

## The Language (OOELL)

*Use all of ELL but add Java-like OO constructs:*

```
<program> ::= { <class-decl> }* <expression>
```

```
<class-decl> ::= class <identifier> extends <identifier>
                { field <identifier> }*
                { <method-decl> }*
```

```
<method-decl> ::= method <identifier> ( {<identifier>}*(',') )
                <expression>
```

```
<expression> ::= new <identifier> ( {<expression>}*(',') )
```

```
<expression> ::= send <expression> <identifier>
                ( {<expression>}*(',') )
```

```
<expression> ::= super <identifier> ( {<expression>}*(',') )
```

**Implementation: Program (will stay the same for all implementations)**

To evaluate a program:

- Process the class declarations
- Evaluate the expression

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

Specification for `elaborate-class-decls`:

- Build an appropriate structure for each class
  - Store the field names (why not values like in an environment?)
  - Build an appropriate structure for each method
- Build a global *class environment* out of the class declarations

**Implementation: New Expressions (will stay the same for all implementations)***New Objects*

```
(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)))
```

*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)))
```

*Super Method 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)))
```

## A Simple Implementation

*Storing classes:*

Just store the raw ASTs for each class declaration

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

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

*Helper—looking up a class in the class environment:*

```
(define lookup-class
  (lambda (name)
    (let loop ((env the-class-env))
      (cond
        ((null? env)
         (eopl:error 'lookup-class
          "Unknown class ~s" name))
        ((eqv? (class-decl->class-name (car env)) name) (car env))
        (else (loop (cdr env)))))))
```

## A Simple Implementation (cont'd)

*Storing objects:*

Store a list of “parts” for each class in the inheritance hierarchy.

Each “part” stores the values of the fields—why not the names as with environments?

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

## A Simple Implementation (cont'd)

*Building a new object:*

Make a part for this class, then recursively create an object (list) for the parent's class

```
(define new-object
  (lambda (class-name)
    (if (eqv? class-name 'object)
        '()
        (let ((c-decl (lookup-class class-name)))
          (cons
            (make-first-part c-decl)
            (new-object (class-decl->super-name c-decl)))))))

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

## A Simple Implementation (cont'd)

*Helper functions for looking up pieces from a class declaration's AST:*

<code>class-decl-&gt;class-name</code>	returns the class's name
<code>class-decl-&gt;field-ids</code>	returns a list of the class's fields
<code>class-decl-&gt;super-name</code>	returns the class's parent's name
<code>class-decl-&gt;method-decls</code>	returns a list of the class's method declarations
<code>method-decl-&gt;method-name</code>	returns a method's name
<code>method-decl-&gt;ids</code>	returns a method's formal parameters
<code>method-decl-&gt;body</code>	returns a method's body
<code>method-decls-&gt;method-names</code>	returns a list of all the method name

## A Simple Implementation (cont'd)

*Some other helpers—some of these are compositions of other accessor helpers*

<code>part-&gt;class-name</code>	returns a part's class name
<code>part-&gt;fields</code>	returns a part's fields
<code>part-&gt;field-ids</code>	returns a part's field names (gets the class first, then the fields)
<code>part-&gt;class-decl</code>	returns a part's corresponding class declaration (look up name)
<code>part-&gt;method-decls</code>	returns a part's method declarations (looks up the class name)
<code>part-&gt;super-name</code>	returns a part's class's parent's name
<code>class-name-&gt;method-decls</code>	returns a class name's method declarations (look up name)
<code>class-name-&gt;super-name</code>	returns a class name's parent's name
<code>object-&gt;class-name</code>	returns an objects class name

## A Simple Implementation (cont'd)

*Finding and applying methods:*

If the host-name is the object class, stop: it has no methods.

Otherwise,

- Look up the host name
- Get the method declarations
- Look up the target method (m-name)
- If found, apply it
- If not found, recursively apply using host class's parent

```
(define find-method-and-apply
  (lambda (m-name host-name self args)
    (if (eqv? host-name 'object)
        (eopl:error 'find-method-and-apply
                    "No method for name ~s" m-name)
        (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))))))
```

## A Simple Implementation (cont'd)

*Applying a method (once found):*

Evaluate just like a function call

- Build a new environment
  - Build an environment out of the “parts” (class fields) *as seen from the host class* (how would you change this to do dynamic inheritance of variables?)
  - Bind formals to actuals
  - Also bind “self” to the current object
  - Bind name of parent class to special variable `'%super`
- Evaluate the body of the method in that environment

```
(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)))))))
```

## A Simple Implementation (cont'd)

*Building an environment out of the fields:*

- Get field names (part -> name -> lookup to get class decl -> field ids)
- Get field values (stored in the object's "part")
- Bind these and extend the environment you get from recursing on the parent classes

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

*Building an object "view" as seen from a particular class:*

Search through the "parts" until you get to the one for the class you're looking for

```
(define view-object-as
  (lambda (parts class-name)
    (if (eqv? (part->class-name (car parts)) class-name)
        parts
        (view-object-as (cdr parts) class-name))))
```