Example: Class Declaration

```
class fish extends object
  field size
  method initialize (s) set size = s
  method get_size() size
  method grow(food)
    set size = +(size, food)
  method eat(other_fish)
    let s = send other_fish get_size()
     in send self grow(s)

let f = new fish(10)
 in begin
     send f grow(2);
     send f get_size()
    end
```

Example: Derived Class

```
class fish extends object
  field size
  method initialize (s) set size = s
  method get_size() size
  method grow(food)
    set size = +(size, food)
  method eat(other_fish)
    let s = send other_fish get_size()
     in send self grow(s)

class colorfish extends fish
  field color
  method set_color(c) set color = c
  method get_color() color
...
```

Example: Overriding and Super

```
class fish extends object
  field size
  method initialize (s) set size = s
  method get_size() size
  method grow(food)
    set size = +(size, food)
  method eat(other_fish)
    let s = send other_fish get_size()
     in send self grow(s)
...
class pickyfish extends fish
  method grow(food)
   super grow(-(food, 1))
...
```

Example: Field Scope

```
class fish extends object
  field size
  method initialize (s) set size = s
  method get_size() size
  method grow(food)
    set size = +(size, food)
  method eat(other_fish)
    let s = send other_fish get_size()
     in send self grow(s)
...
class pickyfish extends fish
  method grow(food)
   set size = +(size, -(food, 1))
...
```

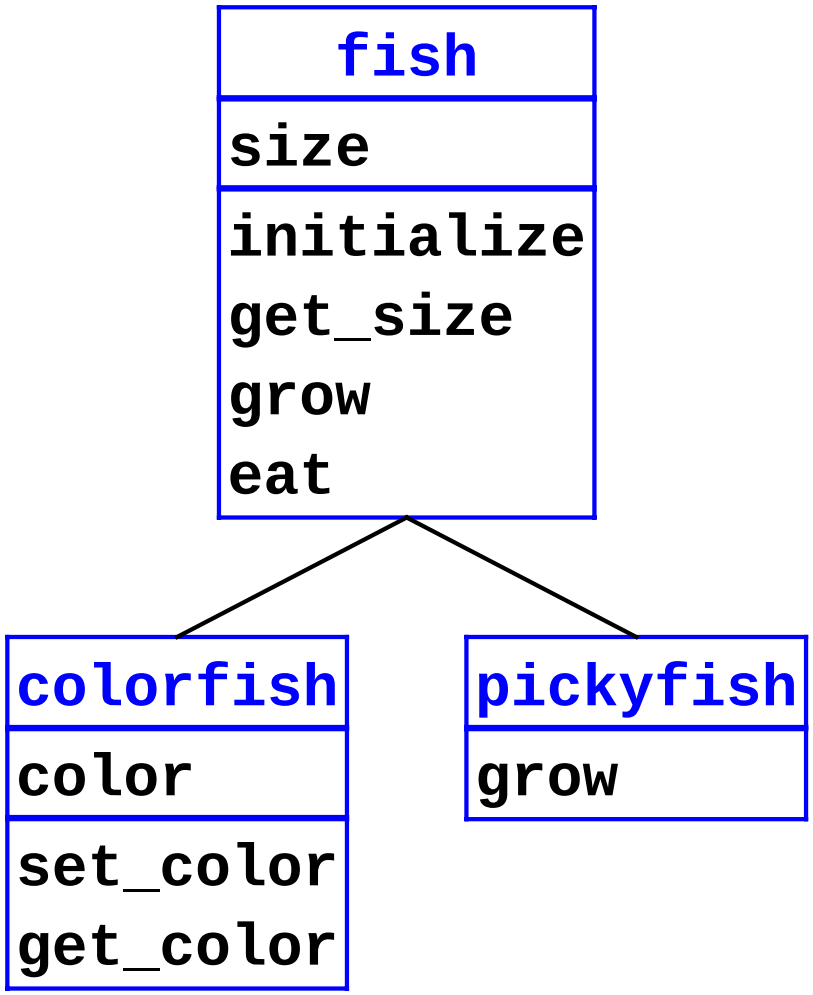
- Scope: methods in a derived class see fields of superclass

Example: Hiding Fields

```
class fish extends object
  field size
  method initialize (s) set size = s
  method get_size() size
  method grow(food)
    set size = +(size, food)
  method eat(other_fish)
    let s = send other_fish get_size()
     in send self grow(s)
...
class bigtailfish extends fish
  field size
  method get_tail_size() size
...
```

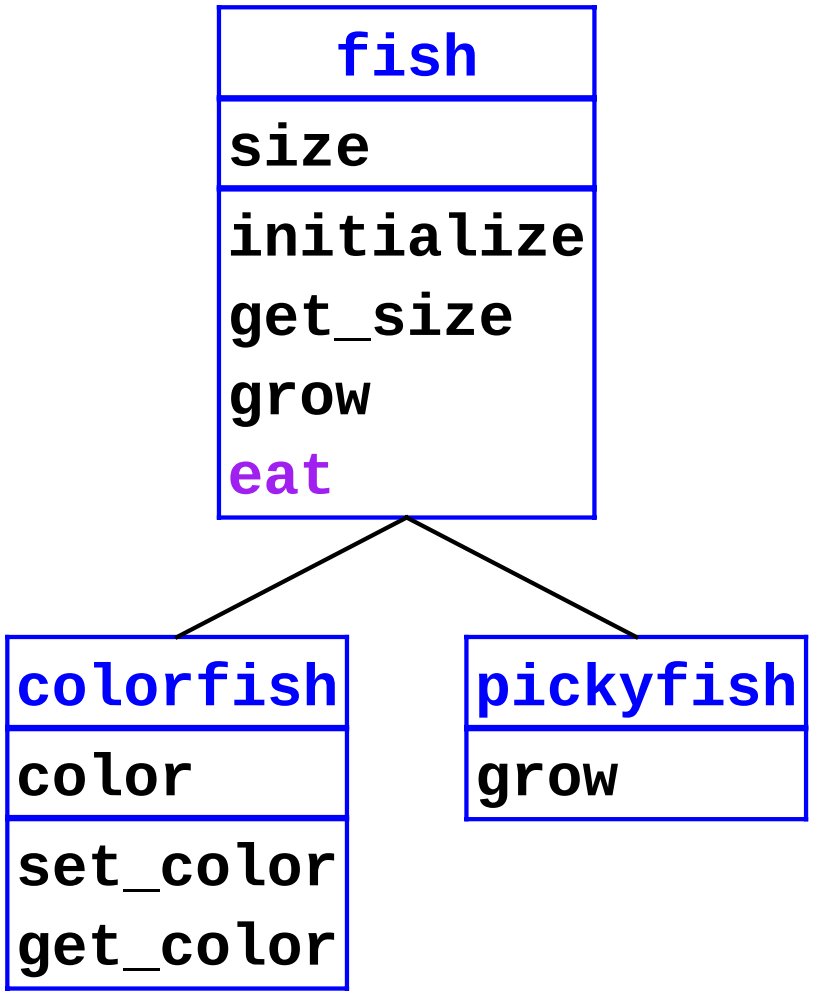
- Scope: local fields can hide superclass fields

Evaluation




```
let
o1 = new colorfish(3)
o2 = new pickyfish(6)
in begin
send o2 eat(o1);
send o2 get_size()
end
```

Evaluation




```
eat(o) let
s = send o get_size()
in send self grow(s)
```

```
let
o1 = new colorfish(3)
o2 = new pickyfish(6)
in begin
send o2 eat(o1);
send o2 get_size()
end
```

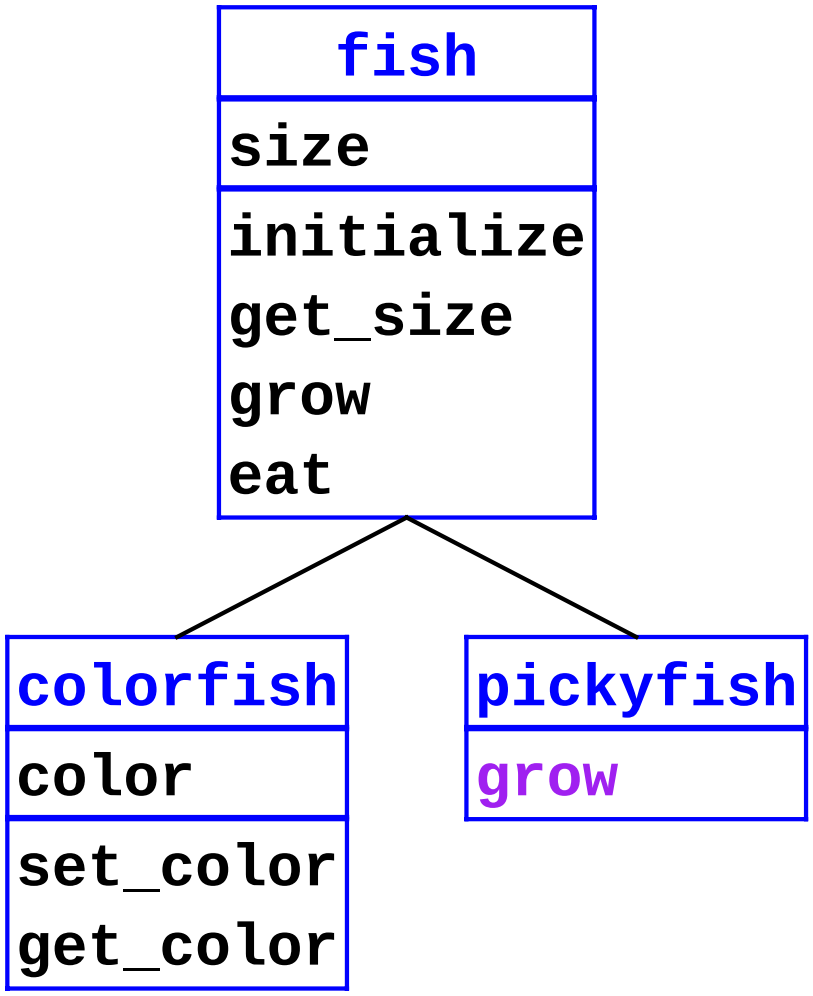
o1 =

colorfish
size = 3
color = 0

o2 =

pickyfish
size = 6

Evaluation




```
grow(f)
super grow(-(f, 1))
```

```
let
o1 = new colorfish(3)
o2 = new pickyfish(6)
in begin
send o2 eat(o1);
send o2 get_size()
end
```

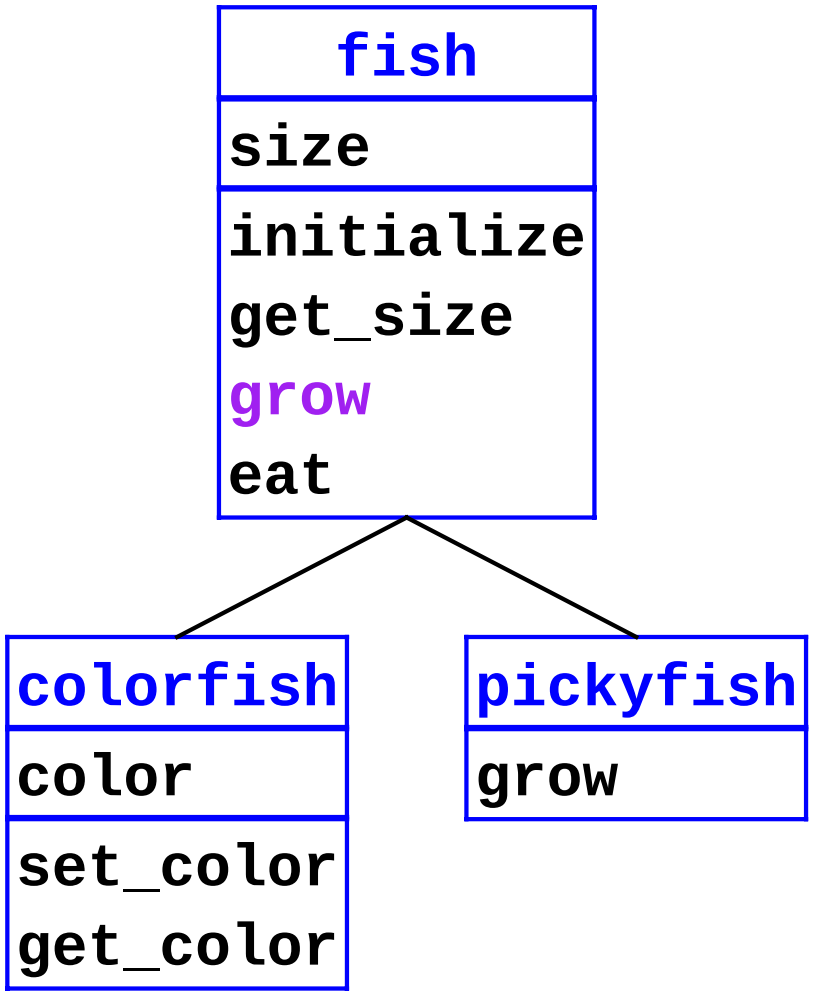
o1 =

colorfish
size = 3
color = 0

o2 =

pickyfish
size = 6

Evaluation




```
grow(f)
set size=+(size,f)
```

```
let
o1 = new colorfish(3)
o2 = new pickyfish(6)
in begin
send o2 eat(o1);
send o2 get_size()
end
```

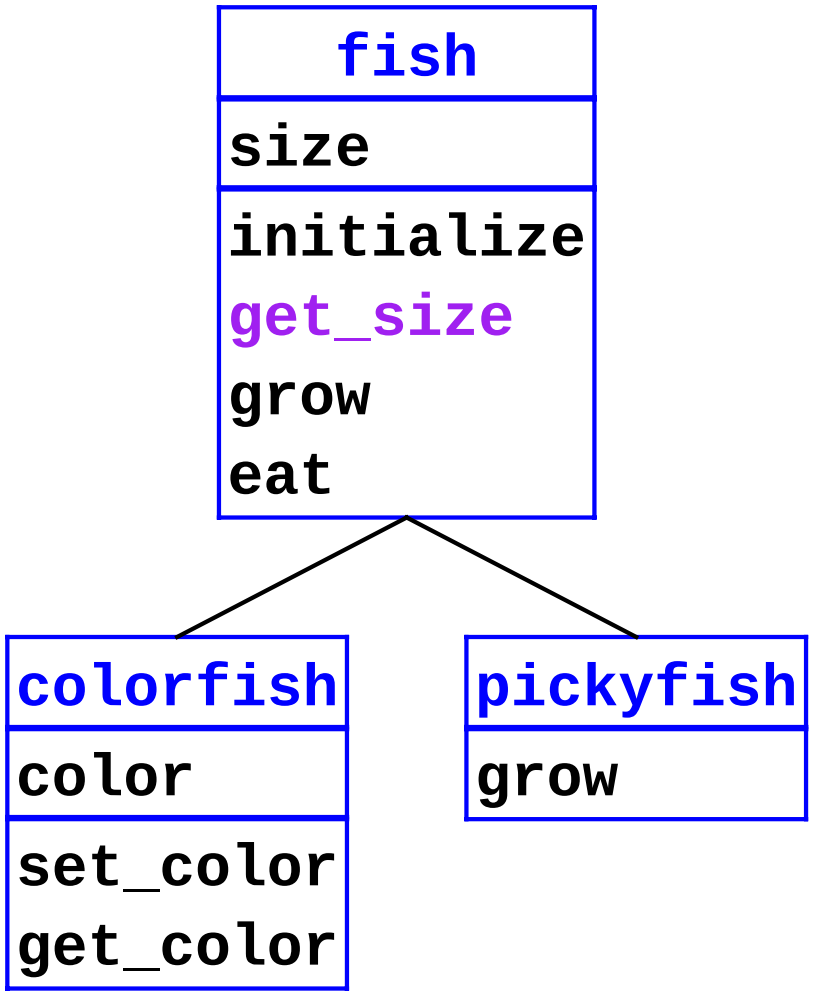
o1 =

colorfish
size = 3
color = 0

o2 =

pickyfish
size = 6

Evaluation



get_size() size

```
let
o1 = new colorfish(3)
o2 = new pickyfish(6)
in begin
send o2 eat(o1);
send o2 get_size()
end
```

o1 =

colorfish
size = 3
color = 0

o2 =

pickyfish
size = 8

Interpreter

- Build class tree

```
(define eval-program
  (lambda (pgm)
    (cases program pgm
      (a-program (c-decls exp)
        (elaborate-class-decls! c-decls)
        (eval-expression exp (init-env))))))
```

```
elaborate-class-decls! : lstof-cls-decl ->
```

Interpreter

- Expression form: object creation

```
(new-object-exp (class-name rands)
  (let ((args (eval-rands rands env))
        (obj (new-object class-name)))
    (find-method-and-apply
      'initialize class-name obj args)
    obj))
```

```
elaborate-class-decls! : lstof-cls-decl ->
new-object : sym -> object
find-method-and-apply : sym sym object
                        lstof-expval -> expval
```

Interpreter

- Expression form: method call

```
(method-app-exp (obj-exp method-name rands)
  (let ((args (eval-rands rands env))
        (obj (eval-expression obj-exp env)))
    (find-method-and-apply
      method-name (object->class-name obj)
      obj args)))
```

```
elaborate-class-decls! : lstof-cls-decl ->
new-object : sym -> object
find-method-and-apply : sym sym object
                        lstof-expval -> expval
```

Interpreter

- Expression form: super call

```
(super-call-exp (method-name rands)
  (let ((args (eval-rands rands env))
        (obj (apply-env env 'self)))
    (find-method-and-apply
      method-name (apply-env env '%super)
      obj args)))
```

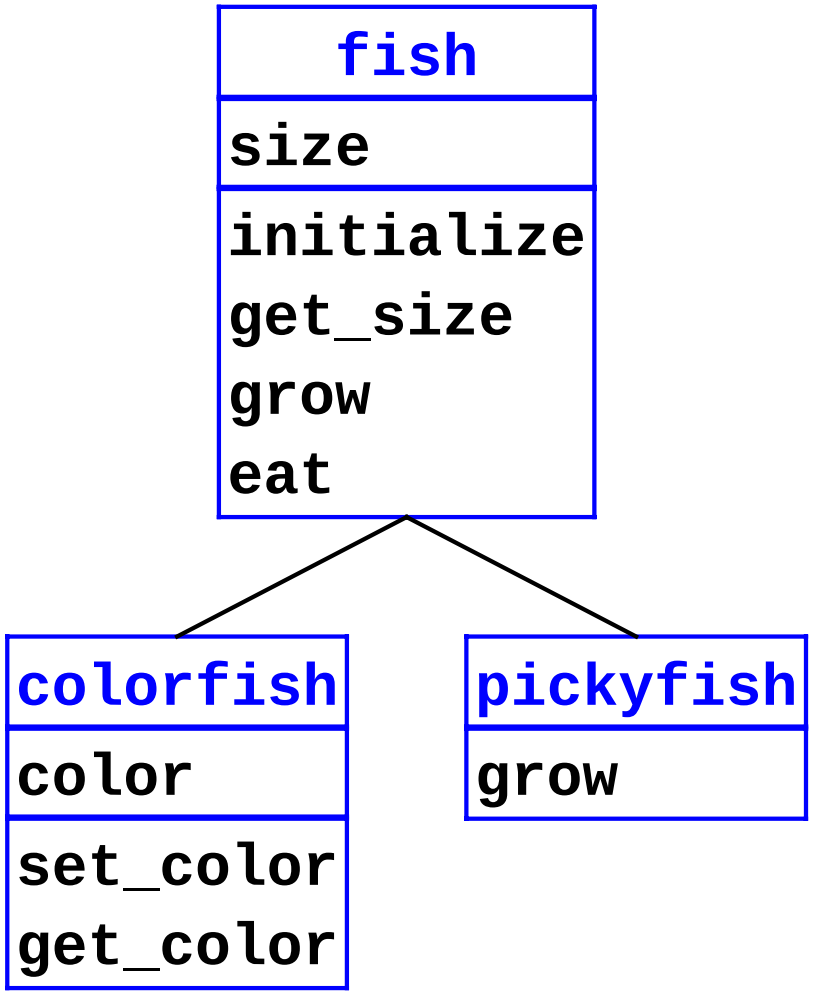
```
elaborate-class-decls! : lstof-cls-decl ->
new-object : sym -> object
find-method-and-apply : sym sym object
                        lstof-expval -> expval
```

Class Elaboration

- Simply keep the declarations

```
class fish ...

class colorfish
  extends fish
 ...

class pickyfish
  extends fish
 ...
```

=



Class Elaboration

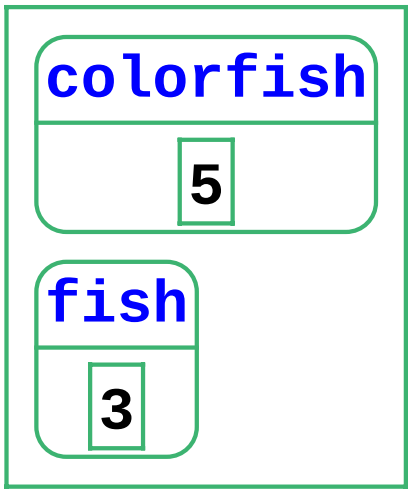
```
(define the-class-env '())

(define (elaborate-class-decls! c-decls)
    (set! the-class-env c-decls)))
```

Class Elaboration

```
;; lookup-class : sym -> class-decl
(define (lookup-class name)
  (lookup name the-class-env))

;; lookup : sym lstof-cls-decl -> class-decl
(define (lookup-class-in-env name env)
  (cond
   [(null? env)
    (eopl:error 'lookup-class
                "Unknown class ~s" name)]
   [(eqv? (class-decl->class-name (car env))
          name)
    (car env)]
   [else (lookup name (cdr env))]))
```

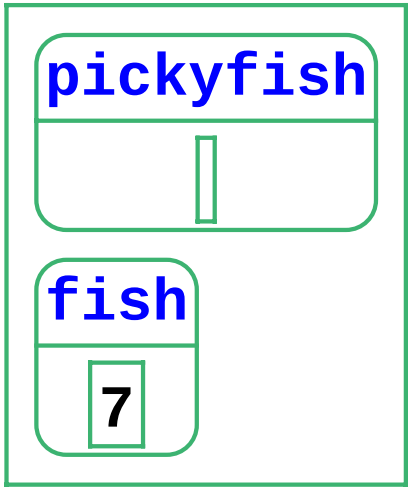
Object Representation

- An object = a list of ***parts***
  - from instantiated class up to base class



Object Representation

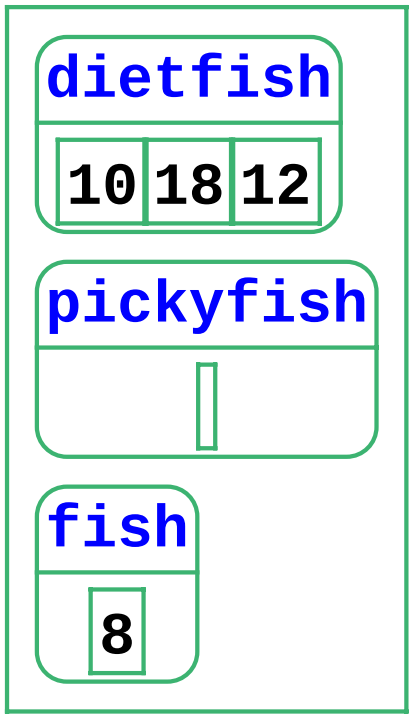
- An object = a list of ***parts***
  - from instantiated class up to base class



Object Representation

- An object = a list of ***parts***
  - from instantiated class up to base class

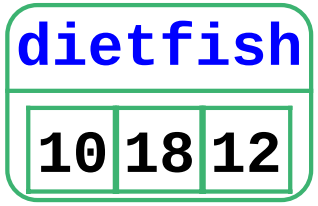
```
class dietfish
  extends pickyfish
  field carbos
  field sodium
  field cholestorol
  ...
```



- Use part vectors in environments

Object Representation

```
(define-datatype part part?
  (a-part
    (class-name symbol?)
    (fields vector?)))
```



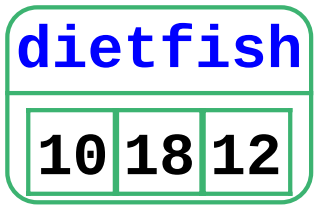

```
;; An object is a list of parts
```

Object Representation

```
;; new-object : sym -> object
(define (new-object cls-name)
  (if (eqv? cls-name 'object)
      '()
      (let ([c-decl (lookup-class cls-name)])
        (cons
         (make-first-part c-decl)
         (new-object (class-decl->super-name
                      c-decl)))))))
```

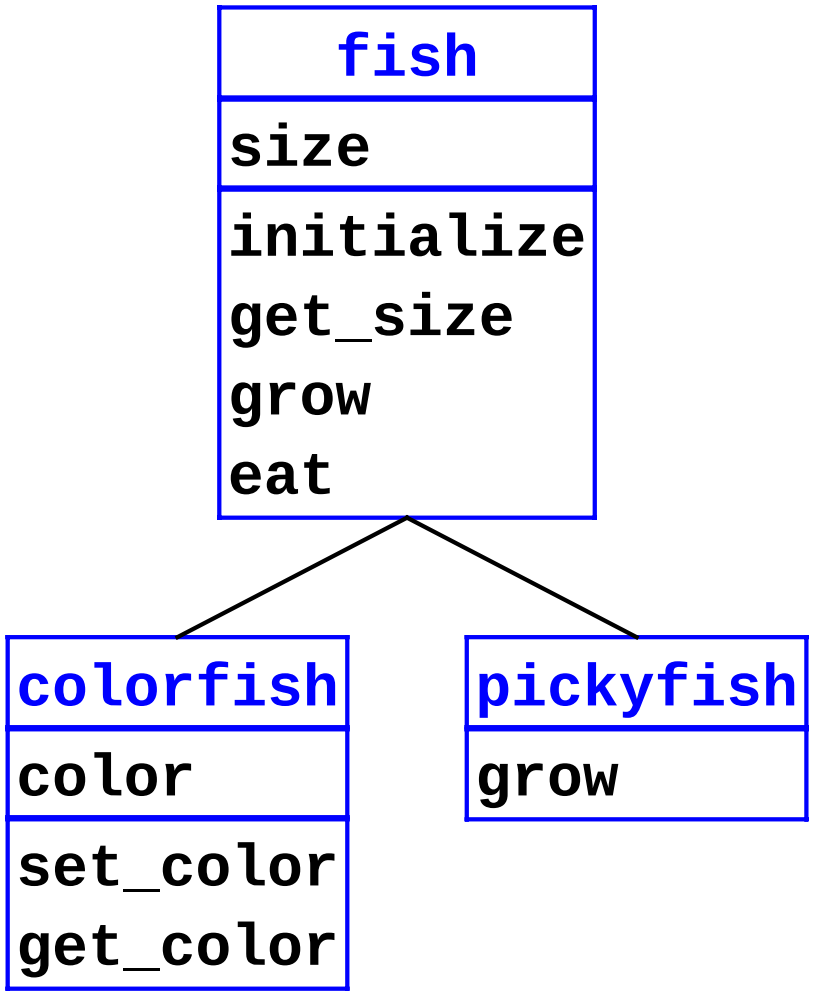
Object Representation

```
;; make-first-part : class-decl -> part
(define (make-first-part c-decl)
  (a-part
   (class-decl->class-name c-decl)
   (make-vector
    (length (class-decl->field-ids
             c-decl)))))
```



Method Search

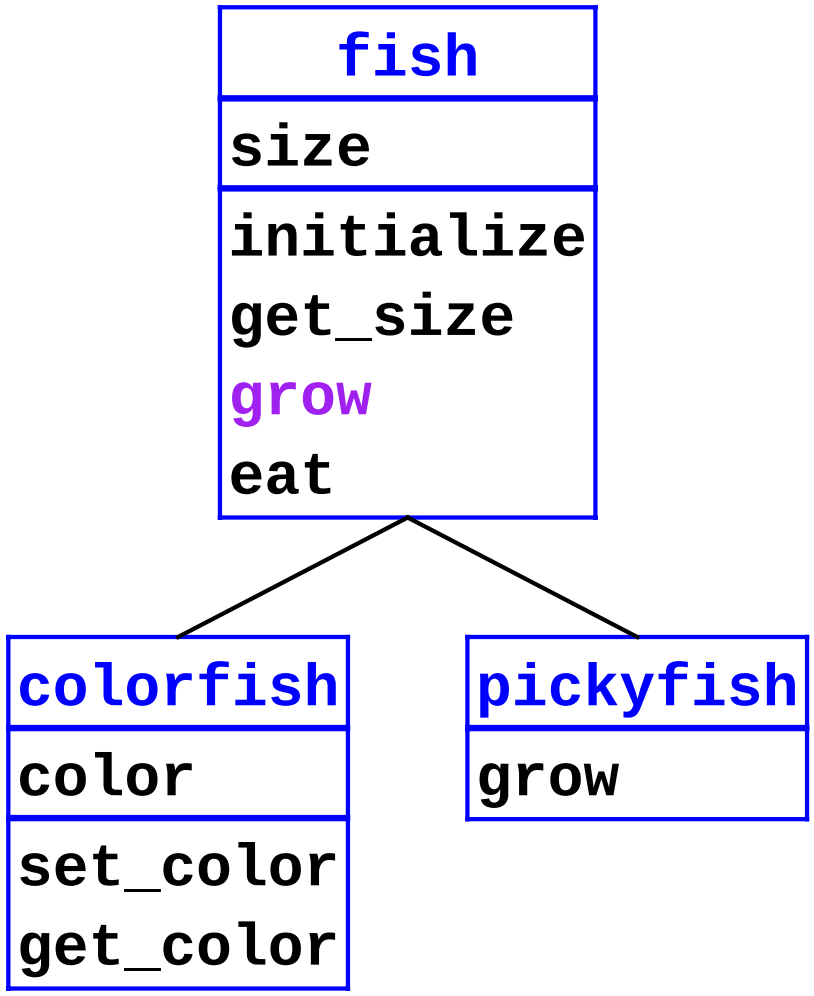
- **`get_size`** in **`colorfish`**: Check **`colorfish`**'s methods, then methods in the superclass **`fish`**, etc.

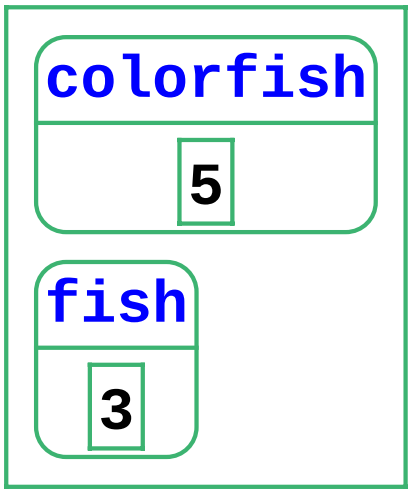


Method Search

```
(define find-method-and-apply
  (lambda (m-name host-name self args)
    (if (eqv? host-name 'object)
      (eopl:error ...) ; not found
      (let ([m-decl
             (lookup-method-decl
              m-name
              (class-name->method-decls
               host-name))])
        (if (method-decl? m-decl)
            (apply-method m-decl host-name
                          self args)
            (find-method-and-apply m-name
             (class-name->super-name
              host-name)
             self args))))))
```

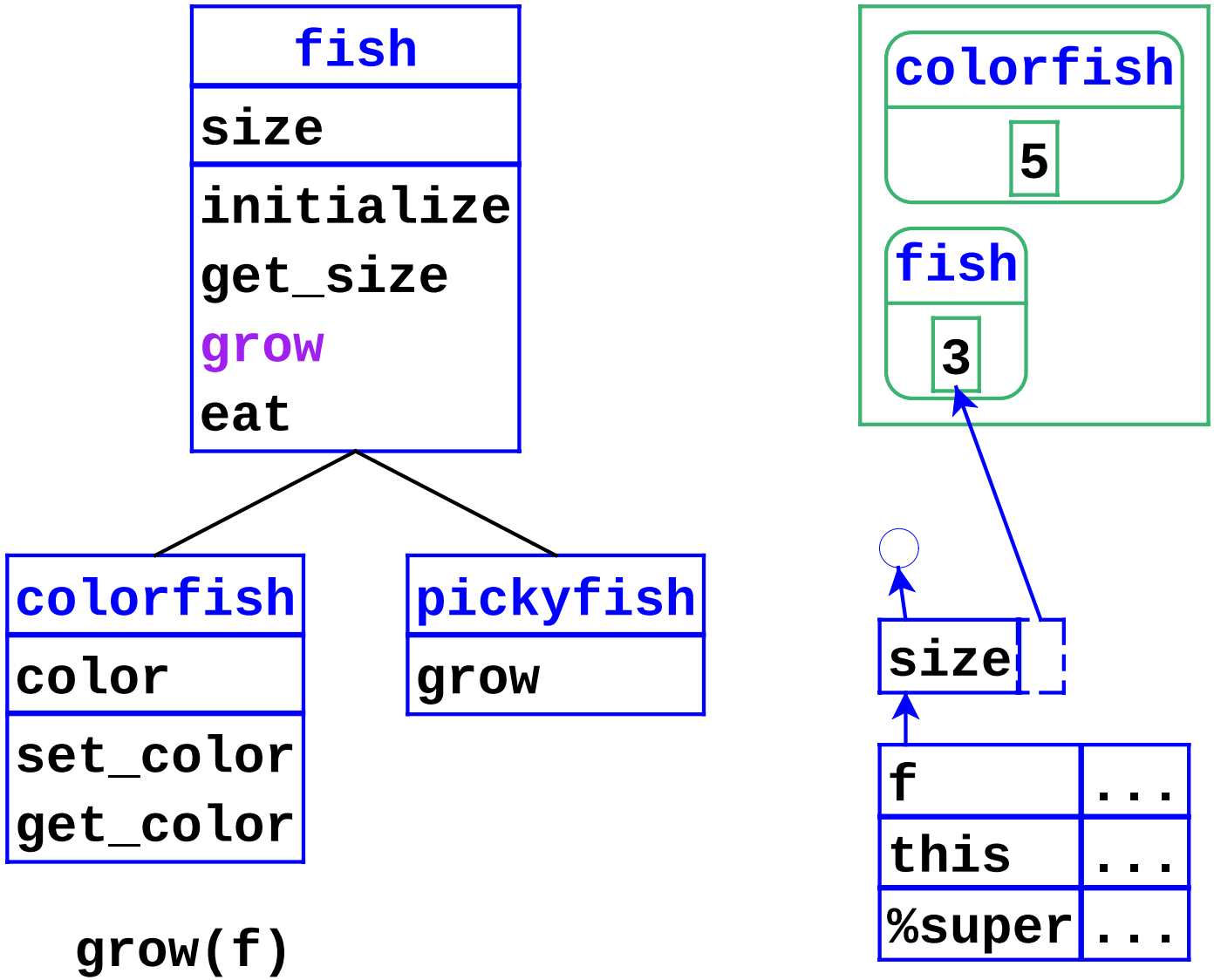
Method Application





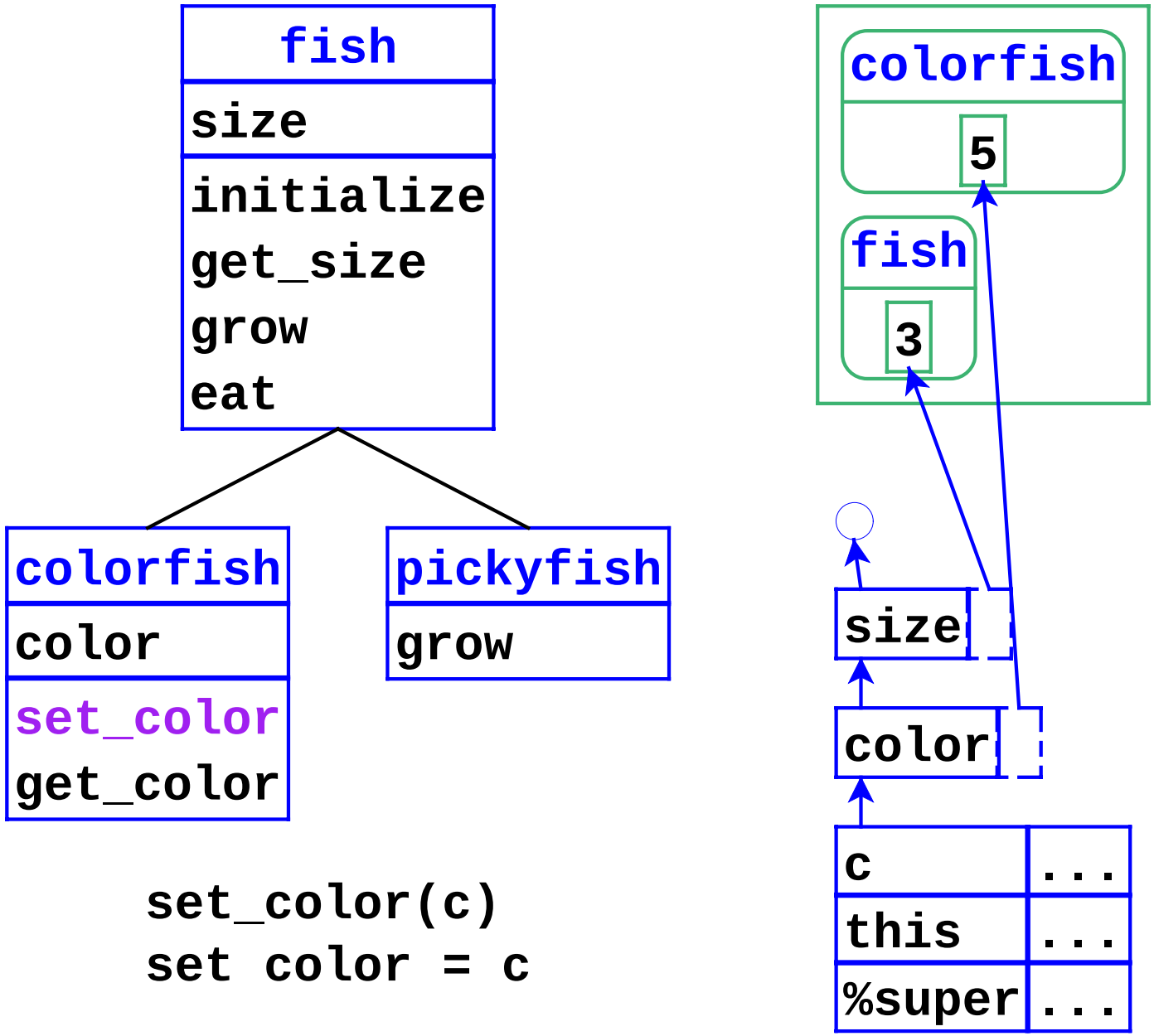

```
grow(f)
set size=+(size,f)
```

Method Application




```
grow(f)
set size=+(size,f)
```

Method Application




```
set_color(c)
set color = c
```

Method Application

```
;; apply-method : method-decl sym object
;;                lstof-expval -> expval
(define apply-method
 (lambda (m-decl host-name self args)
  (let ([ids (method-decl->ids m-decl])
        [body (method-decl->body m-decl])
        [super-name
         (class-name->super-name host-name)])
    (eval-expression
     body
     (extend-env
      (cons '%super (cons 'self ids))
      (cons super-name (cons self args))
      (build-field-env
       (view-object-as self
                       host-name)))))))
```

Method Application

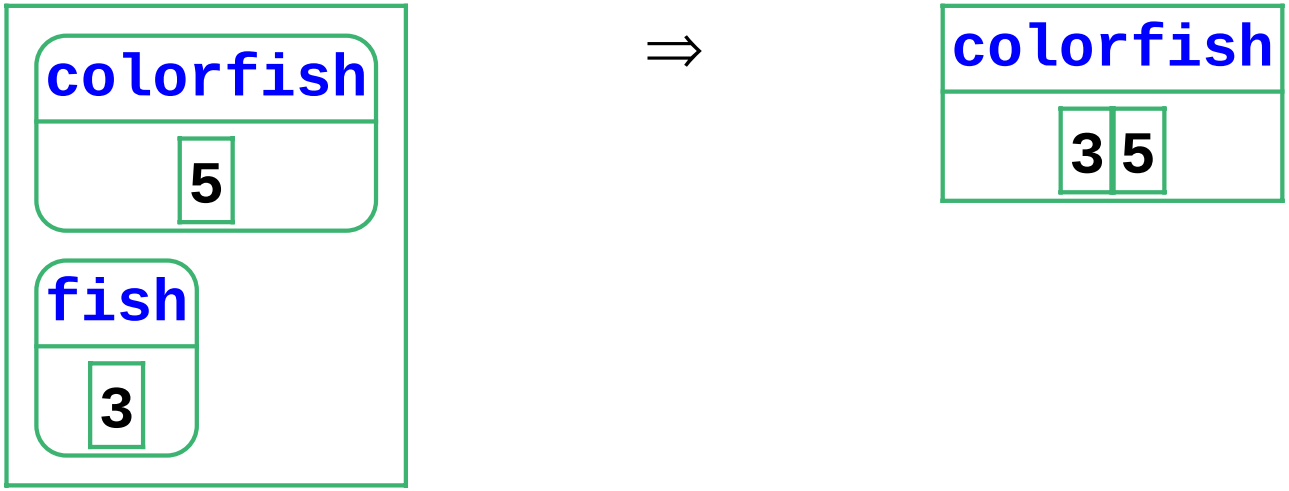
```
;; view-object-as : object sym -> lstof-parts
(define (view-object-as parts class-name)
 (if (eqv? (part->class-name (car parts))
           class-name)
    parts
    (view-object-as (cdr parts) class-name)))

;; build-field-env : lstof-parts -> env
(define (build-field-env parts)
  (if (null? parts)
      (empty-env)
      (extend-env-refs
       (part->field-ids (car parts))
       (part->fields    (car parts))
       (build-field-env (cdr parts)))))
```

Object Implementation Overview

- **Inheritance**: superclass chain for fields and methods, part chain
- **Overriding**: method dispatch uses object tag
- **Super calls**: **`%super`** hidden variable contains superclass name

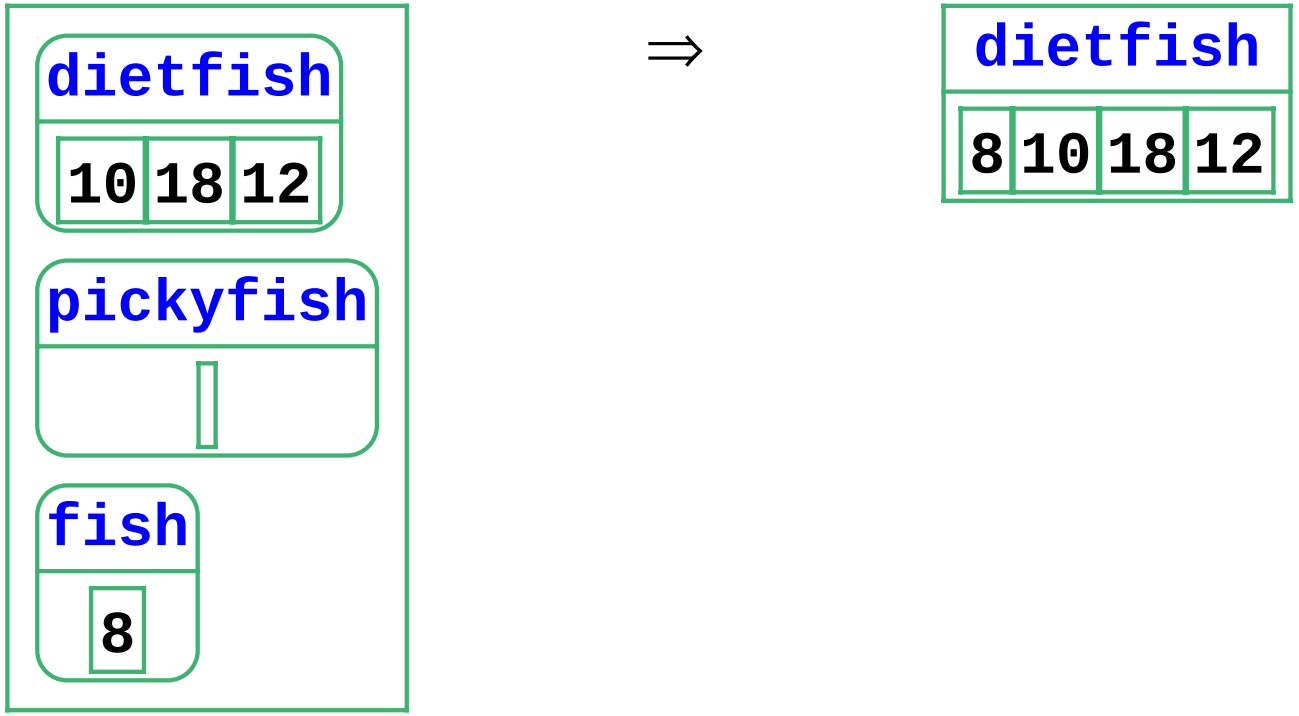
A More Realistic Object Representation

- Chain of parts wastes space
- Collapse vectors into one



A More Realistic Object Representation

- Chain of parts wastes space
- Collapse vectors into one



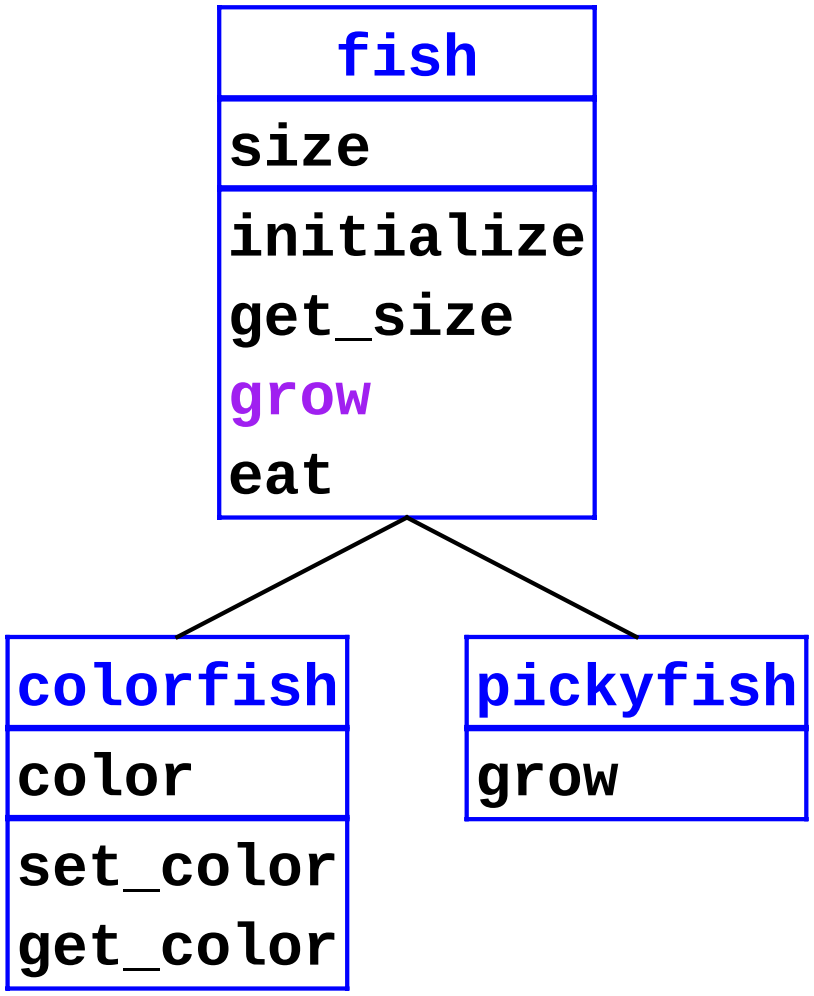
A More Realistic Object Representation

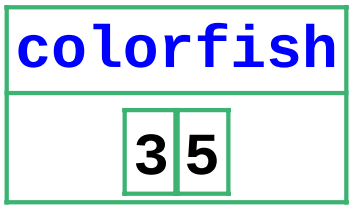
```
(define-datatype object object?
  (an-object
    (class-name symbol?)
    (fields vector?)))

;; new-object : sym -> object
(define (new-object class-name)
  (an-object
   class-name
   (make-vector
    (roll-up-field-length class-name))))
```

A More Realistic Object Representation

```
;; roll-up-field-length : sym -> num
(define roll-up-field-length
  (lambda (class-name)
    (if (eqv? class-name 'object)
        0
        (+ (roll-up-field-length
            (class-name->super-name
             class-name))
           (length
            (class-name->field-ids
             class-name))))))
```

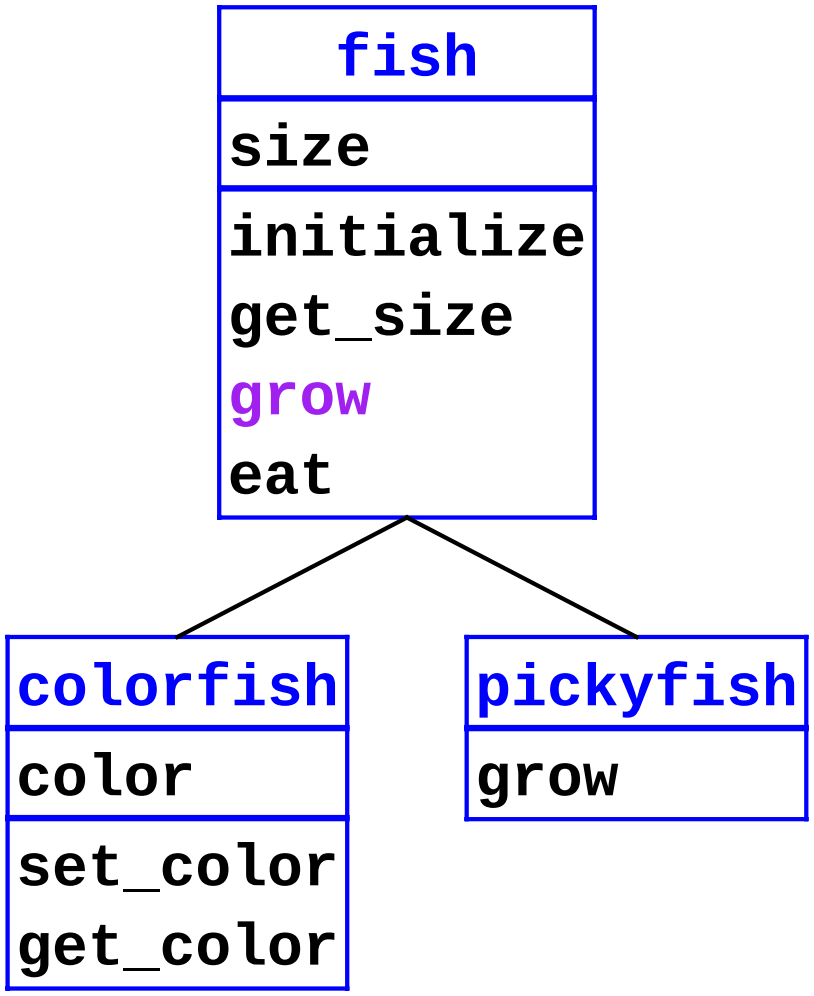
Method Application

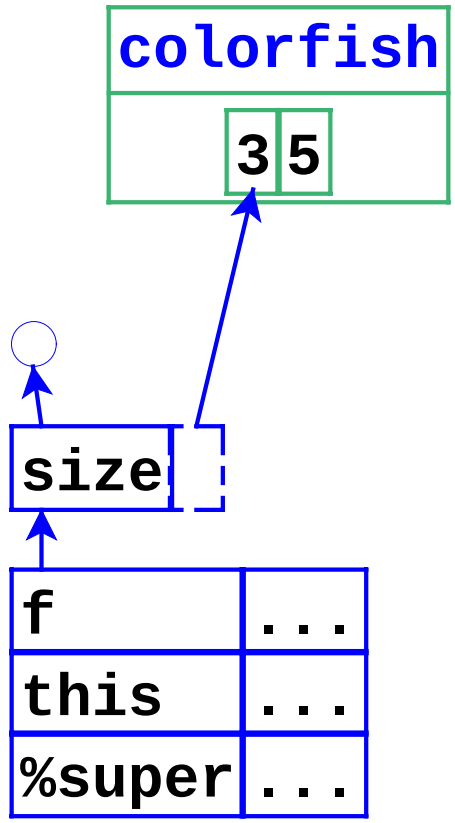





```
grow(f)
set size=+(size,f)
```

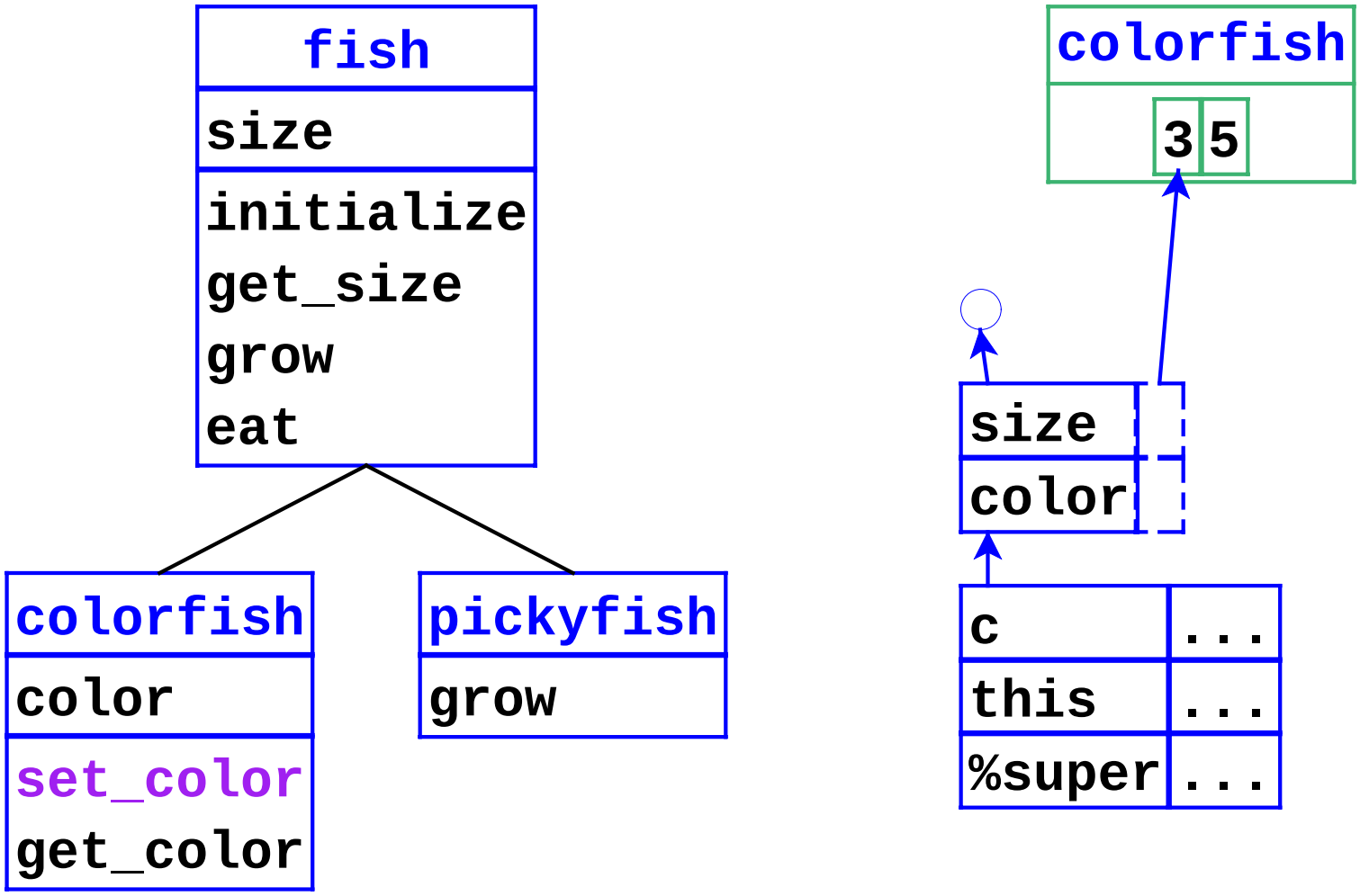
Method Application






```
grow(f)
set size=+(size,f)
```

Method Application




```
set_color(c)
set color = c
```

Method Application

```
(define apply-method
 (lambda (m-decl host-name self args)
  (let ([ids (method-decl->ids m-decl)]
        [body (method-decl->body m-decl)]
        [super-name (class-name->super-name
                     host-name)]
        [field-ids (roll-up-field-ids
                    host-name)]
        [fields (object->fields self)])
    (eval-expression
     body
     (extend-env
      (cons '%super (cons 'self ids))
      (cons super-name (cons self args))
      (extend-env-refs field-ids fields
                       (empty-env)))))))
```