From Functions to Objects

• Functional languages (Scheme, ML)
  ○ ADT is a type and a collection of functions
    
    \[
    \begin{align*}
    \text{make-fish} &: (\text{num} \rightarrow \text{fish}) \\
    \text{grow-fish} &: (\text{fish num} \rightarrow \text{fish}) \\
    \text{fish-size} &: (\text{fish} \rightarrow \text{num})
    \end{align*}
    \]

• Object-oriented languages (Java, C++, Smalltalk)
  ○ ADT is a class

  \textbf{fish class}
  \[
  \begin{align*}
  \text{method initialize} &: (\text{num} \rightarrow ) \\
  \text{method grow} &: (\text{num} \rightarrow ) \\
  \text{method size} &: (\rightarrow \text{num})
  \end{align*}
  \]
From Functions to Objects

We can implement objects with functions:

```
(define (mk-fish size)
  (letrec ([get-size (lambda () size)]
    [grow (lambda (s)
       (set! size (+ s size)))]
    [eat (lambda (fish)
       (grow ((fish 'get-size)))])]
  (lambda (msg)
    (cond
     [(eq? msg 'get-size) get-size]
     [(eq? msg 'grow) grow]
     [(eq? msg 'eat) eat]])))
```

but it's not convenient!
Elements of an OO Language

- (Expressed) values = objects

- Classes
  - superclass
  - fields
  - methods

- Expression forms
  - new
  - method call
  - super call

- Program = class declarations + expression
Syntax

<prog> ::= <class-decl>* <expr>

<class-decl> ::= class <id> extends <id> <field-decl>* <method-decl>*

'field-decl' ::= field <id>

(method-decl) ::= method <id>(<id>*())<expr>

<expr> ::= new <id>(<expr>*())
          ::= send <expr> <id>(<expr>*())
          ::= super <id>(<expr>*())
          ::= ...
Example

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

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

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))

...
class fish ...

class colorfish extends fish ...

class pickyfish extends fish ...

Class Tree

fish
size
initialize
get_size
grow
eat

colorfish
color
set_color
get_color

pickyfish
grow

Evaluation Sketch

```
<table>
<thead>
<tr>
<th>fish</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
<tr>
<td>initialize</td>
</tr>
<tr>
<td>get_size</td>
</tr>
<tr>
<td>grow</td>
</tr>
<tr>
<td>eat</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>colorfish</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
</tr>
<tr>
<td>set_color</td>
</tr>
<tr>
<td>get_color</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pickyfish</th>
</tr>
</thead>
<tbody>
<tr>
<td>grow</td>
</tr>
</tbody>
</table>

new colorfish(1)
```
Evaluation Sketch

\begin{itemize}
  \item \textbf{fish}
    \begin{itemize}
      \item size
      \item initialize
      \item get\_size
      \item grow
      \item eat
    \end{itemize}
  \item \textbf{colorfish}
    \begin{itemize}
      \item color
      \item set\_color
      \item get\_color
    \end{itemize}
  \item \textbf{pickyfish}
    \begin{itemize}
      \item grow
    \end{itemize}
\end{itemize}

new colorfish(1)

\begin{itemize}
  \item \textbf{obj} = \textbf{colorfish}
    \begin{itemize}
      \item size = 1
      \item color = 0
    \end{itemize}
\end{itemize}
Evaluation Sketch

let o1 = new colorfish(3)
   in begin
   send o1 grow(4);
   send o1 get_size()
   end
let o1 = new colorfish(3) in begin
send o1 grow(4); send o1 get_size() end

\[ o1 = \begin{array}{c}
\text{colorfish} \\
\text{size} \quad \text{color} \\
3 \quad \text{0}
\end{array} \]

grow(f)
set size=+(size,f)
Evaluation Sketch

```plaintext
let o1 = new colorfish(3)
in begin
  send o1 grow(4);
  send o1 get_size()
end

o1 = colorfish
   size = 7
   color = 0
```

```
fish
  size
  initialize
  get_size
  grow
  eat

colorfish
  color
  set_color
  get_color

pickyfish
  grow

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

```
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
```

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 =

<table>
<thead>
<tr>
<th>colorfish</th>
</tr>
</thead>
<tbody>
<tr>
<td>size = 3</td>
</tr>
<tr>
<td>color = 0</td>
</tr>
</tbody>
</table>

o2 =

<table>
<thead>
<tr>
<th>pickyfish</th>
</tr>
</thead>
<tbody>
<tr>
<td>size = 6</td>
</tr>
</tbody>
</table>
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

grow(f)
set size=+(size,f)
 Evaluation Sketch

```
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

• First, 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-clst-decl ->
```
Interpreter

• Expression form: object creation

\[
\text{(new-object-exp (class-name rands))}
\]

\[
\text{(let ((args (eval-rands rands env)))}
\]

\[
\text{(obj (new-object class-name))}
\]

\[
\text{(find-method-and-apply 'initialize class-name obj args)}
\]

\[
\text{obj})
\]

\[
\text{elaborate-class-decls! : lstof-cls-decl \rightarrow}
\]

\[
\text{new-object : sym \rightarrow object}
\]

\[
\text{find-method-and-apply : sym sym object lstof-expval \rightarrow expval}
\]
• 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

- Elaboration can just keep the declarations

```java
class fish {
    size
    initialize
    get_size
    grow
    eat
}
class colorfish extends fish {
    ... =
    color
    set_color
    get_color
}
class pickyfish extends fish {
    ... =
    grow
}```
Class Elaboration

(define the-class-env '())
(define (elaborate-class-decls! c-decls)
    (set! the-class-env c-decls))
Class Elaboration

• Finding a node in the tree:

;; 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

```
colorfish
  5
fish
  3
```
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 cholesterol

...
Object Representation

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

```
dietfish
10 18 12
```

`; An object is a list of parts

```
dietfish
10 18 12
```

...
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)))))))

<table>
<thead>
<tr>
<th>dietfish</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 18 12</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
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)))))

\textbf{dietfish}

\begin{tabular}{ccc}
10 & 18 & 12 \\
\end{tabular}
Method Search

• get_size in colorfish: Check colorfish's methods, then methods in the superclass fish, etc.

```
fish
size
initialize
get_size
grow
eat

colorfish
color
set_color
get_color

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