Лекция 9
Scheme и объектно-ориентированное программирование

Объектно-ориентированное программирование

Набросаем план:

- Абстракция данных, объединяющая состояние и функции
- Механизм передачи сообщений
- Объектные модели
  - диаграмма классов
  - диаграмма объектов
- Пример: «текстовая бродилка» (iFiction / interactive fiction). Родоначальник жанра игра «Colossal Cave Adventure»
  https://www.ifiction.org/games/playz.php
  https://jerz.setonhill.edu/if/canon/Adventure.htm

Абстракции

- Процедурные абстракции
- Абстракции данных
- Общее предназначение: выражать сложные понятия через более простые, скрывая детали.
- Вопросы:
  - Как лучше всего представить систему набором абстракций?
  - Как сделать простым расширение системы?
    - Добавлять новые типы данных?
    - Добавлять новые функции?

Подход «от данных»

- Структуры данных
  - Сборка сложных структур с помощью cons, векторов:
    point, line, 2dshape, 3dshape
  - Использование заглавного звена – тега, указывающего тип данных
    (define (make-point x y) (list 'point x y))
  - Реализация структуры данных требует набора функций:
    конструктора, селекторов, мутаторов, …

- Обобщённые операции

```
(define (scale x factor)
      (cond ((point? x)  (point-scale x factor))
            ((line? x)   (line-scale x factor))
            ((2dshape? x)(2dshape-scale x factor))
            ((3dshape? x)(3dshape-scale x factor))
            (else (error ”unknownType”))))
```

Обобщённые операции

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale

Обобщённые операции

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans

Обобщённые операции

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans
color	point-color	line-color	2dshape-color	3dshape-color

Обобщённые операции

- при добавлении новой функции просто добавляем новую обобщённую операцию **new-op**

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans
color	point-color	line-color	2dshape-color	3dshape-color

Обобщённые операции

- при добавлении новой функции просто добавляем новую обобщённую операцию **new-op**

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans
color	point-color	line-color	2dshape-color	3dshape-color
new-op	point-new…	line-new…	2dshape-new…	3dshape-new…

Обобщённые операции

- при добавлении нового типа данных приходится переписывать **каждую** обобщённую функцию, что трудоёмко!

	Point	**Line**	**2-d**	**3-d**	**curve**
scale	point-scale	line-scale	2d-scale	3d-scale	cu-scale
translate	point-trans	line-trans	2d-trans	3d-trans	cu-trans
color	point-color	line-color	2d-color	3d-color	cu-color
new-op	point-new…	line-new…	2d-new…	3d-new…	cu-new…

Два взгляда на мир

	Point	**Line**	**2-d**	**3-d**	**curve**
scale	point-scale	line-scale	2d-scale	3d-scale	cu-scale
translate	point-trans	line-trans	2d-trans	3d-trans	cu-trans
color	point-color	line-color	2d-color	3d-color	cu-color
new-op	point-new…	line-new…	2d-new…	3d-new…	cu-new…

- Выбираем то, что соответствует путям модификации системы:
  обобщённые операции – если не будет новых типов;
  объекты данных – если семейство типов расширяемое.

Что такое объект данных?

- Это структура данных
  - соединённая с набором собственных операций;
  - для которой есть общее определение (Line), и экземпляры (line17);
  - экземпляр хранит свои свойства (состояние)

- Мы изобрели ООП!

ООП в Scheme: Функции и состояние

- У функции есть
  - **параметры** и **тело** описанные в lambda-выражении
  - **окружение,** где хранятся связывания её имён
- Можно использовать функцию для хранения (и сокрытия) данных в локальных переменных и предоставления доступа к ним.
- При вызове функции-конструктора создаётся новое окружение.
- Нужно иметь доступ из функции-объекта к этому окружению:
  - для чтения нужны операции-селекторы (или геттеры);
  - для изменения состояния нужны мутаторы;
  - состояние – это текущие связывания в кадре, созданном при вызове конструктора.

Пример: мутируемые пары как объекты

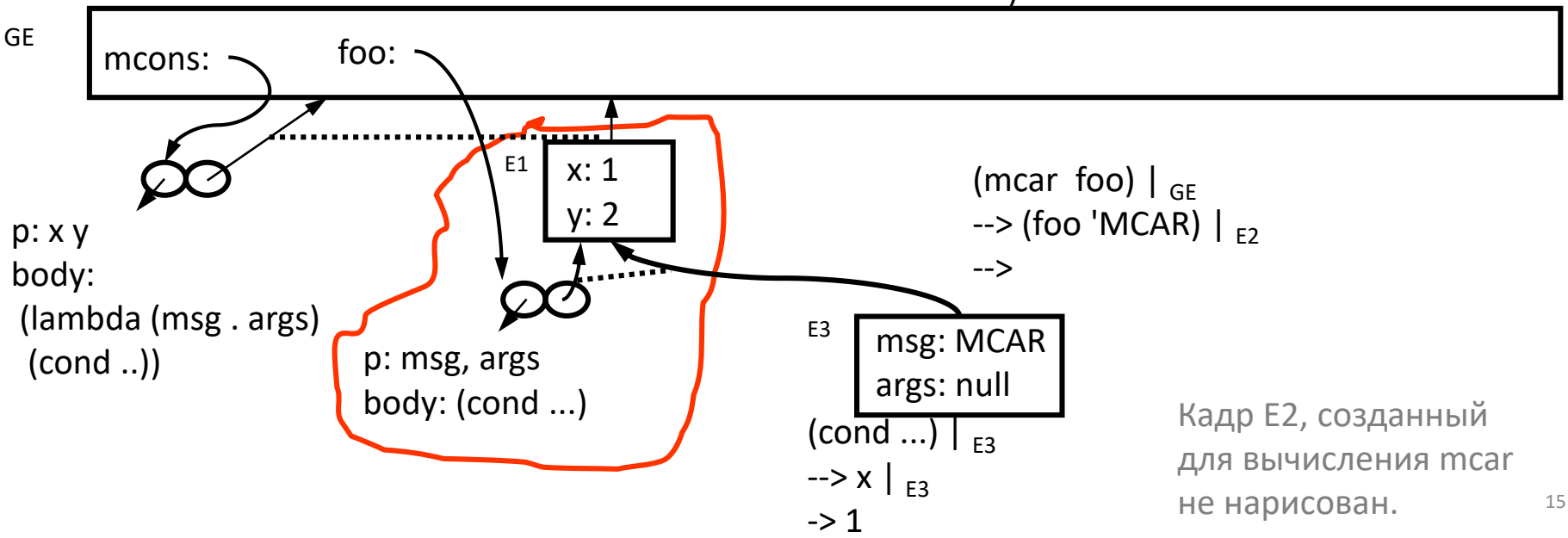
```
(define (mcons x y) ; конструктор мутируемой пары
  (lambda (msg . args) (cond ((eq? msg 'MCAR) x)
                             ((eq? msg 'MCDR) y)
                             ((eq? msg 'MPAIR?) #t)
                             ((eq? msg 'SET-MCAR) (set! x (car args)))
                             ((eq? msg 'SET-MCDR) (set! y (car args)))
                             (else (error ”WRONG MESSAGE”)))))
(define (mcar p)   (p 'MCAR)) ; селекторы
(define (mcdr p)   (p 'MCDR))
(define (set-mcar! p v)   (p 'SET-MCAR v)) ; мутаторы
(define (set-mcdr! p v)   (p 'SET-MCDR v))
(define (mpair? p) (and (procedure? p) (p 'MPAIR?))) ; чеккер
```

Пара-объект с точки зрения MBO

(define foo (mcons 1 2))

(mcar foo) --> (foo 'MCAR)

```
(define (mcons x y)
  (lambda (msg . args)
    (cond ((eq? msg 'MCAR) x)
          ((eq? msg 'MCDR) y)
          ((eq? msg 'MPAIR?) #t)
          ...)
```



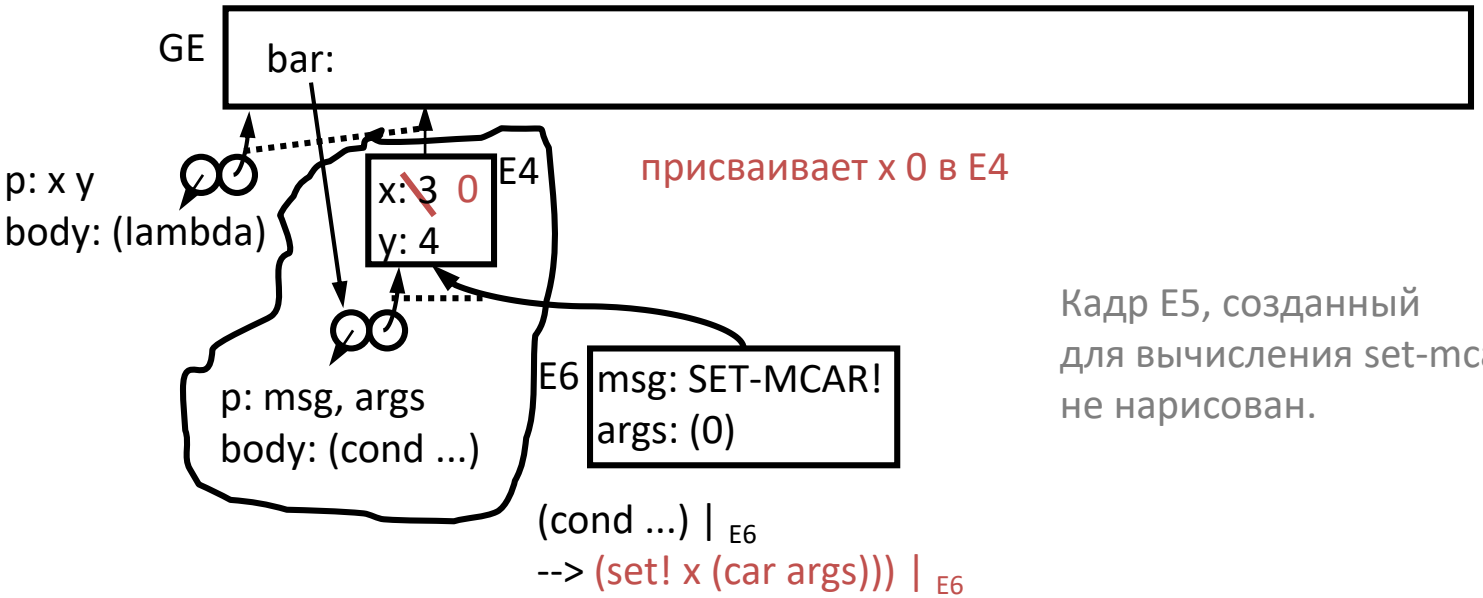
Кадр E2, созданный для вычисления mcar не нарисован.

Пример мутации в MBO

(define bar (mcons 3 4))

(set-mcar! bar 0)

(set-mcar! bar 0) | $_{GE}$
--> (bar 'SET-MCAR! 0) | $_{E5}$



Кадр E5, созданный для вычисления set-mcar! не нарисован.

Стили программирования: функциональный и объектно-ориентированный

- Функциональное программирование:
  - Система организуется как набор функций, обрабатывающих данные

  (do-something <data> <arg> ...)
  (do-another-thing <data>)
- ООП:
  - Система организуется как набор объектов, обменивающихся сообщениями

  (<object> 'do-something <arg>)
  (<object> 'do-another-thing)
  - Объект инкапсулирует данные и операции

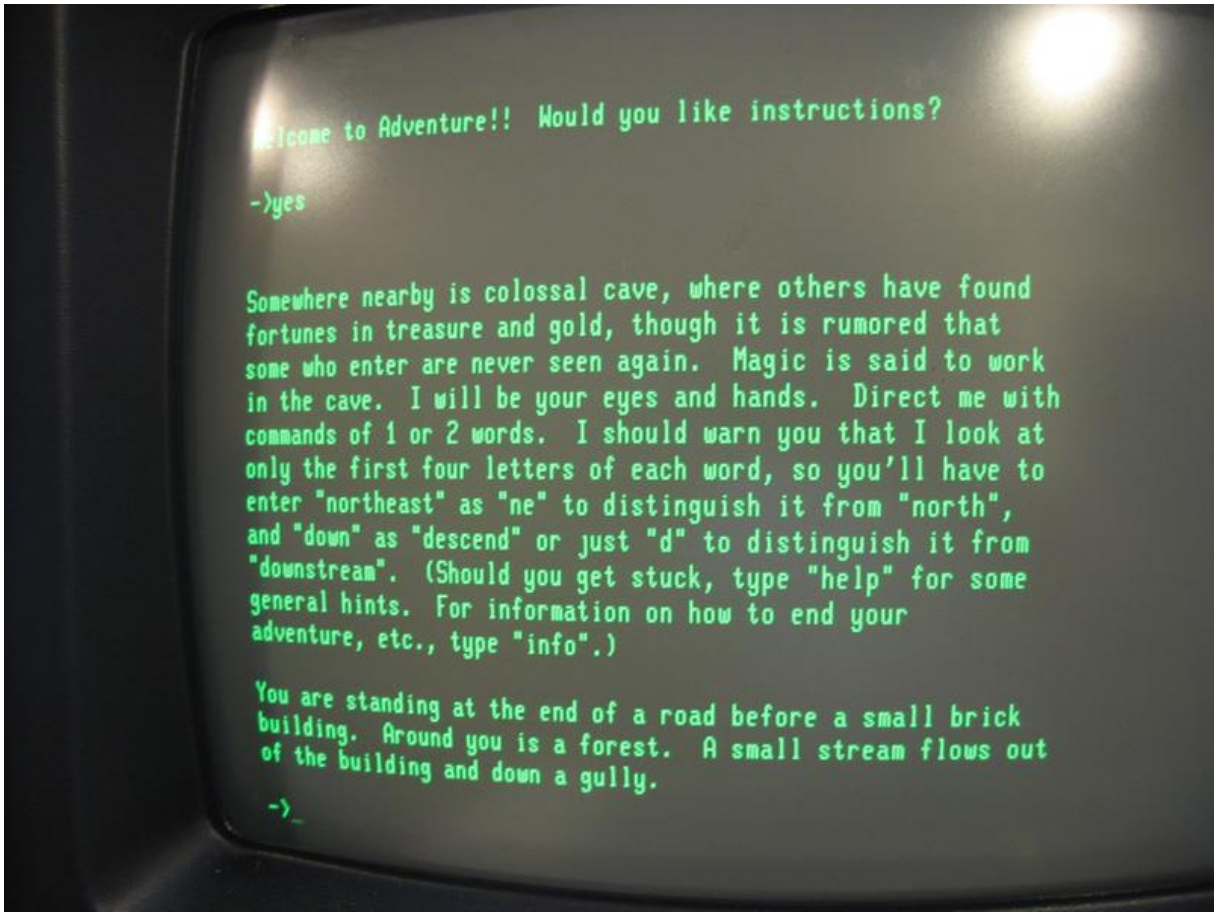
Основные термины ООП

- **Класс**
  - описывает общую структуру и поведение однотипных экземпляров;
  - в Scheme класс описывается функцией-конструктором;
  - в Racket класс описывается спец. формой class.
- **Экземпляр**:
  - отдельный объект некоторого класса;
  - в Scheme, объект – это функция-обработчик сообщений, результат вызова операции-конструктора некоторого класса;
  - в Racket экземпляр порождается new;
  - в Racket отправить сообщение можно send, dynamic-send, а работать со значением атрибута можно через get-field и set-field!.

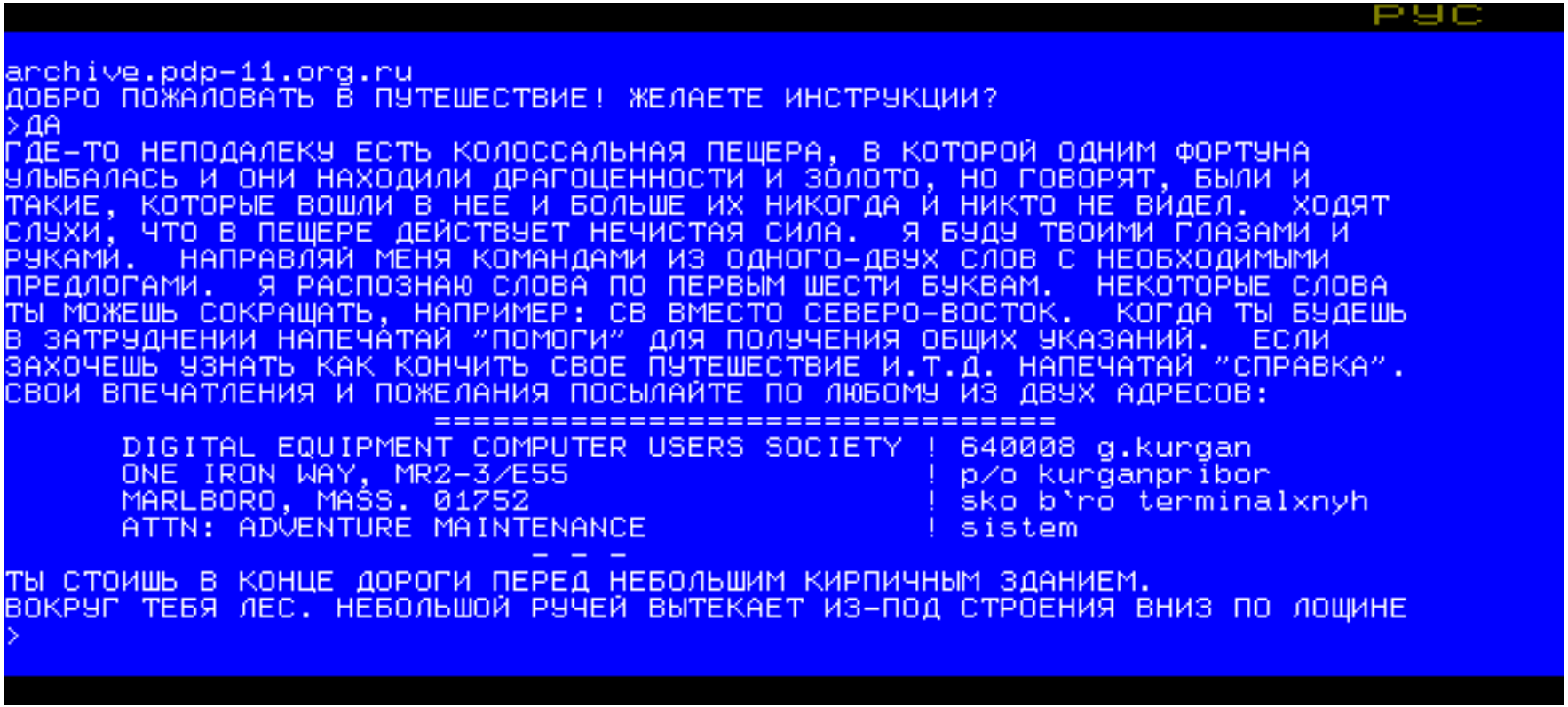
Пример. «Текстовая бродилка по MIT»

- В игре участвуют
  - один аватар игрока (**'my-avatar**)
  - студенты, тролли
  - одни часы для смены состояния игрового мира после каждого хода
  - объекты других типов: места, вещи
- Примеры экземпляров классов
  - **'student-street** место
  - **'sicp** вещь
  - **'registrar** тролль
  - **'alyssa-hacker** студент(ка)

1975 год «Colossal Cave» PDP-11 Уилл Краудер



УК-НЦ или ДВК


РУС
archive.pdp-11.org.ru
ДОБРО ПОЖАЛОВАТЬ В ПУТЕШЕСТВИЕ! ЖЕЛАЕТЕ ИНСТРУКЦИИ?
>ДА
ГДЕ-ТО НЕПОДАЛЕКУ ЕСТЬ КОЛОССАЛЬНАЯ ПЕЩЕРА, В КОТОРОЙ ОДНИМ ФОРТУНА
УЛЫБАЛАСЬ И ОНИ НАХОДИЛИ ДРАГОЦЕННОСТИ И ЗОЛОТО, НО ГОВОРЯТ, БЫЛИ И
ТАКИЕ, КОТОРЫЕ ВОШЛИ В НЕЕ И БОЛЬШЕ ИХ НИКОГДА И НИКТО НЕ ВИДЕЛ. ХОДЯТ
СЛУХИ, ЧТО В ПЕЩЕРЕ ДЕЙСТВУЕТ НЕЧИСТАЯ СИЛА. Я БУДУ ТВОИМИ ГЛАЗАМИ И
РУКАМИ. НАПРАВЛЯЙ МЕНЯ КОМАНДАМИ ИЗ ОДНОГО-ДВУХ СЛОВ С НЕОБХОДИМЫМИ
ПРЕДЛОГАМИ. Я РАСПОЗНАЮ СЛОВА ПО ПЕРВЫМ ШЕСТИ БУКВАМ. НЕКОТОРЫЕ СЛОВА
ТЫ МОЖЕШЬ СОКРАЩАТЬ, НАПРИМЕР: СВ ВМЕСТО СЕВЕРО-ВОСТОК. КОГДА ТЫ БУДЕШЬ
В ЗАТРУДНЕНИИ НАПЕЧАТАЙ "ПОМОГИ" ДЛЯ ПОЛУЧЕНИЯ ОБЩИХ УКАЗАНИЙ. ЕСЛИ
ЗАХОЧЕШЬ УЗНАТЬ КАК КОНЧИТЬ СВОЕ ПУТЕШЕСТВИЕ И.Т.Д. НАПЕЧАТАЙ "СПРАВКА".
СВОИ ВПЕЧАТЛЕНИЯ И ПОЖЕЛАНИЯ ПОСЫЛАЙТЕ ПО ЛЮБОМУ ИЗ ДВУХ АДРЕСОВ:
DIGITAL EQUIPMENT COMPUTER USERS SOCIETY ! 640008 g.kurgan
ONE IRON WAY, MR2-3/E55 ! p/o kurganpribor
MARLBORO, MASS. 01752 ! sko b`ro terminalxnyh
ATTN: ADVENTURE MAINTENANCE ! sistem
ТЫ СТОИШЬ В КОНЦЕ ДОРОГИ ПЕРЕД НЕБОЛЬШИМ КИРПИЧНЫМ ЗДАНИЕМ.
ВОКРУГ ТЕБЯ ЛЕС. НЕБОЛЬШОЙ РУЧЕЙ ВЫТЕКАЕТ ИЗ-ПОД СТРОЕНИЯ ВНИЗ ПО ЛОЩИНЕ
>

Класс **Place** на Scheme

```
(define (create-place name)
   (define (get-name) name) ; описана как пример операции
   (lambda (msg)
     (cond ((eq? msg 'get-name) (get-name))
           (else (error "WRONG MESSAGE")))))
```

Класс **Place** в Racket

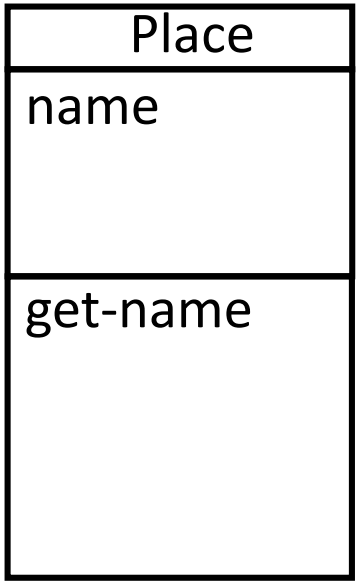
```
(require racket/class)
…
(define place% ; % в конце имени – соглашение об именовании классов
  (class object%
    (super-new)
    (init-field name) ; атрибут, инициируемый конструктором
    (define/public (get-name) name) ; операция-геттер
))
```

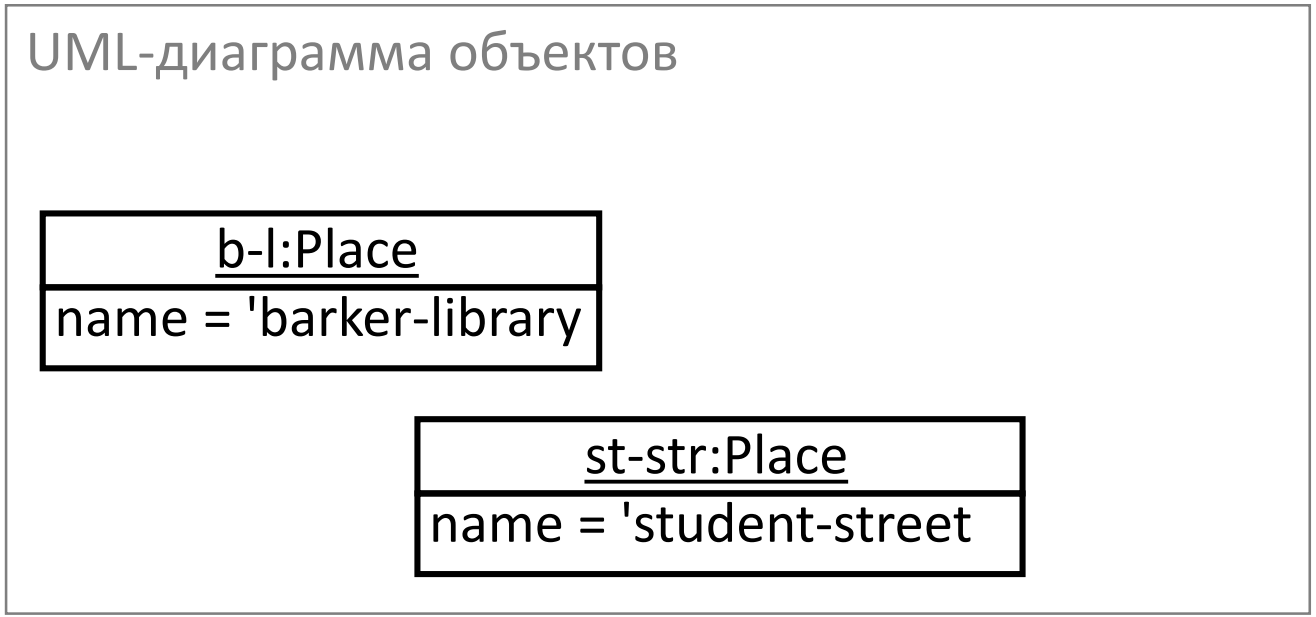
Класс **Place** на UML-диаграмме классов

В UML класс и типы именуют с заглавной буквы, а операции, атрибуты, поля – со строчной.

Объекты на Scheme

```
> (define b-l (create-place 'barker-library))
> (define st-str (create-place 'student-street))
> (b-l 'get-name) -> barker-library
```





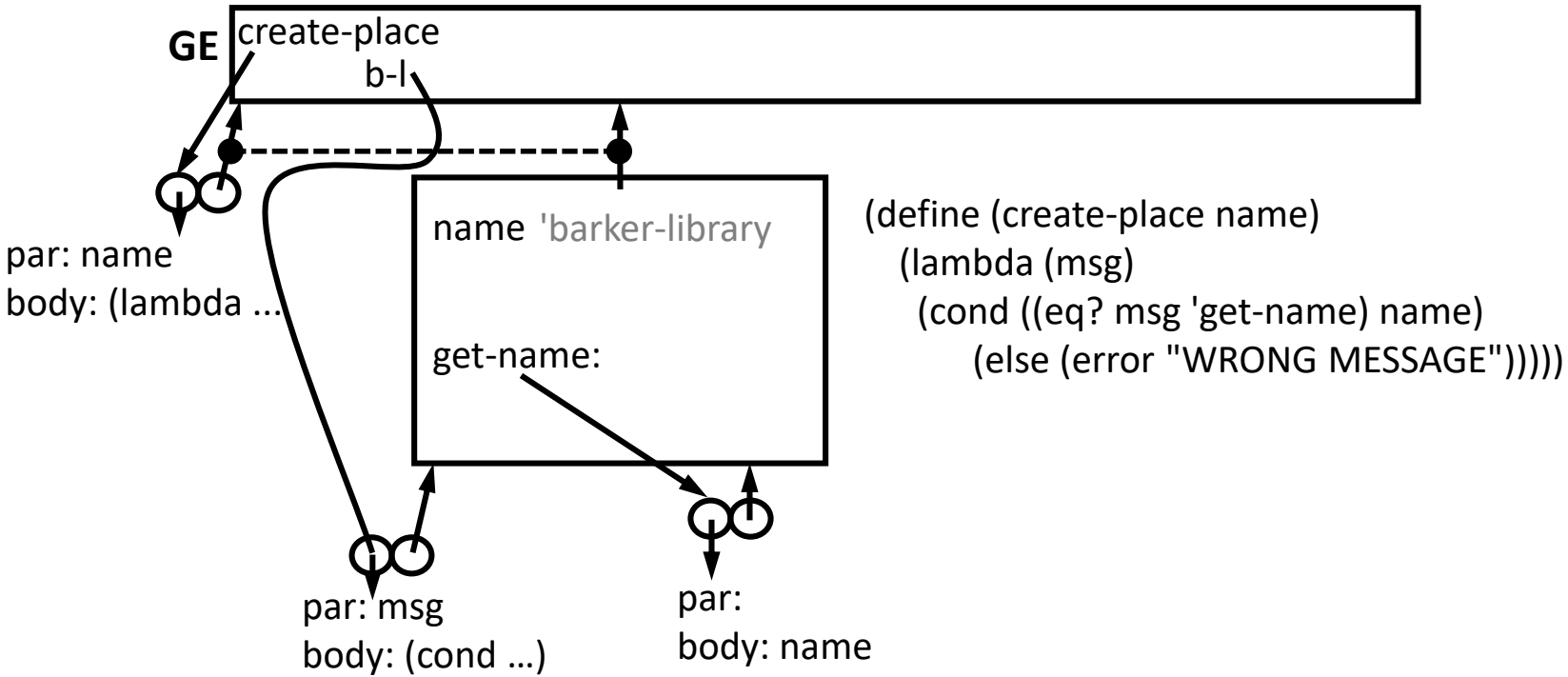
Объекты в Racket

```
(define b-l (new place% (name 'barker-library)))
(define st-str (new place% (name 'student-street)))
> (send b-l get-name) -> barker-library ; вызов операции-геттера
> (get-field name st-str) -> student-street ; вызов геттера по умолчанию
```

С точки зрения МВО

> (define b-l (create-place 'barker-library))

> (b-l 'get-name) -> barker-library




```
(define (create-place name)
   (lambda (msg)
     (cond ((eq? msg 'get-name) name)
           (else (error "WRONG MESSAGE")))))
```

Наполняем мир «MIT»

- Добавим вещи – класс **Thing**. У вещи есть локация – место, экземпляр **Place**. Для этого обновим UML-диаграмму классов:

- Вместо атрибута **Thing::location: Place**
  и атрибута **Place::things: Thing[0..*]**
  мы заводим связь между классами – ассоциацию.

Класс **Thing** на Scheme

```
(define (create-thing name location)
 (define (this msg)
   (cond
        ((eq? msg 'get-name) name)
        ((eq? msg 'get-location) location)
        (else (error "WRONG MESSAGE"))))
   (begin (location 'put-thing! this) this))
```

Обновлённый класс **Place** на Scheme

```
(define (create-place name)
   (let ((things '())) ; список вещей в этой локации
   (lambda (msg . msg-args)
     (cond ((eq? msg 'get-name) name)
           ((eq? msg 'get-things) things)
           ((eq? msg 'put-thing!) (set! things (cons (car msg-args) things)))
           (else (error "WRONG MESSAGE"))))))
> (define b-l (create-place 'barker-library))
> (define st-str (create-place 'student-street))
> (define sicp (create-thing 'sicp b-l))
> (define e-book (create-thing 'engineering-book b-l))
> (define trees (create-thing 'trees st-str))
```

Класс **Thing** в Racket

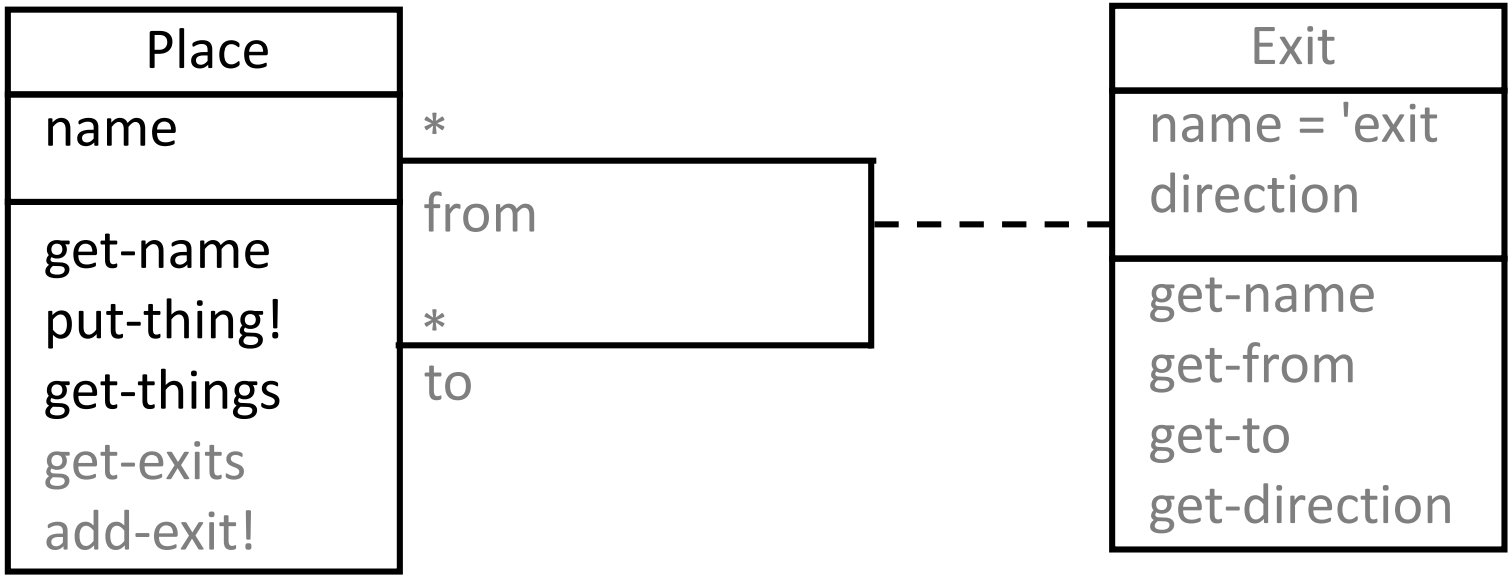
```
(define thing%
  (class object%
    (super-new)
    (init-field name location)
    (define/public (get-location) location)
    (define/public (get-name) name)
    (send location put-thing! this)
  ))
```

Обновлённый класс **Place** в Racket

```
(define place%
  (class object%
    (super-new)
    (init-field name)
    (field (things '()))
    (define/public (get-name) name)
    (define/public (get-things) things)
    (define/public (put-thing! thing) (set! things (cons thing things)))
))
> (define b-l (create-place 'barker-library))
> (define sicp (create-thing 'sicp b-l))
> (define e-book (create-thing 'engineering-book b-l))
```

Продолжаем наполнять мир «MIT»

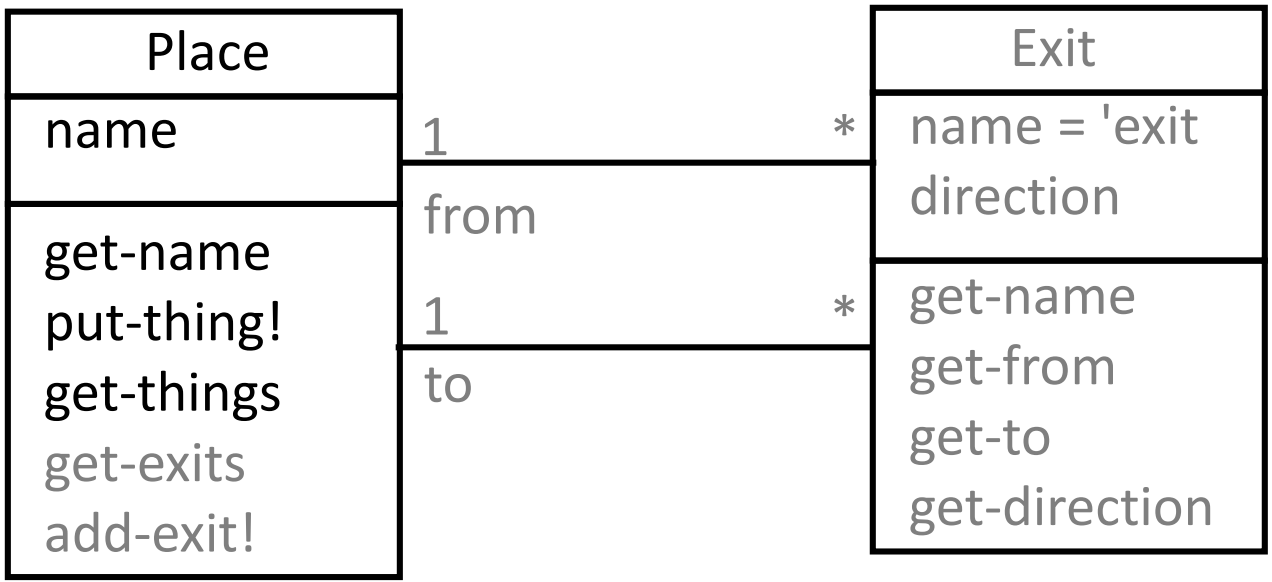
- Добавим выходы (**Exit)**. Выход имеет направление и несимметрично соединяет два экземпляра **Place**. Снова обновим UML-диаграмму классов:



- Заводим связь между классами – класс ассоциации **Exit**.

Класс материализованной связи

- Другой способ показать **Exit** на UML-диаграмме классов:



- Вообще говоря, так мы допускаем наличие нескольких выходов, несимметрично соединяющих одну и ту же пару мест.

Класс ассоциации **Exit** на Scheme

```
(define (create-exit from to direction)
 (define (this msg)
   (cond
        ((eq? msg 'get-name) 'exit)
        ((eq? msg 'get-from) from)
        ((eq? msg 'get-to) to)
        ((eq? msg 'get-direction) direction)
        (else (error "WRONG MESSAGE"))))
    (begin (from 'add-exit! this) this))
```

Обновлённый класс **Place** на Scheme

```
(define (create-place name)
   (let ((things '()) (exits (make-hash)))
   (lambda (msg . msg-args)
     (cond ((eq? msg 'get-name) name)
           ((eq? msg 'get-things) things)
           ((eq? msg 'get-exits) exits)
           ((eq? msg 'put-thing!) (set! things (cons (car msg-args) things)))
           ((eq? msg 'add-exit!)
                 (let ((exit (car msg-args)))
                       (hash-ref! exits ((exit 'get-to) 'get-name) exit)))
            (else (error "WRONG MESSAGE"))))))
```

Класс **Exit** в Racket

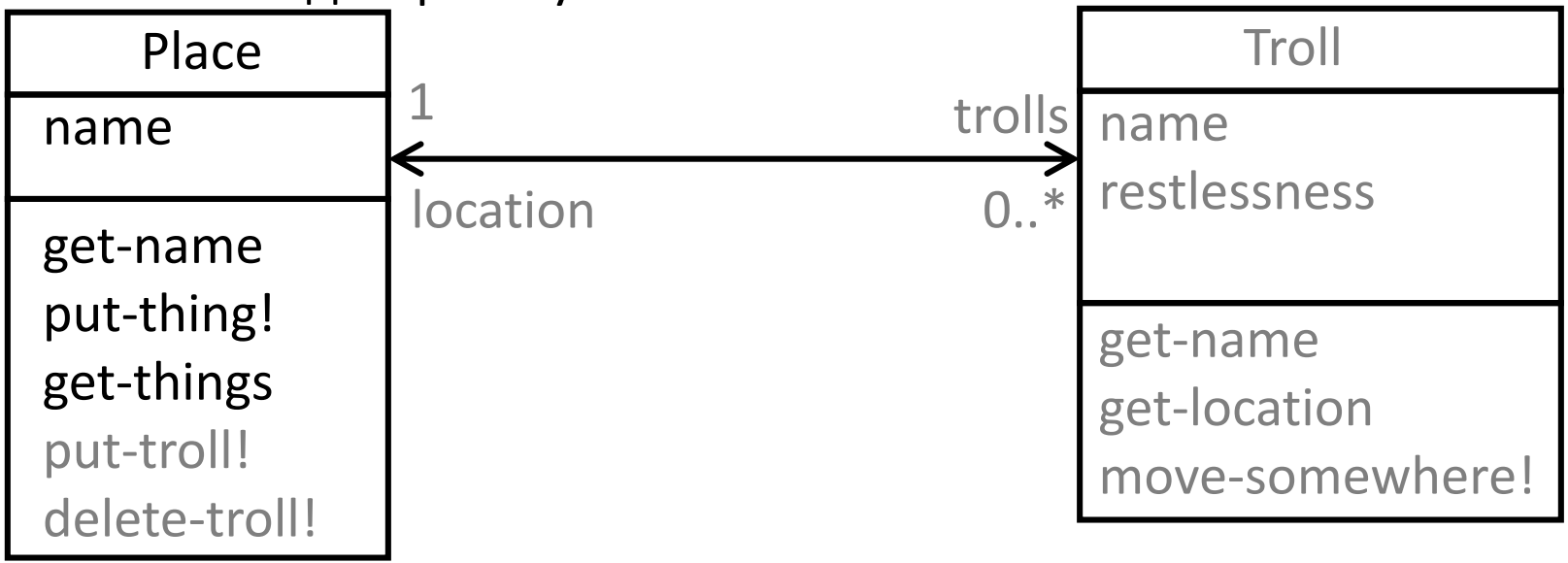
```
(define exit%
  (class object%
    (super-new)
    (init-field from to direction)
    (field (name 'exit))
    (define/public (get-from) from)
    (define/public (get-to) to)
    (define/public (get-direction) direction)
    (define/public (get-name) name)
    (send from add-exit! this)
  ))
```

Обновлённый класс **Place** в Racket

```
(define place%
  (class object%
    (super-new)
    (init-field name)
    (field (things '()) (exits (make-hash)))
    (define/public (get-name) name)
    (define/public (get-things) things)
    (define/public (get-exits) exits)
    (define/public (put-thing! thing) (set! things (cons thing things)))
    (define/public (add-exit! exit)
                   (hash-ref! exits (send (send exit get-to) get-name) exit))
))
```

Продолжаем наполнять мир «MIT»

- Добавим троллей (**Troll)**. Тролль имеет имя, локацию, неугомонность. Он может перемещаться с места на место через выходы. Снова обновим UML-диаграмму классов:



Класс **Troll** на Scheme

```
(define (create-troll name location restlessness)
 (define (this msg)
   (cond
       ((eq? msg 'get-name) name)
       ((eq? msg 'get-location) location)
       ((eq? msg 'move-somewhere)
                (when (< (random) restlessness)
           (let* ((exit (pick-random-list (hash-values (place 'get-exits))))
                                  (to (exit 'get-to))) (when (not (equal? location to))
                               (begin (location 'delete-troll! this)
                               (to 'put-troll! this) (set! location to))))))
       (else (error "WRONG MESSAGE"))))
   (begin (location 'put-troll! this) this))
```

Обновлённый класс **Place** на Scheme

```
(require racket/set)
(define (create-place name)
  (let ((things '()) (exits (make-hash)) (trolls (set)))
  (lambda (msg . msg-args)
    (cond ((eq? msg 'get-name) name)
          ((eq? msg 'get-things) things)
          ((eq? msg 'get-exits) exits)
          ((eq? msg 'put-thing!) (set! things (cons (car msg-args) things)))
          ((eq? msg 'add-exit!) (let ((exit (car msg-args)))
                                  (hash-ref! exits ((exit 'get-to) 'get-name) exit)))
          ((eq? msg 'put-troll!) (set-add! trolls (car msg-args)))
          ((eq? msg 'delete-troll!) (set-remove! trolls (car msg-args)))
          (else (error "WRONG MESSAGE"))))))
```

Класс **Troll** в Racket

```
(define troll%
  (class object%
    (super-new)
    (init-field name location restlessness)
    (define/public (get-location) location)
    (define/public (move-somewhere) (when (< (random) restlessness)
        (let* ((exit (pick-random-list (hash-values (send location get-exits))))
               (to (send exit get-to))) (when (not (equal? location to))
                                          (begin (send location delete-troll! this)
                                          (send to put-troll! this) (set! location to))))))
    (define/public (get-name) name)
    (send location put-troll! this)
  ))
```

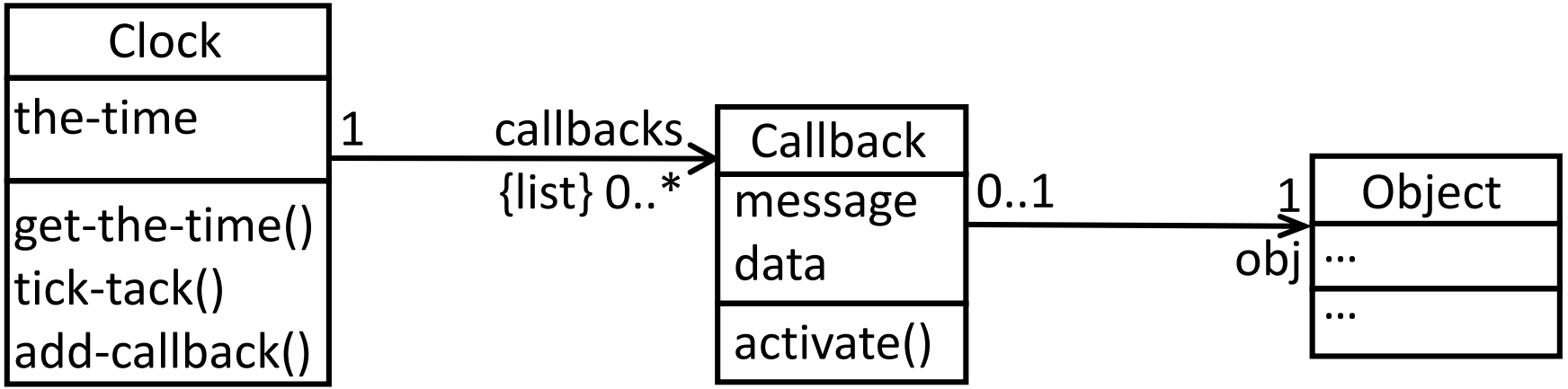
Обновлённый класс **Place** в Racket

```
(require racket/set) ; работа с множествами
(define place%
  (class object%  (super-new)
    (init-field name)
    (field (things '()) (exits (make-hash)) (trolls (set)))
    (define/public (get-name) name)
    (define/public (get-things) things)
    (define/public (get-exits) exits)
    (define/public (put-troll troll) (set-add! trolls troll))
    (define/public (put-thing! thing) (set! things (cons thing things)))
    (define/public (delete-troll! troll) (set-remove! trolls troll))
    (define/public (add-exit! exit)
                   (hash-ref! exits (send (send exit get-to) get-name) exit))))
```

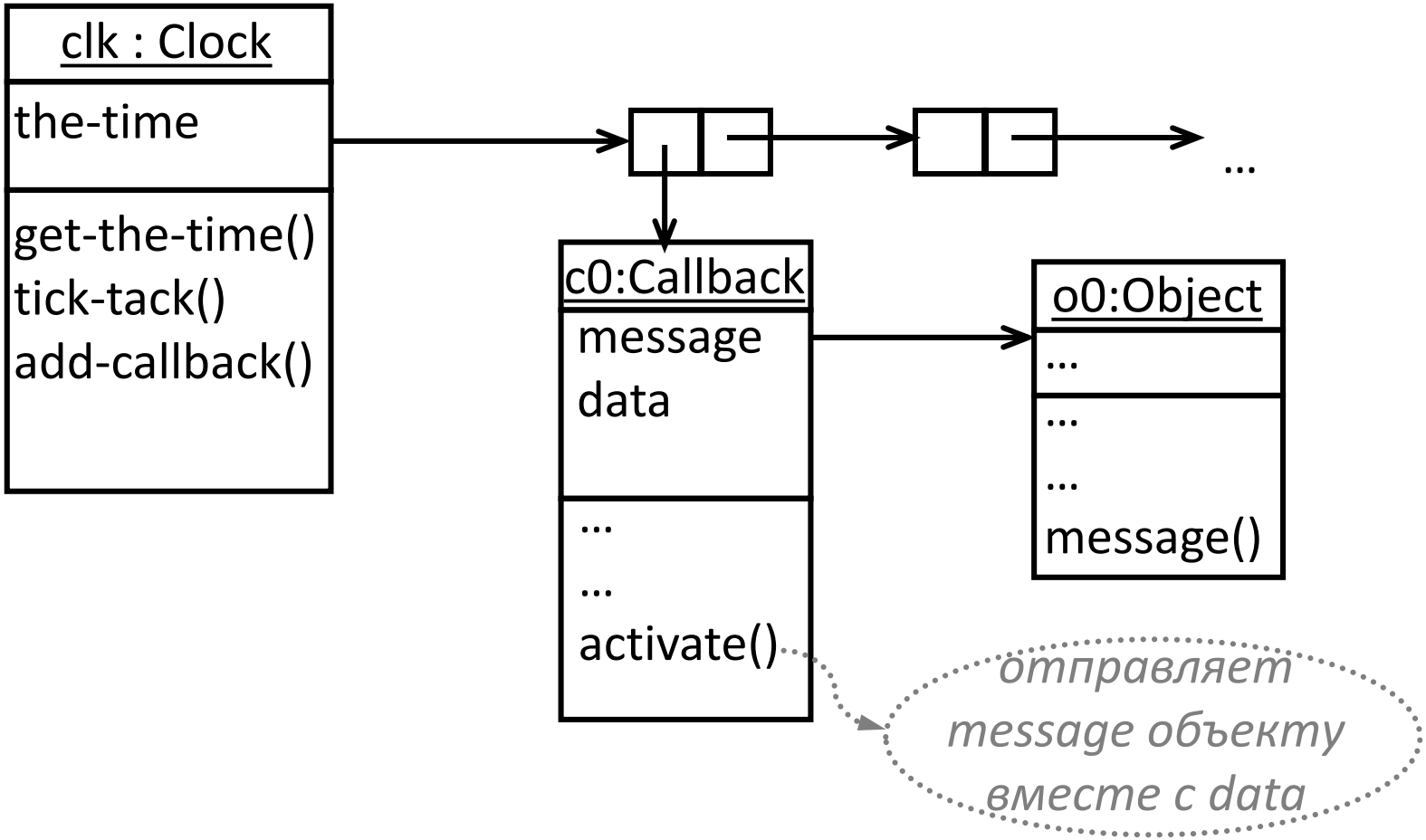
Запустим время

– добавим часы, отсчитывающие время;
– будем отслеживать положение движущихся объектов;
– часы будут посылать сообщения объектам, подписавшимся на события времени, чтобы те обновили своё состояние.

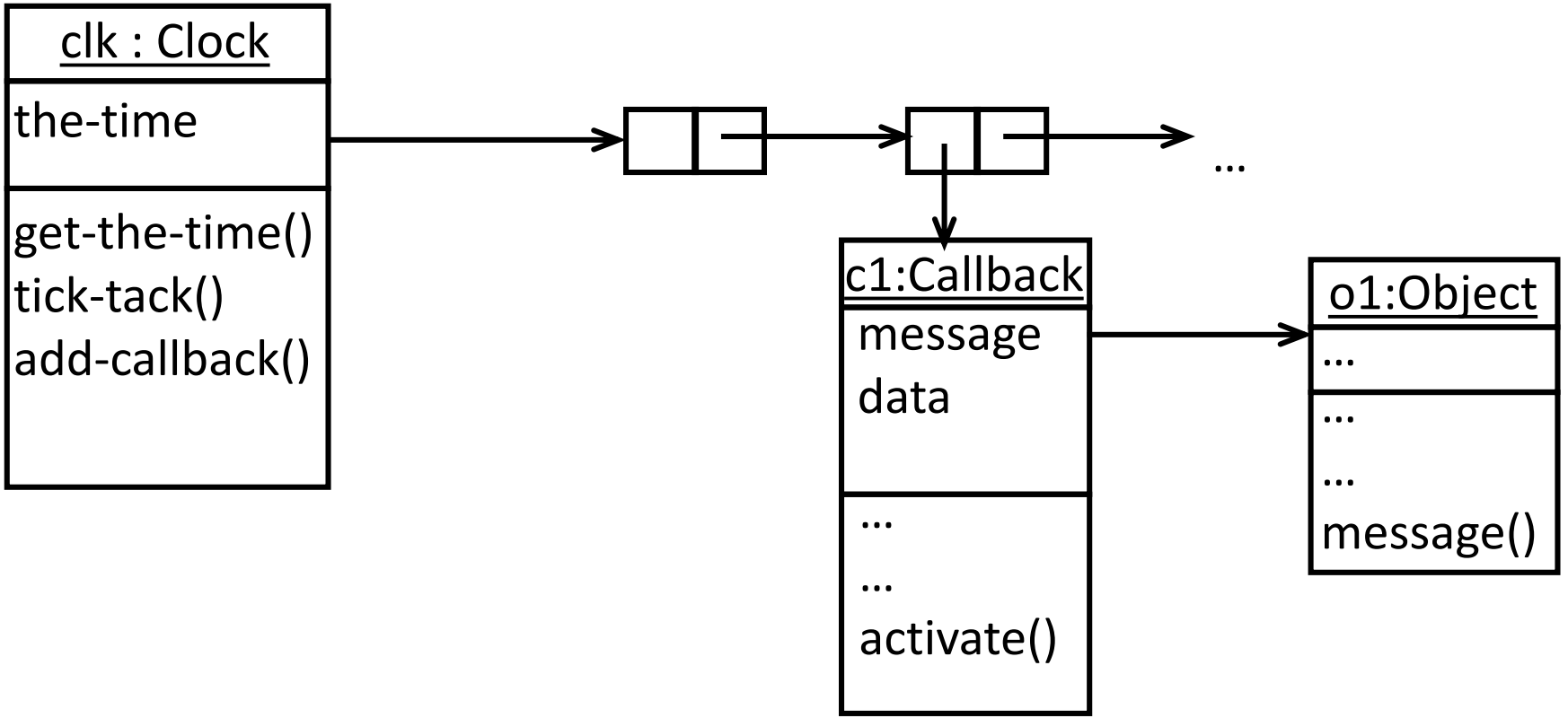
Измененная UML-диаграмма классов



Часы управляют всем



Часы управляют всем



Реализация класса **Clock** на Scheme

```
(define (create-clock . args)
  (let ((the-time 0)
        (callbacks '()))
    (lambda (msg . msg-args)
      (cond
        ((eq? msg 'get-the-time) the-time)
        ((eq? msg 'tick-tack)  (begin (map (lambda (x) (x 'activate)) callbacks)
                                      (set! the-time (+ the-time 1))))
        ((eq? msg 'add-callback)
         (begin
          (set! callbacks (cons (car msg-args) callbacks))
          'added))
          (else (error "WRONG MESSAGE"))))))
(define world (create-clock)) ; часы мира MIT
```

Реализация класса **Callback** на Scheme

```
(define (create-callback obj message . data)
  (lambda (msg)
    (cond
      ((eq? msg 'get-object) obj)
      ((eq? msg 'get-message) message)
      ((eq? msg 'activate) (obj message data))
       (else (error "WRONG MESSAGE")))))
```

Обновлённый класс **Troll** на Scheme

```
(define (create-troll name location restlessness)
 (define (this msg)
  (cond ((eq? msg 'get-name) name)
        ((eq? msg 'get-location) location)
        ((eq? msg 'move-somewhere)
                 (when (< (random) restlessness)
            (let* ((exit (pick-random-list (hash-values (location 'get-exits))))
                                    (to (exit 'get-to))) (when (not (equal? location to))
                                 (begin (location 'delete-troll! this)
                                 (to 'put-troll! this) (set! location to))))))
        (else (error "WRONG MESSAGE"))))
   (begin (location 'put-troll! this) (world 'add-callback (create-callback this
  'move-somewhere)) this))
```

Реализация класса **Clock** в Racket

```
(define clock%
  (class object%
    (super-new)
    (field (the-time 0)
           (callbacks '()))
    (define/public (get-the-time) the-time)
    (define/public (tick-tack)
          (begin (map (lambda (x) (send x activate)) callbacks)
                 (set! the-time (+ the-time 1))))
    (define/public (add-callback cb)
          (begin (set! callbacks (cons cb callbacks))
  ))))

(define world (new clock%)) ; часы мира MIT
```

Реализация класса **Callback** в Racket

```
(define callback%
  (class object%
    (super-new)
    (init-field obj message data)
    (define/public (get-object) obj)
    (define/public (get-message) message)
    (define/public (activate)
          (dynamic-send obj message data))
          ; dynamic-send указывает, что класс объекта obj не статичен
  ))
```

Обновлённый класс **Troll** в Racket

```
(define troll%
  (class object%
    (super-new) (init-field name location restlessness)
    (define/public (get-location) location)
    (define/public (move-somewhere) (when (< (random) restlessness)
        (let* ((exit (pick-random-list (hash-values (send location get-exits))))
               (to (send exit get-to))) (when (not (equal? location to))
               (begin (send location delete-troll! this) (send to put-troll! this)
               (set! location to))))))
    (define/public (get-name) name)
    (define/public (add-to-world world)
        (send world add-callback (new callback% (obj this) (message move-
  somewhere!) (data '())))))
(send location put-troll! this) (add-to-world world) )
```

Промежуточные итоги

- Рассмотрели ОО-стиль программирования:
  – отличный от функционального
  – подходящий для симуляторов, сложных систем, ...

- Рассмотрели объектные модели
  – н/з от языка реализации

    класс – шаблон структуры и поведения объектов

    экземпляр – конкретный объект, созданный по шаблону

    UML-диаграммы классов и UML-диаграммы объектов

Промежуточные итоги

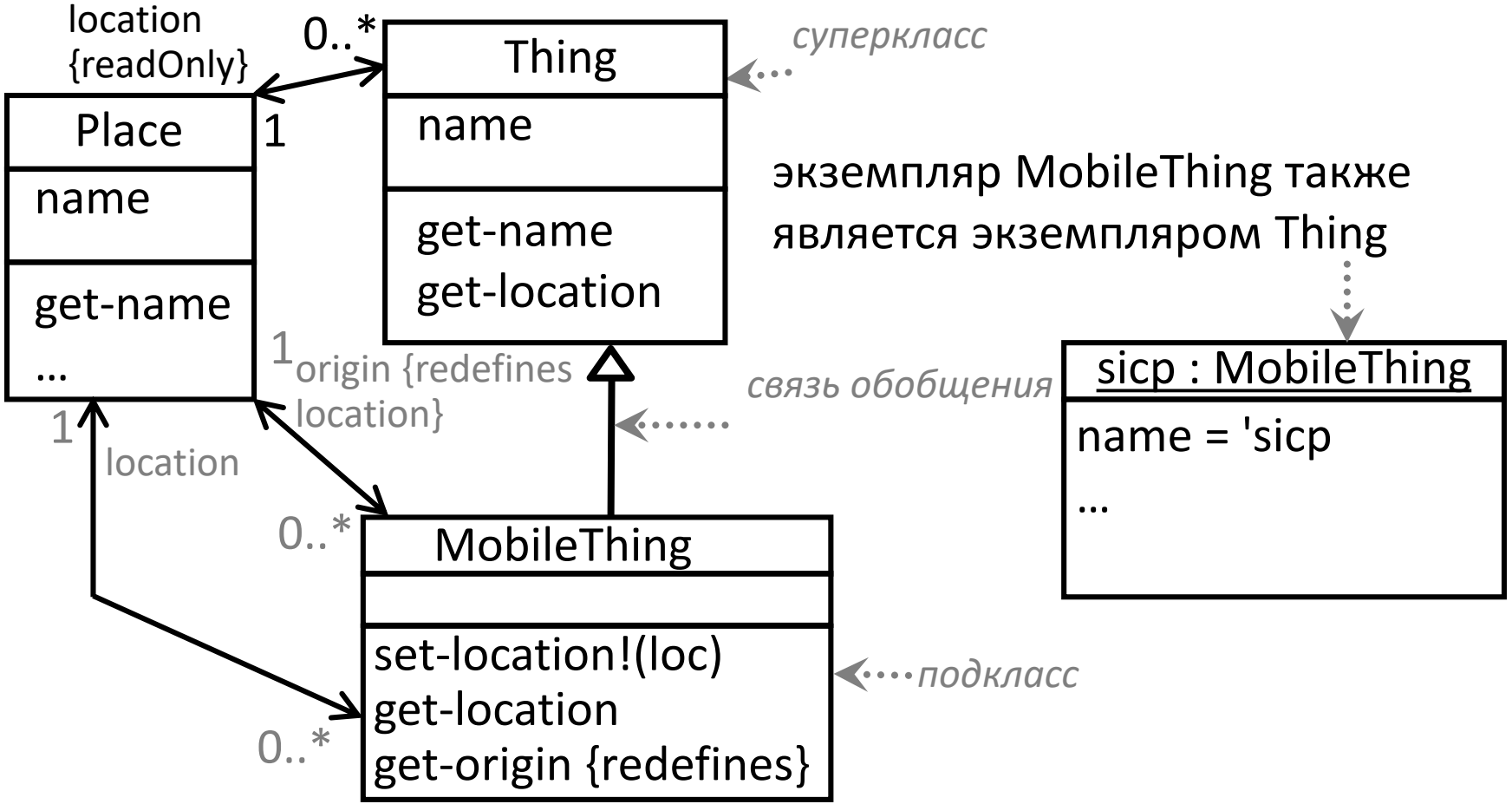
Рассмотрели две реализации ОО-сред

– «наколеночную» на Scheme, где класс – функция-конструктор, а экземпляр – вычисленный вызов конструктора (результат которого – другая функция, обрабатывающая сообщения), а слоты экземпляра – локальные переменные, а методы экземпляра – функции с содержательной обработкой, а отправка сообщения – вызов экземпляра с указанием сообщения;
– готовую – библиотеку racket/class, где класс – описание атрибутов (field, init-field) и операций (define/public, define/private), а экземпляр – вычисленный вызов new, а отправку сообщения осуществляют send, dynamic-send, а доступ к слотам дают get-field, set-field!

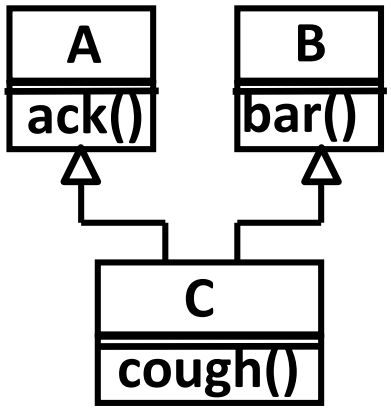
Пояснения по инкапсуляции

- В Scheme можно описывать атрибуты (или операции) как локальные имена внутри конструктора (см. let внутри create-clock). Пользователь, которому доступны лишь сигнатуры конструктора и открытых операций не будет информирован о таких атрибутах (или операциях), для него они закрыты. Скрывать код, реализующий конструктор и операции, можно при помощи механизма модулей (мы его не рассматриваем на лекциях).
- В racket/class есть define/private для описания закрытых членов класса. С ними не работают send-ы, dynamic-send-ы, get-field-ы, set-field!-ы.

UML-диаграмма с обобщением (наследованием)



Множественное наследование



- Суперкласс и подкласс
  - A – суперкласс C;
  - B – тоже суперкласс C;
  - C – подкласс A и B.
- Подкласс наследует атрибуты и операции своих суперклассов
  - экземпляру C можно отправлять сообщения ack(), bar(), cough().

Меры, предпринимаемые ради наследования в Scheme

- Класс описывается функцией-мэйкером, которая определяет общую структуру и поведение экземпляров класса, т. е.
  - его набор атрибутов;
  - обработчик сообщений;
  - определяет суперклассы, наследуемую структуру и поведение.
- Отдельно описывается функция-конструктор, которая с помощью мэйкера порождает экземпляр класса.
- Корневой класс: Object – корень иерархии наследования. Все классы – напрямую или косвенно его наследники.
- Типизация:
  - Каждый класс должен реализовывать операцию get-type, возвращающую все типы его экземпляра вплоть до Object.

Реализация наследования в Scheme

- Экземпляр: порождается конструктором (например create-named-object)
  - все экземпляры различны в смысле eq?
  - новый формат отправки сообщений:

    (ask <instance> '<message> <arg1> … <argn>)
  - у всех экземпляров есть операции:

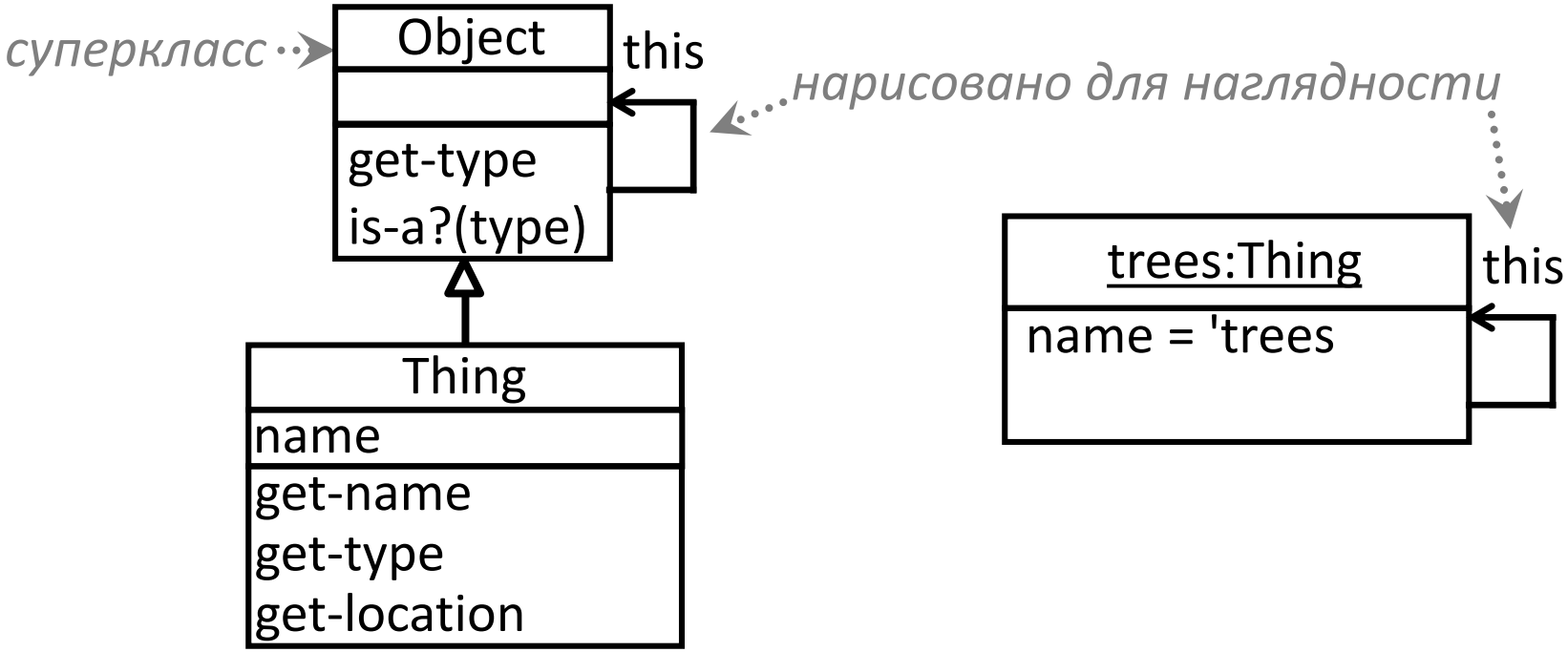
    get-type – получить список типов объекта

    > (ask <instance> 'get-type) -> (<type> <supertype> …)

    is-a? – проверить имеет ли объект данный тип

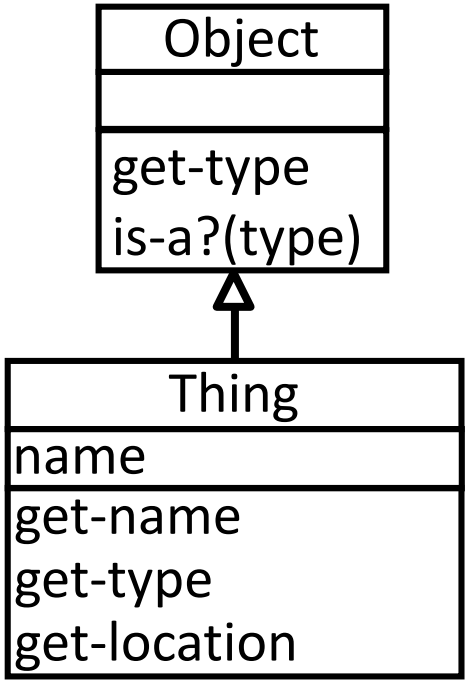
    (ask <instance> 'is-a? <some-type>) -> <boolean>

Обновлённая диаграмма для **Thing**



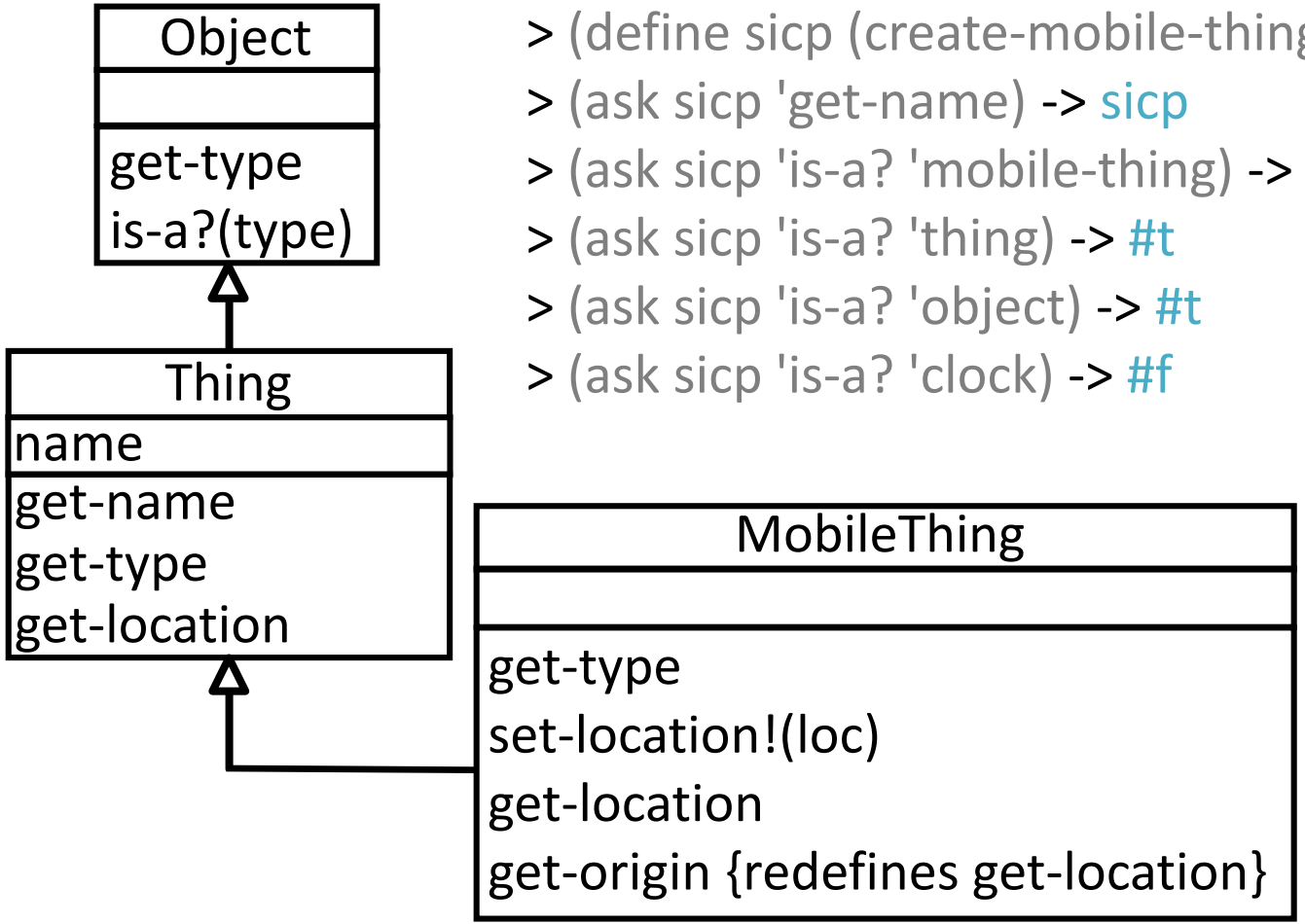
- Thing наследует от Object:
  - атрибут this : у каждого объекта есть ссылка на себя (всего себя)
  - операцию is-a?
  - переопределяет операцию get-type

Обновлённая диаграмма для **Thing**




```
> (define trees (create-thing 'trees st-str))
> (ask trees 'get-name) -> trees
> (ask trees 'get-type) -> (thing object)
> (ask trees 'is-a? 'thing) -> #t
> (ask trees 'is-a? 'clock) -> #f
> (ask trees 'is-a? 'object) -> #t
```

Обновлённая диаграмма с **MobileThing**




```
> (define sicp (create-mobile-thing 'sicp b-l))
> (ask sicp 'get-name) -> sicp
> (ask sicp 'is-a? 'mobile-thing) -> #t
> (ask sicp 'is-a? 'thing) -> #t
> (ask sicp 'is-a? 'object) -> #t
> (ask sicp 'is-a? 'clock) -> #f
```

Мэйкер. Типовое описание класса на Scheme

- Описание класса по имени <type> -- это определение мэйкера. Оно содержит:
  - указание суперклассов
  - состояние (включающее this)
  - обработчик сообщений, определяющий сообщения и операции
    - обязательно обрабатывающий get-type как указано ниже
    - имеющий ветвь (else (get-method …)) для сообщений суперкласса

Код мэйкера класса на Scheme

```
; для класса <type> описываем одноимённый мэйкер
; аргументы – слоты, инициализируемые сразу у экземпляра
(define (<type> this <arg1> <arg2> … <argn> )
  (let ((<super1>-part (<super1> this <args>)
        (<super2>-part (<super2> this <args>)
        <другие суперклассы>
        <не инициализируемые / скрытые слоты> )
    (lambda (msg)
      (cond
        ((eq? msg 'get-type)
                          (lambda () (type-extend '<type> <super1>-part
                           <super2>-part …)) )
        <другие сообщения и операции>
        (else (get-method msg <super1>-part <super2>-part …))))))
```

Порождение экземпляров в Scheme. Объекты Instance

- Для класса с именем <type> описываем конструктор с именем create-<type>. Он использует create-instance – функцию высшего порядка
  - порождающую объект Instance
  - использующую мэйкер класса <type>
  - возвращающую ссылку на объект Instance
- Объект создаётся вызовом конструктора create-<type>

определение create-<type>

```
(define (create-<type> <arg1> <arg2> … <argn>)
  (create-instance <type> <arg1> <arg2> … <argn>))
```

; 1й аргумент в вызове create-instance – мэйкер класса <type>

вызов create-<type>

```
> (define <instance> (create-<type> <arg1> <arg2> … <argn>))
```

Пример. Класс **MobileThing** в Scheme

```
(define (create-mobile-thing name location)
  (create-instance mobile-thing name location))
(define (mobile-thing this name location)
  (let ((thing-part (thing this name location)))
    (lambda (msg) (cond
      ((eq? msg 'get-type) (lambda () (type-extend 'mobile-thing thing-part)))
      ((eq? msg 'set-location!) (lambda (newloc) (set! location newloc)))
      ((eq? msg 'get-location) (lambda () location))
      ((eq? msg 'get-origin) (get-method 'get-location thing-part))
      (else (get-method msg thing-part))))))
```

конструктор для MobileThing

мэйкер MobileThing

атрибуты MobileThing

мэйкер суперкласса

«недоэкземпляр» суперкласса

новое

сообщение может быть обработано в суперклассе

Мэйкер создаёт «недоэкземпляр», которого достаточно для части суперкласса внутри экземпляра подкласса. Полный экземпляр создаётся конструктором – create-mobile-thing.

Ещё пример: обновлённый класс **Thing** в Scheme

```
(define (create-thing name location)
  (create-instance thing name location))
(define (thing this name location)
  (let ((object-part (object this)))
    (lambda (msg)
      (cond
        ((eq? msg 'get-type)
              (lambda () (type-extend 'thing object-part)))
        ((eq? msg 'get-name) (lambda () name))
        ((eq? msg 'get-location) (lambda () location))
        (else (get-method msg object-part))))))
```

Далее: класс **Object** в Scheme

```
; конструктор для Object не обязателен, Object может быть абстрактным
; (define (create-object) (create-instance object))

; мейкер нужен, иначе не сможем создать экземпляр любого класса
(define (object this)
    (lambda (msg)
    (cond
     ((eq? msg 'get-type)
          (lambda () '(object)))
     ((eq? msg 'is-a?) (lambda (type)
                              (if (member (ask this 'get-type)) #t #f)))
     (else (error "WRONG MESSAGE")))))
```

Использование объектов Instance

- поиск метода: get-method для <сообщения> в <instance>
- вызов метода
- оба шага сразу: ask
- get-method возвращает функцию

; породим объект

(define <instance> (create-<type> <arg1> <arg2> … <argn>))

; возьмём обработчик конкретного сообщения

(define sm-method (get-method '<сообщение> <instance>))

;и запустим

(sm-method <m-arg1> <m-arg2> … <m-argm>)

; выбор и запуск обработчика в «одном флаконе»

(ask <instance> '<сообщение> <m-arg1> … <m-argm>)

Класс **Object** в Racket

Класс Object встроен в racket/class. Сделать из него полный аналог **Object** из Scheme не выйдет.

> (define o (new object%))

> (send o get-type) -> no such method … get-type …

но можно вызывать функцию is-a?, так как она тоже встроенная

> (is-a? o named-object%) -> #f

> (is-a? o clock%) -> #f

> (is-a? x object%) -> #t

только стоит иметь в виду, что в Racket is-a? не является операцией класса Object.

> (send o is-a? object?) -> no such method … is-a? …

Обновлённый класс **Thing** в Racket

```
(define thing%
  (class object%
    (super-new)
    (init-field name location)
    (define/public (get-type) ; обеспечим поддержку get-type
          (list 'thing% 'object%))
    (define/public (get-location) location)
    (define/public (get-name) name)
    (send location put-thing! this)
 ))
> (define trees (new thing% (name 'trees) (location st-str)))
> (send trees get-name) -> trees
> (send trees get-type) -> (thing% object%)
> (is-a? trees thing%) -> #t
> (is-a? trees clock%) -> #f
> (is-a? trees object%) -> #t
```

Пример. Класс **MobileThing** в Racket

```
(define mobile-thing% (class thing% ; суперклассом является thing%
    (super-new)
    (inherit-field (origin location))   <... наследуем и переименовываем location
    (define location origin)            <... объявляем новое location
    (define/override (get-location) location)
    (define/public (get-origin) (super get-location))   <... у суперкласса
    (define/public (set-location! newloc)
      (begin (set-field! things  location
                         (remove this (get-field things location)))
             (set! location newloc)
             (send newloc put-thing! this)))
    (define/override (get-type) (cons 'mobile-thing% (super get-type)))))
```

переопределяем

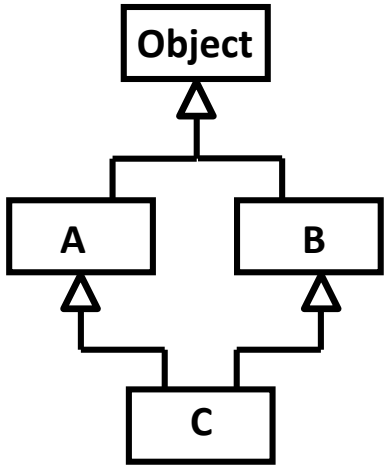
... у суперкласса

Пример. Класс **MobileThing** в Racket

```
> (define sicp (new mobile-thing% (name 'sicp) (location b-l)))
> (send (send sicp get-origin) get-name) -> barker-library
> (send (send sicp get-location) get-name) -> barker-library
> (map (lambda (x) (send x get-name)) (send b-l get-things)) -> (sicp)
> (send sicp set-location! st-str)
> (map (lambda (x) (send x get-name)) (send b-l get-things)) -> ()
> (map (lambda (x) (send x get-name)) (send st-str get-things)) ->
  (sicp trees)
> (send (send sicp get-location) get-name) -> student-street
> (send (send sicp get-origin) get-name) -> barker-library
```

Иерархия типов

- При наследовании у экземпляра несколько типов




```
> (define a-instance (create-a))
> (define c-instance (create-c))
> (ask a-instance 'get-type) -> (a object)
> (ask c-instance 'get-type) -> (c a b object)
> (ask c-instance 'is-a? 'c) -> #t
> (ask c-instance 'is-a? 'b) -> #t
> (ask c-instance 'is-a? 'a) -> #t
> (ask c-instance 'is-a? 'object) -> #t
> (ask a-instance 'is-a? 'c) -> #f
> (ask a-instance 'is-a? 'b) -> #f
> (ask a-instance 'is-a? 'a) -> #t
```

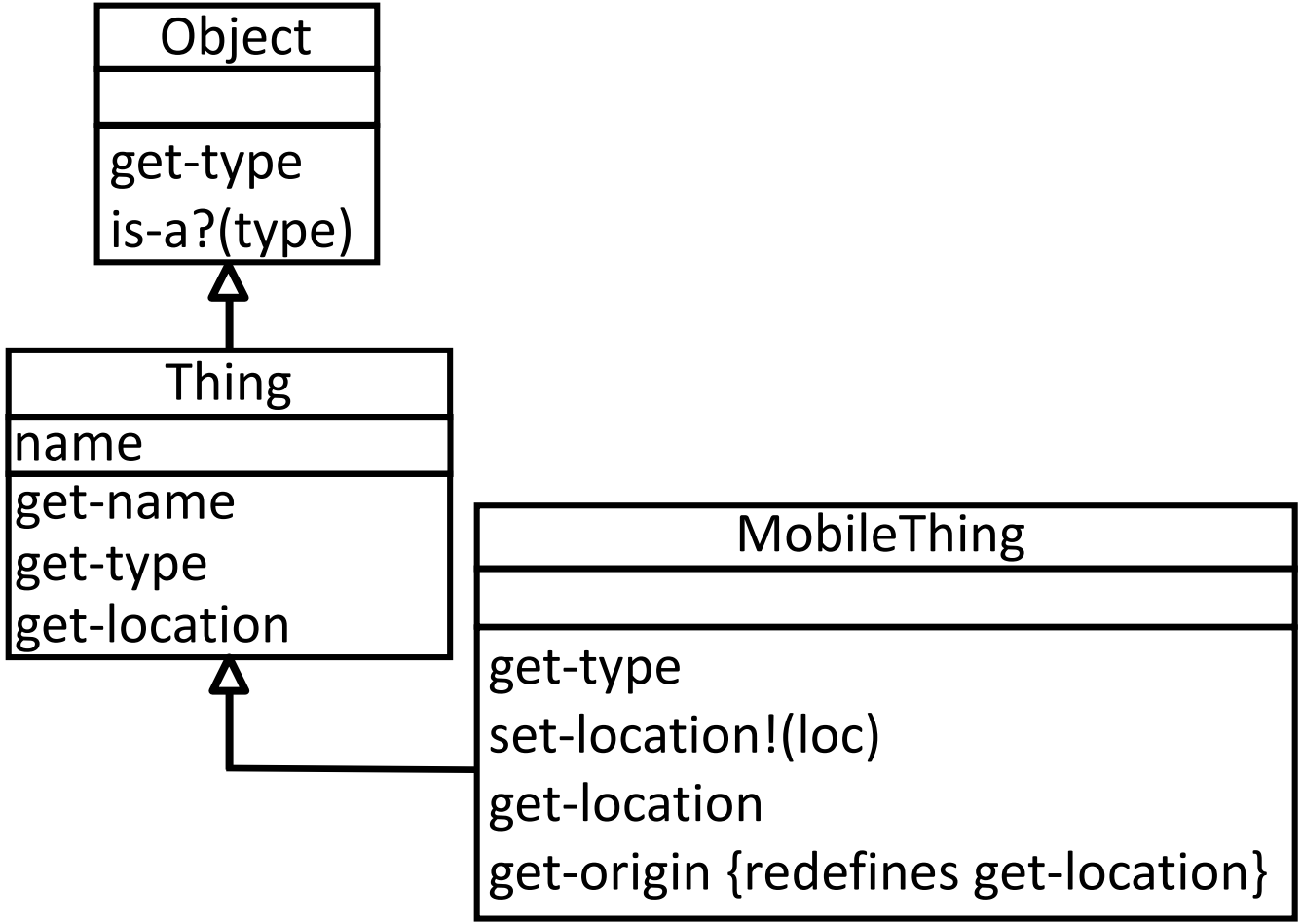
Точки зрения на ООП

- Точка зрения модели
  - диаграмма классов и диаграмма объектов
  - терминология: сообщения, операции, обобщение, суперкласс, подкласс, …
- Точка зрения использования
  - Соглашения о том как писать ОО-код на Scheme:
    - описывать класс
      - указывать его суперклассы
    - порождать экземпляры
    - использовать экземпляры (вызывать операции)
  - Библиотека racket/class

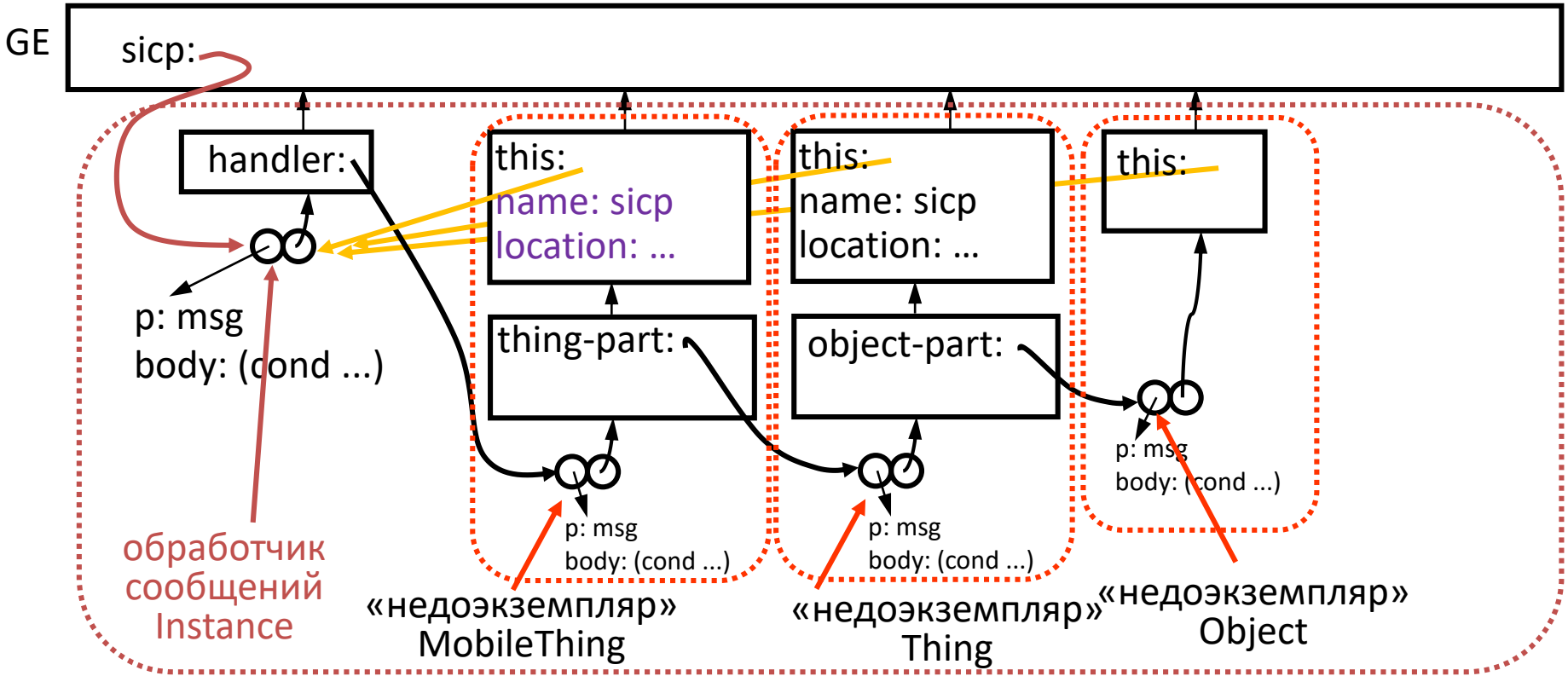
Далее: Точка зрения реализации

- как могут быть реализованы экземпляры, классы, наследование на Scheme

Вернёмся к примеру



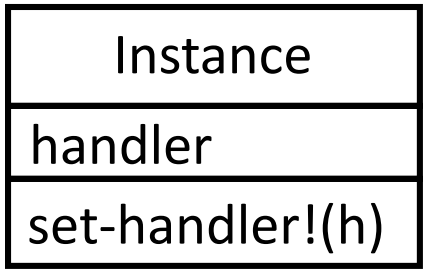
Пример с точки зрения модели окружений



Объект состоит из цепочки обработчиков с «заглавным звеном»

Точка зрения реализации: Instance

```
(define (make-instance)
  (let ((handler #f))
    (lambda (msg)
      (cond
        ((eq? msg 'set-handler!)
         (lambda (handler-proc)
           (set! handler handler-proc)))
        (else (get-method msg handler))))))

(define (create-instance maker . args)
  (let* ((instance (make-instance)) (handler (apply maker instance args)))
    (ask instance 'set-handler! handler)
    instance))
```



Точка зрения реализации: **get-method** и **ask**

получить метод:

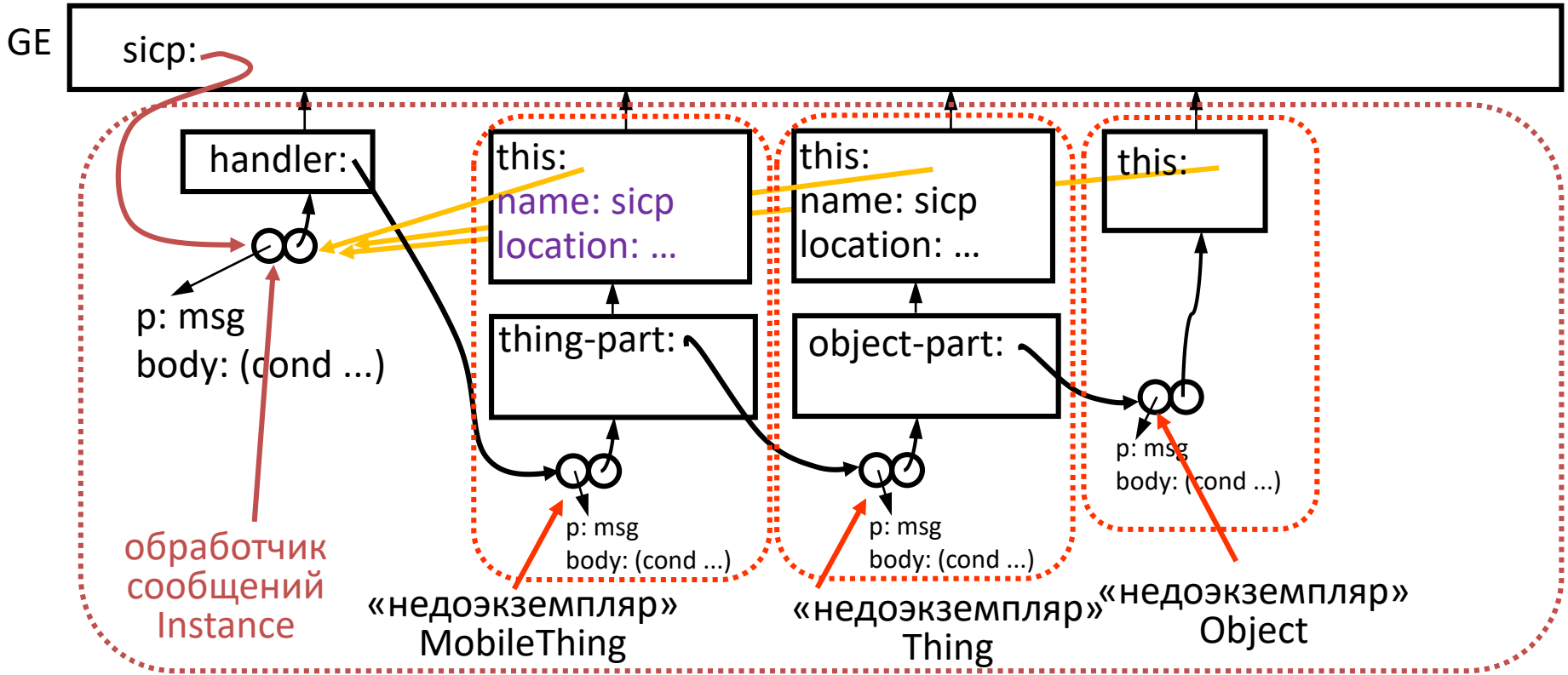
```
(define (get-method message object)
  (object message))
```

"ask" – комбинация поиска метода и его вызова:

```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (procedure? method)
        (apply method args)
        (error "WRONG METHOD"))))
```

вспомним, что (apply op args) это то же что (op arg1 arg2 … argN)

Какова роль **this**?

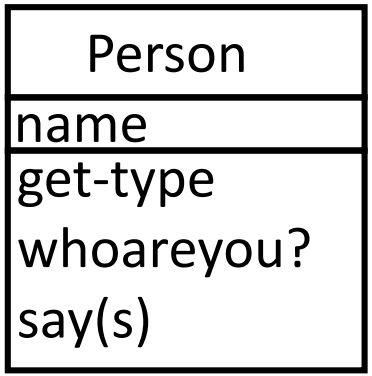


Всюду this указывает на начало цепочки. Зачем?

Точка зрения использования. This

- В каждом классе есть атрибут this
  - this – это ссылка на *весь* экземпляр
- Зачем? Как и когда используется this?
  - При реализации операции объект может послать сообщение своей части: например, внутри операции класса MobileThing, можно (ask thing-part 'get-name).
  - Иногда нужно послать сообщение экземпляру целиком: например (ask this 'get-type) – такой вызов есть в теле is-a? в Object. Важно, что Object заранее не знает действительный тип экземпляра, частью которого он является. Аналогично, можно из части экземпляра, отвечающий за супертип, вызывать новую реализацию операции в подтипе.

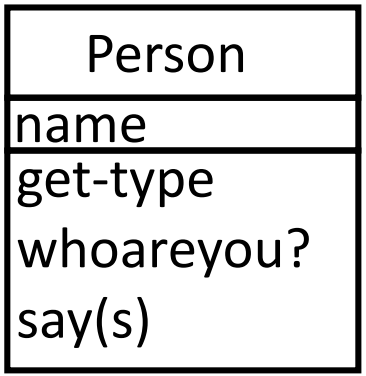
Класс Person




```
> (define alyssa (create-person 'alyssa-hacker))
> (ask alyssa 'whoareyou?) -> alyssa-hacker
> (ask alyssa 'say '(the sky is blue)) -> (the sky is blue)
```

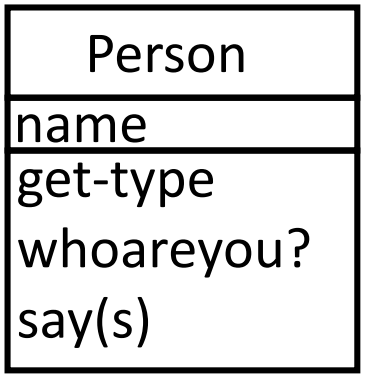
Реализация Person в Racket

```
(define person%
  (class object%
    (super-new)
    (init-field name)
    (define/public (whoareyou?) name)
    (define/public (say stuff) stuff)
    (define/public (get-type) (list 'person% 'object%))
  ))
> (define alyssa (new person% (name 'alyssa-hacker)))
> (send alyssa whoareyou?) -> alyssa-hacker
> (send alyssa say '(the sky is blue)) -> (the sky is blue)
```

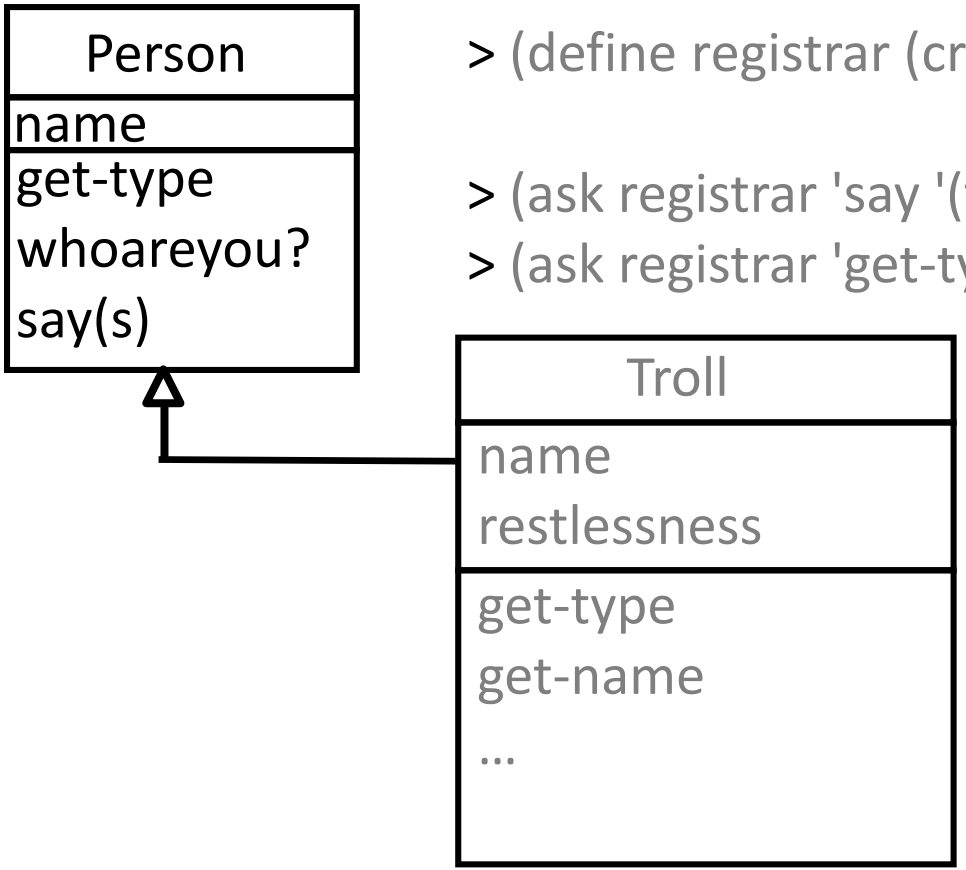


Реализация Person на Scheme

```
(define (create-person name)
  (create-instance person name))
(define (person this name)
  (let ((object-part (object this)))
    (lambda (msg)
      (cond
            ((eq? msg 'get-type) (lambda () (type-extend 'person object-part)))
            ((eq? msg 'whoareyou?)  (lambda () name))
            ((eq? msg 'say)  (lambda (stuff) stuff))
            (else (get-method msg object-part))))))
```

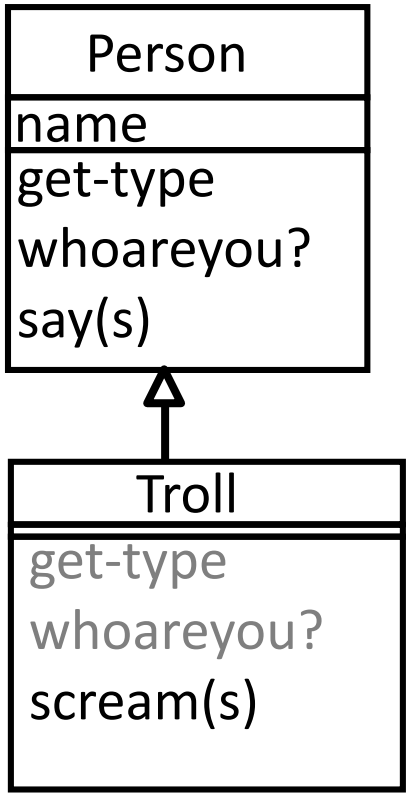


Сделаем Troll подклассом **Person**



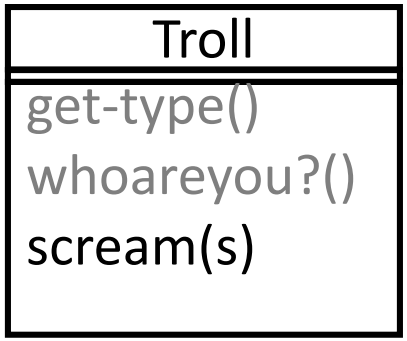

```
> (define registrar (create-troll 'registrar))

> (ask registrar 'say '(the sky is blue)) -> (the sky is blue)
> (ask registrar 'get-type) -> (troll person object)
```

Дополним класс **Troll**



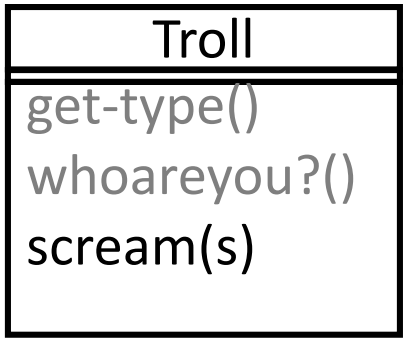
; пусть тролль к имени добавляет, что он тролль
> (ask registrar 'whoareyou?) -> (*troll* registrar)
; пусть операция тролля whoareyou? использует операцию персоны whoareyou?
; пусть тролль, умеет кричать
> (ask registrar 'scream '(i am hungry))
-> (*trololo* i am hungry)
; пусть операция операция scream использует операцию say

Реализация **Troll** в Racket



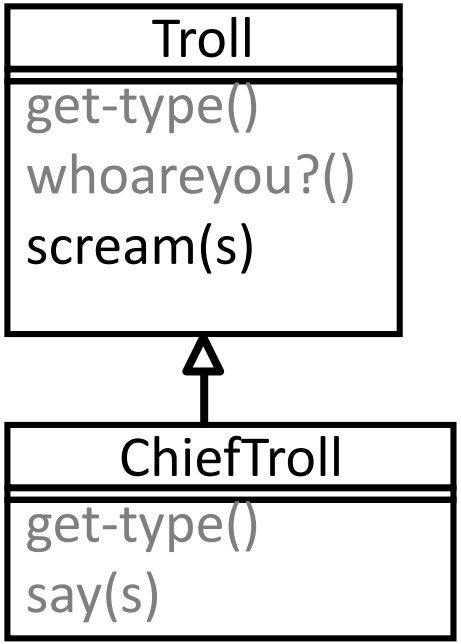

```
(define troll%
  (class person%
    (super-new)
    (define/override (whoareyou?) (list 'troll (super whoareyou?)))
    (inherit/super say)
    (define/public (scream smthng) (cons 'trololo (super say smthng)))
    (define/override (get-type) (cons 'troll% (super get-type)))
  ))
> (define registrar (new troll% (name 'registrar)))
> (send registrar whoareyou?) -> (troll registrar)
> (send registrar say '(the sky is blue)) -> (the sky is blue)
> (send registrar scream '(the sky is blue)) -> (trololo the sky is blue)
```

Реализация **Troll** на Scheme



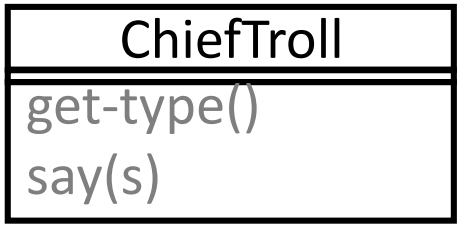

```
(define (create-troll name)
  (create-instance troll name))
(define (troll this name)
  (let ((person-part (person this name)))
    (lambda (msg)
      (cond
        ((eq? msg 'get-type) (lambda () (type-extend 'troll person-part)))
        ((eq? msg 'whoareyou?)
              (lambda () (list 'troll (ask person-part 'whoareyou?))))
        ((eq? msg 'scream)
              (lambda (smthng) (cons 'trololo (ask person-part 'say smthng))))
        (else (get-method msg person-part))))))
```

Введём подкласс **ChiefTroll**



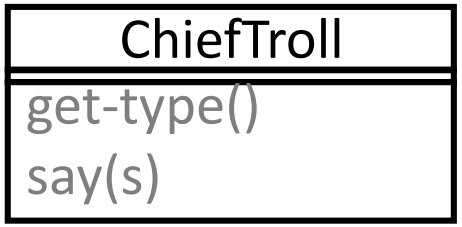

```
(define cht (create-chief-troll 'grendel))
; главный тролль представляется также
> (ask cht 'whoareyou?) -> (troll grendel)
; в конце всего сказанного он добавляет hai
> (ask cht 'say '(the sky is blue))
-> (the sky is blue hai)
```

ChiefTroll в Racket




```
(define chief-troll%
  (class troll%
    (super-new)
    (define/override (say stuff)
                     (append (super say stuff) (list 'hai)))
    (define/override (get-type)
                     (cons 'chief-troll% (super get-type)))
  ))
> (define cht (new chief-troll% (name 'grendel)))
> (send cht whoareyou?) -> (troll grendel)
> (send cht say '(the sky is blue)) -> (the sky is blue hai)
> (send cht scream '(the sky is blue)) -> (trololo the sky is blue)
```

ChiefTroll на Scheme




```
(define (create-chief-troll name)
  (create-instance chief-troll name))
(define (chief-troll this name)
  (let ((troll-part (troll this name)))
    (lambda (msg)
      (cond ((eq? msg 'get-type)
                    (lambda () (type-extend 'chief-troll troll-part)))
            ((eq? msg 'say) (lambda (stuff) (append (ask troll-part 'say stuff)
                                                    (list 'hai))))
            (else (get-method msg troll-part))))))
```

Что не так с **ChiefTroll**?

> (define cht (create-chief-troll 'grendel))

> (ask cht 'scream '(the sky is blue)) -> (trololo the sky is blue)

Почему в конце фразы нет hai?

– scream использует say класса Troll, а не say класса ChiefTroll

Исправленный **ChiefTroll**

(define cht (create-chief-troll 'grendel))

> (ask cht 'scream '(the sky is blue)) -> (trololo the sky is blue *hai*)

Обновлённая реализация **Troll** в Racket

```
(define troll%
  (class person%
    (super-new)
    (define/override (whoareyou?) (list 'troll (super whoareyou?)))
    (define/public (scream notes) (cons 'trololo (send this say notes)))
    (define/override (get-type) (cons 'troll% (super get-type)))
  ))
```

вызываем say актуального класса

```
> (define cht (new chief-troll% (name 'grendel)))
> (send cht scream '(the sky is blue))
-> (trololo the sky is blue hai)
> (send cht say '(the sky is blue))
-> (the sky is blue hai)
```

Обновлённая реализация **Troll** на Scheme

```
(define (create-troll name)
  (create-instance troll name))
(define (troll this name)
  (let ((person-part (person this name)))
    (lambda (msg)
      (cond
            ((eq? msg 'get-type)
       (lambda () (type-extend 'troll person-part)))
            ((eq? msg 'whoareyou?) (lambda () (list 'troll name)))
            ((eq? msg 'scream) (lambda (notes) (cons 'trololo
                                                     (ask this 'say notes))))
           (else (get-method msg person-part))))))
```

вызываем say актуального класса

Промежуточные итоги

- Мы описываем иерархии классов
  - используя соглашение по структуре описания класса
    - для наследования структуры и поведения из суперкласса
- Управление поведением
  - “ask” к своей части – «недоэкземпляру» суперкласса
  - “ask” к this – ко всему экземпляру, т. е. возможно экземпляру подкласса.
- Возможно дополнительное управление на основе типов объектов.

Дальше больше

- Просто объекты
  - соглашение о сообщениях и операциях
  - this

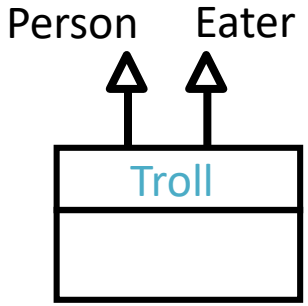
- Наследование (одиночное)
  - внутренние части от суперклассов
  - в операции можно обратиться к внутренней части
  - get-method для суперкласса находит нужный метод

- **Множественное наследование🡐**

Troll и **Eater**

Eater – едок, отъедает куски от персон, находящихся в той же локации. Вероятность укуса зависит от **hunger**.

Обновлённый **Troll** на Scheme. Версия 1




```
(define (create-troll name hunger)
  (create-instance troll name hunger))
(define (troll this name hunger)
  (let ((person-part (person this name))
        (eater-part (eater this hunger)))
    (lambda (msg)
      (cond
        ((eq? msg 'get-type) (lambda () (type-extend 'troll person-part
                                                     eater-part)))
        ((eq? msg 'whoareyou?)
              (lambda () (list 'troll (ask person-part 'whoareyou?))))
        ((eq? msg 'eat-people) (lambda () (...))) ...
        (else (get-method msg person-part eater-part))))))
```

Точка зрения реализации: Множественное наследование в Scheme

Как реализовать новый get-method – просматривать экземпляры по порядку, пока не будет найден тот, который может обработать сообщение.

```
(define (get-method message object) (object message))
```

превращается в

```
(define (get-method message . objects)
  (let try ((objs objects))
    (if (null? objects) (void)
            (let ((method ((car objects) message)))
              (if (procedure? method) method
                  (try (cdr objects))))))
)
```

Точка зрения реализации: Множественное наследование в Scheme

Как реализовать type-extend – собрать всё, убрать дубли. Будем считать, что разработчики классов ответственны и не зацикливают иерархию наследования.

```
(define (type-extend type . super-parts)
  (let loop ((parts super-parts) (result (list type)))
    (if (null? parts) (reverse result)
            (let ((part-types (ask (car parts) 'get-type)))
              (loop (cdr parts)
                    (foldl (lambda (x y) (if (member x y) y (cons x y)))
                           result part-types)))
    ))
)
; можно видеть, что эта реализация не очень эффективна
```

Обновлённый **Troll** в Racket

```
; множественного наследования в Racket нет, поэтому
; заведём интерфейс с перечнем операций eater%
(define eater-interface<%> (interface () get-type get-hunger eat-people))
; опишем troll%, как наследника person% и реализацию интерфейса
(define troll%
  (class* person% (eater-interface<%>)
    (super-new)
; реализуем интерфейс с помощью «внутреннего едока»
    (init-field hunger)
    (field (eater-part (new eater% hunger)))
; get-hunger перенаправляем внутреннему едоку
    (define/override (get-hunger) (send eater-part get-hunger))
    (define/override (get-type)
          (list* 'troll% 'eater-interface<%> (super get-type)))
; eat-people реализуем в troll%
    (define/public (eat-people) (…))
   ))
```

Обновлённый **Troll** в Racket

```
> (define registrar (new troll% (name 'registrar) (hunger 2)))
> (send registrar whoareyou?) -> (troll registrar)
> (send registrar say '(the sky is blue)) -> (the sky is blue)
> (send registrar scream '(the sky is blue)) -> (trololo the sky is blue)
> (send registrar get-type)
-> (troll% eater-interface<%> person% object%)
> (is-a? registrar eater%) -> #f
> (is-a? registrar eater-interface<%>) -> #t
> (is-a? registrar person%) -> #t
> (subclass? registrar person%) -> #t
> (implementation? registrar eater-interface<%>) -> #t
```

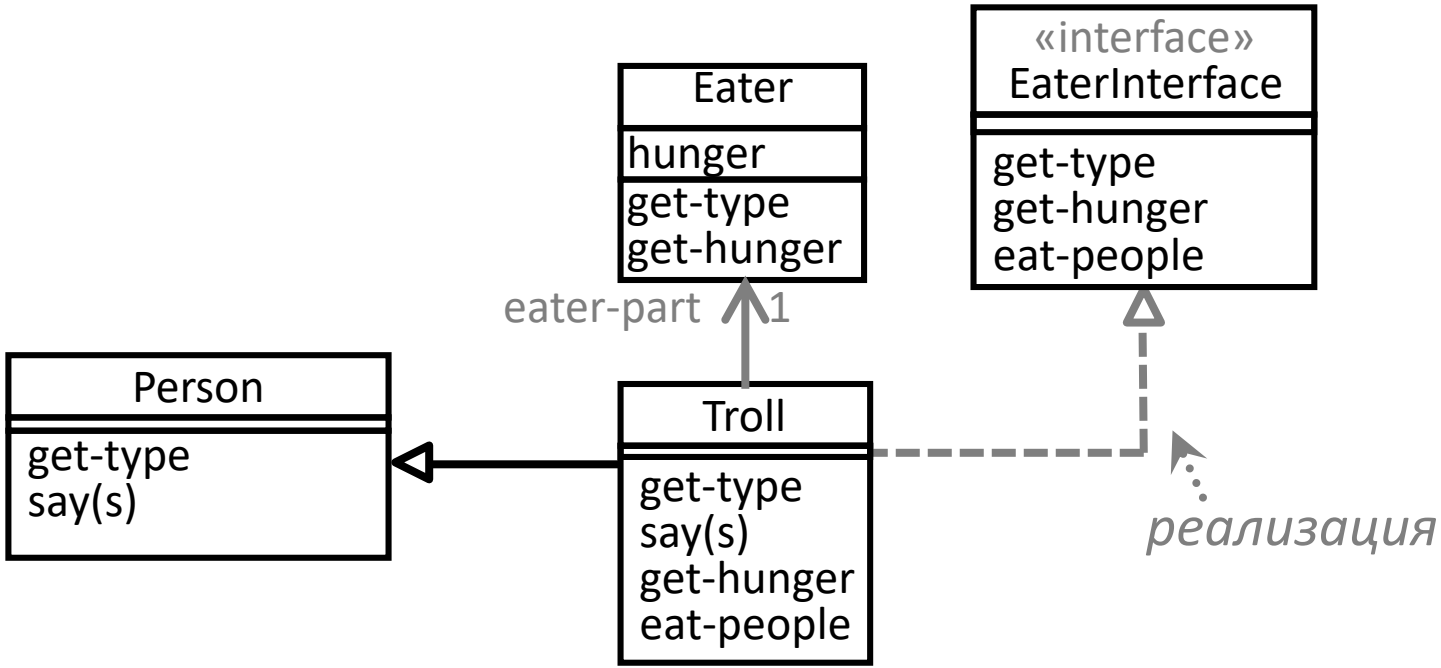
бывает полезным проверить: подкласс ли? реализация ли?

Интерфейс на диаграмме классов

Интерфейс помечается тэгом «interface».

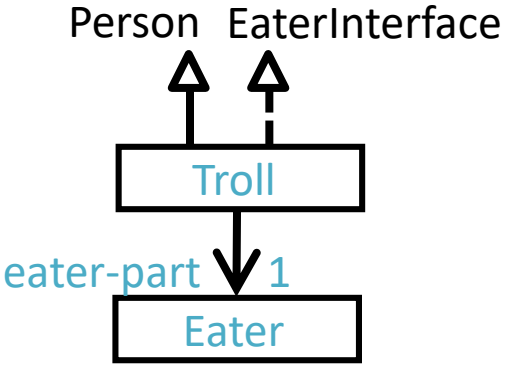
С реализующим классом интерфейс соединяется связью реализации.

Классы и интерфейсы можно соединять ассоциацией.



Обновлённый **Troll** на Scheme. Версия 2

```
(define (create-troll name hunger)
  (create-instance troll name hunger))
(define (troll this name hunger)
  (let ((person-part (person this name))
        (eater-part (create-eater hunger)))
    (lambda (msg)
      (cond ((eq? msg 'get-type) (lambda () (type-extend 'troll
                                   (type-extend 'eater-interface person-part))))
((eq? msg 'whoareyou?) (lambda () (list 'troll (ask person-part 'whoareyou?))))
        ((eq? msg 'get-hunger) (lambda () (ask eater-part 'get-hunger)))
        ((eq? msg 'eat-people) (lambda () (...))) ...
        (else (get-method msg person-part))))))
 ; годятся прежние упрощённые type-extend и get-method
```

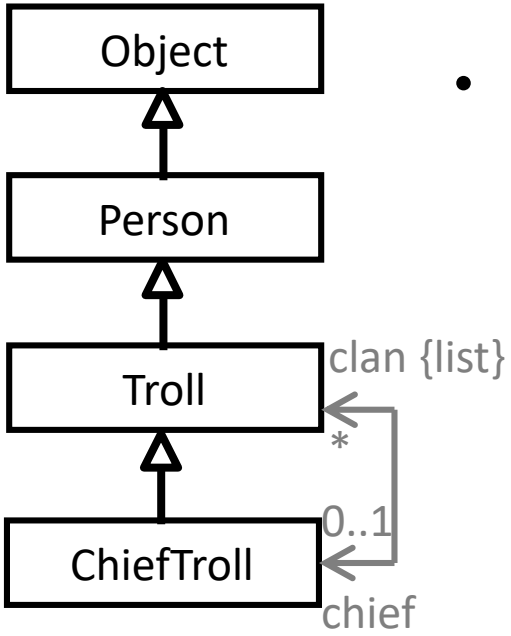


Следующий пример – «Кланы троллей»

- Пример продемонстрирует разницу между
  - “is-a” (связью обобщения)
  - “has-a” (ассоциацией)

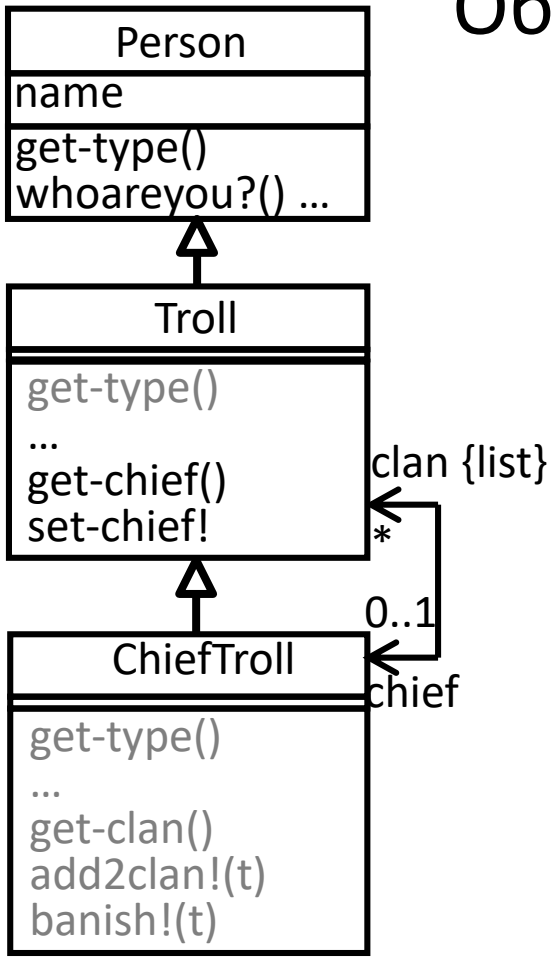
- Добавим ассоциацию внутри иерархии наследования Troll!

Эскиз диаграммы классов



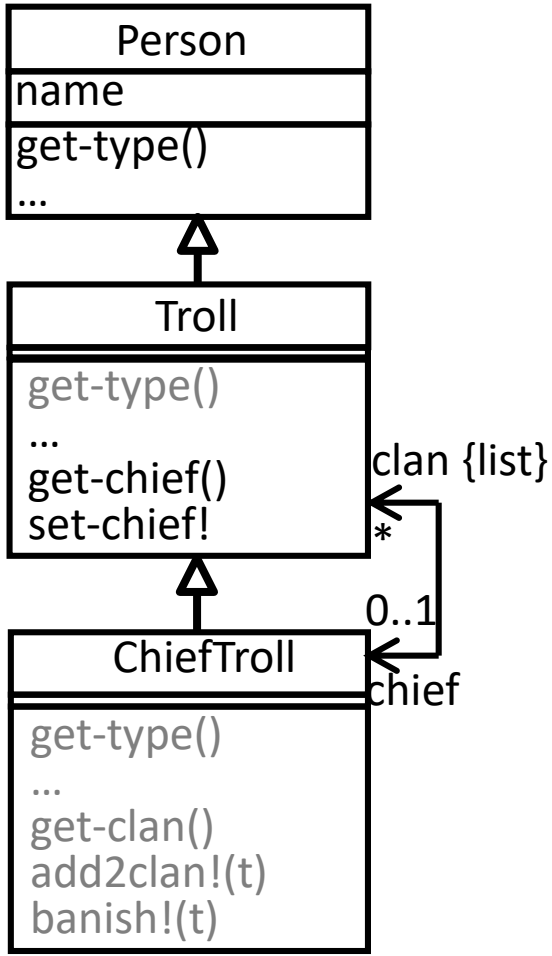
- Добавим ассоциацию между троллем-вождём и его кланом и рассмотрим, что получится с точки зрения
  - диаграммы классов
  - поведения
  - диаграммы объектов
  - описания классов
  - модели вычислений с окружениями

Обновлённые Troll и ChiefTroll



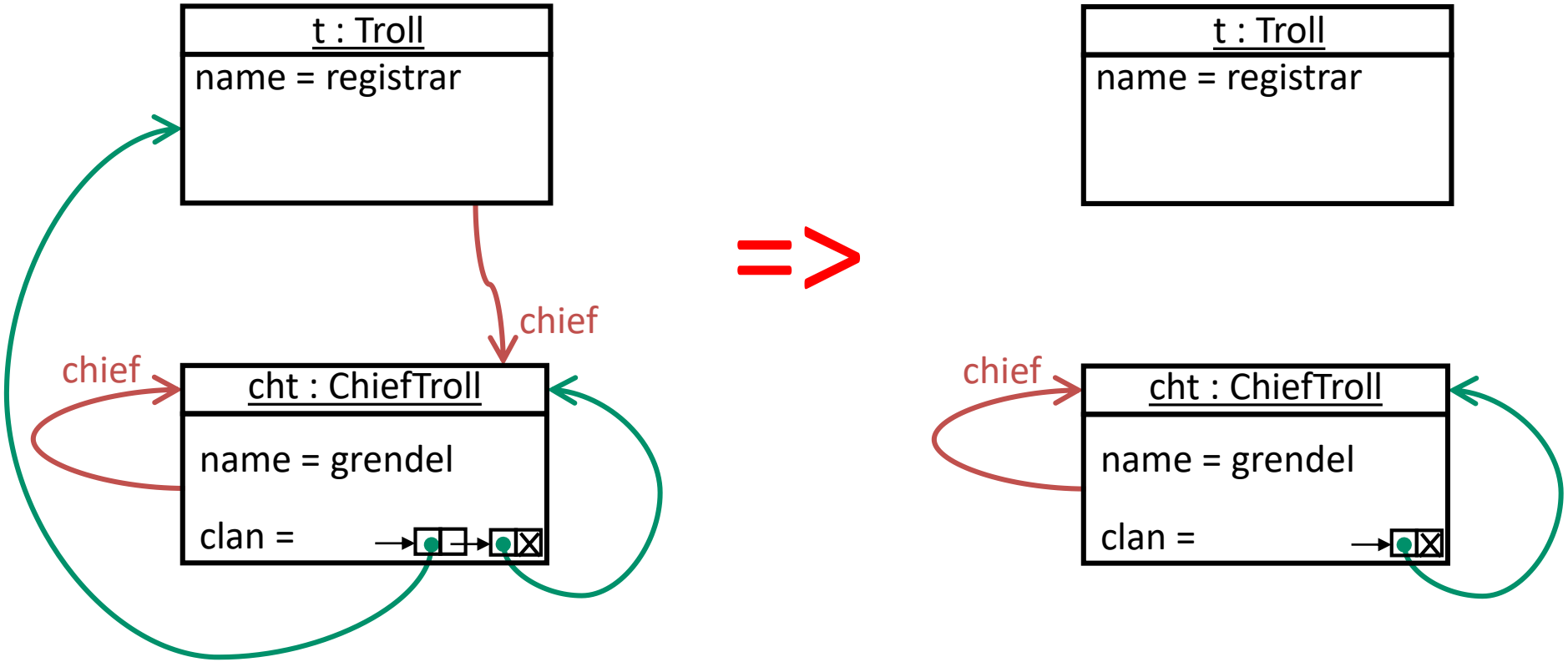
- состояние:
  - у тролля *может быть* вождь
  - у тролля-вождя *может быть* клан
- поведение: добавлены операции для управления состоянием
  - Troll:get-chief()
  - Troll:set-chief!(t)
  - ChiefTroll:get-clan()
  - ChiefTroll:add2clan!(t)
  - ChiefTroll:banish!(t)

Поведение




```
> (define t (create-troll 'registrar 2))
> (define cht (create-chieftroll 'grendel 4))
> (ask t 'get-type)
-> (troll eater person object)
> (ask cht 'get-type)
-> (chieftroll troll eater person object)
> (ask cht 'add2clan! cht)
> (ask cht 'add2clan! t)
> (names-of (ask cht 'get-clan))
-> ((troll registrar) (troll grendel))
> (ask (ask t 'get-chief) 'whoareyou?)
-> (troll grendel)
> (ask cht banish! t)
> (names-of (ask cht 'get-clan))
-> (troll grendel)
```

Диаграмма объектов (скрещенная со стрелочной)



Описание **Troll** (Racket)

```
(define eater-interface<%> (interface () get-type get-hunger eat-people))
(define troll%
  (class* person% (eater-interface<%>)
    (super-new)
    (init-field hunger)
    (field (eater-part (new eater% hunger))
           (chief null))
    (define/override (get-hunger) (send eater-part get-hunger))
    (define/override (get-type)
      (list* 'troll% 'eater-interface<%> (super get-type)))
    (define/public (eat-people) (…))
    (define/public (get-chief) chief)
    (define/public (set-chief! t) (set! chief t))
    (define/override (whoareyou?) (list 'troll (super whoareyou?)))
    (define/public (scream notes) (cons 'trololo (send this say notes)))
   ))
```

Описание **ChiefTroll** (Racket)

```
(define chief-troll%
  (class troll%
    (super-new)
    (field (clan null))
    (define/override (say stuff) (append (super say stuff) (list 'hai)))
    (define/override (get-type) (cons 'chief-troll% (super get-type)))
    (define/public (get-clan) clan)
    (define/public (add2clan! t) (begin (set! clan (cons t clan))
                                        (send t set-chief! this)))
    (define/public (banish! t) (begin (set! clan (remove t clan))
                                      (send t set-chief! null)))
  ))
```

Поведение (Racket)

```
> (define t (new troll% (name 'registrar) (hunger 2)))
> (define cht (new chief-troll% (name 'grendel) (hunger 4)))
> (send cht add2clan! cht)
> (send cht add2clan! t)
> (names-of (send cht get-clan))
-> ((troll registrar) (troll grendel))
> (send (send t get-chief) whoareyou?) -> (troll grendel)
> (send cht banish! t)
> (names-of (send cht get-clan))
-> (troll grendel)
```

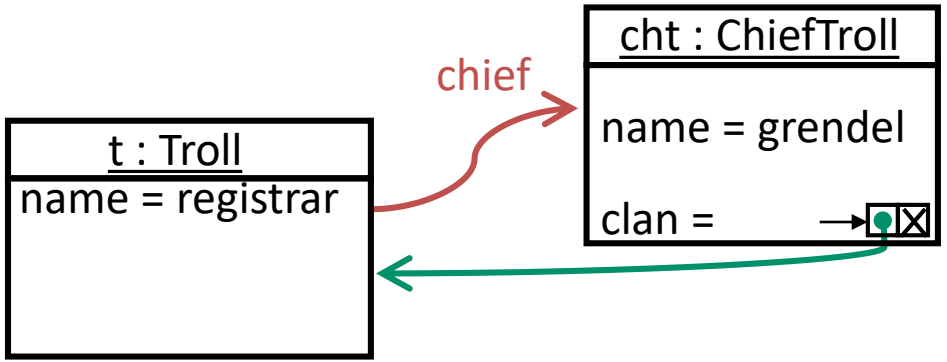
Описание **Troll** (Scheme)

```
(define (create-troll name hunger) (create-instance troll name hunger))
(define (troll this name hunger)
  (let ((person-part (person this name)) (eater-part (create-eater hunger))
        (chief null))
    (lambda (msg)
      (cond ((eq? msg 'get-type) (lambda () (type-extend 'troll
                                              (type-extend 'eater-interface person-part))))
            ((eq? msg 'get-chief) (lambda () chief))
            ((eq? msg 'set-chief!) (lambda (t) (set! chief t)))
            ((eq? msg 'whoareyou?) (lambda () (list 'troll (ask person-part 'whoareyou?))))
            ((eq? msg 'get-hunger) (lambda () (ask eater-part 'get-hunger)))
            ((eq? msg 'eat-people) (lambda () (...))) ...
            (else (get-method msg person-part))))))
```

Описание **ChiefTroll** (Scheme)

```
(define (create-chief-troll name hunger)
  (create-instance chief-troll name hunger))
(define (chief-troll this name hunger)
  (let ((troll-part (troll this name hunger)) (clan null))
    (lambda (msg)
      (cond ((eq? msg 'get-type)
                    (lambda () (type-extend 'chief-troll troll-part)))
            ((eq? msg 'get-clan) (lambda (t) clan))
            ((eq? msg 'add2clan!) (lambda (t) (begin (set! clan (cons t clan))
                                                     (ask t 'set-chief this))))
            ((eq? msg 'banish!) (lambda (t) (begin (set! clan (remove t clan))
                                                   (ask t 'set-chief null))))
            ((eq? msg 'say) (lambda (stuff) (append (ask troll-part 'say stuff)
                                                    (list 'hai))))
            (else (get-method msg troll-part))))))
```

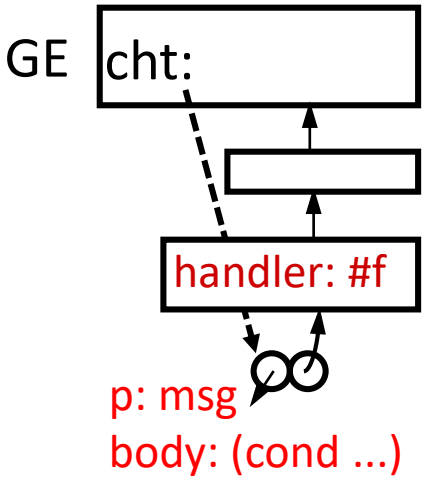
Добавление в клан




```
(lambda (t) (begin (set! clan (cons t clan))
                   (ask t 'set-chief this)))
```

B MBO

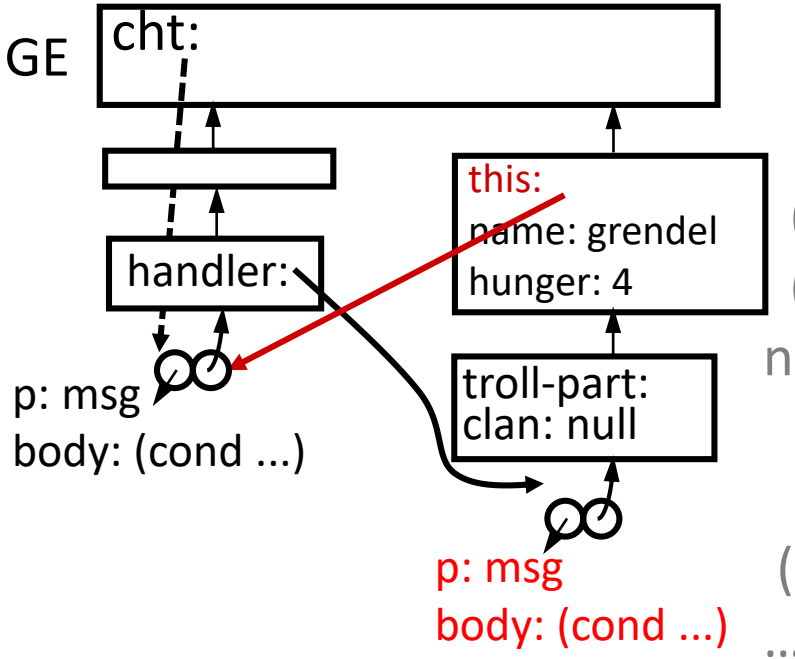
> (define cht (create-chief-troll 'grendel 4) ==>
(create-instance chief-troll 'grendel 4)




```
(define (make-instance)
 (let ((handler #f))
  (lambda (msg)
   (cond
    ((eq? msg 'set-handler!)
     (lambda (handler-proc)
      (set! handler handler-proc)))
    (else (get-method msg handler)))) ))
(define (create-instance maker . args)
 (let* ((instance (make-instance) )
        (handler (apply maker instance args) ))
  (ask instance 'set-handler! handler)
  instance))
```

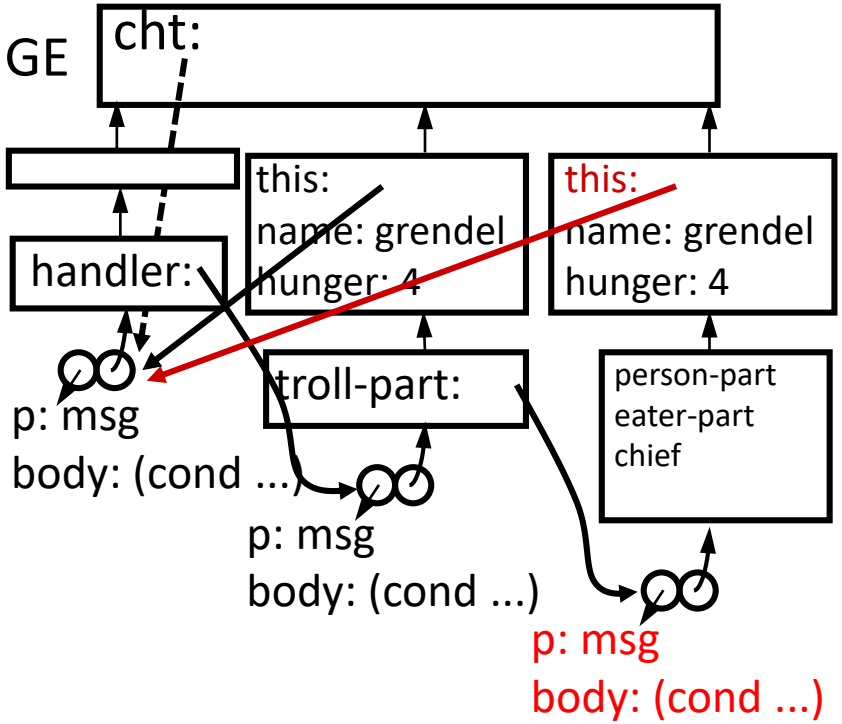
B MBO

```
> (define cht (create-chief-troll 'grendel 4) ==>
    (create-instance chief-troll 'grendel 4) ==>
        (chief-troll this 'grendel 4)
```




```
(define (chief-troll this name hunger)
 (let ((troll-part (troll this name hunger)) (clan
null))
   (lambda (msg)
    (cond ((eq? msg 'get-type)
 (lambda () (type-extend 'chief-troll troll-part)))
...
```

B MBO

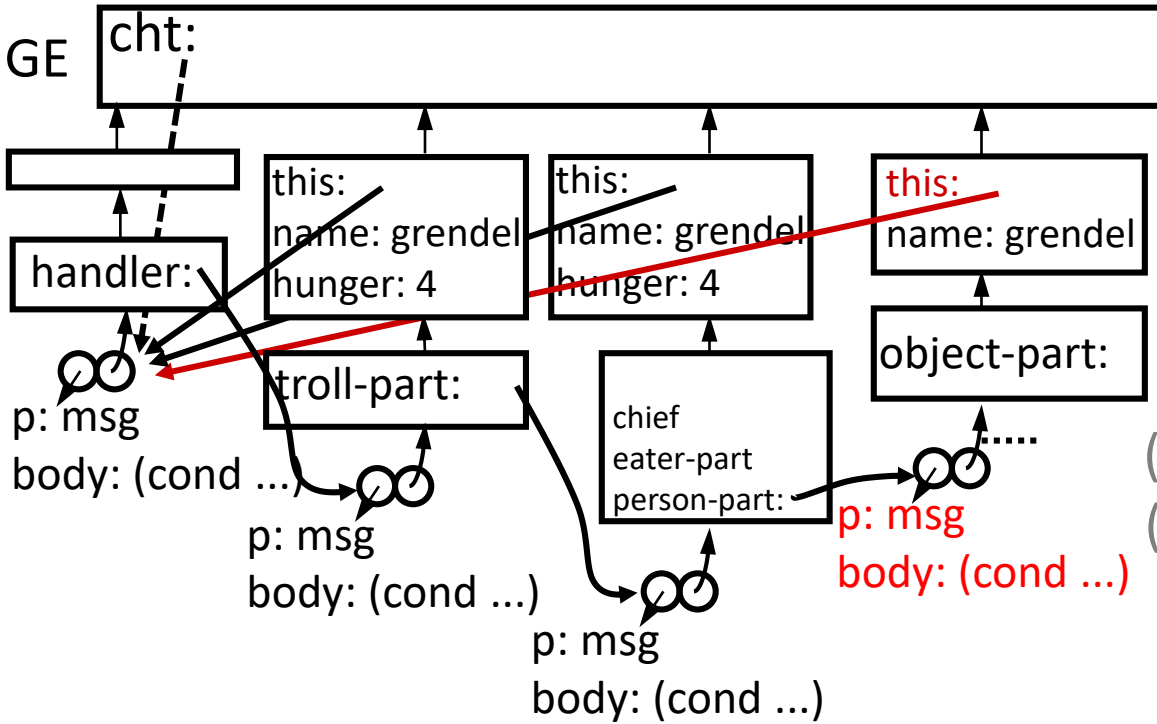



```
> (define cht (create-chief-troll 'grendel 4) ==>
    (create-instance chief-troll 'grendel 4) ==>
      (chief-troll this 'grendel 4) ==>
        (troll this 'grendel 4)
```

```
(define (troll this name hunger)
(let ((person-part (person this name))
   (eater-part (create-eater hunger))
   (chief null))
  (lambda (msg)
    (cond ((eq? msg 'get-type) (lambda ()
...)
```

B MBO

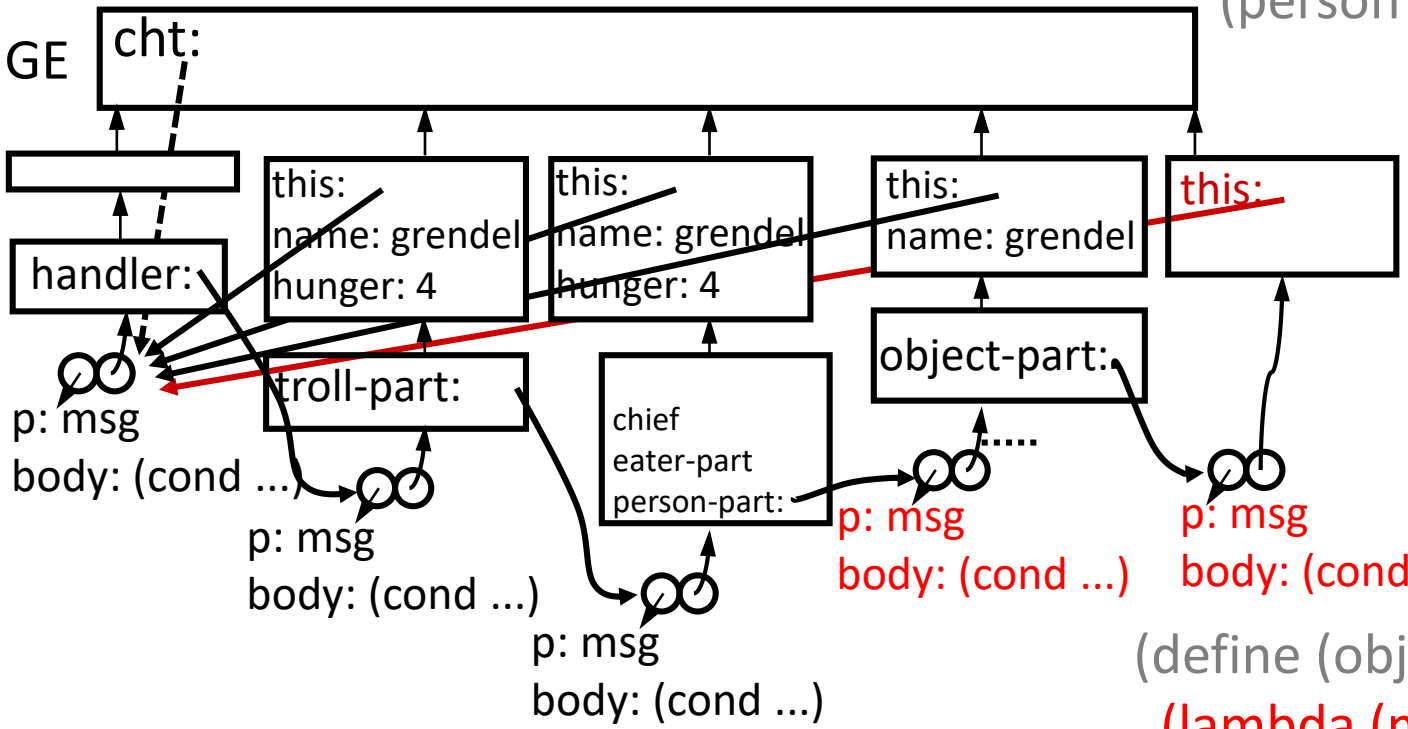
```
> (define cht (create-chief-troll 'grendel 4) ==>
    (create-instance chief-troll 'grendel 4) ==>
      (chief-troll this 'grendel 4) ==>
        (troll this 'grendel 4) ==>
          (person this 'grendel)
```




```
(define (person this name)
  (let ((object-part (object this)))
    (lambda (msg)
      (cond
        ...))))
```

B MBO

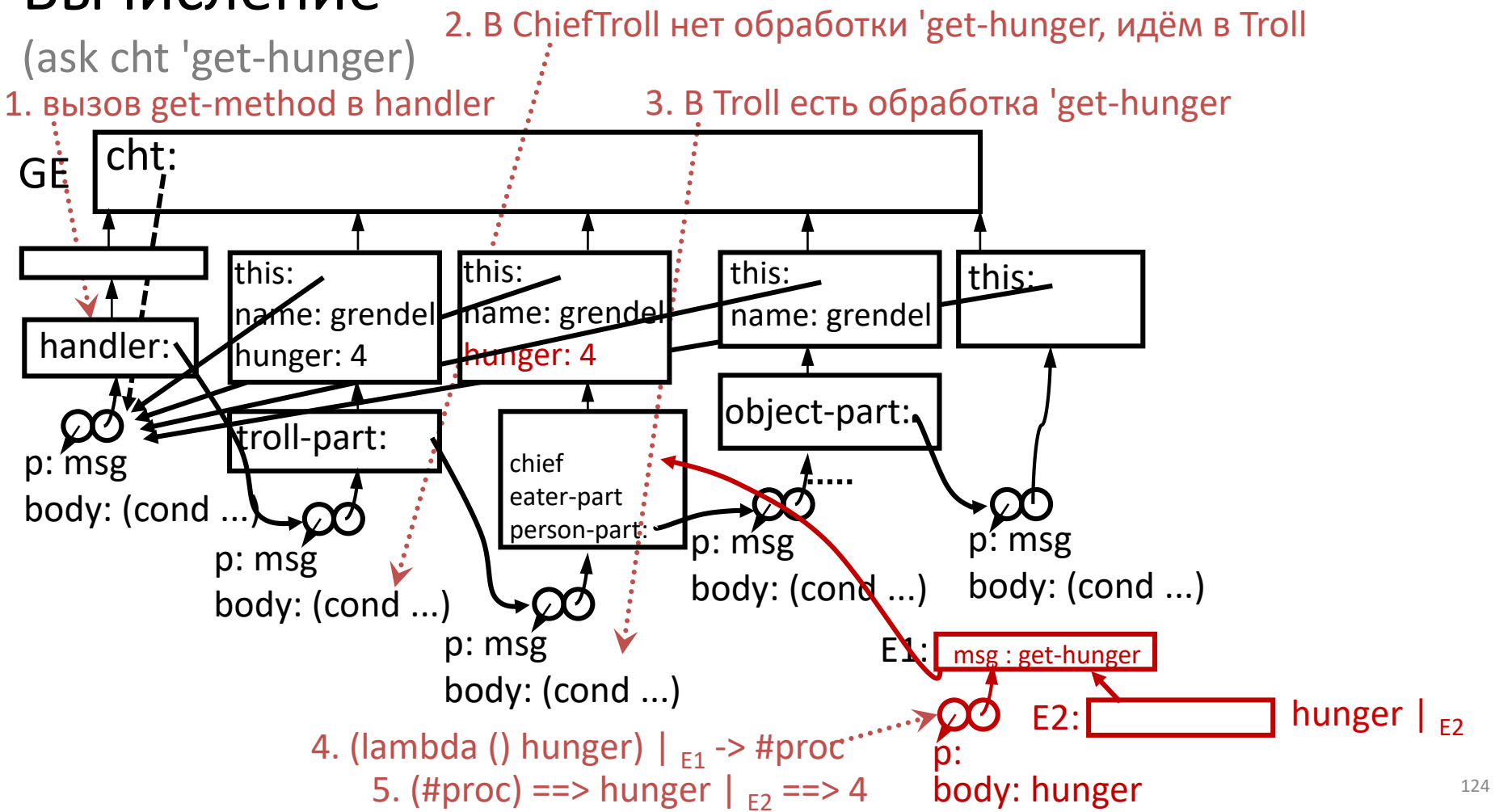
```
> (define cht (create-chief-troll 'grendel 4) ==>
(create-instance chief-troll 'grendel 4) ==>
(chief-troll this 'grendel 4) ==>
(troll this 'grendel 4) ==>
(person this 'grendel) ==>
(object this)
```




```
(define (object this)
  (lambda (msg) ( cond...))))
```

Вычисление

(ask cht 'get-hunger)



Итог лекции 9

- Классы: описывают общую структуру и поведение экземпляров.
- Экземпляры с точки зрения реализации на Scheme содержат:
  - цепочку обработчиков, в которой «заглавное звено» (this) цепочки = instance
- Иерархия классов
  - Наследование структуры и поведения от суперклассов
  - Множественное наследование: правила поиска операций
  - Интерфейсы и реализации как альтернатива множественному наследованию
- Точки зрения на ООП
  - **Модель:** диаграммы классов и объектов
  - **Использование:** способы описания классов и создания экземпляров в Scheme и Racket (racket/class)
  - **Реализация:** отображение понятий ООП в Scheme

(детали реализации в Racket не известны)