**6 Розділ**

Об’єктна модель Фаро

Модель програмування Pharo в основному натхненна моделю Smalltalk. Вона є простою і однотипною: все є об’єктом і об’єкти комунікують тільки через передачу повідомлень одне одному. Зміна екземпляр є приватною для об’єкта. Всі методи є публічно доступні і динамічно зв’язні. В цьому розділі ми представляємо основоположні ідеї об’єктної моделі Фаро. Ми переглянемо такі ідея, як self і super і точно визначимо їхню семантику. Потім ми обговоримо ідеї представлення класів, як об’єктів. Про це більше детально поговоримо в розділі:”Класти та Метакласи”

**6.1 Правила моделі**

Об’єктна модель заснована на декількох простих правила, які однотипною застосовується.

Правило 1. Все є об’єктом.

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

Правило 3. Кожен клас має суперклас.

Правило 4. Все відбувається через відправку повідомлень.

Правило 5. Пошук метода динамічно слідує ієрархії класів.

Давайте подивимось на кожне з цих правил більш детально.

**6.2 Все є об’єктом**

Мантна, що все все є об’єктом є дуже заразна. Зразу після короткого часу роботи з Фаро, ви здивуєтесь, як це правило спрощує все, що ви робите.

Цілі числа, для прикладу, є також справжні об'єкти, тому можна відсилати повідомлення і до них, так які до звичайних об’єктів.

В кінці цього розділу, ми додали коментарі до реалізації об’єктів для допитлих читачів.

“відправити ‘+ 4’ до 3, видасть 7”

3+4

>>> 7

"відпавити factorial, видасть велике число"

20 factorial

>>> 2432902008176640000

Об’єкт 7 є інший ніж об’єкт повернутий виразом “20 factorial”, але через те, що вони обидва є полеморфними об’єктами, але ні цей код ні реалізація factorial, не повинні це знати.

Повертаючись до правила “все є об’єктом”, найбільш фундаментальним результатом, якого є що класи це також об’єкти.

Класи це не, не якийсь окремий тип об’єктів, вони є звичайними об’єктами, яким ти можеш надсилати повідомлення, переглядати і змінювати. Це означає, що фаро є справді рефлективною системою, що дає багато гнучкості для розробників.

Важливі класи - це також об’єкти.

**6.3 Кожен об’єкт є екземпляром класу.**

Кожен об’єкт має свій клас; Ти можеш дізнатись який, надіславши повідомлення “class” до цього об’єкту.

1 class

>>> SmallInteger

20 factorial class

>>> LargePositiveInteger

'hello' class

>>> ByteString

(4@5) class

>>> Point

Object new class

>>> Object

Клас визначає структуру його екземпляра через нестатичні поля класу, і повідінка циї полів через методи. Кожен метод має назву, яка назувається його селектором, який є унікальним в межах класу.

Оскільки класи є об’єктами і кожен об’єкт є екземпляром якогось класу, це слідує тому, що класи також повинні бути інстансами якогось класу. Клас екземпляр якого є класом називаються метакласом.

Всякий раз, коли ви створюєте клас, система автоматично створює метаклас для вас. Метаклас визначає структуру і поведінку класу, який є його екземпляром. 99% часу вам не потрібно буде думати про метакласи, і можете з успіхом їх ігнорувати.(Ми подивимось поближче на метакласи в розділі “Класи і Метакласи”).

**6.4 Структура екземпляру і його поведінка.**

Зараз ми коротко оглянемо структуру і поведінку екземплярів класу.

**Поля езкемпляру**

Поля екземпляру в Фаро є приватними для самого екземпляру. Це на відміну від Java і C++, який дозволяє, щоб поля класу були доступні для будь якого іншого екземпляра, який просто виявився такого самого типу.

Ми говоримо, що границя інкапсуляції об’єктів в Java і C++ є клас, коли в Фаро це є екземпляр.

В Фаро, два екземпляра того самого типу не мають доступу до полів іншого класу, хіба клас оголошує методи для доступу. Не існує такого синтаксису, який дозволяє прямий доступ до полів екземпляру будь-якого об’єкту.

(Насправді, існує механізм, який називається Рефлексія, що надає можливість запитати інший об’єкт про значення його полів; метапрограмування призначене для написання інструментів, таких як, інспектор об’єктів, чия роль полягає в пошуку в середині інших об’єктів)

До поля екземпляру можна доступитись за іменем в будь-якому методі класу, що оголошує їх, а також в методах оголошених в підкласі.

Це означає, що в Фаро поля езкемплярів є подібними до захищених полів в С++ і Java. Хоча ми називаємо їх приватними, оскільки це вважається паганим стилем програмування, доступатись до полів екземпляру напряму з субкласу.

Приклад інкапсуляції екземпляра.

Метод Point>>dist: обчислює дистанцює між приймачом і іншою точкою. Поля екземпляру приймача, x і y, напряму доступаються у тілі метода.

Хоча, до пулів езкземпляра точки ми повинні доступатись тільки через надсилання повідомлення до x і y.

Point >> dist: aPoint

"Поверне відстань між aPoint і приймачом."

| dx dy | dx := aPoint

x - x. dy := aPoint y - y

^ ((dx \* dx) + (dy \* dy)) sqrt

1@1 dist: 4@5

>>> 5.0

Основна причина, щоб надавати перевагу інкапсуляції заснованій на езкемплярах над класовою інкапсуляцію полягає в тому, що це дозволяє різним реалізаціям той самої абстракції співіснувати.

Для прикладу, метод dist: ми не знаєм і не переймаємось ти чи аргумент aPoint є екземпляром того самого класу, як і приймач. Об’єкт аргумента може бути представлений в полярних координатах, чи як запис в базі даних, або як інший комп’ютер в мережі.

До тих під, поки він може відповідати на повідомлення x і y, код метода dist(показаний вище) буде досі працювати.

**Методи**

Всі методи є публічними і віртуальними(шукаються динамічно). Методи групуються в протоколах за їх намірами. Декілька звичних назв протоколів вже вважаються нестандартними: accessing для всіх методів доступу, initialization для методів створення валідного початкового стану об’єкта. Протокол private деколи використовується для групування методів, які не повинні використовуватись зовні. Хоча, ніщо не забороняє вам надсилати повідомлення, яке оголошене в такому “приватному” методі

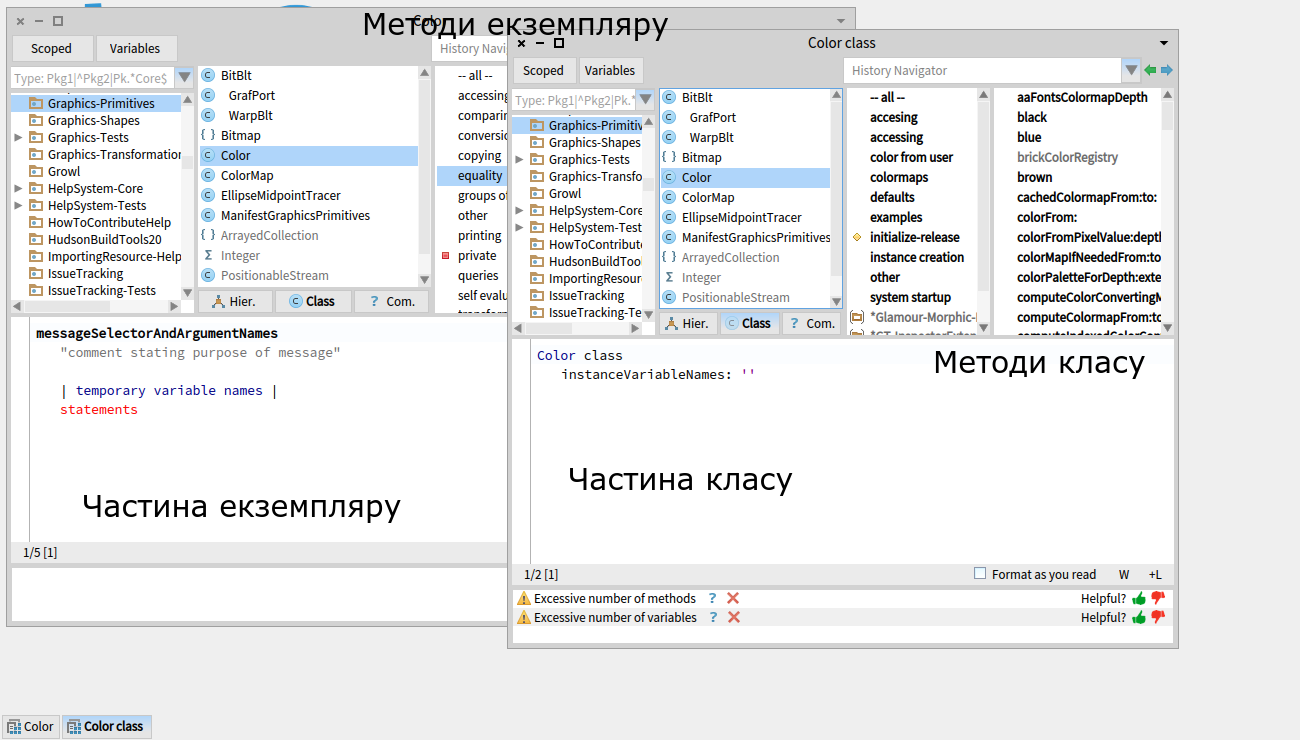
Методи можуть мати доступ до всіх полів екземпляру об’єкта. Деякі розробники надають перевагу підходу, що доступатись до полів екземпляру потрібно тільки через методи доступу. Ця практика має право на істування, але вона також засмічує інтерфейс класу, ба більше, розголошує інформацію, про приватний стан об’єкта

**6.5 Сторона екземпляру і сторона класу**

Оскільки класи є об’єктами, вони можуть мати свої поля і свої власні методи. Ми називаємо їх полями класу і методами класу, але вони не відрізняються від звичайних полів полів і методів екземпляру. Вони просто оперують над різними об’єктами(класами в даному випадку). Поле екземпляру в даному випадку, описує стан езкемпляру, а методи екземпляру описують поведінку.

Подібно, поля метакласу є такими ж полями тільки оголошеними метакласом(і описує стан класів - екземпляри метакласу), і методи метакласу є просто методами оголошеними метакласом (які будуть виконані над класом)

Клас і його метаклас - це два окремих класи, навіть враховуючи, що перший є екземпляром другого. Проте, це в основному не має значення для пересічного розробника. В основному розробник зосереджується на описані поведінки його об’єктів і класів, що їх створюють.

З цієї причини, браузер помагаю тобі переглядати класи, як і метакласи, ніби вони одне частина одного цілого, тільки розділені на половину: частина екземпляру і частина класу, як показано на малюнку 6.1

Малюнок 6.1: Перегляд метакласу і класу

За замовчування, коли ти вибираєш клас у браузері ти розглядаєш частину екземпляру(тобто методи, що виконується коли повідомлення відправлється до екземпляра класу Color). Натискаючи на “Class side” кнопку вікно перейде до частини класу(методи, що будуть виконувати, коли повідомлення надсилається до класу Color, як такого).

Для прикалу, Color blue надсилає повідомлення blue до класу Color. Ви в свою чергу знайдете метод blue оголошений на частині класу Color, а не на частині екземпляру.

"Метод blue з частини класу (зручний спосіб створення об’єкту)"

aColor := Color blue.

>>> Color blue

"Екземпляри Color є самовиконувані"

"Методу доступу езкемпляра класу red (повертає червоне значення з RGB)"

Color blue red

>>> 0.0

"Методу доступу езкемпляра класу blue (повертає синє значення у RGB)" Color blue blue

>>> 1.0

Ти оголошуєш клас фільтруючи шаблон запропонований на частині екземпляра. Коли ти приймаєш шаблон, система створює не тільки клас, який ти оголосив, але й відповідний метаклас (який потім можна по-редагувати нажавши на “Class side” кнопку).

Єдина частина з процесу створеня метакласу з шаблона, яку вам є сенс модифікувати, це список імен полів метакласу.

Як тільки клас був створений, переглядаючи частину екземпляру(“Class side” не є увімкнутий) ви можете редагувати і переглядати методи, що будуть оброблені екземпляром цього класу(або його підкласів).

**Методи класів**

Методи класів можуть бути досить корисними; Можна переглянути Color class, як приклад. Ти побачиш, що є 2 типа методі визначених в класі: методи створення, як Color class>>wheel, і ті, що виконують допоміжні функції, як like Color class>>wheel:. Це є типово, хоча ти іноді знайдеш методи класу організовані по-іншому.

Зручно поміщати допоміжні функції на частину класу, оскільки вони можуть бути виконані навіть не створивши будь-яких додаткових об’єктів. Справді, багато з них міститимуть коментар, створений для того, щоб спростити їх використання.

Можна переглянути метод Color class>>wheel:, Подвійно клікнувши на початок коментаря t "(Color wheel: 12) інспектувати" і нажавши CMD-d. Ти побачиш ефект виконання методу.

Для тих хто знайомий з Java і C++, методи класів можуть здатися подібними до статичних методів. Проте, лднорідність об’єктної моделі Фаро(де класи - це звичайні об’єкти) означає, що вони трохи різні: коли статичні методи в Java є просто статично визначені процедури, методи класу динамічно визначаються в Фаро. Це означає, що наслідування і перевантаження працюють в Фаро, але вони не працють в Java.

**Поля класів**

Зі використовуючи звичайні поля класу, всі екземпляри класу мутимуть такі ж імена своїх полів(хоча кожен екземпляр має також свої власні приватні поля), і екземпляри його підкласів також унаслідують ці імена.

Так само працюють і поля метакласів: кожен клас має свої власні приватні унаслідувані поля. Підклас унаслідує ці поля, але підклас буде мати власні приватні копії цих значень. Як і об’єкти не дають досуп до своїх полів, так і класи і їх підкласи не дають доступ до своїх полів.

Для прикладу, ти можеш використовувати поле екземпляра count, щоб слідкувати, як багато екземплярів класу було створено. Проте, будь-який підклас буде слідкувати тільки, за своєю власною count змінно, тому екземпляри підкласу будуть рахуватись окремо.

Приклад: Поля класу і підкласу

Для прикладу, ми оголосимо клас Dog, і його підклас Hyena. Припустимо, що ми добавили поле класу cout до класу Dog(тобто ми оголосили його в метакласі). Тоді Hyena автоатично унаслідує поле з класу Dog.

Object subclass: #Dog

instanceVariableNames: ' '

classVariableNames: ' '

package: 'PBE-CIV'

Dog class

instanceVariableNames: 'count'

Dog subclass: #Hyena

instanceVariableNames: ' '

classVariableNames: ' '

package: 'PBE-CIV'

Зараз припустимо, що ми визначили метод класу Dog, щоб ініціалізувати count нулем, і збільшити його, коли навиий екземпляр буде створений:

Dog class >> initialize

count := 0.

Dog class >> new

count := count +1.

^ super new

Dog class >> count

^ count

Зараз коли ми сворили новий екземпляр класу Dog, значення поля count збільшилось на одиницю, так і для класу Hyena(але екземпляри класу для гієни рахуються окремо).

Примітка: Звірніть увагу на повідомлення initialize у класі, в даному коді. В Фаро, коли ти створюєш об’єкти якогось класу, для прикладу Dog new, то initialize викликається автоматична, як частина повідомлення new (це можна перевірити переглянувши поведінку цього методу Behavior>>new). Але у випадку з класами, просто їх оголошення не означає автоматичний виклик initialize, тому ми повинні явно викликати його тут. За замовчування методи ініціалізації автоматично виконуються тільки коли класи звантажуються. Дивись далі обговорення про ліниву ініціалізацію.

Dog initialize.

Hyena initialize.

Dog count

>>> 0

Hyena count

>>> 0

| aDog |

aDog := Dog new.

Dog count

>>> 1 "Зросло"

Hyena count

>>> 0 "Незмінилось"

Поля в класі є приватні до класу в такій самій мірі, як поля екземплярі класу приватні для екземплярів. Оскільки класи і їх екземпляри є визначими об’єктами, це має такі наслідки:

Клас не має доступу до полів його власний екземплярів. Тому, для прикладу, клас Color не має доступу до об’єкта створенемо з нього aColorRed. Іншими словами, тільки тому, що клас використовується для створення езкемпляру(використовуючи new або інші допоміжні методи, як Color red), це не дає йому якого небуть прямого доступу до полів його екземпляра. Клас замість цього повинен використовувати методи для доступу, як і будь-які інші об’єкти.

2. Обернене також вірно: екземпляри класу не мають доступу до полів свого класу. З прикладу вище, aDog(екземплря класу Dog) не має прямого доступу до поля count у класі Dog(звісно крім варіантом з використанням методів для доступу).

Важливо! Клас не має доступу до поляв свого власного екземпляру, так як і екземпляр класу не має доступ до полів класу.

Через це, методи іціціалізації повинні буи завжди визначені на стороні екземпляру, сторона класу не має доступу до полів, і тому не може ініціалізувати їх! Все, що клас може зробити, це відправити повідомлення initialization, використовувати методи доступу, для тільки що створених екземплярів.

Java не має нічого подібного до полів класу. Java і C++ статичні змінні є більше схожі на поля класу(про них детально в розділі 6.9), оскільки в цих мовах всі підкласи і всі їх екземпляри розділяють ті самі статичні змінні.

Приклад: Оголошеня сингилтона

Паттерн сингилтон є звичайним прикладом використання полів класу і методів класу. Уявимо, що нам потрібно реалізувати клас WebServer, і використаємо паттерн сингелтон, щоб гарантувати існування тільки одного екземпляра цього класу.

Ми так оголосимо клас WebServer.

Object subclass: #WebServer

instanceVariableNames: 'sessions'

classVariableNames: ''

package: 'Web'

Потім перейшовши на сторну класу(нажавши “Class side”), ми додамо поле класу uniqueInstance.

WebServer class

instanceVariableNames: 'uniqueInstance'

В результаті, клас WebServer отримає нове поле(на додачу до тих, що унаслідувалась від Behavior, такі як superclass і methodDict). Це означає, що значення додаткового поля класу буде описувати екземпляр класу WebServer, а отже і клас WebServer.

Object class allInstVarNames

>>> "#('superclass' 'methodDict' 'format' 'layout' 'instanceVariables'

'organization' 'subclasses' 'name' 'classPool' 'sharedPools'

'environment' 'category' 'traitComposition' 'localSelectors')"

WebServer class allInstVarNames

>>>"#('superclass' 'methodDict' 'format' 'layout' 'instanceVariables'

'organization' 'subclasses' 'name' 'classPool' 'sharedPools'

'environment' 'category' 'traitComposition' 'localSelectors'

#uniqueInstance)"

Тепер ми можемо оголосити метод класу uniqueInstance, як показано нижче.

Цей метод спочатку провірить чи uniqueInstance був ініціалізований. Якщо ні, метод створить екземпляр класу і присвоїть його полю uniqueInstance. В кінці значення uniqueInstance буде повернено. Оскільки uniqueInstance є полем класу, метод на пряму може доступатись до нього.

WebServer class >> uniqueInstance

uniqueInstance ifNil: [ uniqueInstance := self new ].

^ uniqueInstance

Перший раз, коли WebServer uniqueInstance буде виконуватись, екземпляр класу WebServer буде створений і збережений в полі uniqueInstance. Наступного разу, попередньо створений екземпляр буде повернутий, а не буде створюватись новий.(Цей шаблон проектування, перевіряти чи змінна є nil в методах доступу, і ініціалізувати її, якщо nill - називається “лінива ініціалізація”).

Зауважте, що код для створення екземпляру вище, написаний, як self new, а не як WebServer new. Яка різниця? Оскільки uniqueInstance метод визначений в класі WebServer, ви можете подумати, що різниця немає. Справді, допоки хтось не створить підклас WebServer, вони є однакові. Але припустимо, що ReliableWebServer є підкласом WebServer і наслідує uniqueInstance метод. Ми очікуємо, що ReliableWebServer uniqueInstance поверне нам ReliableWebServer. Використання self гарантує нам це, оскільки self ніби прив’язний до відповідного отримувача. В даному випадку до класів WebServer і ReliableWebServer. Зауважте також, що WebServer і ReliableWebServer будуть мати різні значення їх поля uniqueInstance.

Декілька зауважень до “лінивої ініціалізації”. Не використовуйте його надмірно. Ініціалізація початкових даних екземпляра, зазвичай належить до прерогативи методів ініціалізації. Помістивши весь код ініціалізації в методи ініціалізації підвищить читабільність коду - вам не потрібно переглядати всі методи доступу, щоб дізнатись, які є початкові значення. Хоча ініціалізація полів у відповідних методах класу( за допомогою перевірок на nil) може здатись приваблива, її варто уникати, хіба що у вас є причини її використовувати.

Для прикладу, в нашому методі uniqueInstance вище, ми використали “ліниву ініціалізацію”, бо зазвичай від користувачів не очікуються виклику WebServer initialize. Замість цього, вони очікують, щоб клас був готовий повернути нові унікальні значення. Через це, “лінива ініціалізація” тут оправдана. Подібно до цього, якщо змінну дуже дорого ініціалізувати (для прикладу, відкриття зв’язку до бази даних чи сокет), ви можете вибрати відтермінування ініціалізації, допоки вам справді не буде потрібен об’єкт.

**6.6 Кожен клас має клас клас предок**

Кожен клас в фаро унаслідує свою поведінку і опис структури з якогось єдиного суперкласу (класу нащадка). Це означає, що в Smalltalk можна наслідуватись тільки від одного класу.

SmallInteger superclass

>>> Integer

Integer superclass

>>> Number

Number superclass

>>> Magnitude

Magnitude superclass

>>> Object

Object superclass

>>> ProtoObject

ProtoObject superclass

>>> nil

За традицією, коренем наслідування є клас Object (оскільки все є об’єктом). В Фаро, об’єктом коренем є Proto Object, але ви зазичай не будете звертати ніякої уваги на цей клас. Proto Object інкапсулює мінімальний набір повідомлень, що кожен об’єкт має мати, і Proto Object проективний так, щоб повертати якомога більше помилок, щоб підтримувати оголошення через посередника. Проте, більшість класів наслідуються від класу Object, який визначає багато інших додаткових повідомлень, що майже всі об’єкти знають, як опрацювати. Створюючи класи для своєї програми вам слід наслідуватись від класу Object (або від його підкласів), хіба у вас є дуже добрі причини не робити цього.

Новий клас зазвичай створюється за допомогою повідомлення “instance- Variable Names: ... ” до вже існуючого класу. Ще є декілька інших методів, щоб створювати класи. Щоб зрозуміти, що вони таке, подивимось на протокол створення класу і його підкласу.

Хоча Фаро не надає можливості множинного успадкування, воно надає механізм під назвою Traits(особливості), щоб ділитись кодом між декількома незв’язними класами. Traits це набір методів, що можуть бути перевикористаними різними класами, що не наслідуються один від одного. Traits дозволяють інкапулювати код в одному місці і не дублювати його в різних місцях.

**Абстрактні методи і абстрактні класи**

Абстрактний клас - це клас, який існує для того, щоб від нього наслідувались, а не щоб з нього створювались екземпляри. Абстрактний клас зазвичай є не завершений, тобто він наприклад не оголошує всіх методів, що використовує. Методи заповнювачі в абстрактних класах є пустими методами, так що інші методи можуть викликати їх, очікуючи, що ці пусті методи будуть визначені в класах нащадках. Ці методи є абстрактними методами.

Фаро не має соціального синтаксису, щоб виокремити, що клас чи метод є абстрактними. Замість, за загально прийнятими правилами, тіло абстрактного методу має містити вираз “self subclassResponsibility”. Це визначає, що підклас відповідальний за визначення конкретної реалізації цього методу.

Метод з виразом “self subclassResponsibility” повинен завжди бути перевантаженим, а отже ніколи не повинен бути викликаний. Якщо ви забули про це і викликали метод, тоді буде генерована помилка.

Подібно до цього, клас вважається абстрактним, якщо хоч один з його методів є абстрактним. Ніщо насправді не забороняє вам створити екземпляр абстрактного класу, все буде працювати, допоки ви не викличете абстрактний метод.

Для прикладу, абстрактний клас Magnitude

Magnitude - це абстрактний клас, що помагає визначити об’єкти, що можуть бути порівнвані один з одним. Підкласи Magnitude повинні реалізувати оператори <,= і хеш. Використовуючи ці повідомлення Magnitude визначає інші оператори: >, >=, <=, max:, min: between:and: і інші для порівнювання об’єктів. Ці методи будуть унаслідувані підкласами. Оператор >>< класу Magnitude є абстрактний і визначений, як показано нижче.

< aMagnitude

“Визначає чи отримувач є менший ніж аргумент.”

^self subclassResponsibility

На противагу цьому, оператор >= є конкретний і визначений в термінах оператора <.

>= aMagnitude

"Визначає чи отримувач є більший чи рівний аргументу"

^(self < aMagnitude) not

Це саме можна засусувати і для інших операторів порівняння(вони всі визначенні в термінаї оператора <)

Character(символ) є підкласом Magnitude, він перевизначає метод < (який є позначений, як абстракний у класі Magnitude, через використання виразу”self subclassResponsibility” ) і заманяє його своєю версією(дивись визначення методу нижче).

Character також явоно визначає оператори = і хеш. Клас також наслідує оператори >=, <=, ~= з класу Magnitude.

< aCharacter

“Повертає True(правда), якщо значення отримувача є меншим ніж значення аргумента”

^self asciiValue < aCharacter asciiValue

**Traits**

Trait - це колекція методів, що можуть бути використані в іншому класі, без використання механізму наслідування. Це дозволяє класу мати тільки один клас предок і в цей самий час ділити корисні методи з невз’язними з собою класами.

Щоб визначити новий trait, вам слід просто клікнути правою кнопкою миші на панель класів і вирати пункт меню “Add Trait”, або замінити шаблон створення підкласу на шаблон створення trait(приклад нижче).

Trait named: #TAuthor

uses: { }

package: 'PBE-LightsOut'

Ми визначаємо trait TAuthor в пакеті PBE-LightsOut. Цей trait не використовує ніякиї інших traits. Також за допомогою ключового слова “uses” ми можемо додати вираз з іншими traits, які будуть використовуватись. В даному випадку ми просто використовуємо порожній масив.

Traits можуть містити тільки методи, а не поля. Припустимо ми б хотіли додати методи author до декількох класів, незалежно від того де вони є в ієрархії класів.

Ми можемо це зробити так:

TAuthor >> author

"Повертає ініціали автора"

^ 'on' "oscar nierstrasz"

Тепер ми можемо використати цей trait в класі, що вже має суперклас. Для прикладу клас LOGame, що ми визначили в розділі “Перша програма”. Ми просто модифікуємо шаблон для створення для класу LOGame, додавши TAuthor trais, до ключового слова “uses”.

BorderedMorph subclass: #LOGame

uses: TAuthor

instanceVariableNames: 'cells'

classVariableNames: ''

package: 'PBE-LightsOut'

Якщо зараз створити екземпляр класу LOGame, то відповість на повідомлення author, як очікувалось.

LOGame new

author >>> 'on'

Вираз, для комбнації trait, може міститити їх декілька, використовуючи оператор “+”. У випадку конфліктів(якщо декілька traits визначають метод з однаковим ім’ям), їх можна уникнути, якщо явно видалити ці методи(за допомогою “-”), або перевизначивши ці методи в класі або trait, що ви оголошуючи, також можливо дати іншу назву методу(за допомогою @).

Traits використовуються в ядрі системи. Добрим прикладом може бути клас Behavior.

Object subclass: #Behavior

uses: TPureBehavior @

{#basicAddTraitSelector:withMethod:->#addTraitSelector:withMethod:} instanceVariableNames: 'superclass methodDict format' classVariableNames: 'ObsoleteSubclasses'

package: 'Kernel-Classes'

Ту ми бачимо, що метод addTraitSelector:withMethod: визначний в trait TPureBehavior був переіменований на basicAddTraitSelector:withMethod:.

**6.7 Все відбувається через відправку повідомлень.**

Це правило відображає суть програмування в Фаро.

В процедурних мовах програмування(і в деяких об'єктно-орієнтованих мовах, з статичними методами, таких як Java), вибір того, який код виконувати робиться на стороні коду, що викликає дані процедури. Сторона коду, що викликає, вказує процедуру, яку потрібно статично виконати.

В Фаро, ми не оперуємо понянням “викликати метод”. Замість цього ми відправляємо повідомлення. Це просто термінологія, але має велике значення. Це означає, що це не відповідальність користувача вашого класу вибирати метод для виконання, а це відповідальність утримувача повідомлення вибрати відповідний метод.

Відправляючи повідомлення, ми не верішуємо, який метод буде виконаний. Замість цього ми кажемо об’єкту зробити щось, надіславши йому повідомлення. Повідомленя це ніщо інше, як назва і список аргументів. Отримувач повідомлення потім вирішує, як саме зреагувати, викликавши певний свій метод. Оскільки різні об’єкти мають різні методи, щоб відповідати на однакове повідомлення, методод для виклику повинен вибиратись динамічно, коли повідомлення отримано.

3 + 4

>>> 7 “відправлене повідомлення + з аргументом 4 до цілого числа 3”

(1@2) + 4

>>> 5@6 "відправлене повідомлення + з аргументом 4 до точки (1@2)"

Як результат, ми можемо відправити те саме повідомлення до різних об’єктів, кожен з яких може мати свій власний метод, щоб відповісти на повідомлення. Ми не кажемо SmallInteger 3 чи точці (1@2), як відповісти на повідомлення +4. Кожен має свій власний метод, для повідомлення +, і відповість на +4 відповідно.

Один з результатів такої моделі надсилання повідомлень в Фаро є те, що вона заохочує стиль програмування при якому, об’єкти мають малі методи і делегують завдання до інших об’єктів, замість того, щоб реалізовувати великі, процедурні методи, що беруть на себе дуже багато відповідальності. Джозеф Перлайн висловлює цей принцип наступним чином:

“Не роби нічого, що ти можеш делагувати до когось іншого.”

Багато об’єктно-орієнтованих мов надають статичні і динамічні механізми для об’єктів. В Фаро є тільки динамічний підхід, для надсилання повідомлень. Для прикладу, замість того, щоб надати статичні методи для класів, ми просто надсилаємо повідомлення до класів(як є просто об’єктами)

Добре, майже все у Фаро відбуваєтсья через надсилання повідомлень. У якась момент дій мають відбуватись:

Оголошення змінних не виконуються надсиланням повідомлень. Насправді, оголошення змінних навіть невиконувані вирази. Оголошення змінної просто спричиняє виділення пам’яті для посилання на об’єкт..

Присвоєння не надсилає повідомлення.Присвоєння змінної призводить до того, що назва змінної буде заново пов’язана в межах його визначення.

Return не задіює повідомлень. Return просто повертає обчислений результат до відправника.

Примітиви(і прагми/анотації) також не повідомлення. Вони визначені у віртуальній машині.

Решта, крім цих декількох виразів, тобто майже все решта насправді стається через надсилання повідомлень. Оскільки в Фаро не має публічних полів, єдиний спосіб змінити поле екземпляру іншого об’єкту - це надіслати повідомлення питаючи, щоб об’єкт поміняв своє власне поле. Звісно, надаючи методи для отримання і зміни всіх полів екземпляру не є добрм прикладом об’єктно-орієнтованого програмуванняє. Джозеф Перлайн підкрасли це дуже вдало

“Не дозволяй нікому гратись з твоїм даними”

**6.8 Пошук метода динамічно слідує ієрархії класів**

Що насправді стається коли об’єкт отримує повідомення? Це є двоетапний процес: пошук метода і його виконання.

**Пошук метода.** Спочатку знаходиться метод який має таку ж назву, як і повідомлення.

**Виконання метода**. По-друге, знайдений метод застосовується до отримувача з аргументами повідомлення. Коли повідомлення знайдене, аргументи зв’язуются з параметрами метода, і віртуальна машина виконує його.

Пошук метода є досить простий:

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

Це насправді так, просто як ці павила. Проте є ще декілька питань, на які нам слід відповісти.

* *Що стається з повідомлення, що явно не повертає значення?*
* *Що стається, коли клас перезавантажує метод свого суперкласу?*
* *Яка різниця між self і super self?*
* *Що стається коли не знайшовся метод?*

Правила для пошуку методу, що є тут згадані є концептуальні. Розробники віртуальних машин використовують всі можливі оптимізації і хитрощі, щоб пришвидшити пошук методів. Це їхня робота, але ви ніколи не повинні виявити оптимізації, що суперечать цим правилам.

Спочатку подивимось на базову стратегію пошуку, а потім розглянемо додаткові питання.

**Пошук методу**

Припустимо ми створили екземпляр класу EllipseMorph

Suppose we create an instance of EllipseMorph.

anEllipse := EllipseMorph new.

Якщо ми зараз надішлемо повідомлення defaultColor, ми отримаємо жовтий колір.

anEllipse defaultColor

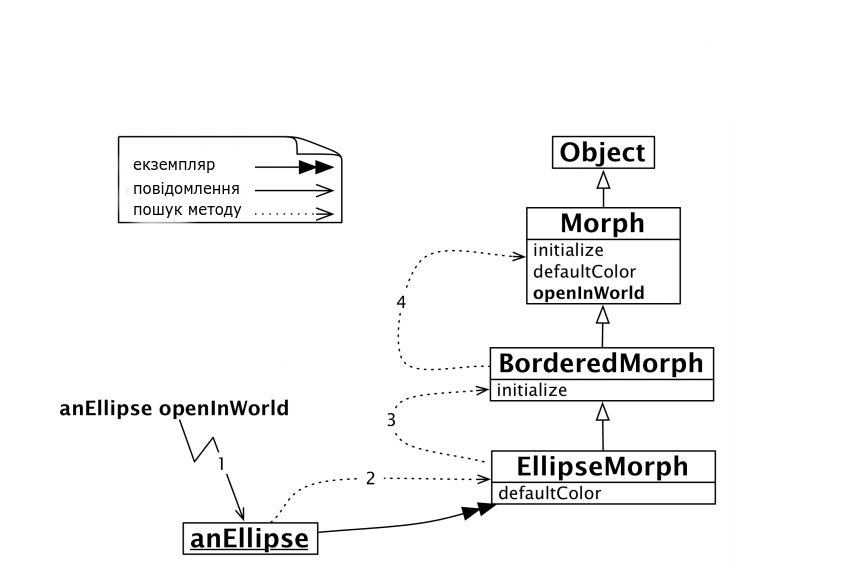
>>> Color yellow

Клас EllipseMorph реалізує метод defaultColor, тому порібний метод знайдений миттєво.

EllipseMorph >> defaultColor

"Повертає колір за замовчуванням"

^ Color yellowFigure



Малюнок 6.2: Пошук метода слідує ієрархії класів.

Навідміну від цього, якщо ми відправимо повідомлення openInWorld до anEllipse, методу не буде зразу знайдений, оскільки клас не реалізовує openInWorld. Пошук тому продовжиться в суперкласі BorderedMorph і так дальше, жопоки openInWorld метод не буде знайдений в класі Morph(дивись малюнок 6.2)

Morph >> openInWorld

"Додай цього morph до світу"

self openInWorld: self currentWorld

**Returning self**

Зауважте, що EllipseMorph>>defaultColor явно повертає Color yellow(жовтий колір), в той час як Morph>>openInWorld ніщо не повертає.

Насправді повідомлення завжди повертає значення на повідомлення(яке є звісно об’єктом). Результат можу бути визначений задопомогою виразу з “^” в методі, але якщо виконання методу завершилось, без виконання “^”, метод також поверне значення. Повідомлення поверне об’єкт, що його отримав. Ми зазвичай говоримо, що метод повертає self. Тому в Фаро псевдо змінна self представляє отримувача повідомлення, подібно до цього ключового слова в Java. Інші мови, такі як Рубі, за замовчування повертають значення останього виразу в методі. В Фаро це не так, замість цього ви можктк собі уявити “^ self” в кінці кожного метода.

**Важливо** self представляє отримувача повідомлення.

Це означає, що openInWorld є еквівалентний openInWorldReturnSelf, визначеного нижче.

This suggests that openInWorld is equivalent to openInWorldReturnSelf, defined below.

Morph >> openInWorld

"Додати цього morph до світу."

self openInWorld: self currentWorld

^ self

Чому явне згадування “^ self” не є дуже добрим підходом? Коли ви повертаєте щось явно, ви повідомляєте, що відправляєте, щось корисне відправнику. Коли ви явно певертаєте self, ви говорите, що відправник очікує це значення. Але це не у цьому випадку, тому краще явно не повертати self. Ми повертаємо self тільки, щоб підкреслити, що отримувач повідомлення повертається методом.

Це є стандатний підхід у фаро, що Кент Бек називає цікавим значенням, що повертається.

*“Повертай значення тільки, коли ти хочеш, щоб відправник повідомлення використав його”*

**Важливо** За замовчуванням(якщо інше не визначено) метод повертає отримувача повідомлення.

**Перезавантаження і виконання**

Якщо ми знову подивимось на ієрархію, для класу EllipseMorph на малюнку 6.2. Ми побачимо, що класи Morph and EllipseMorph реалізовують defaultColor. Насправді, якщо ми відкриємо новий morph(Morph new openInWorld), ми побачимо, що отримали синьго morph, коли еліпс буде жовтий за замовчуванням.

Ми кажемо, що EllipseMorph перезавантажує метод defaultColor, що наслідує його від Morph. Унаслідуваний метод, вже не існує з точки зору anEllipse.

Деколи ми не хочемо перевантажувати унаслідувані методи, а радше розширити їх новою функціональністю. Ми б хотіли мати можливість викликати перезавантажений методу і додатково нову функціональність , яка є визначенна в підкласах. В Фаро, як і в багатьох інших об’єктно-орієнтованих мовах, що підтримують одинична наслідування, це можна реалізувати задопомогою повідомлення до super.

Часте застосування цього механізму можна бачити в методах ініціалізації. Коли новий езкземпляр класу ініціалізується, дуже важливо ініціалізувати всі інші унаслідувані поля. Проте, інформація про те, як це зробити вже знаходиться в методах ініціалізації суперкласів. Підклас навіть не повинен пробувати ініціалізувати поля свої суперкласів!

Доброю практикою є надсилати super initialize в методаї ініціалізації, перед тим, як проводити хоч якусь ініціалізацію.

BorderedMorph >> initialize

"Ініціалізувати стан отримувача"

super initialize.

self borderInitialize

Нам потрібен super, щоб комбінувати унаслідувану поведінку, яка б у іншому випадку була перезавантажена.

**Важливо** Доброю практикою є початок методу ініціалізації з “ super initialize”

**Надсилання повідомлень до self і super**

Self представляє собою отримувача повіломлення іпошук метода почитається з класу отримувача. Що таке super? Super - це не суперклас! Це стандартна помилка, що всі допускають.Також помилково думати, що пошук метода почитанється з суперкласу отримувача повідомлення.

**Важливо** Self представляє отримувача овідомлення і пошук методу починається з класу отримувача.

Як насдсилання повідомлень до self і super відрізняються одне від одного?

Як і self, super представляє собою отримувача повідомлення. Так ви правильно прочитали! Одна річ, що помінялась це пошук метода. Замість того, щоб почати пошук в класі отримувача, пошук починається в супекрласі клас де визначений цей метод..

**Важливо** Super представляє отримувача повідомлення і пошук метода починається в суперкласі класу в якому виконується команда.

За допомогою наступного ми детально побачимо, як це працює. Уявіть собі, що ми визначили наступні три методи:

Спочатку ми визначимо метод fullPrintOn в класі Morph, що просто додає до потоку ім’яє класу з послідуючою стрічкою ‘ new’ - ідея в тому, що ми можемо виконати результуючу стрічку і отримати екземпляр подібний до отримувача.

First we define the method fullPrintOn: on class Morph that just adds to the stream the name of the class followed by the string ’ new’ - the idea is that we could execute the resulting string and gets back an instance similar to the receiver. Morph >> fullPrintOn: aStream

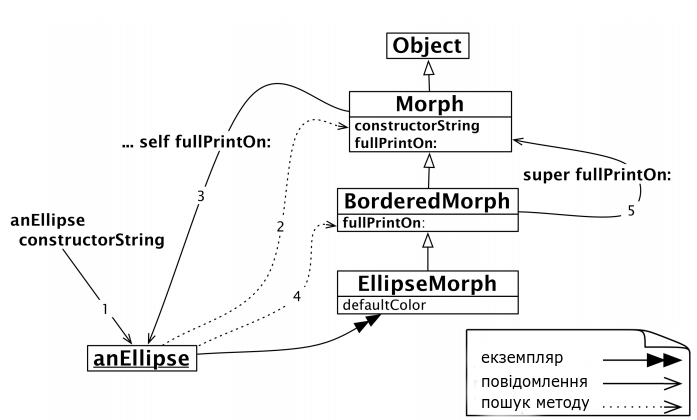
aStream nextPutAll: self class name, ' new'

По-друге, ми визначимо метод constructorString, що відправлятиме поідомлення fullPrintOn:.

Morph >> constructorString

^ String streamContents: [ :s | self fullPrintOn: s ].

І на останок, ми визначимо метод fullPrintOn: в класі BorderedMorph, що є суперкласом EllipseMorph. Цей новий метод розширює поведінку суперкласу.



**Малюнок 6.3: Повідомлення self і superself**

BorderedMorph >> fullPrintOn: aStream

aStream nextPutAll: '('.

super fullPrintOn: aStream.

aStream

nextPutAll: ') setBorderWidth: ';

print: borderWidth;

nextPutAll: ' borderColor: ', (self colorString: borderColor)

Розглянемо повідомлення constructorString відправлене до екземпляру класу EllipseMorph:

EllipseMorph new constructorString

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

Як саме результат отриманий за допомогою комбінації self і super надсилається? По-перше, anEllipse constructorString спричинить знаходження методу constructorString в класі Morph(дивись малюнок 6.3).

Метод Morph>>constructorString виконує відправлення собі(self send) повідомлення fullPrintOn:. Пошук повідомлення fullPrintOn: починається з класу EllipseMorph, і метод BorderedMorph>>fullPrintOn: буде знайдений в BorderedMorph(див. Малюнок 6.3). Важливо зауважити, що відправлення повідомлень до self, спричинює пошук метода з класу отримувача повідомлення, в нашому випадку класу об’єкта anEllipse.

З іншого боку, BorderedMorph>>fullPrintOn: надсилає повідомлення до super, щоб розширити функціональність метода fullPrintOn, що унаслідував від свого суперкласу. Через те, що це є повідомення до super, пошук методу тепер починається в предку класу, в методі якого відбувається надсилання повідомлення до super. Ми тоді зразу знайдемо і виконаємо.

Крок назад

Надсилання повідомлення до self є динамічним в сенсі того, що подивившись на метод, який його містить, ми не можемо знати, який метод буде виконаний. Насправді екземпляр підкласу може отримати повідомлення, яке містить вираз з self і перевизначити метод в підкласі. Тут EllipseMorph може перевизначити метод fullPrintOn: і цей метод буде виконаний методом constructorString. Зауважте, що дивлячись тільки на метод constructorString, ми не можемо передбачити який fullPrintOn: метод ( визначений в класах: EllipseMorph, BorderedMorph і Morph) буде виконаний, якщо виконувати метод constructorString, оскільки це залежить від отримувача повідомлення constructorString.

**Важливо** Повідомлення до self, змушує шукати метод в класі отримувача. Цей пошук є динамічний в сенсі того, що ми не можемо перебачити, корий метод буде виконаний.

Зауважте, що повідомлення до super НЕ починається в суперкласі отримувача. Це б спричинило початок пошуку з класу BorderedMorph, що б виявилось нескінченим циклом.

Якщо добре подумати про повідомлення до super і малюнок 6.3 стане зрозуміло, що прив’язки super є статичними. Все, що має значення - це клас в якому є повідомлення надіслане до super. На відміну від цього, значення self є динамічним. Воно завжди представляє собою утримувача повідомлення, яке зараз виконується. Це означає, що всі повідомлення надістані до self, будуть шукатись починаючи з класу отримувача повідомлення.

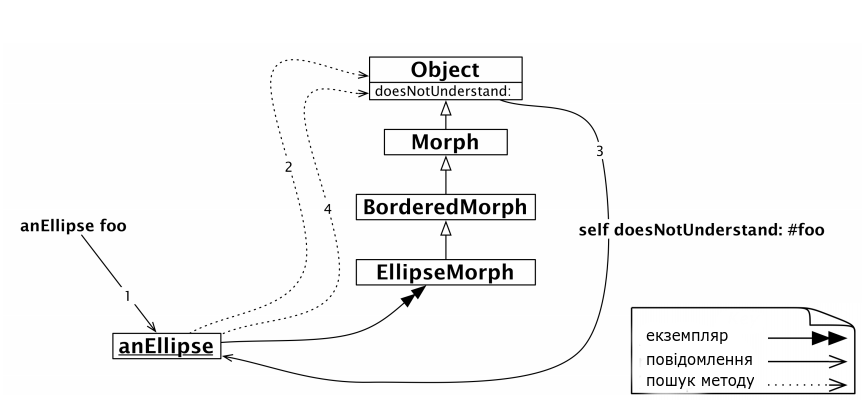
**Важливо** Повідомлення до super, змушує шукати метод починаючи з суперласу класа в якому є метод, що виконує надсиання до super. Ми говоримо, що надсилання повідомлення є статичним, оскільки ми вже знаємо клас в якому пошук повідомлення почнеться.

**Повідомлення незрозуміле**

Що станеться, якщо метод, що ми шукаєм, не буде знайдений?

Припустимо ми надсилаємо повідомлення foo до нашого еліпса. Спочатку звичний пошук методу пройде по всій ієрархії аж до класу Object(у впадку Фаро ProtoObject), шукаючи цей метод. Коли цей метод не буде знайдений, віртуальна машина змусить об’єкт надіслати повідомлення self doesNotUnderstand: #foo.(див. Малюнок 6.4)

Зараз, це є звичне надсилання повідомлення до self, тому пошук почнеться знову з класу EllipseMorph, але цього разу шукаючи метод doesNotUnderstand:. Як ви’являється, Object реалізує метод doesNotUnder- stand:. Цей метод створить об’єкт MessageNotUnderstood, який може запустити відлагоджувач для даного контексту.



Малюнок 6.4: Повідомлення foo є незрозумілим.

Чому ми робими такий довгй шлях, щоб опрацювати таку просту помилку?

Це надає можливість розробникам моживість переривати такі помилки і робити альтернативні дії. Можна перевартажити метод Object>>doesNotUnderstand в будь-якому підкласі і реалізувати свій спосіб опрацювання помилок.

Насправді, це є досить простим способом реалізувати автоматичну делегацію повідомлень з одного об’єкта іншому. Об’єкт делегат може просто перенаправити всі свої незрозумілі повідомлення до іншого об’єкта, відповідалність якого опрацювати їх, і самому згенерувати помилку!

**6.9 Спільні змінні**

Зараз ми оглянимо один з аспектів Фаро, що важко інтерпритувати за допомогою наших п’яти правил:спільні змінні.

Фара надає три різні види спільних змінних:

1. Глобальні спільні змінні
2. Змінні класу: змінні, що є спільними між класом і екземпляром(не путайте з полями класу)
3. Змінні спільні між групою класів.

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

**Глобальні змінні**

В Фаро, всі глобальні змінні зберігаються в просторі імен Smalltalk, який є екземпляром класу SystemDictionary. Глобальні змінні є дуступними зідусіль. Кожен клас є назаний якось глобальною змінною. Також, декілька глобальних змінних використовуються, щоб називати спеціальні або часто використовувані об’єкти.

Змінна Processor є екземпляром класу ProcessScheduler, який є основним планувальником процесів у Фаро.

Processor class

>>> ProcessorScheduler

Інші корисні глобальні змінні

Змінна Smalltalk є екземпляром класу SmalltalkImage. Вона містить функіональність для налаштувань системи. Зокрема, вона містить посилання на основний простір імен глобальних об’єктів Smalltalk. Простір імен включає самого Smalltalk, оскільки це глобальна змінна. Ключі для цього простору імен - це символи, що називають глобальні об’єкти в Фаро.

Приклад:

Smalltalk globals at:

#Boolean >>> Boolean

Оскільки Smalltalk також є глобальною змінною:

Smalltalk globals at: #Smalltalk

>>> Smalltalk

(Smalltalk globals at: #Smalltalk) == Smalltalk

>>> true

World є екземпляром класу PasteUpMorph. Що представляє собою екран.World представляє прямокутник, що визначає простір всього екрану. Всі трансформації на екрані є підтрвнсформацією World.

ActiveHand є теперішнім езкземпляром HandMorph, графічної репрезинтації курсора. Всі submorphs класу ActiveHand містять бсе, що зараз пересувається мишкою.

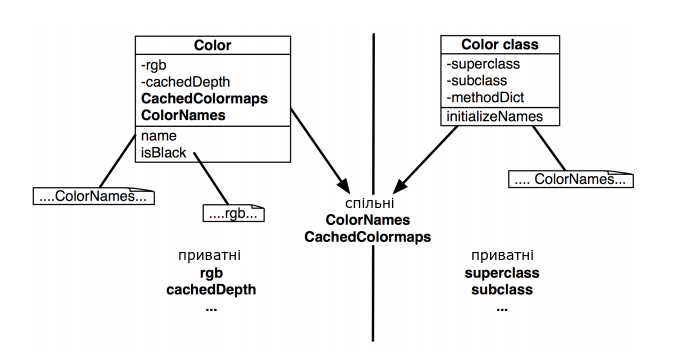
ActiveHand is the current instance of HandMorph, the graphical representation of the cursor. ActiveHand’s submorphs hold anything being dragged by the mouse.

Undeclared - це є інший словник, що містить всі невизначені змінні. Якщо ви напишее метод, що посилається на невизначину змінну, браузер зазвичай нагадає вам визначити її. Для приклажу, якщо ви потім видалете визначення змінної, код буде посилатись на невизначену змінну. Переглянувши Undeclared можна деколи пояснити незрозумілу поведінку.

Використання глобальних змінних в коді.

Рекомендованою практикою є строге обмеження використання глобальних змінних. Зазвичай краще використати поле езкемпляра чи класа, надаючи методи доступу. Якщо б Фаро перписувався заново, то б більшість гобалих змінних було б замінена сингилтонами.

Звичний спосіб, щоб визначити глобальну змінну - це виконати “Do it” над присвоєнням друкованим великим літерами, ще невизначеним, індентифікаторм(ім’ям). Синтаксичний аналізатор запропонує визначити глобальну змінну. Якщо ви хочете визначити глобальну змінну програмним способом, просто виконайте “Smalltalk globals at: #НазваГлобальноїЗмінної put: nil”. Щоб видалити її, виконайте Smalltalk globals removeKey: #НазваГлобальноїЗмінної.



Малюнок 6.5: Доступ методів класу і екземпляру до різних типів змінних.

**Поля класу**

Іноді нам потрібно ділити дані між всіма екземплярами класу і самим класом. Це є можливим за допомогою полів класу. Термін поле класу означає, що час життя змінної є спільний між всіма екземплярами класу і самим класом, як показано на малюнку 6.5. Справді кращою назвою було б спільні змінні оскільки це явно вказує їхню роль, і явно попереджує про небезпеку їх викорстання

На малюнку 6.5 ми бачимо що rgb і cachedDepth полями екземпляру Color, отже доступні тільки екземплярам класу Color. Ми також бачимо, що суперклас, підклас, methodDict і так далі є полями екземпляра класу, так як поля доступні тільки класу Color.

Але ми можемо також побачити дещо нове, ColorRegistry and CachedColormaps є полями класу визначеними в класі Color. Великі букви в назвах цих змінних дають нам підказку, що вони є спільними. Насравді, не тільки всі екземпляри класу можуть до них доступатись, але й клас Color такод, і всіх його підкласи. Також і методи класу і методи екземпляру можуть доступатись до спільних змінних.

Змінна класу є визначена в шаблоні класу. Для приклауд, клас Color визначає велике число полів класу, щоб прищвидчити створення екземпляру. Визначення класу показано нижче.

Object subclass: #Color

instanceVariableNames: 'rgb cachedDepth cachedBitPattern alpha' classVariableNames: 'BlueShift CachedColormaps ColorRegistry

ComponentMask GrayToIndexMap GreenShift HalfComponentMask

IndexedColors

MaskingMap RandomStream RedShift'

package: 'Graphics-Primitives'

Змінна класу ColorRegistry є екземпляром класу IdentityDictionary, що містить часто використовувані об’єкти, доступні за іменами. Цей словник є спільний міє всіма екземплярами Color, як і самим класом. Він доступний зі всіх методів екземпляра і класу.

Ініціалізація класу.

Наявність полів класу спонукає до питання, як їх ініціалізувати?

Один з способів - це “лінива ініціалізація”(описана раніше). Це може бути зроблено додавши метод для доступу до цього поля, який під час свого виконання створить змінну, якщо вона ще не була створена. Це означає, що ми повинні завжди використовувати метод для доступу і ніколи не використовувати поле напряму. Це означає, що ми платиму ціно відпавки повідомлення і перевірки на ініціалізацію. Це також спірно обмежує точку доступу до поля, оскільки наспавді поле вже не є загальнодоступним.

Інший спосіб це перевантажити метод класу initialize(ми це бачили в прикладі з собакою)

Color class >> initialize

...

self initializeColorRegistry.

...

Якщо ви використовуватиме цей варіант, вам слід буде пам’ятати про виклик методу initialize після того як визначили його(викликавши Color initialize). Хоча методи ініціалізації на стороні класу виконуються автоматично, коли код завантажується в пам’яті. Вони не виконуються автоматично, коли спочатку записані в браузері і скомпільовані, або вони змінені чи перекомпільовані.

**Змінні в Пулі**

Змінні на пулі - це змінні, що х спільними для декількох класів, що можуть бути не пов’язані наслідуванням. Змінні в пулі зазвичай зберігались в спільних словниках. Зараз вони повинні бути визначені, як поле наперед визначеного класу(який є підкласмо SharedPool). Наша порада - уникати їх. Вам вони можуть здатись у нагоді тільки у рідкісних і специфічних умовах. Наша мета полягає у тому, щоб пояснити про них такою мірою, щоб ви змогли зрозуміти їх читаючи код.

Клас повинен згадати назву пула клас у визначенні, щобдоступатись до змінних на пулі. Для прикладу, клас Text визнаає, що він буде використовуваи словник TextConstants, що містить всі текстові константи, такі, як CR і LF. Цей словник має ключ #CR, який з’язаний з значенням символа переводу каретки.

ArrayedCollection subclass: #Text

instanceVariableNames: 'string runs'

classVariableNames: ''

poolDictionaries: 'TextConstants'

package: 'Collections-Text'

Це дозволить методам класа доступатись до ключів словника в тілі метода, так і використовувати синтаксис для змінних, замість явного пошука в словнику. Для прикладу ми можемо написати такий метод:

Text >> testCR

^ CR == Character cr

І знову ж таки, ми рекомендуємо вам уникати викорстання змінних на пулі.

**6.10 Зауваження до внутрішньої реалізації об’єктів**

Тут є додаткова інформація про внутрішню реалізацію об’єктів в Фаро, для тих хто хоче глибше зрозуміти Фаро. Реалізація відрізняється для двох типів об’єктів:

1. Об’єкти, що передаються через посилання і існують на купі.
2. Об’єкти, що передаються за значенням. Залежно від версії, це є різні типи цілих чисел: SmallInteger, всі символи, і можливо, якась підмножина чисел з плавоючою комою, як SmallFloat64. Такі внутрішні об’єкти займають місце вказівника на об’єкт, більшість бітів якого кодують проміжне внутрішнє представлення значення і деякі з бітів кодують клас об’єкта.

Перший тип об’єктів це звичайні об’єкти, існують в декількох варіантах:

1. Нормальні об’єкти, що мають нулі чи більше полів класу, такі як Point, який має поля x та y. Кожен екземпляр класу містить вказівник на об’єкт, який може бути посиланням на інший об’єкт чи на вбудований об’єкт.
2. Об’єкти, що можуть індексуватись, як і масиви, що мають нуль чи більше полів класу пронумерованих від 1 до N. Кожен індексований екземпляр тримає вказівник на об’єкт, який може бути посиланням на інший об’єкт чи вбудований об’єкт. Доступитись до об’єктів, що індексуються, можна за допомогою повідомлень at: і at:put:. Для прикладу,((Array new: 1) at: 1 put: 2; at: 1) поверне 2.
3. Об’єкти, як Closure чи Context, які мають які мають, які і іменовані поля, так і індексовані. В такому об’єкті індексовані поля слідують за іменованими.
4. Об’єкти такі, як ByteString or Bitmap, що мають індексовані поля пронумеровані від 1 до N, що містять необроблені дані(внутрішня репрезентація даних). Кожна дана величина може займати 8,16 чи цідих 32 біти, залежучи від визначення класу. До даних можу досупатись, як до цілих цілих чисел, символів чи до дробових чисел, в залежності від того, як методи at: і at:put: реалізовані. Методи at: і at:put: конвертують необроблені дані до об’єктів Фаро і навпаки, ховаючи внутрішню реалізацію, дозволяючи системі ефективна зберігати такі типи, як стрічки, чи бітмапи.

Краса Фаро полягає в тому, що вам зазвичай не потріно перейматись різницею, цих трьох типів об’єктів.

**6.11 Підсумки озділу**

Об’єктна модель фаро є одночасно простою і універсальною. Все є об’єктом, і майже все відбувається через надсилання повідомлень.

* Все є об’єктом. Примітивні сутності, як цілі числа, так і класи і їх метакласи.
* Кожен об’єкт є екземпляром класу. Класи визначають структуру їхніх екземплярів через приватні поля і їхню поверідку через їхні публічні методи. Кожен клас є унікальним екземпляром метакласу. Поля класу є приватні змінні доступні всіма екземплярами класу і самим класом. Класи не можуть напряму доступатись до полів екземпляру, екземпляри не можуть доступатись до полів їх класів. Методи доступу повинні бути визначені, якщо це потрібно.
* Кожен клас має суперклас. Корнем наслідування всіх об’єктів є ProtoObject. Класи, що визначає користувач, зазвичай повинні наслідуватись від класу Object/, або його підкласів. Не існує синтаксису, щоб визначити, що клас є абстрактним. Абстрактний клас, це просто клас з одним чи більше абстрактним методом(який містить вираз self subclassResponsibility). Хоча Фаро підтримує тільки одинична наслідування, досить просто ділити реалізацію методів між класами використовуючи traits.
* Все відбувається через надсилання повідомлень. Ми не виконуємо методи, ми надсилаємо повідомлення. Отримувач поідомлення вибирає власний метод, щоб відовісти на дане повідомлення.
* Пошук методу слідує ієрархії насліжування. Відправлення повідомлень до self є динамічне і починє пошук методу з класу, що тримав повідомлення, коли повідомення до super починають пошук методу в суперкласі класу, який містить метод з даним повідомленням. З цієї очки зору повідомлення до super є більшеш статичні.
* Існують три види спільних змінних. Глобальні змінні доступні будь-де в системі. Поля класу, є спільними для класу, його підкласів і екземплярів. Змінні на пулі є спільними між декількома вибраними класами. Вам слід уникати спільних змінних.