CSC324: Principles of Programming Languages

Lecture 9

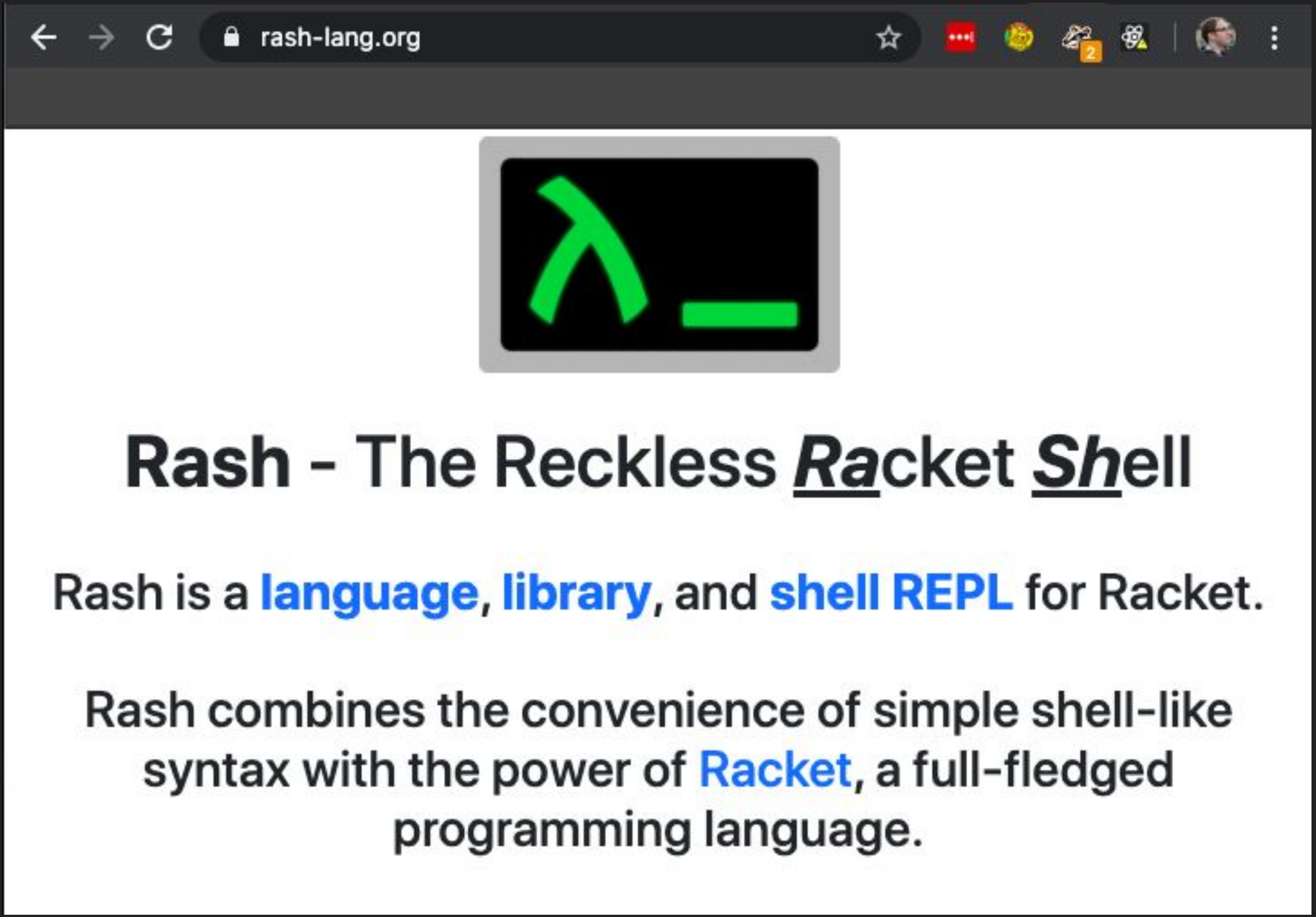
Friday, 5 June, 2020

Announcements

Assignment 1 should be out by EOD today.

Someone brought this to my attention…

...I dare you to try it!



rash-lang.org

Rash - The Reckless ***Ra***cket ***Sh***ell

Rash is a **language**, **library**, and **shell REPL** for Racket.

Rash combines the convenience of simple shell-like syntax with the power of Racket, a full-fledged programming language.

In previous classes...

… we discussed how **lexically-scoped closures** can **close over** identifiers and "carry bound identifiers with it"

… how new syntax can be defined using **syntax transformers (**"macros")

Today, we'll talk about how we can use closures to build a simple **object system** within a non-object oriented language.

Object systems

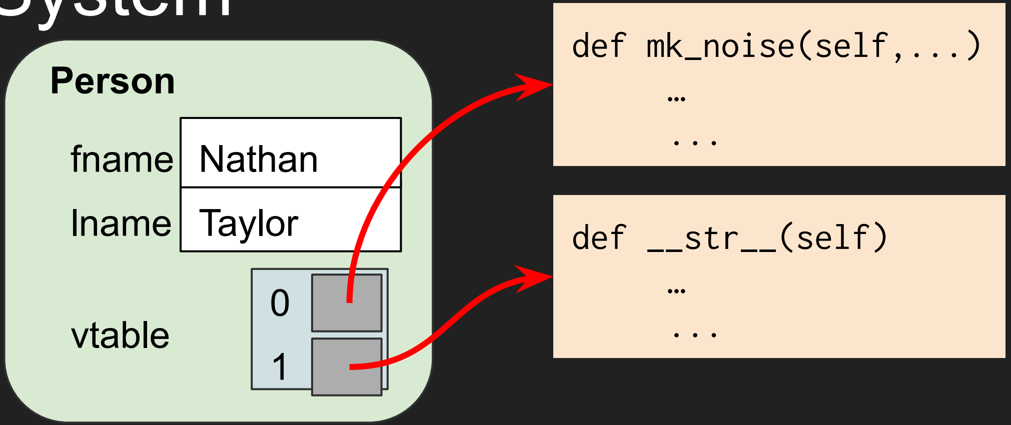
Definition: An **object system** is either the components in an object-oriented language, or a library in a non-object oriented language, that allows for programming in the OOP style.

A vtable-based Object System

- What you're used to if programming in C++ or Java

A vtable-based object is:

- A structure containing the object's fields
- A function pointer table containing the object's methods



```
def mk_noise(self,...)
      …
      ...
```

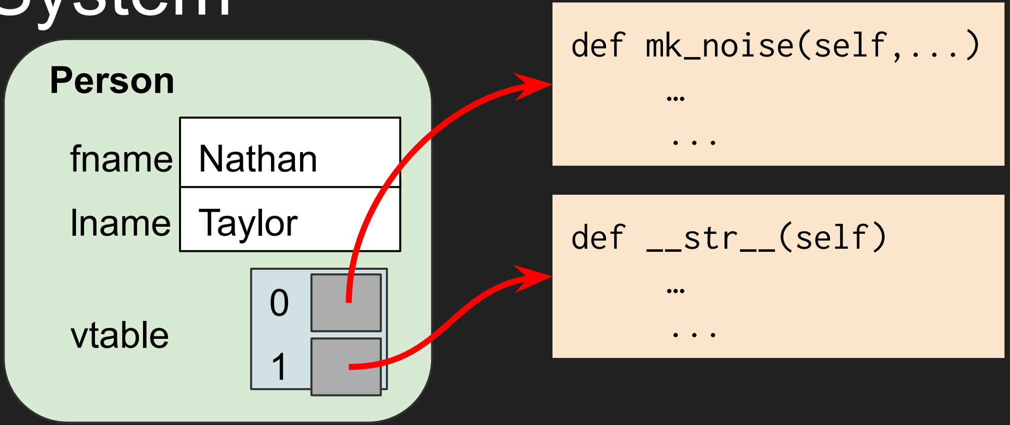
```
def __str__(self)
      …
      ...
```

A vtable-based Object System

- What you're used to if programming in C++ or Java

A vtable-based object is:

- A structure containing the object's fields
- A function pointer table containing the object's methods



```
def mk_noise(self,...)
    …
    ...
```

```
def __str__(self)
    …
    ...
```

The method call

```
    somePerson.__str__()
```

turns into

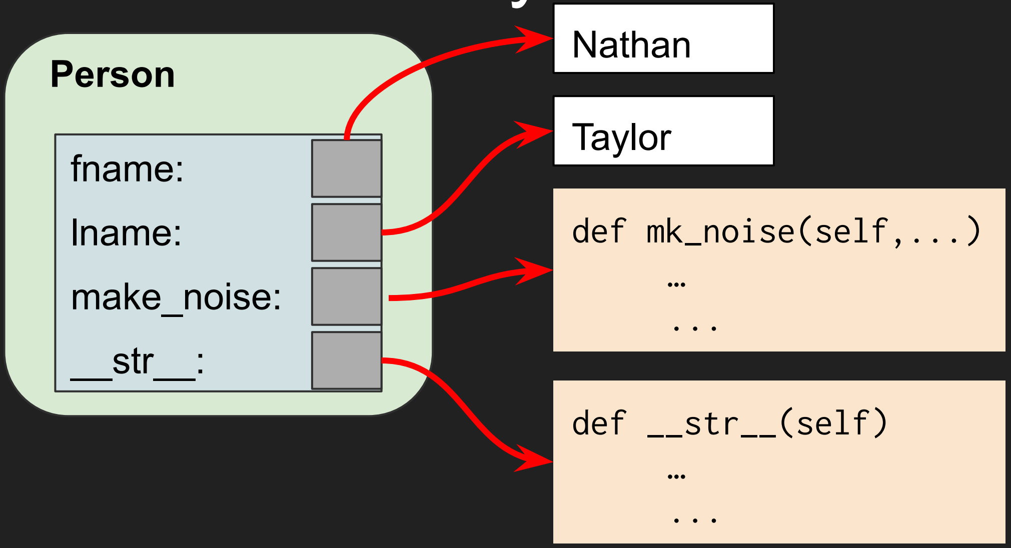
```
    somePerson.vtable[1]()
```

Method and field lookup via a dictionary

Recall that a vtable-based object is:

- A structure containing the object's fields
- A function pointer table containing the object's methods

Such a vtable could be implemented by a **dictionary** data structure, mapping field names to values and method names to functions



```
def mk_noise(self,...)
    …
     ...
```

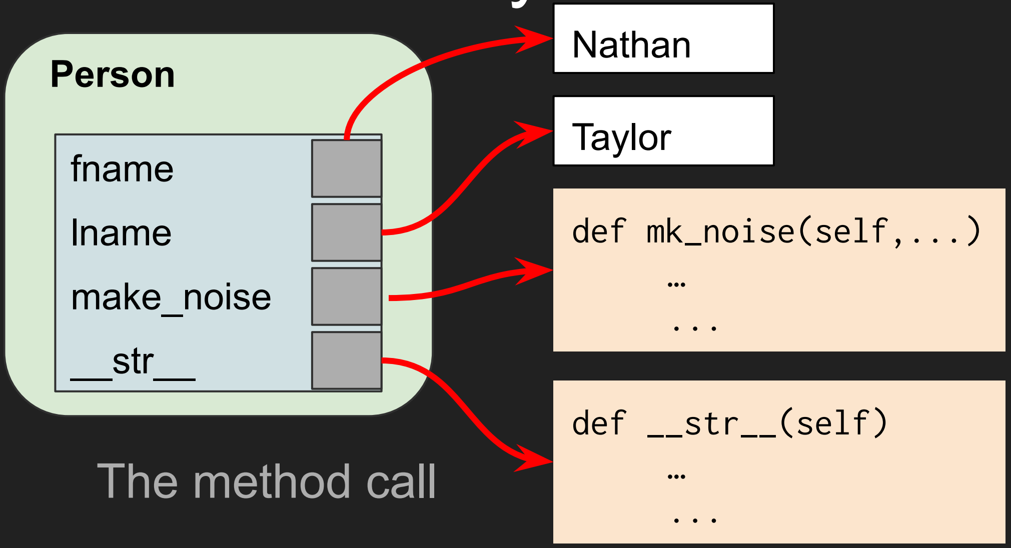
```
def __str__(self)
    …
     ...
```

Method and field lookup via a dictionary

Recall that a vtable-based object is:

- A structure containing the object's fields
- A function pointer table containing the object's methods

Such a vtable could be implemented by a **dictionary** data structure, mapping field names to values and method names to functions



```
def mk_noise(self,...)
     …
      ...
```

```
def __str__(self)
     …
      ...
```

The method call

```
somePerson.__str__()
```

turns into

```
somePerson.method_dict["__str__"]()
```

Note that the only difference between the "user-facing" usage of these method calls and what "actually happens" is a difference in syntax!

...we know how to add custom syntax to a language now ;-)

The vtable method call

```
    somePerson.__str__()
```

turns into

```
    somePerson.vtable[1]()
```

The method dictionary method call

```
    somePerson.__str__()
```

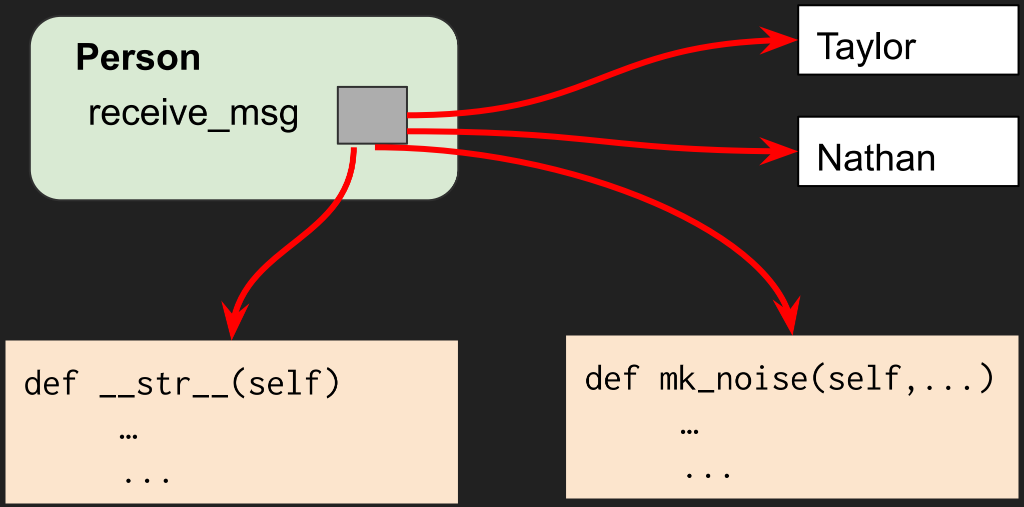
turns into

```
    somePerson.method_dict["__str__"]()
```

A message passing-based Object System

An object is an entity that can can be called with a message, and performs different behaviour depending on the message.

"All You Can Do Is Send A Message"



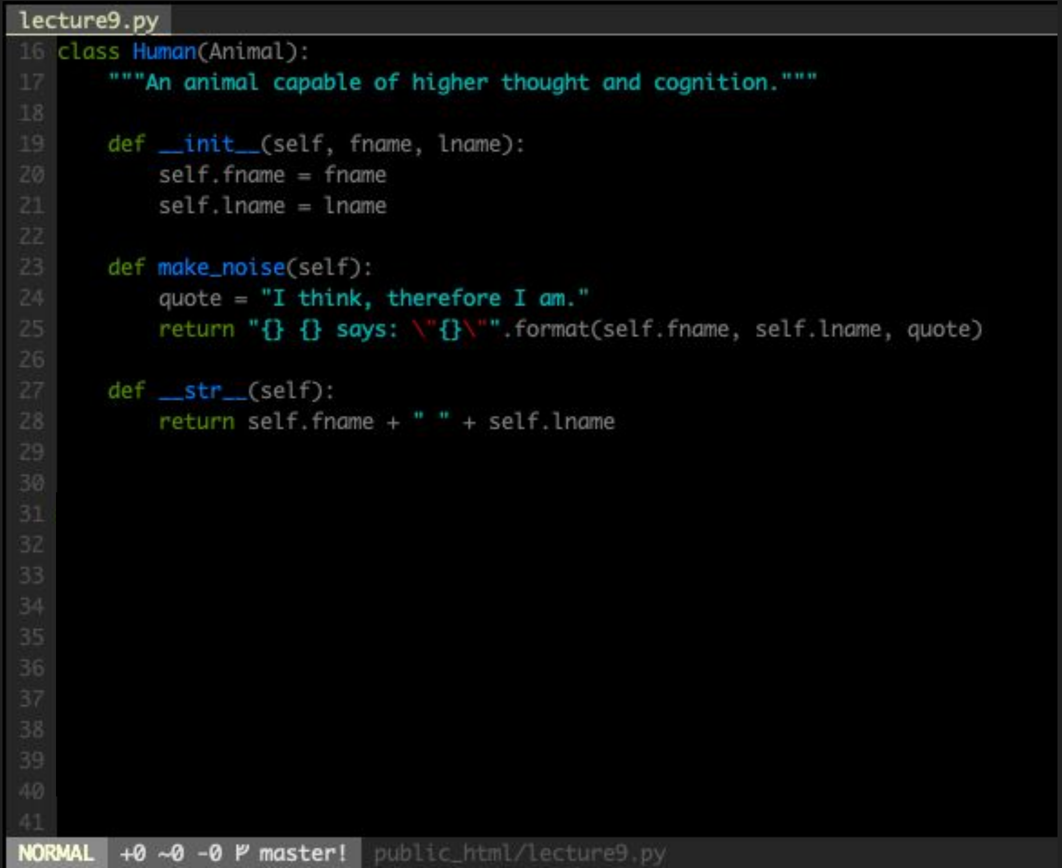
```
def __str__(self)
     …
      ...
```

```
def mk_noise(self,...)
     …
      ...
```

A message passing-based Object System

An object is an entity that can can be called with a message, and performs different behaviour depending on the message.

"All You Can Do Is Send A Message"



lecture9.py

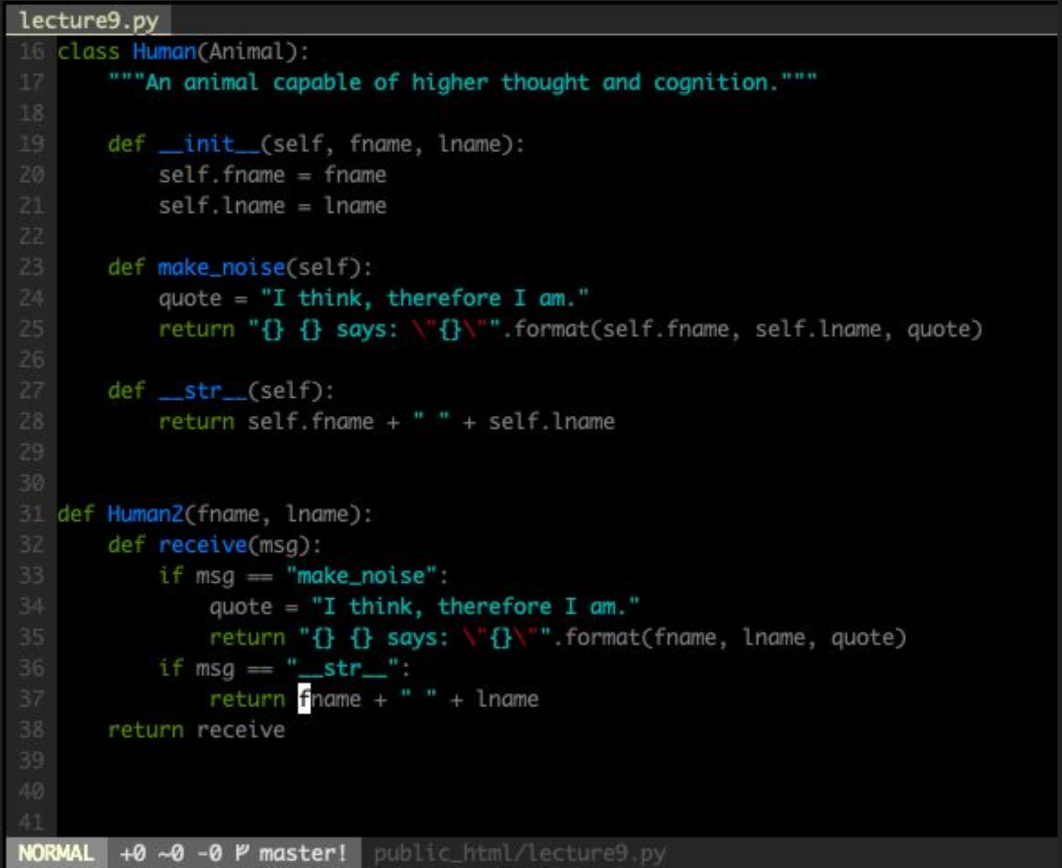
```python
class Human(Animal):
    """An animal capable of higher thought and cognition."""

    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def make_noise(self):
        quote = "I think, therefore I am."
        return "{} {} says: \"{}\"".format(self.fname, self.lname, quote)

    def __str__(self):
        return self.fname + " " + self.lname
```

NORMAL +0 ~0 -0 master! public_html/lecture9.py

A message passing-based Object System

An object is an entity that can can be called with a message, and performs different behaviour depending on the message.

"All You Can Do Is Send A Message"


lecture9.py

```
class Human(Animal):
    """An animal capable of higher thought and cognition."""

    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def make_noise(self):
        quote = "I think, therefore I am."
        return "{} {} says: \"{}\"".format(self.fname, self.lname, quote)

    def __str__(self):
        return self.fname + " " + self.lname


def Human2(fname, lname):
    def receive(msg):
        if msg == "make_noise":
            quote = "I think, therefore I am."
            return "{} {} says: \"{}\"".format(fname, lname, quote)
        if msg == "__str__":
            return fname + " " + lname
    return receive
```

NORMAL +0 ~0 -0 master! public_html/lecture9.py

Object Protocol

Definition: An **object protocol** specifies the semantics of how **objects** in a language and users of those classes interface with one another. The object system implements the semantics of the object protocol.

Programmers are using a language's object protocol when they:

- Construct new objects in a program
- Access fields / call methods in an object
- Access fields / call methods in an object's inheritance hierarchy

Metaobject Protocol

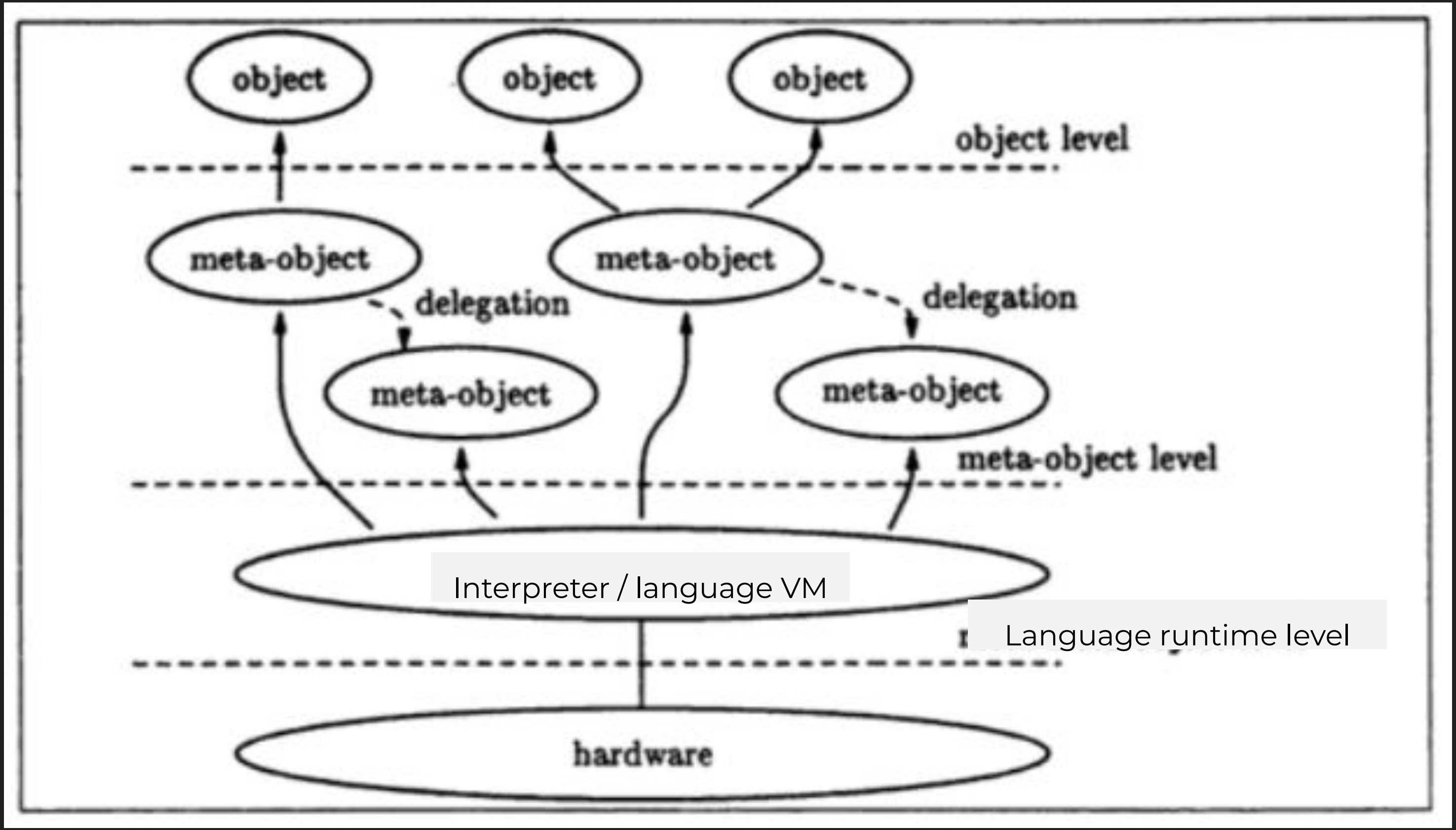
Definition: A **metaobject protocol** specifies the semantics of how **classes** in a language and users of those classes interface with one another.

Programmers are using a metaobject protocol when they:

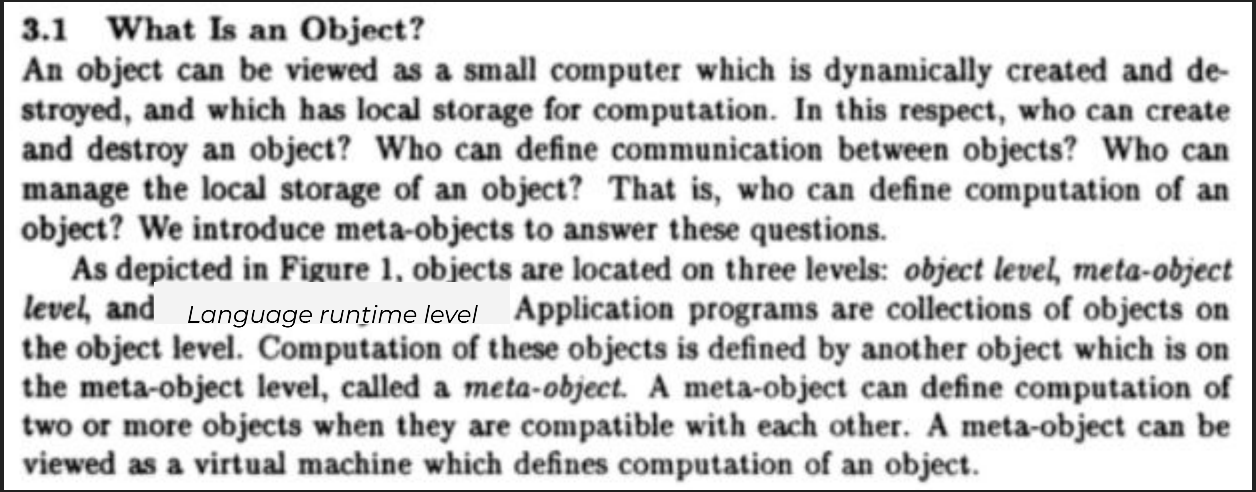
- Define new classes in a program
- Add / remove fields in an object, or circumvent encapsulation / private members
- Change an object's class hierarchy, code, or internal representation

Metaobject Protocol

- Crucially, metaobject protocols are implemented in terms of an underlying object protocol!
- That is to say: <u>if a language has a metaobject protocol, it is implemented within the object system</u>
- In this way, we can create new or modify existing aspects of the programming language using the same tools and terminology (ie. the same "nouns" and "verbs") as what we use to create new or modify existing aspects of a program



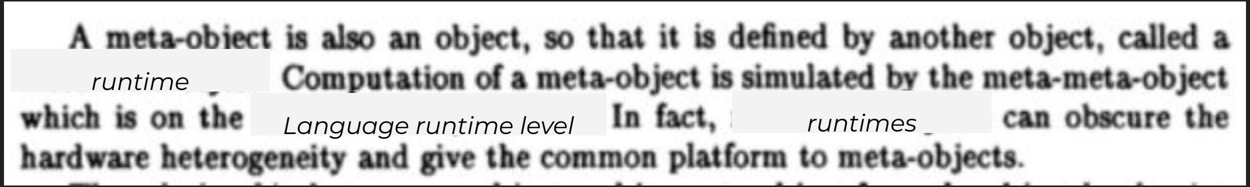
Yokote, et al. A reflective architecture for an object-oriented distributed operating system. Proceedings ofECOOP 1989



3.1 What Is an Object?

An object can be viewed as a small computer which is dynamically created and destroyed, and which has local storage for computation. In this respect, who can create and destroy an object? Who can define communication between objects? Who can manage the local storage of an object? That is, who can define computation of an object? We introduce meta-objects to answer these questions.

As depicted in Figure 1, objects are located on three levels: *object level*, *meta-object level*, and Language runtime level Application programs are collections of objects on the object level. Computation of these objects is defined by another object which is on the meta-object level, called a *meta-object*. A meta-object can define computation of two or more objects when they are compatible with each other. A meta-object can be viewed as a virtual machine which defines computation of an object.



A meta-object is also an object, so that it is defined by another object, called a runtime Computation of a meta-object is simulated by the meta-meta-object which is on the Language runtime level In fact, runtimes can obscure the hardware heterogeneity and give the common platform to meta-objects.

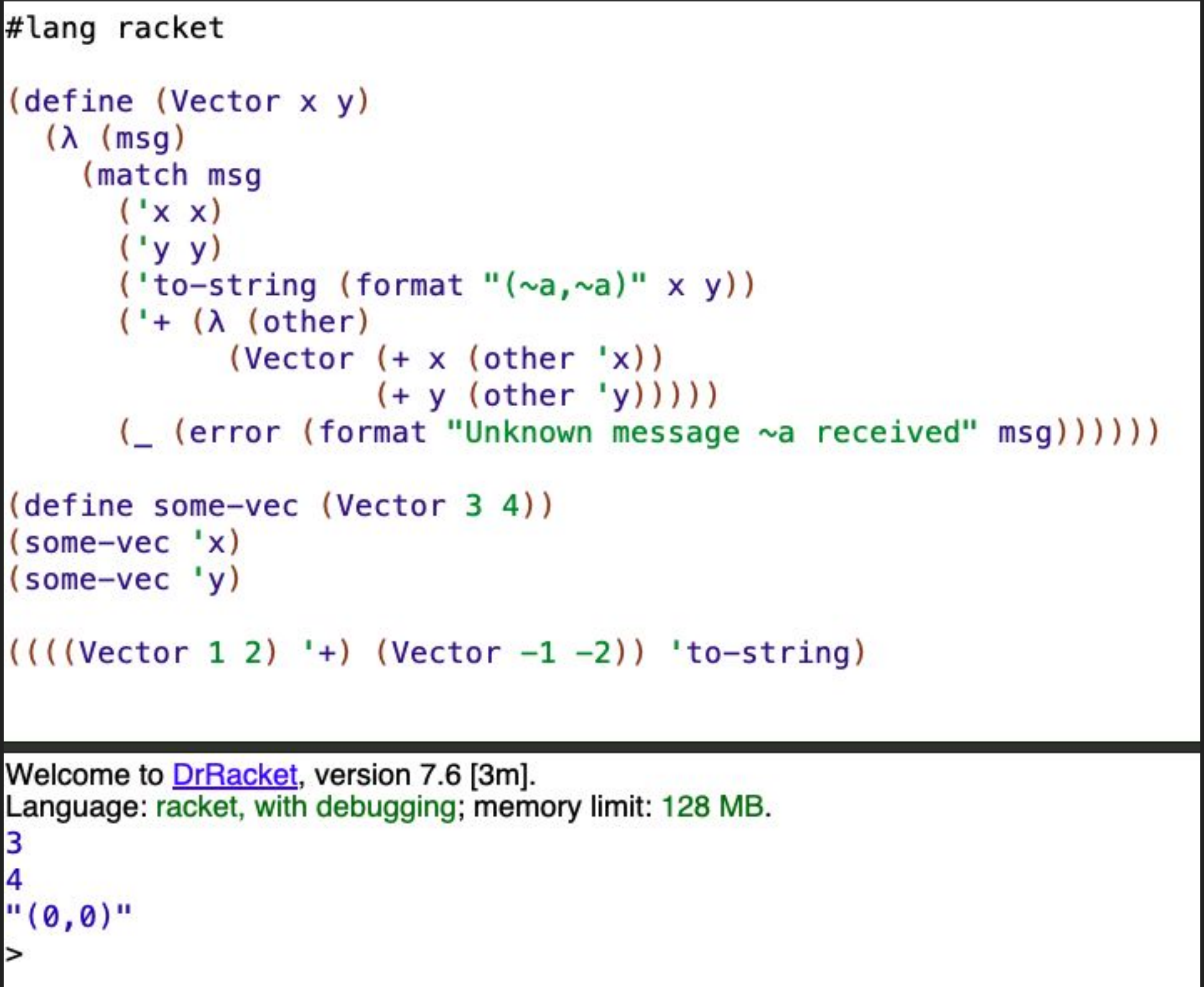
Yokote, et al. A reflective architecture for an object-oriented distributed operating system. Proceedings ofECOOP 1989

Metaobject Protocol

One of the conclusions that we reached was that the "object" need not be a primitive notion in a programming language; one can build objects and their behaviour from little more than assignable value cells and good old lambda expressions.

—Guy Steele on the design of Scheme

A continuation of Lab 4: The Vector (Point) class



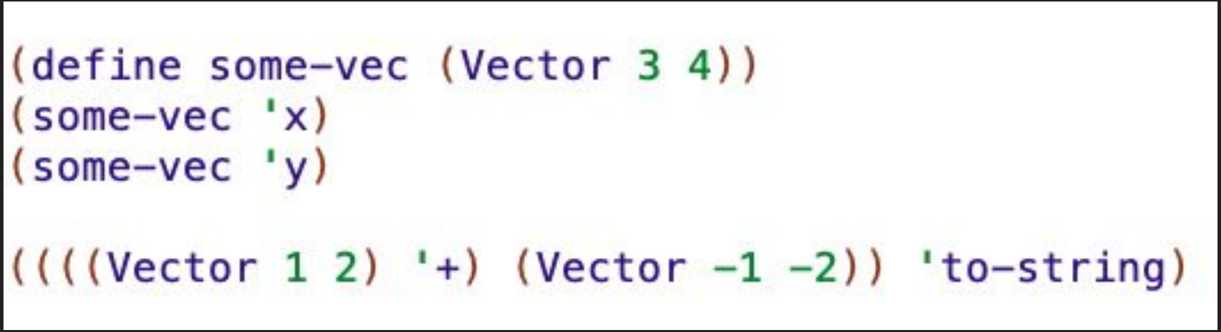
```racket
#lang racket

(define (Vector x y)
  (λ (msg)
    (match msg
      ('x x)
      ('y y)
      ('to-string (format "(~a,~a)" x y))
      ('+ (λ (other)
            (Vector (+ x (other 'x))
                    (+ y (other 'y)))))
      (_ (error (format "Unknown message ~a received" msg))))))

(define some-vec (Vector 3 4))
(some-vec 'x)
(some-vec 'y)

((((Vector 1 2) '+) (Vector -1 -2)) 'to-string)
```

```
Welcome to DrRacket, version 7.6 [3m].
Language: racket, with debugging; memory limit: 128 MB.
3
4
"(0,0)"
>
```

Modifying the syntax of our object model

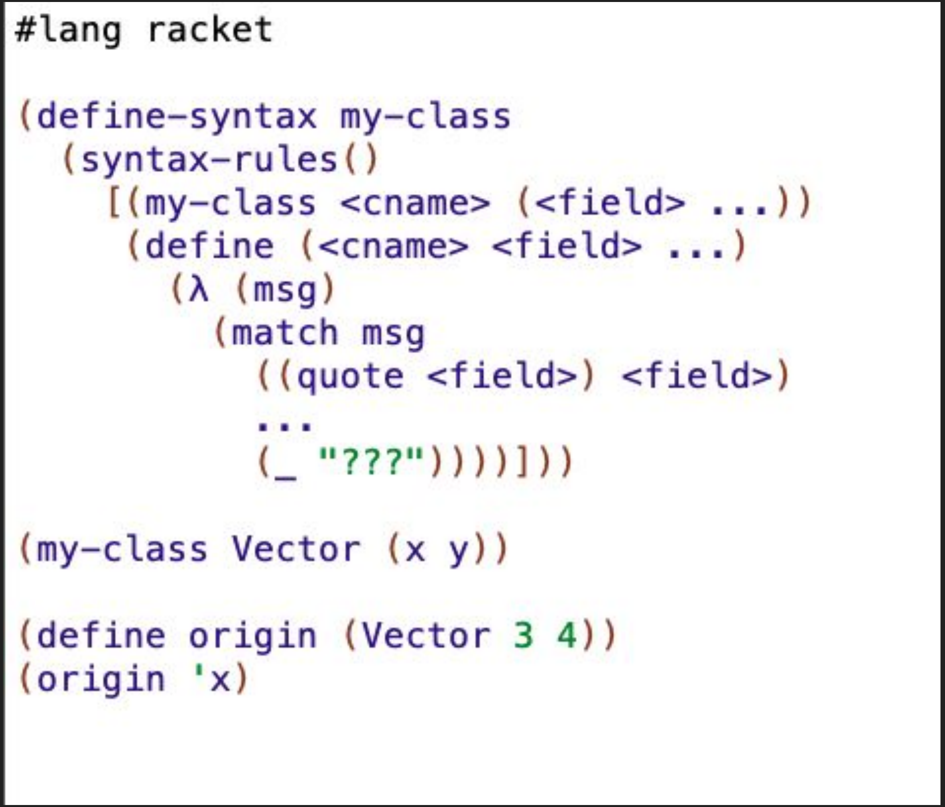
This constructor and accessor syntax "works", but nobody is going to mistake this for an actual OOP language.

Can we apply some syntax transformers to improve the ergonomics of this?



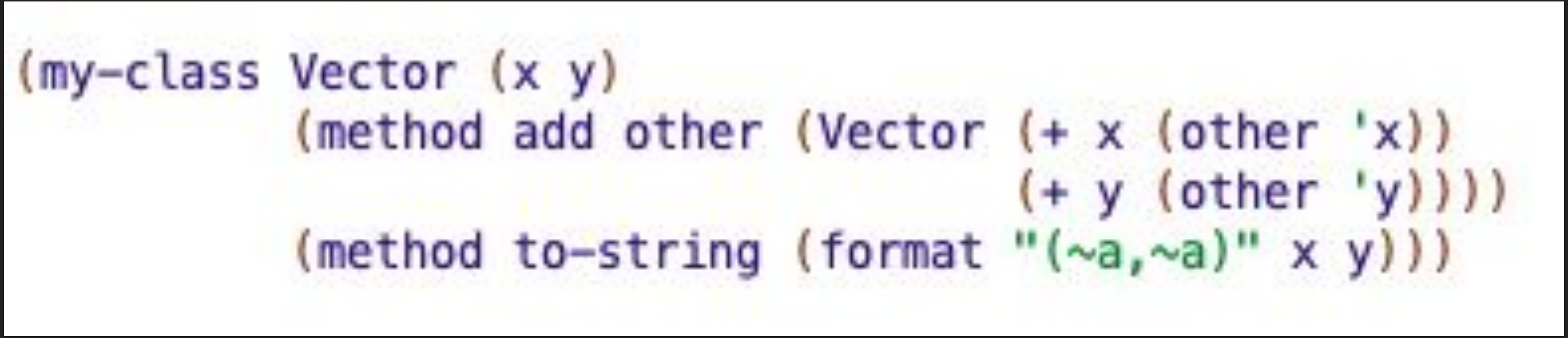
```
(define some-vec (Vector 3 4))
(some-vec 'x)
(some-vec 'y)

((((Vector 1 2) '+) (Vector -1 -2)) 'to-string)
```

A macro for class definition



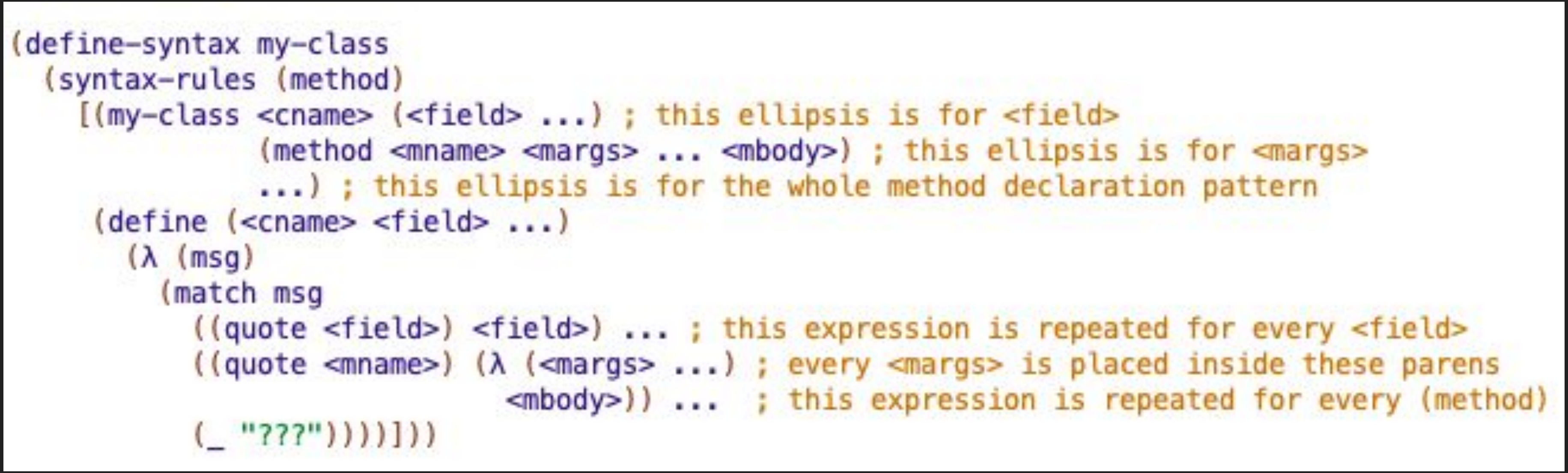
```
#lang racket

(define-syntax my-class
  (syntax-rules()
    [(my-class <cname> (<field> ...))
     (define (<cname> <field> ...)
       (λ (msg)
         (match msg
           ((quote <field>) <field>)
           ...
           (_ "???"))))]))

(my-class Vector (x y))

(define origin (Vector 3 4))
(origin 'x)
```

Implementing methods



```
(my-class Vector (x y)
          (method add other (Vector (+ x (other 'x))
                                    (+ y (other 'y))))
          (method to-string (format "(~a,~a)" x y)))
```

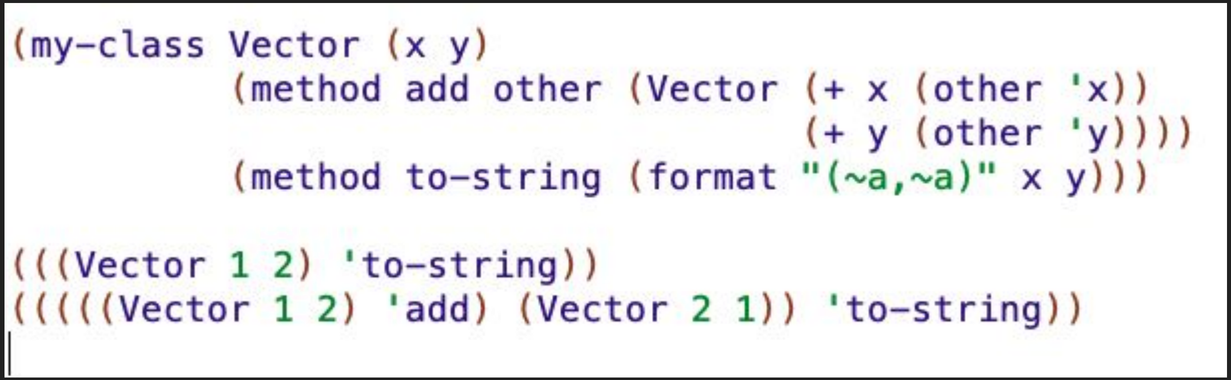
A class macro with the (method …) form



```
(define-syntax my-class
  (syntax-rules (method)
    [(my-class <cname> (<field> ...) ; this ellipsis is for <field>
               (method <mname> <margs> ... <mbody>) ; this ellipsis is for <margs>
               ...) ; this ellipsis is for the whole method declaration pattern
     (define (<cname> <field> ...)
       (λ (msg)
         (match msg
           ((quote <field>) <field>) ... ; this expression is repeated for every <field>
           ((quote <mname>) (λ (<margs> ...) ; every <margs> is placed inside these parens
                              <mbody>)) ...  ; this expression is repeated for every (method)
           (_ "???"))))]))
```

Inconsistency in accessing fields within a method

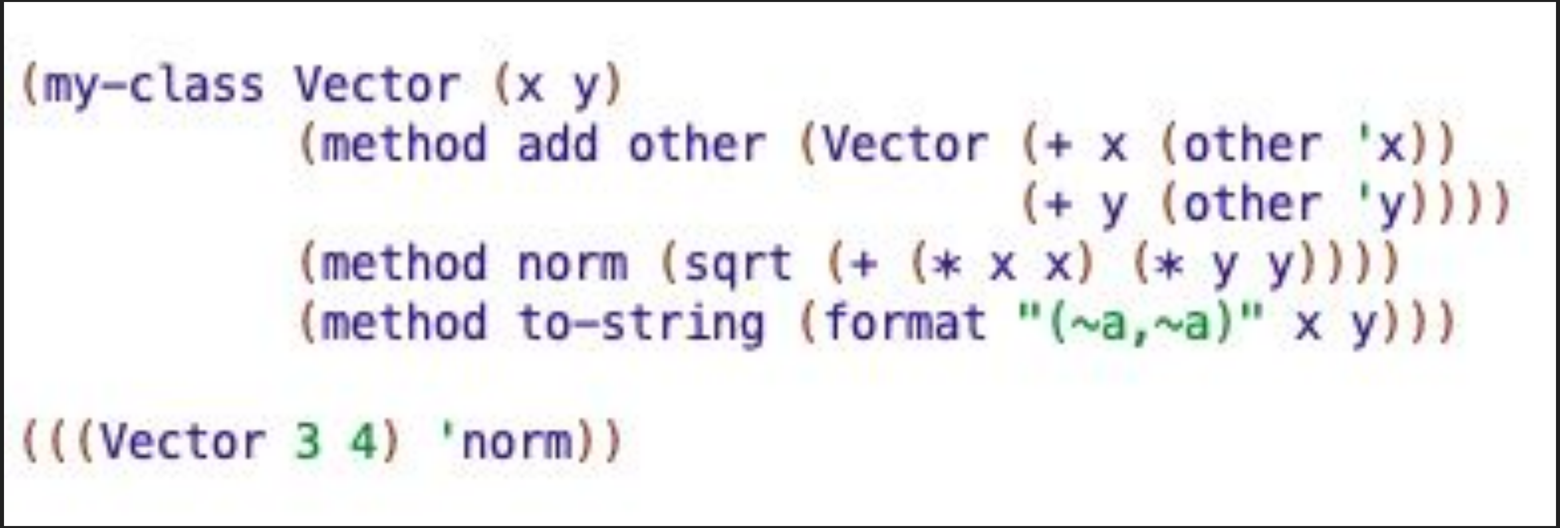
This implementation uses Java-style "implicit this pointer" style syntax. Notice that this mixes using the fields `x` and `y` directly, and their quoted form when we access `(other 'x)`. Ugly!!!!!!!



```
(my-class Vector (x y)
          (method add other (Vector (+ x (other 'x))
                                    (+ y (other 'y))))
          (method to-string (format "(~a,~a)" x y)))

(((Vector 1 2) 'to-string))
(((((Vector 1 2) 'add) (Vector 2 1)) 'to-string))
```

Inability to call a method within a method

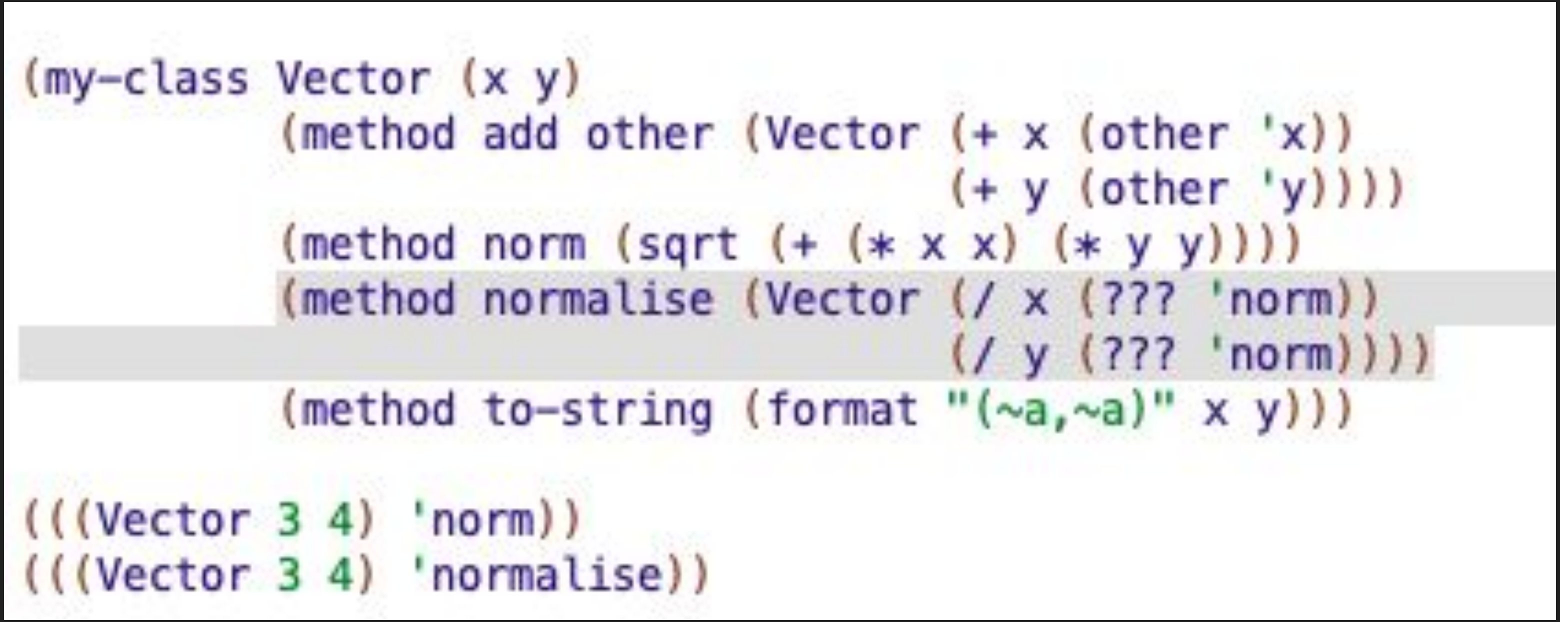
Imagine I implemented a method to return the norm of a Vector. No problem...



```
(my-class Vector (x y)
          (method add other (Vector (+ x (other 'x))
                                    (+ y (other 'y))))
          (method norm (sqrt (+ (* x x) (* y y))))
          (method to-string (format "(~a,~a)" x y)))

(((Vector 3 4) 'norm))
```

Inability to call a method within a method

But what if I want to call a method inside another method? We need to send `norm to something, but the thing we send it to...is the thing we're constructing!

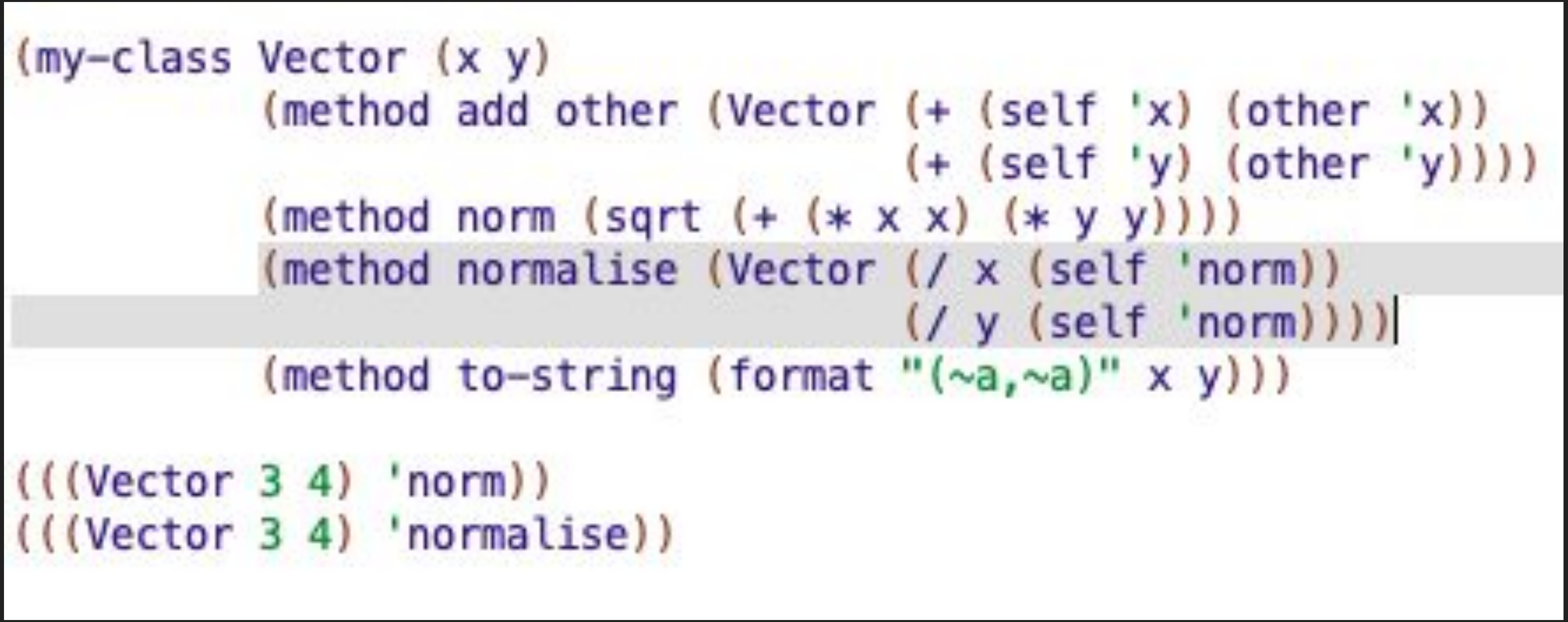


```
(my-class Vector (x y)
          (method add other (Vector (+ x (other 'x))
                                    (+ y (other 'y))))
          (method norm (sqrt (+ (* x x) (* y y))))
          (method normalise (Vector (/ x (??? 'norm))
                                    (/ y (??? 'norm))))
          (method to-string (format "(~a,~a)" x y)))

(((Vector 3 4) 'norm))
(((Vector 3 4) 'normalise))
```

Inconsistency in accessing fields within a method

Next time, we'll talk about implementing a python-like 'self' reference.

Have a `classy` weekend!



```
(my-class Vector (x y)
          (method add other (Vector (+ (self 'x) (other 'x))
                                    (+ (self 'y) (other 'y))))
          (method norm (sqrt (+ (* x x) (* y y))))
          (method normalise (Vector (/ x (self 'norm))
                                    (/ y (self 'norm))))
          (method to-string (format "(~a,~a)" x y)))

(((Vector 3 4) 'norm))
(((Vector 3 4) 'normalise))
```