Object Implementation Overview

- **Inheritance**: superclass chain for fields and methods, part chain for objects
- **Overriding**: method dispatch uses object tag
- **Super calls**: `%super` hidden variable contains superclass name
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 \textit{parts}
  - from instantiated class up to base class

```java
class dietfish
    extends pickyfish
    field carbos
    field sodium
    field cholesterol
...
```

- Use part vectors in environments
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

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

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

<table>
<thead>
<tr>
<th>dietfish</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 18 12</td>
</tr>
</tbody>
</table>
(define find-method-and-apply
  (lambda (m-name host-name self args)
    (if (eq? 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

```
fish
size
initialize
get_size
grow
eat

colorfish
5

fish
3

pickyfish

color
set_color
grow

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

```
fish
size
initialize
get_size
grow
eat

colorfish

pickyfish

color

grow

set_color
get_color

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-part
(define (view-object-as parts class-name)
  (if (eq? (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)))))
Complete Implementation

(implement in DrScheme)
A More Realistic Object Representation

- A chain of parts wastes space
- Collapse vectors into one

\[
\text{colorfish} \\
5 \\
\text{fish} \\
3
\]

\[\Rightarrow\]

\[
\text{colorfish} \\
35
\]
A More Realistic Object Representation

- A 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 with Flat Objects

```
fish
size
initialize
get_size
grow
eat

colorfish
3 5

colorfish
color
set_color
get_color
pickyfish
grow

grow(f)
set size=+(size,f)
```
Method Application with Flat Objects

```
grow(f)
set size = +(size, f)
```
Method Application with Flat Objects

```
fish
- size
- initialize
- get_size
- grow
- eat

colorfish
- 35

pickyfish
- grow

set_color(c)
set color = c
```
(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 ... (cons super-name (cons self args))
          (extend-env-refs field-ids fields
            (empty-env)))))))
Complete Implementation of Flat Objects

(implement in DrScheme)
A More Realistic Class Representation

Eliminate tree walks: object creation, method calls

```
fish
  size
  initialize
  get_size
  grow
  eat

colorfish
  color
  set_color
  get_color

pickyfish
  grow
```
Object Creation without Tree Walks

Current interpreter:

1. Find class
2. Get field list (walk tree)
3. Allocate field array and object

To eliminate tree walks:

2. Extract flat field list from class
Method Calls without Tree Walks

After object and arguments are determined:

1. Lookup object class
2. Find class containing method (walk tree)
3. Get field variables for class (walk tree)
4. Create environment: fields + %super + self + args
5. Evaluate method body

To eliminate tree walks:

2 & 3. Find method in current class, extract field list and superclass name
Class Elaboration

- fish
  - size
  - initialize
  - get_size
  - grow
  - eat

- colorfish
  - color
  - set_color
  - get_color

- pickyfish
  - grow
Class Elaboration

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

<table>
<thead>
<tr>
<th>colorfish</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
<tr>
<td>color</td>
</tr>
<tr>
<td>initialize, {size}, object</td>
</tr>
<tr>
<td>get_size, {size}, object</td>
</tr>
<tr>
<td>grow, {size}, object</td>
</tr>
<tr>
<td>eat, {size}, object</td>
</tr>
<tr>
<td>set_color, {size color}, fish</td>
</tr>
<tr>
<td>get_color, {size color}, fish</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pickyfish</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
<tr>
<td>initialize, {size}, object</td>
</tr>
<tr>
<td>get_size, {size}, object</td>
</tr>
<tr>
<td>grow, {size}, fish</td>
</tr>
<tr>
<td>eat, {size}, object</td>
</tr>
</tbody>
</table>
```

33
## Class Elaboration

<table>
<thead>
<tr>
<th>fish</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
</tr>
<tr>
<td>initialize, {size}, object</td>
<td></td>
</tr>
<tr>
<td>get_size, {size}, object</td>
<td></td>
</tr>
<tr>
<td>grow, {size}, object</td>
<td></td>
</tr>
<tr>
<td>eat, {size}, object</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>colorfish</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td></td>
</tr>
<tr>
<td>initialize, {size}, object</td>
<td></td>
</tr>
<tr>
<td>get_size, {size}, object</td>
<td></td>
</tr>
<tr>
<td>grow, {size}, object</td>
<td></td>
</tr>
<tr>
<td>eat, {size}, object</td>
<td></td>
</tr>
<tr>
<td>set_color, {size color}, fish</td>
<td></td>
</tr>
<tr>
<td>get_color, {size color}, fish</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pickyfish</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
</tr>
<tr>
<td>initialize, {size}, object</td>
<td></td>
</tr>
<tr>
<td>get_size, {size}, object</td>
<td></td>
</tr>
<tr>
<td>grow, {size}, fish</td>
<td></td>
</tr>
<tr>
<td>eat, {size}, object</td>
<td></td>
</tr>
</tbody>
</table>

new colorfish(3)
# Class Elaboration

## fish

- **size**
- **initialize**, {size}, object
- **get_size**, {size}, object
- **grow**, {size}, object
- **eat**, {size}, object

## colorfish

- **size**
- **color**
- **initialize**, {size}, object
- **get_size**, {size}, object
- **grow**, {size}, object
- **eat**, {size}, object
- **set_color**, {size color}, fish
- **get_color**, {size color}, fish

## pickyfish

- **size**
- **initialize**, {size}, object
- **get_size**, {size}, object
- **grow**, {size}, fish
- **eat**, {size}, object
- **send cf get_size()**
Implementation

• Change \texttt{elaborate-class-decls!} to build annotated tree
• Change \texttt{new-object} to use class's immediate field list
• Change \texttt{apply-method} to work with annotated methods

(implement in DrScheme)