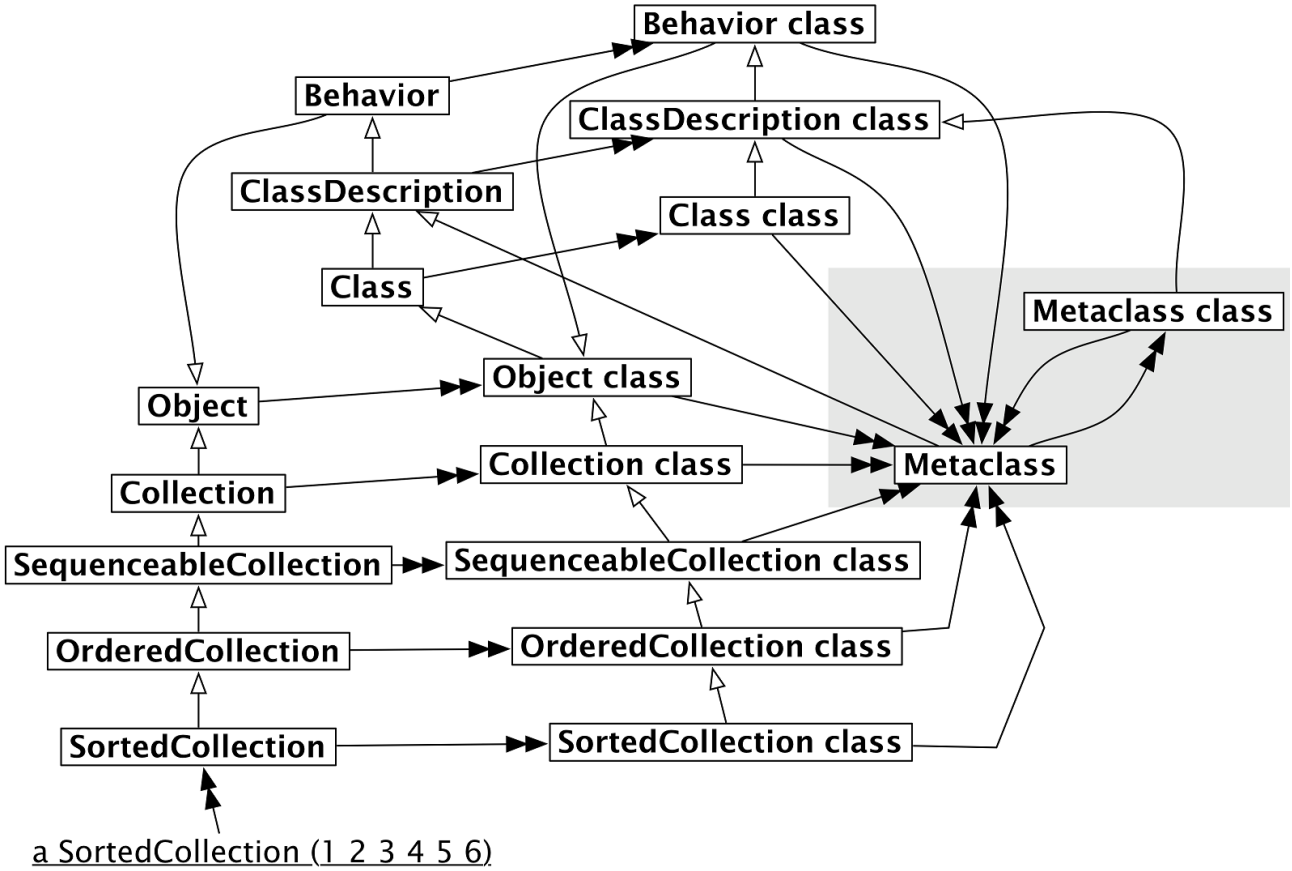

Рис. 17.9. Усі метакласи є екземплярами класу *Metaclass*, навіть метаклас класу *Metaclass*

17.12. Метаклас класу *Metaclass* є екземпляром класу *Metaclass*

Останнє питання, на яке потрібно дати відповідь: а що є класом класу *Metaclass* class? Відповідь проста – це метаклас, тому він має бути екземпляром класу *Metaclass*, як і всі інші метакласи в системі (див. рис. 17.9).

З рис. 17.9 видно, що всі метакласи є екземплярами класу *Metaclass* включно з метакласом самого класу *Metaclass*. Порівнявши рисунки 17.8 і 17.9, можна побачити, що ієрархія метакласів ідеально віддзеркалює ієрархію класів на всьому шляху аж до *Object class*.

Наведений нижче приклад ілюструє, як можна дослідити ієрархію класів, щоб продемонструвати правильність рис. 17.9. (Правду кажучи, ми дещо прибрехали – *Object class superclass >>> ProtoObject class*, не *Class*. У Pharo потрібно піднятись на один рівень вище, щоб досягти класу *Class*).

```
Collection superclass
>>> Object


Collection class superclass
>>> Object class


Object class superclass superclass
>>> Class


Class superclass
>>> ClassDescription


ClassDescription superclass
>>> Behavior


Behavior superclass
>>> Object


"Класом кожного метакласу є Metaclass"
Collection class class
>>> Metaclass

Object class class
>>> Metaclass

Behavior class class
>>> Metaclass

Metaclass class class
>>> Metaclass


"Metaclass є особливим різновидом класу"
Metaclass superclass
>>> ClassDescription
```

17.13. Підсумки розділу

Цей розділ дав змогу заглянути в глибину однорідної об'єктної моделі, і ретельніше пояснив, як організовані класи. Якщо ви розгубилися чи заплуталися, то пригадайте, що ключем є надсилання повідомлень: метод потрібно шукати у класі отримувача. Це працює для *будь-якого* отримувача. Якщо метод не знайдено в класі отримувача, то пошук продовжується у його надкласах.

- Кожен клас є екземпляром свого метакласу. Метакласи неявні. Метаклас створюється автоматично кожного разу, коли створюється клас, який є його єдиним екземпляром. Простіше кажучи, метаклас – це клас, єдиним екземпляром якого є клас.
- Ієрархія метакласів паралельна ієрархії класів. Пошук методу класу такий самий, як пошук методу звичайного об'єкта, і прямує ланцюжком наслідування метакласів.
- Кожен метаклас наслідує *Class* і *Behavior*. Метакласи – також класи, тому вони наслідують *Class*. *Behavior* реалізує поведінку, спільну для всіх сутностей, що мають екземпляри.
- Кожен метаклас є екземпляром класу *Metaclass*. *ClassDescription* підтримує все, що є спільного в класах *Class* і *Metaclass*.
- Метаклас класу *Metaclass* є екземпляром класу *Metaclass*. Відношення «*є екземпляром*» утворює замкнуте коло, тому клас класу *Metaclass class* – це *Metaclass*.

Розділ 18
Рефлексія

Pharo – рефлексивна мова програмування. Рефлексія передбачає можливості для того, щоб і досліджувати структуру та поведінку програмної системи під час її виконання, і змінювати їх. Засоби дослідження часто називають *інтроспекцією* або самоаналізом, а можливості змін – *адаптивністю* або заступництвом[18]. У цьому розділі представлено кілька аспектів щодо засобів інтроспекції Pharo: як отримати доступ і змінити значення змінної екземпляра, як переміщуватися системою або як виконувати перехресні посилання. Також описано деякі аспекти поведінкової рефлексії, тобто, можливості змінити систему та розширити її.

18.1. Суть рефлексії

Рефлексія втілює ідею, що програма здатна *аналізувати* свої власні структуру та хід виконання. З рефлексією внутрішні механізми, які підтримують виконання програм (класи, методи, стек викликів), доступні для розробника так само, як і його звичайні програми. У Pharo це означає, що ці внутрішні механізми описані як звичайні об'єкти, і що розробник може надсилати їм повідомлення. Об'єкти, які підтримують виконання програм, часто називають *метаоб'єктами*, щоб наголосити на тому факті, що вони перебувають на іншому рівні, ніж звичайні об'єкти.

Точніше кажучи, *метаоб'єкти* програмної системи на етапі виконання *втілюються* в звичайні об'єкти, які можна запитувати та інспектувати. Метаоб'єкти Pharo – це класи, метакласи, словники методів, скомпільовані методи, а також стек часу виконання, процеси тощо. Таку форму рефлексії також називають *інтроспекцією*, її підтримує багато сучасних мов програмування (чому активно сприяв Smalltalk-80, попередник Pharo).

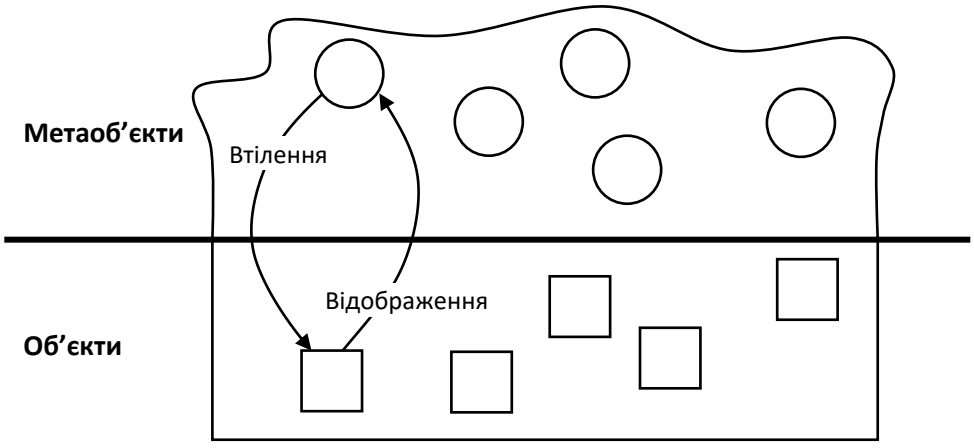

Рис. 18.1. Втілення та відображення

З іншого боку, у Pharo можна модифікувати втілені метаоб'єкти та *відобразити* ці зміни назад у виконувану програму (рис. 18.1). Така здатність називається *заступництвом* і підтримується в здебільшого динамічними мовами програмування, і лише дуже

[18] В оригіналі – *introspection* and *intercession* (*прим. – Ярошко С.*).

обмежено – статичними мовами. Тому, зверніть увагу, якщо люди кажуть, що Java рефлексивна мова, то насправді це інтроспективна, а не рефлексивна мова.

Програма, яка керує іншими програмами (або навіть сама собою), є метапрограмою. Щоб бути рефлексивною, мова програмування має підтримувати і самоаналіз, і заступництво. Інтроспекція – це здатність *досліджувати* структури даних, які визначають мову: об'єкти, класи, методи та стек виконання. Адаптивність – це здатність *модифікувати* ці структури, іншими словами, змінювати семантику мови та поведінку програми зсередини самої програми. *Структурна рефлексія* стосується дослідження та модифікації структур середовища виконання, а *поведінкова рефлексія* стосується зміни інтерпретації цих структур.

У розділі йдеться головно про структурну рефлексію. Розглянуто багато практичних прикладів, які ілюструють, як Pharo підтримує самоаналіз і метапрограмування.

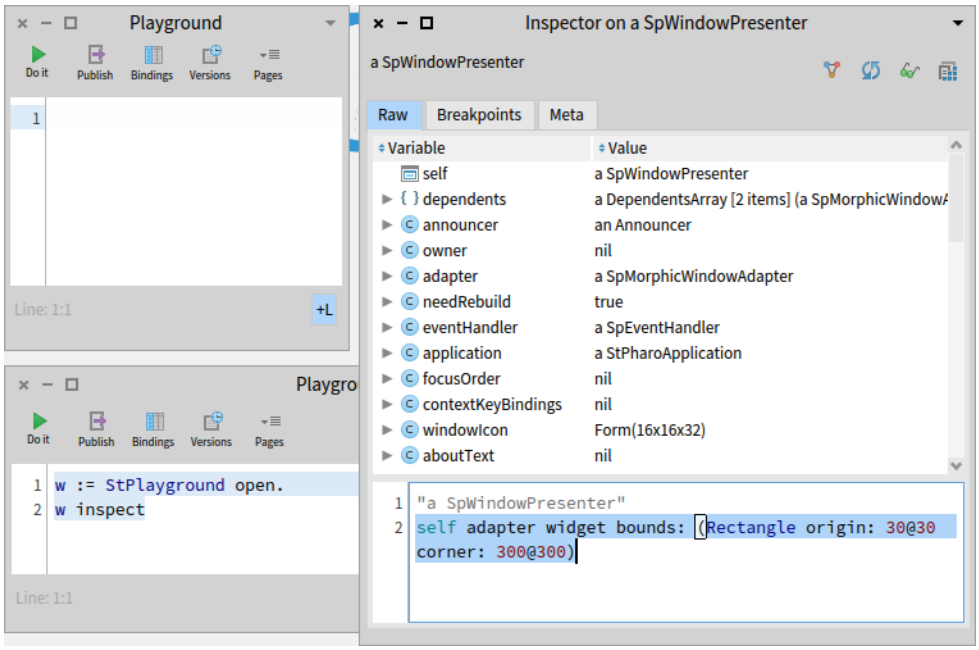

Рис. 18.2. Інспектування екземпляра *StPlayground*

18.2. Інтроспекція

За допомогою інспектора можна переглядати об'єкт, змінювати значення його змінних екземпляра та навіть надсилати йому повідомлення.

Виконайте в Робочому вікні наведений код.

```
w := StPlayground open.
w inspect
```

Він відкриє ще одне Робоче вікно й інспектор. Інспектор показує внутрішній стан нового вікна як список його змінних екземпляра: у лівому стовпці – імена змінних (*announcer*, *owner*, ...), а в правому – значення змінної навпроти відповідного імені.

Тепер виберіть вікно інспектора і клацніть на його панелі редагування коду (вона містить коментар угорі). Введіть у ній повідомлення «*self adapter widget bounds: (Rectangle origin: 30@30 corner: 300@300)*», як зображено на рис. 18.2, і виконайте його командою «*Do it*», як у Робочому вікні.

Ви побачите, що створене Робоче вікно відразу переміститься та змінить розмір.

18.3. Доступ до змінних екземпляра

Як працює інспектор? У Pharo всі змінні екземпляра захищені. Теоретично неможливо доступитися до них з іншого об'єкта, якщо в класі не визначені методи доступу. На практиці інспектор отримує доступ до змінних екземпляра без таких методів, бо він використовує рефлексивні можливості Pharo. Класи визначають або іменовані змінні екземпляра, або індексовані. Для доступу до них інспектор використовує визначені в класі *Object* методи: *instVarAt: index* та *instVarNamed: aString* – для отримання значення змінної екземпляра з номером *index* або з ідентифікатором *aString*, відповідно. Так само, щоб призначити нові значення цим змінним екземпляра, він використовує *instVarAt: index put: value* та *instVarNamed: aString put: value*.

Продовжимо експерименти з Робочими вікнами, створеними в попередньому параграфі. Доповніть код у першому вікні наведеним нижче рядком і виконайте його.

```
w adapter widget
   instVarNamed:'bounds' put: (Rectangle origin: 150@30 corner: 450@300)
```

Клацніть на другому вікні, щоб актуалізувати оновлення та побачити результат. Щойно ви змінили стан об'єкта без допомоги методів доступу.

Змінні екземпляра

Метод *allInstVarNames* повертає імена всіх змінних екземпляра заданого класу.

```
StPlayground allInstVarNames
>>> #(#dependents #announcer #owner #adapter #needRebuild #eventHandler
    #application #focusOrder #contextKeyBindings #windowIcon #aboutText
    #askOkToClose #titleHolder #additionalSubpresentersMap #layout
    #visible #extent #styles #millerList #model #lastPageSelectedTabName
    #withHeaderBar)
```

Наступний фрагмент коду показує, як зібрати значення змінних довільного екземпляра класу *StPlayground*. Виконайте його командою «*Print it*».

```
w := StPlayground someInstance.
w class allInstVarNames collect: [:each | each -> (w instVarNamed: each)]
```

Подібним способом можна відібрати екземпляри класу, які мають певні властивості, перебираючи їх за допомогою ітератора *select:*. Наприклад, щоб отримати всі дочірні об'єкти морфи *world* (кореневої морфи відображених графічних елементів), спробуйте виконати наведений нижче вираз.

```
Morph allSubInstances
   select: [ :each |
      | own |
      own := (each instVarNamed: 'owner').
      own isNotNil and: [ own isWorldMorph ] ]
```

18.4. Про застосування рефлексії

Такі вирази зручні для налагодження коду або створення інструментів розробки, але використання їх для розробки звичайних програм є поганою ідеєю: рефлексивні методи порушують інкапсуляцію об'єктів. Вони порушують правила доступу. Вони роблять ваш код набагато складнішим для розуміння та обслуговування. Розумна теза, яку варто запам'ятати: «Разом із суперсилою приходить супервідповідальність». Тому не варто застосовувати рефлексію тільки тому, що це можна зробити.

18.5. Про примітиви

Обидва методи: і *instVarAt:*, і *instVarAt:put:* – примітивні, тобто реалізовані як вбудовані операції віртуальної машини Pharo. Якщо ви ознайомитеся з їхнім кодом, то побачите прагму *<primitive: N>*, де *N* – ціле число. Вона позначає метод, який потрібно опрацьовувати не так, як інші.

```
Object >> instVarAt: index
  "Primitive. Answer a fixed variable in an object. ..."

  <primitive: 173 error: ec>
  self primitiveFailed
```

Будь-який код Pharo розташований після оголошення примітиву виконується лише тоді, коли примітив зазнає невдачі. У цьому конкретному випадку не існує способу реалізувати бажану дію поза примітивом, тому метод запускає виняток.

Багато методів реалізовано примітивами віртуальної машини для швидшого виконання. Наприклад, деякі арифметичні операції над *SmallInteger*.

```
* aNumber
"Primitive. Multiply the receiver by 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: 9>
^ super * aNumber
```

Якщо віртуальна машина не вміє обробляти тип аргументу, то примітив зазнає невдачі, і керування перейде до коду Pharo. Код методів з примітивами можна змінювати, як і будь-яких інших, але варто пам'ятати, що це може бути ризикованою справою для стабільності цілої системи Pharo.

18.6. Опитування класів та інтерфейсів

Інструменти розробки в Pharo: системний оглядач, налагоджувач, інспектор та інші – всі використовують згадані рефлексивні функції.

Ось ще кілька повідомлень, які можуть бути корисними для створення інструментів розробки.

Повідомлення *isKindOf: aClass* повертає *true*, якщо отримувач є екземпляром *aClass* або одного з його підкласів.

```
1.5 class
>>> SmallFloat64

1.5 isKindOf: Float
>>> true

1.5 isKindOf: Number
>>> true

1.5 isKindOf: Integer
>>> false
```

Повідомлення *respondsTo: aSymbol* повертає *true*, якщо отримувачу доступний метод з селектором *aSymbol*.

```
1.5 respondsTo: #floor
>>> true "бо клас Number реалізує метод floor"

1.5 floor
>>> 1

Exception respondsTo: #,
>>> true "класи винятків можна групувати"
```

Стережіться!

Хоча рефлективні методи особливо корисні для створення інструментів розробки, зазвичай вони не підходять для типових програм. Запит до об'єкта щодо його належності до конкретного класу або щодо розуміння певних повідомлень є типовими ознаками проблем проєктування, бо вони порушують принцип інкапсуляції. Водночас інструменти розробки не є звичайними програмами, бо є частиною самого програмного забезпечення. Саме тому вони мають право глибоко досліджувати внутрішні деталі коду.

18.7. Прості метрики коду

Давайте подивимося, як можна використовувати функції інтроспекції Pharo для швидкого обчислення деяких метрик коду. Метрики коду вимірюють такі аспекти – глибина ієрархії успадкування, кількість безпосередніх або опосередкованих підкласів, кількість методів або змінних екземпляра у кожному класі, або кількість локально визначених методів чи змінних екземпляра. Нижче наведено кілька показників для класу *Morph*, який є надкласом усіх графічних об'єктів у Pharo. Вони засвідчують, що це дуже великий клас, і що він є коренем величезної ієрархії. Такі показники наштовхують на думку, що він потребує деякого рефакторингу!

```
"глибина наслідування"
Morph allSuperclasses size.
>>> 2

"кількість методів"
Morph allSelectors size.
>>> 1436

"кількість змінних екземпляра"
Morph allInstVarNames size.
>>> 6
```

```
"кількість нових методів"
Morph selectors size.
>>> 931

"кількість нових змінних"
Morph instVarNames size.
>>> 6

"безпосередні підкласи"
Morph subclasses size.
>>> 73

"загальна кількість підкласів"
Morph allSubclasses size.
>>> 448

"кількість рядків коду"
Morph linesOfCode.
>>> 5088
```

Однією з найцікавіших метрик в області об'єктно-орієнтованих мов є кількість методів, які розширюють методи, успадковані від надкласу. Це інформує нас про відношення між класом і його надкласами. У наступних параграфах ми з'ясуємо, як використати наші знання про структуру часу виконання, щоб відповісти на такі запитання.

18.8. Дослідження екземплярів

У Pharo все є об'єктом. Зокрема, класи – це об'єкти, які надають корисні функції для відшукання їхніх екземплярів. Більшість повідомлень, які ми зараз розглянемо, реалізовані в *Behavior*, тому їх розуміють усі класи.

Наприклад, можна отримати випадковий екземпляр певного класу, надіславши йому повідомлення *someInstance*.

```
Point someInstance
>>> (-1@-1)
```

Також можна зібрати всі екземпляри за допомогою *allInstances* або довідатися кількість активних екземплярів у пам'яті за допомогою *instanceCount*.

```
ByteString allInstances
>>> #('collection' 'position' ...)

ByteString instanceCount
>>> 54837

String allSubInstances size
>>> 147289
```

Згадані методи отримують доступ до екземплярів, які зберігаються всередині методів. Оскільки Pharo має близько 130 000 методів, то такі цифри не видаються божевільними.

18.9. Від методів до змінних екземпляра

Описані нижче функції можуть бути дуже корисними для налагодження програми, тому що можна попросити клас перерахувати ті його методи, які демонструють певні властивості. Ось ще кілька цікавих і корисних методів для дослідження коду через рефлексію.

whichSelectorsAccess: повертає список усіх селекторів методів, які читають або записують змінну екземпляра, ім'я якої задано аргументом.

whichSelectorsStoreInto: повертає селектори методів, які змінюють значення змінної екземпляра.

whichSelectorsReferTo: повертає селектори методів, які надсилають задане аргументом повідомлення.

```
Point whichSelectorsAccess: 'x'
>>> #(#octantOf: #roundDownTo: #+ #asIntegerPoint #transposed ...)

Point whichSelectorsStoreInto: 'x'
>>> #(#fromSton: #setX:setY: #setR:degrees: #bitShiftPoint:)

Point whichSelectorsReferTo: #+
>>> #(#+)
```

Наведені нижче методи переглядають ланцюжок наслідування.

whichClassIncludesSelector: повертає надклас, який реалізує задане аргументом повідомлення.

unreferencedInstanceVariables повертає список змінних екземпляра, які не використовуються ні в класі отримувача, ні в будь-якому з його підкласів.

```
Rectangle whichClassIncludesSelector: #inspect
>>> Object

Rectangle unreferencedInstanceVariables
>>> #()
```

18.10. Про *SystemNavigation*

SystemNavigation – це фасад, який підтримує різні корисні методи для запитів до вихідного коду системи і відшукання методів за заданим критерієм. *SystemNavigation default* повертає екземпляр, який можна використовувати для навігації системою.

```
SystemNavigation default allClassesImplementing: #yourself
>>> an OrderedCollection(Object)
```

Призначення наведених нижче повідомлень описано їхніми селекторами.

```
SystemNavigation default allSentMessages size
>>>43985

(SystemNavigation default allUnsentMessagesIn: Object selectors) size
>>> 44
```

```
SystemNavigation default allUnimplementedCalls size
>>> 335
```

Зауважимо, що реалізовані методи, для яких не надіслані повідомлення, не обов'язково надлишкові, оскільки повідомлення можуть бути надіслані неявно (наприклад, за допомогою *perform:*). Проблематичніші повідомлення, надіслані, але не реалізовані, бо методи, які надсилають ці повідомлення, не будуть виконані. Це може бути ознакою незавершеної реалізації, застарілих API або відсутності бібліотек. Вони також часто трапляються в тестах щодо реалізації інструментів.

Point allCallsOn повертає колекцію всіх випадків явного надсилання повідомлень класу *Point*. Екземпляри такої колекції мають вигляд «*Ім'яКласу>>селекторМетоду*».

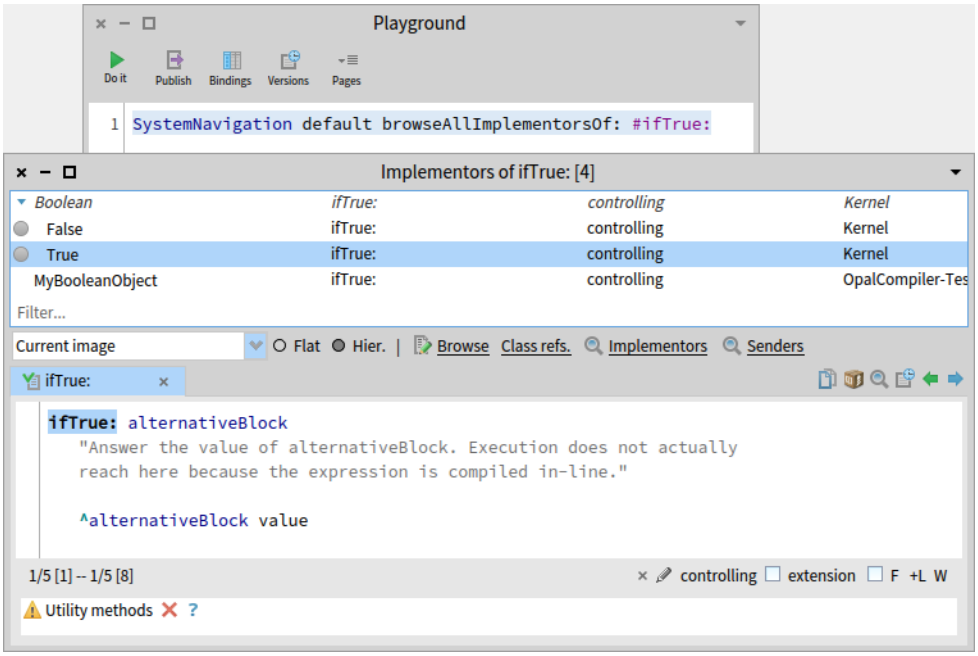

Рис. 18.3. Відшукання всіх реалізацій методу *ifTrue:*

Усі згадані методи інтегровані в середовище програмування Pharo, зокрема, в Оглядач коду. Як вже згадували раніше, існують зручні комбінації клавіш для перегляду всіх реалізаторів [*Cmd-M*] і відправників [*Cmd-N*] кожного повідомлення. Можливо, не так широко відомо, що багато таких вбудованих запитів, реалізовано у вигляді методів класу *SystemNavigation* у протоколі *query*. Наприклад, усі реалізатори повідомлення *ifTrue:* можна переглянути програмно, надіславши відповідне повідомлення (див. рис. 18.3).

```
SystemNavigation default browseAllImplementorsOf: #ifTrue:
```

Особливо корисними є методи *browseAllSelect:* і *browseMethodsWithSourceString:matchCase:*. Ось два різні способи перегляду всіх методів у системі, які надсилають повідомлення до *super* (перший спосіб – це скоріше груба сила, другий – кращий, усуває деякі помилкові спрацьовування).

```
SystemNavigation default
   browseMethodsWithSourceString: 'super'
   matchCase: true

SystemNavigation default
   browseAllSelect: [:method | method sendsToSuper ]
```

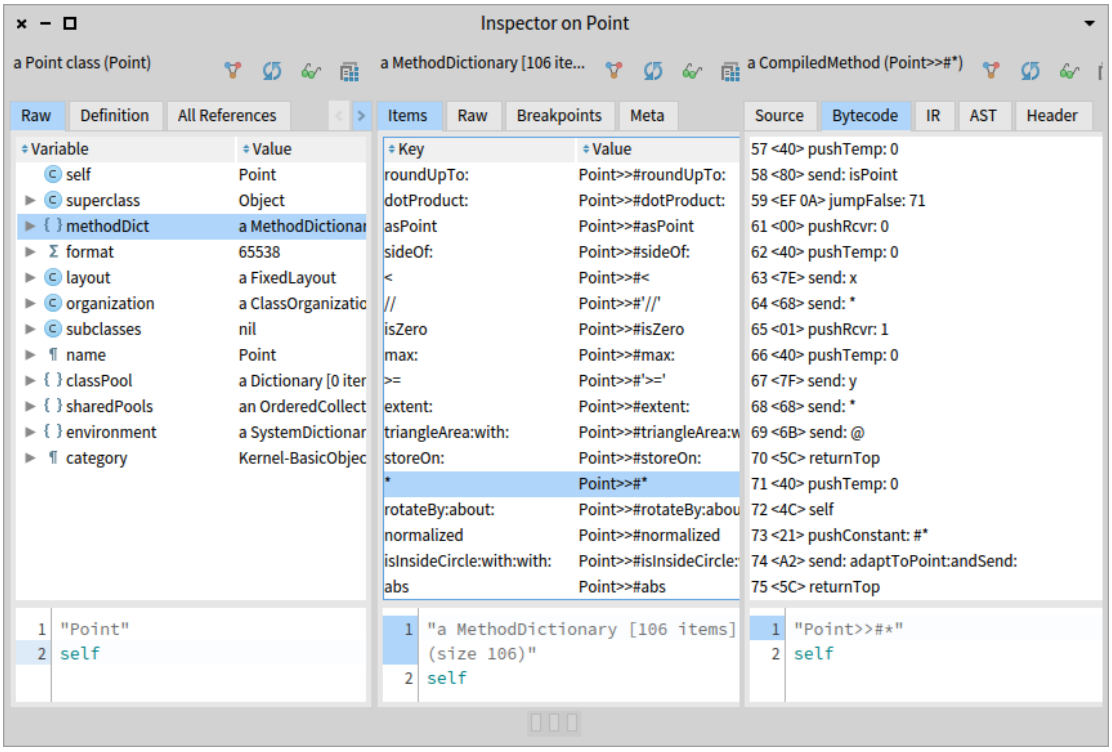

Рис. 18.4. Інспектування класу *Point* і байт-коду його методу #*

18.11. Класи, словники методів і методи

Класи є об'єктами, тому їх можна інспектувати або досліджувати, як і будь-який інший об'єкт.

Виконайте вираз *Point inspect*. На рис. 18.4 інспектор показує структуру класу *Point*. Видно, що клас зберігає свої методи в словнику, ключами якого є селектори. Селектор *#** вказує на декомпільований байт-код методу *Point >> **.

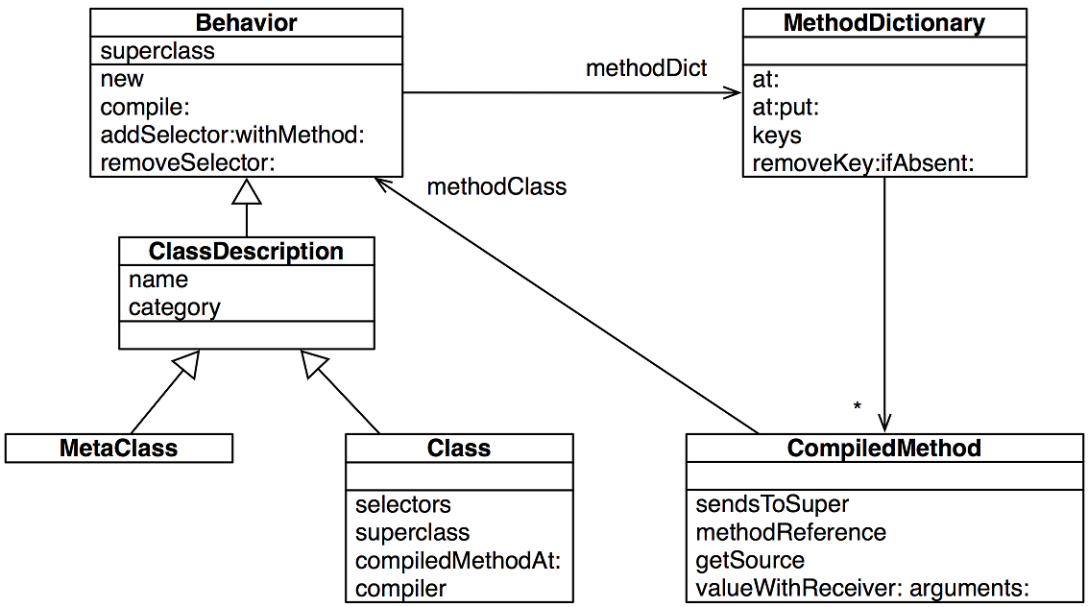

Рис. 18.5. Класи, словники методів і компільовані методи

Розглянемо зв'язок між класами та методами. На рис. 18.5 видно, що класи та метакласи мають спільний надклас *Behavior*. Тут визначено *new* серед інших ключових методів класів. Кожен клас має словник методів, який відображає селектори методів на

скомпільовані методи. Кожен скомпільований метод знає клас, якому він належить. На рис. 18.4 навіть можна побачити декомпільовані байт-коди методу.

Зв'язки між класами та методами можна використовувати для формування запитів щодо системи. Наприклад, щоб дізнатися, які методи класу не перевизначають методи надкласу, можна перейти від класу до словника методів, як зображено нижче, і вибрати потрібне.

```
| aClass |
aClass := SmallInteger.
aClass methodDict keys select: [ :aMethod |
   (aClass superclass canUnderstand: aMethod) not ]
>>> an IdentitySet(#threeDigitName #printStringBase:nDigits: ...)
```

Скомпільований метод – об'єкт, який не тільки зберігає байт-код методу, а також надає численні корисні методи для запитів до системи. Одним із таких методів є *isAbstract* (який повідомляє, чи надсилає метод повідомлення *subclassResponsibility*). Його можна використати для ідентифікації всіх абстрактних методів класу.

```
| aClass |
aClass := Number.
aClass methodDict keys select: [ :aMethod |
   (aClass >> aMethod) isAbstract ]
>>> #(#* #asFloat #storeOn:base: #printOn:base: #+ #round: #/ #-
    #adaptToInteger:andSend: #nthRoot: #sqrt #adaptToFraction:andSend:)
```

Зверніть увагу, що цей код надсилає класові повідомлення >>, щоб за заданим селектором отримати скомпільований метод.

Щоб переглянути всі надсилання повідомлень до *super* в межах певної ієрархії, наприклад, в межах ієрархії колекцій, можна сформувати складніший запит. Для зручного перегляду результатів повідомлення *browseMessageList:name:* відкриє спеціальне вікно (екземпляр класу *ClyOldMessageBrowserAdapter*).

```
class := Collection.
SystemNavigation default
   browseMessageList: (class withAllSubclasses gather: [ :each |
      each methodDict associations
         select: [:assoc | assoc value sendsToSuper ]
         thenCollect: [:assoc |
            RGMethodDefinition realClass: each selector: assoc key]])
   name: 'Supersends of ', class name, ' and its subclasses'
```

Зверніть увагу на те, як перейшли від класів до словників методів, а потім до скомпільованих методів, щоб відібрати ті з них, які нас цікавлять. RGMethodDefinition – це легкий проксі-сервер для скомпільованого методу, який використовується багатьма інструментами. З екземпляра скомпільованого методу легко отримати посилання на сам метод за допомогою *CompiledMethod >> methodReference*.

```
(Object >> #=) methodReference selector
>>> #=
```

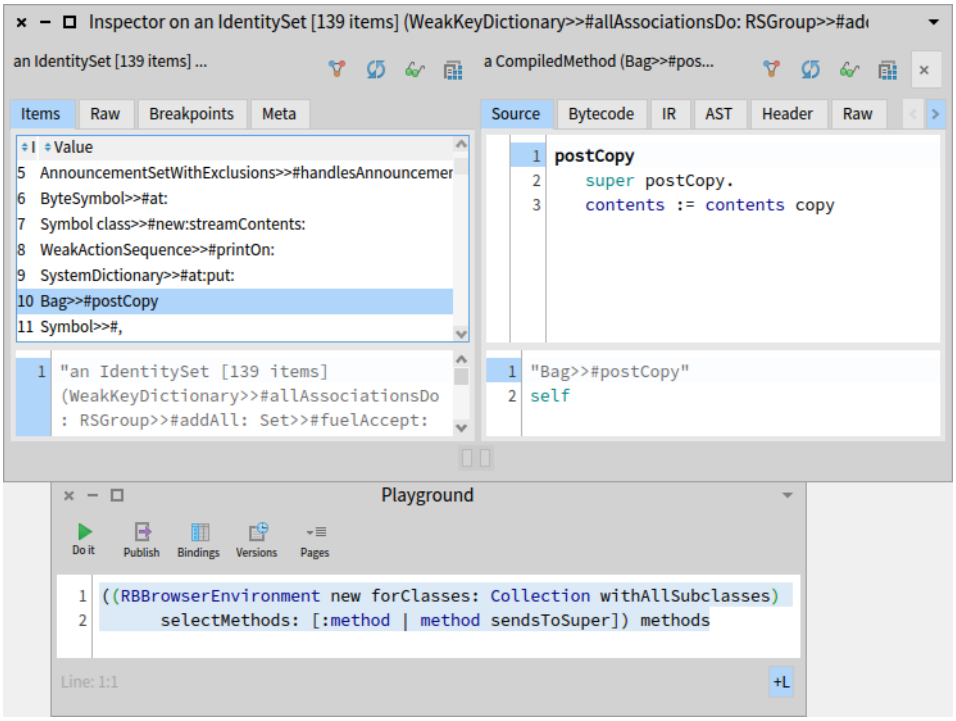

Рис. 18.6. Інспектування методів, які надсилають повідомлення до *super*

18.12. Середовища перегляду

SystemNavigation надає зручні засоби програмного формування запитів і перегляду системного коду, проте існують й інші способи. Інтегрований у Pharo Системний оглядач дає змогу обмежити середовище для виконання пошуку.

Припустимо, нам потрібно дізнатися, які класи посилаються на клас *Point*, але тільки з його власного пакета.

Відкрийте Оглядача на класі *Point*. Клацніть на пакеті верхнього рівня *Kernel* у панелі пакетів і увімкніть перемикач ***Scoped View***. Оглядач покаже лише пакет *Kernel* і всі класи в ньому (і деякі класи, які мають методи розширення з цього пакета). Тоді відкрийте контекстне меню класу *Point* і виберіть команду «*Class refs. Cmd+N*». Вона покаже всі методи, які мають посилання на клас *Point*, точніше, ті з них, які належать до пакета *Kernel*. Порівняйте отримане з результатом пошуку в Оглядачі без обмежень.

Таку ділянку пошуку називають *середовищем перегляду* (клас *RBBrowserEnvironment* – базовий для ієрархії класів, які реалізують середовища перегляду). Усі інші пошуки, як-от пошук *відправників повідомлення* або *реалізацій методу* в Оглядачі, також будуть обмежені цим середовищем.

Середовище перегляду можна створити також програмно. Ось, наприклад, як створити нове середовище *RBBrowserEnvironment* для класу *Collection* та його підкласів, вибрати методи, що надсилають повідомлення до *super*, і переглянути отримане середовище.

```
((RBBrowserEnvironment new forClasses: Collection withAllSubclasses)
      selectMethods: [:method | method sendsToSuper]) methods
```

Зверніть увагу, що цей фрагмент значно компактніший, ніж попередній еквівалентний приклад із використанням *SystemNavigation*. Щоб побачити результат запиту, відкрийте його в інспекторі: виконайте його командою «*Cmd+I*».

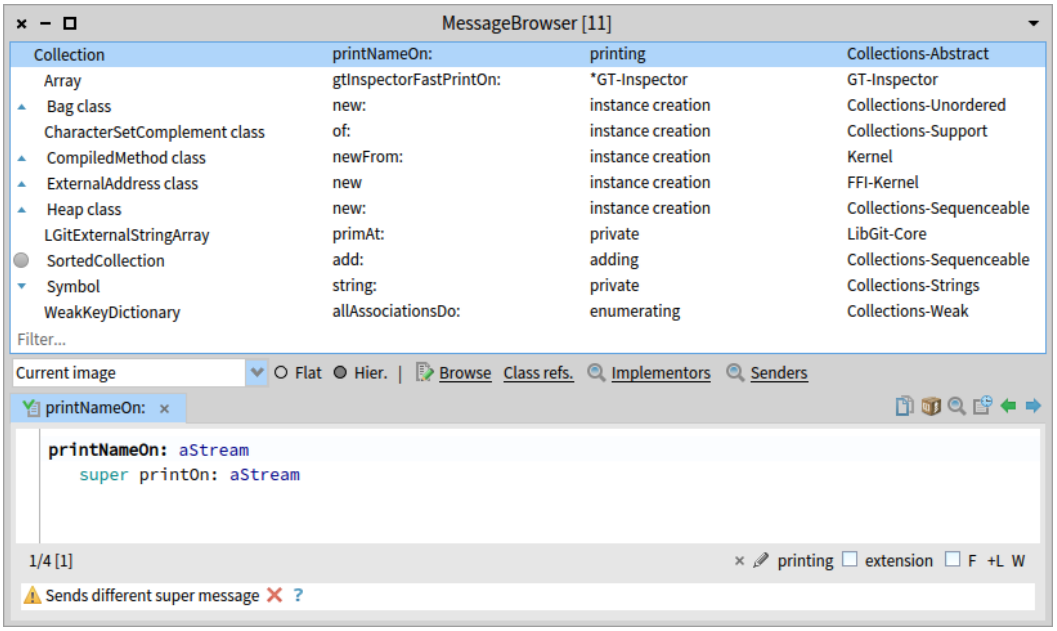

Рис. 18.7. Інспектування методів, які надсилають повідомлення до *super*

Нарешті, програмно можна знайти ті методи, які надсилають до *super* повідомлення, що відрізняються від селектора самого методу.

```
Smalltalk tools messageList browse:
((RBBrowserEnvironment new forClasses: (Collection withAllSubclasses))
   selectMethods: [:method | method sendsToSuper and:
      [(method parseTree superMessages includes: method selector) not]
   ]) methods
```

Тут у кожного скомпільованого методу запитують його дерево аналізу (*Refactoring Browser*), щоб з'ясувати, чи повідомлення до *super* відрізняються від селектора методу. Для перегляду результатів використано оглядача повідомлень. На рис. 18.7 зображено, що в ієрархії *Collection* було знайдено одинадцять таких методів, враховуючи *Collection >> printNameOn:*, який надсилає *super printOn:*.

Перегляньте протокол *querying* класу *RBProgramNode*, щоб побачити, що можна запитати в дерева аналізу.

18.13. Прагми – анотації методів

Прагма – це анотація методу, яка зазначає дані про програму, але не бере участі у виконанні програми. Вона не має прямого впливу на роботу методу, який анотує. Прагми застосовують, зокрема, для постачання інформації компіляторові та для опрацювання на етапі виконання.

Інформація для компілятора. Компілятор може використати прагму, щоб зробити виклик методу примітивною функцією. Така функція має бути визначена віртуальною машиною або зовнішнім плагіном.

Опрацювання на етапі виконання. Деякі прагми доступні для перевірки під час виконання.

У методі можна оголосити одну або кілька прагм, оголошення прагм розташовують перед першим виразом Pharo. Внутрішньо прагма – це статичне повідомлення, надіслане з літеральними аргументами.

Ми вже бачили в цьому розділі окремі прагми, коли говорили про примітиви. Примітив – це не що інше, як оголошення прагми. Розглянемо вираз *<primitive: 173 error: ec>*, визначений у методі *Object >> #instVarAt:*. Селектор прагми – *primitive:error:*, а його аргументи – буквальне значення літерала *173* і змінна *ec*, код помилки, який заповнює віртуальна машина, якщо виконання реалізації зазнає невдачі.

Компілятор, ймовірно, найбільший користувач прагм. SUnit – ще один інструмент, який використовує анотації. SUnit може оцінити охоплення програми тестами за кодом модульних тестів. Окремі методи можна виключити з оцінки охоплення за допомогою відповідної прагми, як у методі *documentation* метакласу *SplitJointTest class*:

```
SplitJointTest class >> documentation
  <ignoreForCoverage>
  "self showDocumentation"

  ^ 'This package provides function.... "
```

Анотуючи метод прагмою *<ignoreForCoverage>*, можна керувати сферою охоплення.

Прагми – це об'єкти першого класу, екземпляри класу *Pragma*. Скомпільований метод відповідає на повідомлення *pragmas*, повертаючи масив прагм.

```
(SplitJoinTest class >> #documentation) pragmas.
>>> an Array(<ignoreForCoverage>)
```

```
(SmallFloat64>>#+) pragmas
>>> an Array(<primitive: 541>)
```

За допомогою повідомлення *allNamed:in:* класові *Pragma* можна перебрати методи, що містять певну прагму. Наведений нижче приклад демонструє, що на стороні класу *SplitJoinTest* є два методи, анотованих *<ignoreForCoverage>*.

```
Pragma allNamed: #ignoreForCoverage in: SplitJoinTest class
>>> an Array(<ignoreForCoverage> <ignoreForCoverage>)
```

18.14. Доступ до контексту етапу виконання

Ми побачили, як інтроспективні можливості Pharo дають змогу формувати запити та досліджувати об'єкти, класи та методи. Але як щодо середовища виконання?

Контексти методу

Контекст етапу виконання запущеного (на виконання) методу фактично перебуває у віртуальній машині – його взагалі немає в образі! З іншого боку, налагоджувач, вочевидь, має доступ до цієї інформації, і ми можемо з задоволенням і користю досліджувати контекст виконання, як і будь-який інший об'єкт. Як таке можливо?

Насправді нічого чарівного в налагоджувачі немає. Секрет полягає в псевдозмінній *thisContext*, з якою ми раніше ознайомилися лише побіжно. Щоразу, коли у запущеному

методі трапляється звертання до *thisContext*, увесь контекст виконання цього методу втілюється та стає доступним образові як ряд пов'язаних екземплярів класу *Context*.

Ми можемо легко поекспериментувати з цим механізмом самі.

Змініть визначення *Integer >> slowFactorial*, вставивши вираз «*thisContext inspect. self halt.*», як зображено нижче.

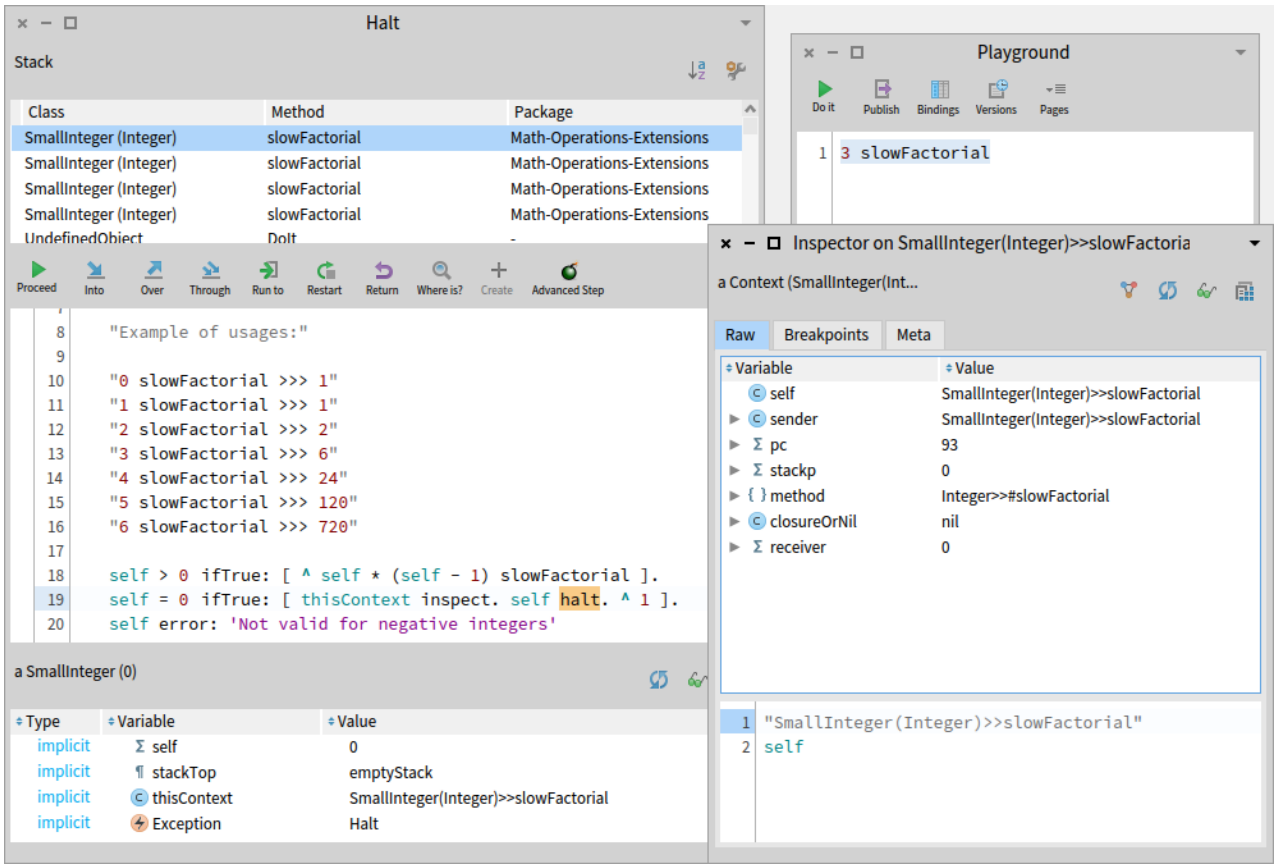

Рис. 18.8. Інспектування *thisContext*

```
Integer >> slowFactorial
  "Повертає факторіал отримувача."

  self > 0 ifTrue: [^ self * (self - 1) slowFactorial].
  self = 0 ifTrue: [thisContext inspect. self halt. ^ 1].
  self error: 'Not valid for negative integers'
```

Тепер виконайте *3 slowFactorial* у Робочому вікні. Ви мали б отримати і вікно налагоджувача, і вікно інспектора, як показано на рис. 18.8.

Інспектування *thisContext* дає повний доступ до поточного контексту виконання, стеку викликів, локальних змінних і аргументів, ланцюжка відправників і отримувача. Ласкаво просимо до налагоджувача бідної людини! Якщо тепер переглянути клас досліджуваного об'єкта (тобто, виконати «*self browse*» у нижній панелі інспектора), то виявиться, що це екземпляр класу *Context*, як і кожен відправник у ланцюжку.

Змінну *thisContext* не призначено для щоденного програмування, але вона важлива для створення таких інструментів, як налагоджувачі, і для доступу до інформації про стек викликів. Щоб дізнатися, які методи використовують *thisContext*, виконайте такий вираз (його виконання може зайняти трохи часу).

```
SystemNavigation default
   browseMethodsWithSourceString: 'thisContext' matchCase: true
```

Як з'ясувалося, одним із найпоширеніших застосувань є виявлення відправника повідомлення. Типове застосування полягає в наданні повнішої інформації розробнику. Розглянемо приклад. За домовленістю метод, який надсилає *self subclassResponsibility*, вважається абстрактним. Але як *Object >> subclassResponsibility* надає вичерпне повідомлення про помилку і зазначає, який абстрактний метод було викликано? Дуже просто, запитавши *thisContext* про відправника.

```
subclassResponsibility
"This message sets up a framework for the behavior of the class'
 subclasses. Announce that the subclass should have implemented this
 message."

SubclassResponsibility signalFor: thisContext sender selector
```

18.15. Інтелектуальні контекстні точки переривання

Щоб задати точку переривання у Pharo, в потрібному місці методу записують «*self halt*». Виконання такого виразу призводить до втілення *thisContext*, і відкривання вікна налагоджувача в точці переривання. На жаль, це створює проблеми для методів, які інтенсивно використовуються в системі.

Припустимо, наприклад, що потрібно дослідити виконання *Morph >> openInWorld*. Визначити точку переривання в цьому методі досить проблематично.

Будьте уважні, наступний експеримент зруйнує все! Створіть *новий* образ системи та задайте таку точку переривання, як подано нижче.

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

Зверніть увагу, як система відразу зависає, щойно ви намагаєтеся відкрити будь-яку нову морфу (меню/вікно/...)! Ви навіть не отримаєте вікно налагоджувача. Проблема стане зрозумілою, коли ми додумаємося, що, по-перше, *Morph >> openInWorld* використовується багатьма частинами системи, тому точка переривання спрацьовує дуже скоро після взаємодії з інтерфейсом користувача; по-друге, *сам налагоджувач* надсилає *openInWorld*, як тільки пробує відкрити вікно, запобігаючи відкриттю налагоджувача! Потрібен спосіб умовної зупинки, який перериватиме виконання лише за умови перебування в певному контексті. Це саме те, що надає *Object >> haltIf:*.

Припустимо тепер, що потрібно зупинитися, лише якщо *openInWorld* надсилається, скажімо, з контексту *MorphTest >> testOpenInWorld*.

Знову запустіть новий образ та встановіть таку точку зупинки:

```
Morph >> openInWorld
   "Add this morph to the world."
   self haltIf: #testOpenInWorld.
   self openInWorld: self currentWorld
```

Цього разу образ не зависає. Спробуйте запустити *MorphTest*. Він зупиниться та відкриє налагоджувача.

```
MorphTest run: #testOpenInWorld.
```

Як це працює? Давайте розглянемо *Object >> haltIf:*. Він надсилає повідомлення *if:* з умовою класу винятків *Halt*. Умовою може бути логічне значення, блок або символ. Метод *Halt class >> if:* сам перевірить, чи умова є символом, а в нашому прикладі так і є, і тому виконає «*self haltIfCallChain: thisContext home sender contains: condition*». Текст відповідного методу наведено нижче.

```
Object >> haltIf: condition
  <debuggerCompleteToSender>
  Halt if: condition.
```

```
Halt class >> haltIfCallChain: haltSenderContext contains: aSelector
  | cntxt |
  cntxt := haltSenderContext.
  [ cntxt isNil ] whileFalse: [
     cntxt selector = aSelector ifTrue: [ self signalIn:
        haltSenderContext ]. cntxt := cntxt sender ]
```

Починаючи з *thisContext*, *haltIfCallChain:contains:* проходить стеком виконання і перевіряє, чи ім'я викликаного методу збігається з переданим аргументом. Якщо це так, то він сам сигналізує про виняток, який за замовчуванням відкриває налагоджувача.

Аргументом для *haltIf:* можна також надати логічне значення або логічний блок, але це прості випадки, які не використовують *thisContext*.

18.16. Перехоплення незрозумілих повідомлень

У попередніх параграфах рефлексивні функції Pharo використовували здебільшого для формування запитів і дослідження об'єктів, класів, методів і стеку на етапі виконання. Тепер розглянемо, як використати знання про структуру системи для перехоплення повідомлень і зміни поведінки під час виконання.

Коли об'єкт отримує повідомлення, то для його опрацювання шукає відповідний метод спочатку в словнику методів свого класу, а якщо такого методу немає, то продовжує пошук ієрархією класів догори, доки не досягне *Object*. Якщо метод для цього повідомлення так і не знайдеться, то об'єкт *надішле сам собі* повідомлення *doesNotUnderstand:* і передасть селектор незрозумілого повідомлення як аргумент. Потім процес пошуку розпочнеться заново, доки не буде знайдено *Object >> doesNotUnderstand:* і не буде запущено налагоджувач.

Але що, якщо *doesNotUnderstand:* перевизначено одним із підкласів *Object* на шляху пошуку? Виявляється, це зручний спосіб реалізації певних видів дуже динамічної поведінки. Перевизначивши *doesNotUnderstand:*, об'єкт, який не розуміє повідомлення, може вдатися до альтернативної стратегії, щоб відповісти на нього.

Є два дуже поширених застосування цієї техніки: по-перше, реалізація легких проксі-серверів (обгорток) для об'єктів; по-друге, динамічна компіляція або завантаження коду, якого бракує. Це те, про що йтиметься в наступних параграфах.

18.17. Легкий проксі-сервер

Щоб реалізувати легкий проксі, вводять *мінімальний об'єкт*, який діятиме як обгортка існуючого об'єкта. Оскільки обгортка практично не реалізує власних методів, будь-яке надіслане їй повідомлення перехоплює функція *doesNotUnderstand:*. Реалізуючи це повідомлення, обгортка може виконувати спеціальні дії перед тим, як делегувати повідомлення справжньому суб'єкту, для якого вона є проксі-сервером.

Давайте розглянемо, як це можна зробити. Визначаємо *LoggingProxy*, як подано нижче.

```
ProtoObject subclass: #LoggingProxy
   instanceVariableNames: 'subject invocationCount'
   classVariableNames: ''
   package: 'PBE-Reflection'
```

Зауважте, що оголошено підклас *ProtoObject*, а не *Object*, щоб запобігти успадкуванню від *Object* більше 470 (!) методів.

```
Object methodDict size
>>> 473
```

Така обгортка має дві змінні екземпляра: об'єкт *subject*, для якого вона проксі, і *count* – кількість перехоплених повідомлень. Потрібно ініціалізувати обидві змінні екземпляра та надати метод читання лічильника повідомлень. Спочатку змінна *subject* вказує на сам проксі-об'єкт.

```
LoggingProxy >> initialize
  invocationCount := 0.
  subject := self.

LoggingProxy >> invocationCount
  ^ invocationCount
```

Завдання обгортки – перехоплювати всі повідомлення, друкувати їх у *Transcript*, оновлювати лічильник повідомлень і пересилати повідомлення справжньому об'єктові.

```
LoggingProxy >> doesNotUnderstand: aMessage
  Transcript show: 'performing ', aMessage printString; cr.
  invocationCount := invocationCount + 1.
  ^ aMessage sendTo: subject
```

А тепер трохи магії. Створимо новий об'єкт *Point* і новий об'єкт *LoggingProxy*, а тоді попросимо проксі стати точкою!

```
point := 1@2.
LoggingProxy new become: point.
```

Виконання повідомлення *become:* спричиняє обмін посилань: заміни всіх посилань на об'єкт, на який вказує змінна *point*, на посилання на екземпляр проксі та навпаки. Найважливіше те, що змінна *subject* екземпляра проксі тепер посилається на точку!

```
point invocationCount
>>> 0
```

```
point + (3@4)
>>> (4@6)

point invocationCount
>>> 1
```

В консолі з'явиться рядок «*performing + (3@4)*».

У більшості випадків це працює добре, але є деякі недоліки.

```
point class
>>> LoggingProxy
```

Насправді метод *class* реалізовано в *ProtoObject*, але навіть якби він був реалізований в *Object*, від якого *LoggingProxy* не наслідує, повідомлення *class* фактично не надсилається до обгортки або до її суб'єкта. На повідомлення безпосередньо відповідає віртуальна машина. *yourself* також ніколи не надсилають по-справжньому.

Нижче перелічено інші повідомлення, які залежно від отримувача може інтерпретувати безпосередньо віртуальна машина.

```
+- < > <= >= = ~= * / \\ =\=
@ bitShift: // bitAnd: bitOr:
at: at:put: size
next nextPut: attend
blockCopy: value value: do: new new: x y
```

Ніколи не надсилаються повідомлення, методи яких компілятор перетворює на вбудовані байт-коди порівняння та переходу. Нижче перелічено їхні селектори.

```
ifTrue: ifFalse: ifTrue:ifFalse: ifFalse:ifTrue:
and: or:
whileFalse: whileTrue: whileFalse whileTrue
to:do: to:by:do:
caseOf: caseOf:otherwise:
ifNil: ifNotNil: ifNil:ifNotNil: ifNotNil:ifNil:
```

Спроби надіслати ці повідомлення об'єктові невідповідного типу зазвичай призводять до винятку з боку віртуальної машини, оскільки вона не може використовувати вбудовану диспетчеризацію для неправильних отримувачів. Можна перехопити виняток і визначити належну поведінку, наприклад, перевизначивши *mustBeBoolean* в отримувачі, або перехопивши виняток *NonBooleanReceiver*.

Навіть якщо можемо не брати до уваги надсилання таких спеціальних повідомлень, то існує ще одна фундаментальна проблема, яку неможливо подолати за допомогою описаного підходу: не вдасться перехопити надсилання *self*.

```
point := 1@2.
LoggingProxy new become: point.
point invocationCount
>>> 0

point rectangle: (3@4)
>>> (1@2) corner: (3@4)
```

```
point invocationCount
>>> 1
```

У консолі з'явиться рядок «*performing rectangle: (3@4)*», а інформації про доступ до полів точки немає. Використання *self* аргументом повідомлення в методі *rectangle:* обмануло обгортку.

```
Point >> rectangle: aPoint
"Answer a Rectangle that encompasses the receiver and aPoint.
 This is the most general infix way to create a rectangle."

^ Rectangle
   point: self
   point: aPoint
```

Порівняйте отримане раніше з результатом створення такого самого прямокутника без використання *self*.

```
point := 1@2.
LoggingProxy new become: point.
point invocationCount
>>> 0

Rectangle point: point point: (3@4)
>>> (1@2) corner: (3@4)

point invocationCount
>>> 4
```

У консолі – чотири повідомлення про доступ до полів: два до *x*, і два до *y*.

За допомогою описаної техніки можна перехоплювати повідомлення, але потрібно пам'ятати про природні обмеження використання проксі-серверів. У параграфі 18.19 описано інший, загальніший підхід до перехоплення повідомлень.

Від перекладача. Автори книги описали одну загальну обгортку, яку можна використовувати з об'єктами різних типів. Зрозуміло, що на стільки загальна сутність однаково реагує на всі повідомлення: просто заносить їх до протоколу. Поміркуйте, як за допомогою перевизначення певних методів у проксі задати спеціальну поведінку об'єкта-обгортки. Наприклад, як убезпечитися від використання точок з від'ємними координатами під час створення прямокутників.

18.18. Генерування методів, яких бракує

Іншим поширеним застосуванням перехоплення незрозумілих повідомлень є динамічне завантаження або створення методів, яких бракує. Уявіть дуже велику бібліотеку класів з багатьма методами. Замість того, щоб завантажувати всю бібліотеку, ми могли б завантажити заглушку для кожного класу бібліотеки. Заглушки знають, де знайти вихідний код усіх своїх методів, перехоплюють усі незрозумілі повідомлення та на вимогу динамічно генерують методи, яких немає. У якийсь момент цю поведінку можна деактивувати, а завантажений код зберегти як мінімально необхідну для клієнтської програми підмножину.

Давайте розглянемо простий варіант цієї методики на прикладі класу, який на вимогу автоматично додає методи читання своїх змінних екземпляра. Логіка поведінки така: перехоплюємо кожне незрозуміле повідомлення, якщо існує змінна екземпляра з таким іменем, як селектор перехопленого повідомлення, та просимо клас скомпілювати метод доступу до неї і повторно надсилаємо те саме повідомлення.

```
Object subclass: #DynamicAccessors
   instanceVariableNames: 'x'
   classVariableNames: ''
   package: 'PBE-Reflection'
```

```
DynamicAcccessors >> doesNotUnderstand: aMessage
  | messageName |
  messageName := aMessage selector asString.
  (self class instVarNames includes: messageName)
     ifTrue: [
        self class compile: messageName, String cr, ' ^ ', messageName.
        ^ aMessage sendTo: self ].
  ^ super doesNotUnderstand: aMessage

DynamicAcccessors >> initialize
  x := 55
```

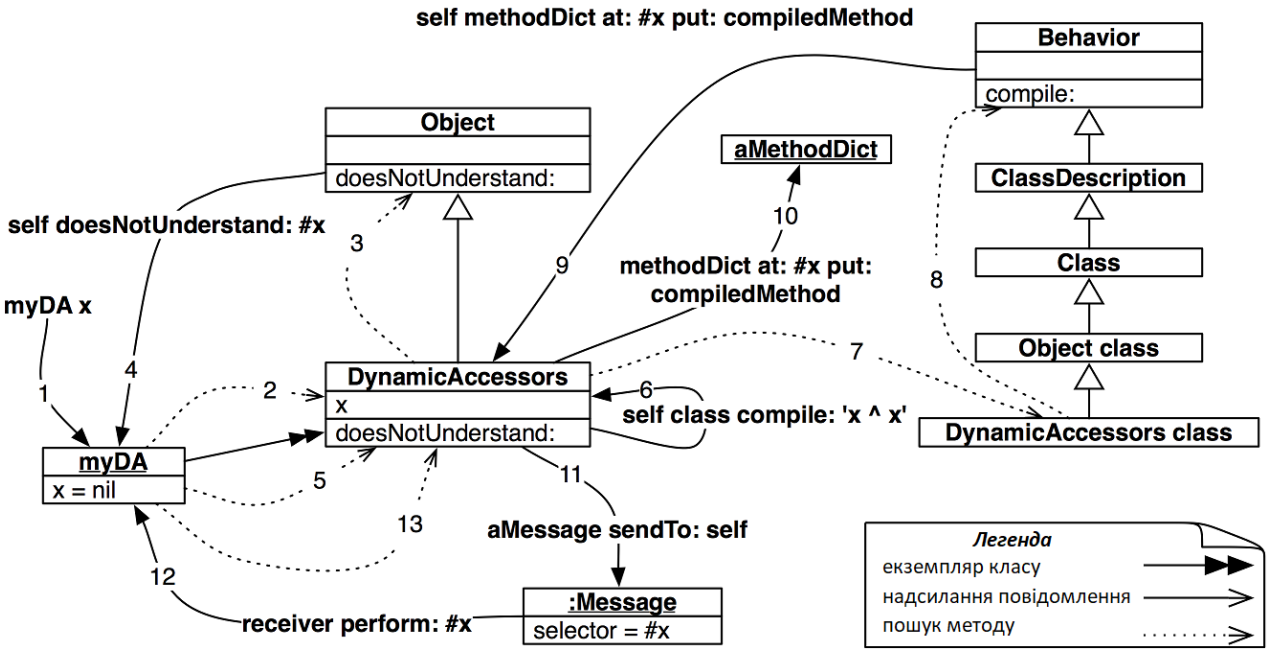

Рис. 18.9. Динамічне створення методів доступу

Припустимо, що клас *DynamicAccessors* має змінну екземпляра *x*, але не має попередньо визначеного методу доступу. Наведений нижче код динамічно згенерує такий метод і отримає значення змінної.

```
myDA := DynamicAccessors new.
myDA x
>>> 55
```

Давайте розглянемо, що відбувається, коли повідомлення *x* вперше надсилають екземплярові *DynamicAccessors* (див. рис. 18.9).

На рисунку номерами позначено послідовні кроки опрацювання повідомлення. Пояснимо кожен з них. (1) Повідомлення *x* надсилаємо до *myDA*. (2) Відбувається пошук методу в класі та (3) в ієрархії класів, пошук зазнає невдачі. (4) Це призводить до того, що «*self doesNotUnderstand: #x*» повертається до об’єкта (5) і запускає новий пошук. Цього разу метод *doesNotUnderstand:* відразу знайдено в *DynamicAccessors*, (6) він просить свій клас скомпілювати рядок *'x ^ x'*. Метод *compile:* шукається (7) в ієрархії метакласів і (8) нарешті знайдено в *Behavior*. Він (9-10) додає новий скомпільований метод до словника методів *DynamicAccessors*. Нарешті, (11-13) повідомлення надсилається повторно (за допомогою *sendTo:*), і цього разу метод доступу знайдено.

Якщо початкове надіслане повідомлення не збігається з іменами змінним екземпляра, то за допомогою *super* керування передається методові *doesNotUnderstand:* за замовчуванням. Таку саму техніку можна використовувати для генерування модифікаторів змінних екземпляра або інших видів шаблонного коду.

Про повідомлення *sendTo:*

Метод *sendTo:* реалізовано в класі *Message*, як подано нижче.

```
Message >> sendTo: receiver
   "повертає результат надсилання себе до отримувача"

   ^ receiver perform: selector withArguments: args
```

Метод *Object >> perform:* можна використовувати для надсилання повідомлень, створених на етапі виконання:

```
5 perform: #factorial
>>> 120

6 perform: ('fac', 'torial') asSymbol
>>> 720

4 perform: #max: withArguments: (Array with: 6)
>>> 6
```

18.19. Об’єкти як обгортки методів

Ми вже бачили, що скомпільовані методи є об’єктами в Pharo. Вони підтримують низку методів, які дають змогу програмісту надсилати запити до системи виконання.

От що, напевно, може викликати здивування, так це те, що будь-який об’єкт може відігравати роль скомпільованого методу та бути значенням у словнику методів. Все, що такий об’єкт має зробити, це відповісти на повідомлення *run:with:in:* і кілька інших важливих повідомлень. За допомогою такого механізму можна створювати інші шпигунські інструменти. Підхід можна описати так: створити об’єкт, який посилається на оригінальний скомпільований метод, щоб можна було його виконати, та замінити метод в словнику методів таким об’єктом (так званою обгорткою методу). Об’єкт-обгортка може виконувати різну додаткову роботу, наприклад, реєструвати, чи було виконано метод, скільки разів, коли він був виконаний.

Випробуємо такий підхід обгортання методів. Pharo постачається з простим класом *ObjectsAsMethodsExample*, щоб можна було проілюструвати його.

Визначте порожній клас *Demo*. Виконайте «*Demo new answer42*» і зверніть увагу, що виникає звичайна помилка *Message Not Understood*.

```
Object subclass: #Demo
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-Reflection'
```

```
Demo new answer42
```

Тепер додайте до словника методів класу *Demo* звичайний об'єкт.

```
Demo methodDict
   at: #answer42
   put: ObjectsAsMethodsExample new
```

Спробуйте ще раз надрукувати результат виконання «*Demo new answer42*». Цього разу ви мали б отримати відповідь 42.

```
Demo new answer42
>>> 42
```

Якщо переглянути клас *ObjectsAsMethodsExample*, то можна знайти наведені нижче методи.

```
ObjectsAsMethodsExample >> run: oldSelector with: arguments in: aReceiver
  ^ self perform: oldSelector withArguments: arguments

ObjectsAsMethodsExample >> answer42
  ^ 42
```

Коли екземпляр *Demo* отримує повідомлення *answer42*, пошук методу відбувається як зазвичай, однак віртуальна машина виявить, що замість скомпільованого методу цю роль намагається відіграти звичайний об'єкт Pharo. Тоді віртуальна машина надішле йому нове повідомлення *run:with:in:* з аргументами – початковим селектором методу, його аргументами та отримувачем. Оскільки *ObjectsAsMethodsExample* реалізує такий метод, то він перехоплює повідомлення та делегує його собі.

Видалити підробку можна як зі звичайного словника.

```
Demo methodDict removeKey: #answer42 ifAbsent: []
```

Якщо уважніше придивитися до *ObjectsAsMethodsExample*, то виявиться, що його надклас також реалізує деякі методи: *flushcache*, *methodClass:* і *selector:*, але всі вони порожні. Скомпільованому методу можуть надсилати такі повідомлення, тому об'єкт, який прикидається скомпільованим методом, мусить бути реалізованим. *flushcache* є найважливішим методом, який потрібно реалізувати; інші можуть знадобитися деяким інструментам залежно від того, чи визначено метод за допомогою *Behavior >> addSelector:withMethod:* чи безпосередньо за допомогою *MethodDictionary >> at:put:*.

А для серйозного використання ідеї, поданої в цьому параграфі, наполегливо рекомендуємо застосовувати бібліотеку, що називається *MethodProxies*, доступну на http://github. com/pharo-contributions/MethodProxies, оскільки вона набагато надійніша і безпечніша.

18.20. Підсумки розділу

Рефлексія означає здатність запитувати, досліджувати та навіть змінювати метаоб'єкти системи виконання як звичайні об'єкти.

- Інспектор використовує *instVarAt:* і пов'язані методи для перегляду приватних змінних екземплярів об'єктів.
- Метод *Behavior >> allInstances* повертає всі екземпляри класу.
- Повідомлення *class*, *isKindOf:*, *respondsTo:* тощо корисні для збору показників або створення інструментів розробки, але їх треба уникати у звичайних програмах: вони порушують інкапсуляцію об'єктів і ускладнюють розуміння та підтримку коду.
- *SystemNavigation* – службовий клас, який містить багато корисних запитів для навігації та перегляду ієрархії класів. Наприклад, використовуйте *SystemNavigation default browseMethodsWithSourceString: 'pharo' matchCase: true*, щоб знайти та переглянути всі методи з заданим вихідним рядком. (Повільно, але ретельно!).
- Кожен клас Pharo вказує на екземпляр *MethodDictionary*, який відображає селектори на екземпляри *CompiledMethod*. Скомпільований метод вказує на свій клас, замикаючи коло.
- *RGMethodDefinition* – легкий проксі-сервер для скомпільованого методу, який надає додаткові зручні методи та використовується багатьма інструментами Pharo.
- *RBBrowserEnvironment*, частина інфраструктури Refactoring Browser, пропонує досконаліший ніж *SystemNavigation* інтерфейс для запитів до системи. Його перевага в тому, що результат запиту можна використовувати як область нового запиту. Доступні графічний і програмний інтерфейси.
- *thisContext* – псевдозмінна, яка втілює стек віртуальної машини на етапі виконання. Здебільшого використовується налагоджувачем для динамічної побудови інтерактивного перегляду стеку. Вона також особливо корисна для динамічного визначення відправника повідомлення.
- Контекстні точки переривання можна встановити за допомогою *haltIf:*, якщо вказати аргументом селектор певного методу. *haltIf:* перерве виконання лише тоді, коли названий метод буде у стеку викликів.
- Поширеним способом перехоплення повідомлень, надісланих певній сутності, є використання *мінімального об'єкта* як обгортки для неї. Такий проксі-сервер реалізує якомога менше методів і перехоплює всі надіслані повідомлення, реалізуючи *doesNotunderstand:*. Після перехоплення він може виконати деякі додаткові дії, а потім переслати початкове повідомлення за призначенням.

- Надішліть *become:*, щоб поміняти місцями посилання на два об'єкти, наприклад, на проксі та його суб'єкт.
- Будьте обережні, деякі повідомлення, як-от *class* і *yourself*, насправді ніколи не надсилаються, а інтерпретуються віртуальною машиною. Інші, як-от +, – та *ifTrue:* можуть бути безпосередньо інтерпретовані або вбудовані віртуальною машиною залежно від отримувача.
- Ще одне типове використання перевизначення *doesNotUnderstand:* – це ліниве завантаження або компіляція методів, яких бракує. Метод *doesNotUnderstand:* не може перехопити надсилання *self*.

Навчальне електронне видання

Стефан ДЮКАС, Джордана РАКІЧ,
Себастьян КАПЛАР, Квентін ДЮКАС

PHARO 9 НА ПРИКЛАДАХ

Навчальний посібник

Переклад з англійської
з доповненнями
Сергій ЯРОШКО

Редактор *Н. Плиса*
Комп'ютерне макетування *С. Ярошко*

Фото на обкладинці *Джозефа Гегана*

Формат 60×84/8. Ум. друк. арк. 31,4.

Видавець та виготовлювач:
Львівський національний університет імені Івана Франка,
79000, Львів, вул. Університетська, 1
Свідоцтво про внесення суб'єкта
видавничої справи до Державного реєстру видавців, виготівників
і розповсюджувачів видавничої продукції.
Серія ДК № 3059 від 12.12.2007 р.