# COSE212: Programming Languages

# Lecture 11 — Design and Implementation of PLs
# (7) Classes and Objects

Hakjoo Oh
2026 Fall

# Review: Implicit-Refs

Syntax

$$
\begin{array}{rcl}
P & \to & E \\
E & \to & n \mid x \\
 & \mid & E + E \mid E - E \\
 & \mid & \texttt{iszero}\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
 & \mid & \texttt{let}\ x = E\ \texttt{in}\ E \\
 & \mid & \texttt{proc}\ x\ E \mid E\ E \\
 & \mid & x := E \\
 & \mid & E; E
\end{array}
$$

- Every variable denotes a (mutable) memory cell.
- $x := E$ updates the cell that $x$ refers to.
- $E_1; E_2$ sequences effects: evaluate $E_1$, then $E_2$.

# Review: Implicit-Refs

Semantic Domain

$$
\begin{aligned}
\boldsymbol{Val} &= \mathbb{Z} + \boldsymbol{Bool} + \boldsymbol{Procedure} \\
\boldsymbol{Procedure} &= \boldsymbol{Id} \times \boldsymbol{E} \times \boldsymbol{Env} \\
\boldsymbol{\rho} \in \boldsymbol{Env} &= \boldsymbol{Id} \to \boldsymbol{Loc} \\
\boldsymbol{\sigma} \in \boldsymbol{Mem} &= \boldsymbol{Loc} \to \boldsymbol{Val}
\end{aligned}
$$

Evaluation Rule

$$\boldsymbol{\rho}, \boldsymbol{\sigma} \vdash \boldsymbol{E} \Rightarrow \boldsymbol{v}, \boldsymbol{\sigma}'$$

- Environments map variables to *locations*, not values.
- Each evaluation step threads the memory: the result is a value *and* the updated memory after side-effects.

# Review: Implicit-Refs

$$\frac{}{\rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\rho, \sigma \vdash x \Rightarrow \sigma(\rho(x)), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 + E_2 \Rightarrow n_1 + n_2, \sigma_2} \qquad \frac{\rho, \sigma_0 \vdash E \Rightarrow 0, \sigma_1}{\rho, \sigma_0 \vdash \texttt{iszero}\ E \Rightarrow \mathit{true}, \sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \mathit{true}, \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v, \sigma_2}$$

$$\frac{}{\rho, \sigma \vdash \texttt{proc}\ x\ E \Rightarrow (x, E, \rho), \sigma} \qquad \frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash x := E \Rightarrow v, [\rho(x) \mapsto v]\sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \quad [x \mapsto l]\rho, [l \mapsto v_1]\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_2}\ l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{\begin{array}{c}\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \\ [x \mapsto l]\rho', [l \mapsto v]\sigma_2 \vdash E \Rightarrow v', \sigma_3\end{array}}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \mathrm{Dom}(\sigma_2)$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

# Object-Oriented Programming: Motivation

We often want to bundle a piece of state together with the operations that may touch it, e.g., a counter you can both *incr* and *get*, and nothing else:

```
let counter = 0
in let incr = proc (x) (counter := counter + 1; counter)
   in let get  = proc (x) counter
      in ...
```

- This bundles state (`counter`) and operations (`incr`, `get`) by hand.
- Object-oriented programming makes this bundling a *language feature*.
- Today: design a small object-oriented language called CLASSES (an extension of IMPLICIT-REFS).

# Object-Oriented Programming: Key Concepts

- An *object* bundles
  - *fields* (a.k.a. instance variables): stored state,
  - *methods*: procedures that have access to the fields.
- Calling a method is sometimes viewed as *sending a message* to the object: *message passing*.
- A *class* specifies the fields and methods of all its objects. Each object is an *instance* of its class.
- *Inheritance*: a new class can be defined as a small modification of an existing one by adding fields, adding methods, or overriding methods.

# Example 1: A Counter Class

```
class Counter extends object {
  field count
  method initialize () { count := 0 }
  method incr ()       { count := count + 1 }
  method get ()        { count }
}
in let c = new Counter() in
   begin c.incr();          % count = 1
         c.incr();          % count = 2
         c.get()            % returns 2
   end
```

- One object bundles a piece of state (`count`) with the methods that may touch it (`incr`, `get`).
- `new Counter()` creates the object and *automatically* runs `initialize()`, so `count` starts at **0**.

# Example 2: Dynamic Dispatch

```
class interior-node extends object {
  field left   field right
  method initialize (l, r) { left := l; right := r }
  method sum () { left.sum() + right.sum() }
}
class leaf-node extends object {
  field value
  method initialize (v) { value := v }
  method sum () { value }
}
in let o1 = new interior-node(
              new interior-node(new leaf-node(3), new leaf-node(4)),
              new leaf-node(5))
   in o1.sum()
```

- The same call `left.sum()` works whether `left` is an `interior-node` or a `leaf-node`; the method is picked at runtime by the receiver's class.
  - In `o1.sum()`: `left.sum()` runs `interior-node`'s `sum` (`o1.left` is interior).
  - Recursing: `left.sum()` runs `leaf-node`'s `sum` (`o1.left.left` is a leaf).

# Example 3: Accessing the Receiver Object via self

```
class oddeven extends object {
  method initialize () { 1 }
  method even (n) {
    if iszero(n) then 1 else self.odd(n - 1)
  }
  method odd (n) {
    if iszero(n) then 0 else self.even(n - 1)
  }
}
in let o1 = new oddeven()
   in o1.odd(13)
```

- Inside a method body, `self` refers to the current receiver object, so `self.X(...)` lets a method invoke another method on the same object.
- As a side benefit, mutual recursion comes for free here: `even` and `odd` call each other through `self` with no `letrec`.

# Inheritance

If $c_2$ extends $c_1$, we say $c_1$ is the *parent* or *superclass* of $c_2$, and $c_2$ is a *child* or *subclass* of $c_1$.

- Every class except a predefined root class `object` has exactly one parent: *single inheritance*.
- The `extends` relation imposes a *tree* on the set of classes, rooted at `object`.
- A child inherits all fields and methods of its parent unless they are redeclared.
- Therefore any instance of a child can be used wherever an instance of the parent is expected (*subclass polymorphism*).

# Example 4: ColorPoint

```
class point extends object {
  field x   field y
  method initialize (initx, inity) { x := initx; y := inity }
  method move (dx, dy)             { x := x + dx; y := y + dy }
  method getX ()  { x }     method getY ()  { y }
}
class colorpoint extends point {
  field color
  method set-color (c) { color := c }
  method get-color ()  { color }
}
in let cp = new colorpoint(10, 20)
   in begin cp.set-color(87);
            cp.move(3, 4);
            cp.getX();              % 13 (inherited move + getX)
            cp.get-color()          % returns 87
      end
```

- `colorpoint` inherits all five members from `point` and adds `color`, `set-color`, `get-color`.
- Subclass polymorphism: a `colorpoint` can be used wherever a `point` can (code calling `p.move(...)` or `p.getX()` works on either).

# Example 5: Field Shadowing

```
class c1 extends object {
  field x   field y
  method initialize () { 1 }
  method setx1 (v) { x := v }   method getx1 () { x }
  method sety1 (v) { y := v }   method gety1 () { y }
}
class c2 extends c1 {
  field y                        % shadows c1's y
  method sety2 (v) { y := v }    method getx2 () { x }
  method gety2 () { y }
}
in let o2 = new c2() in
   begin o2.setx1(101); o2.sety1(102); o2.sety2(999);
         o2.getx1();   % 101
         o2.gety1();   % 102 (c1's y, set by sety1)
         o2.getx2();   % 101 (c1's x; c2 didn't redeclare)
         o2.gety2()    % 999 (c2's y, shadowed)
   end
```

- A method resolves field names against the fields of its *host class* (where it is declared), not the receiver's class.
- So `sety1` writes c1's `y`; `sety2` writes c2's `y`; `getx1` and `getx2` both see the same (inherited) `x`.

# Example 6: Method Overriding

```
class c1 extends object {
  method initialize () { 1 }
  method m1 () { 11 }
  method m2 () { self.m1() }
}
class c2 extends c1 {
  method m1 () { 22 }                    % overrides c1's m1
}
in let o1 = new c1() in
   let o2 = new c2() in
   begin o1.m1();       % 11 (c1's m1)
         o2.m1();       % 22 (c2's m1 overrides)
         o2.m2()        % 22, not 11
   end
```

- c2's `m1` *overrides* c1's: `o1.m1()`=11, `o2.m1()`=22.
- `o2.m2()`=22, not 11: `m2` is inherited from `c1`, but its `self.m1()` still dispatches on c2, reaching the override.

# Example 7: super calls

`colorpoint` wants its own `initialize`. Naive (duplicates parent):

```
method initialize (initx, inity, initcolor) {
  x := initx; y := inity; color := initcolor
}
```

Better, using a *super call*:

```
method initialize (initx, inity, initcolor) {
  super.initialize(initx, inity);
  color := initcolor
}
```

`super.initialize(...)` called from `colorpoint.initialize` runs `point.initialize`. In general: a super call invokes a method in the parent of the method's host class, not in the parent of `self`'s class. (See the next example)

# Example 8: super vs. self

```
class c1 extends object {
  method initialize () { 1 }
  method m1 () { self.m2() }
  method m2 () { 13 }
}
class c2 extends c1 {
  method m1 () { 22 }   method m2 () { 23 }
  method m3 () { super.m1() }
}
class c3 extends c2 {
  method m1 () { 32 }   method m2 () { 33 }
}
in let o3 = new c3() in o3.m3()
```

- o3.m3() runs c2's m3, body super.m1().
- Inside c2.m3, super.m1() dispatches *statically*: m3's host class is c2, so super looks in c2's parent (c1) and runs c1's m1.
  - Not in the parent of self's class (c3's parent = c2): that would call c2.m1 = 22.
- Inside c1.m1, self.m2() dispatches *dynamically*: self is still the c3 object, so c3's m2 runs, returning 33.

# Syntax of Classes

| | | | |
|---|---|---|---|
| $P$ | $\rightarrow$ | $D^*\ E$ | program |
| $D$ | $\rightarrow$ | $\texttt{class}\ C\ \texttt{extends}\ C'\ \{F^*\ M^*\}$ | class declaration |
| $F$ | $\rightarrow$ | $\texttt{field}\ x$ | field declaration |
| $M$ | $\rightarrow$ | $\texttt{method}\ m(x)\ E$ | method declaration |
| $E$ | $\rightarrow$ | $n \mid x \mid E + E \mid E - E$ | |
| | $\mid$ | $\texttt{iszero}\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E$ | |
| | $\mid$ | $\texttt{let}\ x = E\ \texttt{in}\ E$ | |
| | $\mid$ | $\texttt{proc}\ x\ E \mid E\ E$ | |
| | $\mid$ | $x := E \mid E; E$ | |
| | $\mid$ | $\texttt{new}\ C(E)$ | object creation |
| | $\mid$ | $E.m(E)$ | method call |
| | $\mid$ | $\texttt{self}$ | current object |
| | $\mid$ | $\texttt{super}.m(E)$ | super call |

# Semantic Domain: Objects

An object carries its class name and a mapping from fields to locations (so fields can be mutated):

$$\boldsymbol{Obj} = \boldsymbol{Class} \times (\boldsymbol{Id} \to \boldsymbol{Loc})$$

We write an object as $(C, \rho_f)$ where $\rho_f = [f_1 \mapsto l_1, \ldots, f_n \mapsto l_n]$.

Objects are now values:

$$\begin{aligned} \boldsymbol{Val} &= \mathbb{Z} + \boldsymbol{Bool} + \boldsymbol{Procedure} + \boldsymbol{Obj} \\ \boldsymbol{\rho} \in \boldsymbol{Env} &= \boldsymbol{Id} \to \boldsymbol{Loc} \\ \boldsymbol{\sigma} \in \boldsymbol{Mem} &= \boldsymbol{Loc} \to \boldsymbol{Val} \end{aligned}$$

# Semantic Domain: Class Environment

Before running the program, all class declarations are processed into a global *class environment* $\zeta$:

$$\zeta \in \boldsymbol{ClassEnv} = \boldsymbol{Class} \to \boldsymbol{ClassDecl}$$

$$\boldsymbol{ClassDecl} = \boldsymbol{Class} \times \boldsymbol{Id}^* \times (\boldsymbol{Id} \to \boldsymbol{Method})$$

i.e. each class name maps to (parent, field names, method table).

A method definition remembers the class in which it was *declared*:

$$\boldsymbol{Method} = \boldsymbol{Id} \times \boldsymbol{E} \times \boldsymbol{Class}$$

- $\boldsymbol{Id}$: the formal parameter
- $\boldsymbol{E}$: the method body
- $\boldsymbol{Class}$: the *host class* where the method is declared

# Building the Class Environment

Given a program $D_1\ D_2\ \ldots\ D_n\ \ E$, build $\zeta$ once before evaluating $E$. Start with only the predefined root `object`:

$$\zeta_0 = [\texttt{object} \mapsto (\bot,\ (),\ \emptyset)]$$

For each $D_i = \texttt{class}\ C_i\ \texttt{extends}\ C_i'\ \{\vec{f}_i\ \vec{M}_i\}$, add an entry:

$$\zeta_i = \zeta_{i-1}[C_i \mapsto (C_i',\ \vec{f}_i,\ \mu_i)]$$

where the method table records $C_i$ as each method's host class:

$$\mu_i = [\, m \mapsto (x, E_b, C_i)\ \mid\ (\texttt{method}\ m(x)\ E_b) \in \vec{M}_i\,]$$

The final environment is $\zeta = \zeta_n$. Inheritance is not flattened here; `lookup` and `fields` resolve it later by walking the chain.

**Field shadowing**: as part of this construction, each field is $\alpha$-renamed to a globally unique identifier; method bodies are renamed to match. E.g., for `c2 extends c1` with both declaring `y`: `c1`'s `y` becomes `y_c1` (and `c1`'s methods rewrite `y` to `y_c1`), `c2`'s `y` becomes `y_c2`.

# Method Lookup

Given a class $C$ and a method name $m$,

$$\textbf{lookup}(\zeta, C, m) = \begin{cases} \mu(m) & \text{if } m \in \text{Dom}(\mu) \\ \textbf{lookup}(\zeta, C', m) & \text{if } m \notin \text{Dom}(\mu) \\ \text{error} & \text{if } C = \texttt{object} \end{cases}$$

where $\zeta(C) = (C', \_, \mu)$.

- When $m \in \text{Dom}(\mu)$: $C$ itself declares (or overrides) the method, so we return $\mu(m)$ immediately. This is where a subclass's *override* takes effect: the receiver's own method table is consulted first.
- When $m \notin \text{Dom}(\mu)$: $C$ has no entry for $m$, so we walk up the class chain to $C$'s parent $C'$ and continue looking. The method is *inherited* from the nearest ancestor that defines it.
- When $C = \texttt{object}$ (and still no $m$): we have reached the root of the class tree without finding $m$. The method name is undefined anywhere on the chain, so we report an error (method-not-found).

# Field Lookup

Walk up the class chain and collect all fields (including inherited):

$$\textbf{fields}(\zeta, C) = \begin{cases} () & \text{if } C = \texttt{object} \\ \textbf{fields}(\zeta, C') +\!\!+ (f_1, \ldots, f_n) & \text{otherwise} \end{cases}$$

where $\zeta(C) = (C', (f_1, \ldots, f_n), \_)$.

- $+\!\!+$ denotes sequence concatenation.
- Parent's fields come first, then $C$'s locally-declared fields (matching the layout of an object of class $C$).
- Recursion bottoms out at `object`, which has no fields.

# Evaluation Rule

With the class environment $\zeta$ available globally, the evaluation rule becomes

$$\zeta, \rho, \sigma \vdash E \Rightarrow v, \sigma'$$

- $\zeta$ is built once from the program's class declarations and does not change during evaluation.
- Existing rules of IMPLICIT-REFS simply thread $\zeta$ unchanged (next slide).
- When a method body runs, the environment $\rho$ is extended with two pseudo-variables: `self` (the receiver object) and `host` (the method's host class, needed by `super`).
- Four new rules: `new`, method call, `self`, `super`.

# Semantic Rules: Inherited Constructs

The IMPLICIT-REFS rules are unchanged except they now carry $\zeta$:

$$\frac{}{\zeta,\ \rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\zeta,\ \rho, \sigma \vdash x \Rightarrow \sigma(\rho(x)), \sigma}$$

$$\frac{\zeta,\ \rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \quad \zeta,\ \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\zeta,\ \rho, \sigma_0 \vdash E_1 + E_2 \Rightarrow n_1 + n_2, \sigma_2} \qquad \frac{\zeta,\ \rho, \sigma_0 \vdash E \Rightarrow 0, \sigma_1}{\zeta,\ \rho, \sigma_0 \vdash \texttt{iszero}\ E \Rightarrow \mathit{true}, \sigma_1}$$

$$\frac{\zeta,\ \rho, \sigma_0 \vdash E_1 \Rightarrow \mathit{true}, \sigma_1 \quad \zeta,\ \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\zeta,\ \rho, \sigma_0 \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v, \sigma_2}$$

$$\frac{}{\zeta,\ \rho, \sigma \vdash \texttt{proc}\ x\ E \Rightarrow (x, E, \rho), \sigma} \qquad \frac{\zeta,\ \rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\zeta,\ \rho, \sigma_0 \vdash x := E \Rightarrow v, [\rho(x) \mapsto v]\sigma_1}$$

$$\frac{\zeta,\ \rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \zeta,\ [x \mapsto l]\rho, [l \mapsto v_1]\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\zeta,\ \rho, \sigma_0 \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_2}\ l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{\begin{array}{c}\zeta,\ \rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \quad \zeta,\ \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \\ \zeta,\ [x \mapsto l]\rho', [l \mapsto v]\sigma_2 \vdash E \Rightarrow v', \sigma_3\end{array}}{\zeta,\ \rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \mathrm{Dom}(\sigma_2)$$

$$\frac{\zeta,\ \rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \zeta,\ \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\zeta,\ \rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

# Rule: Object Creation

$$\frac{\begin{array}{c}\textbf{fields}(\zeta, C) = (f_1, \ldots, f_k), \quad \rho_f = [f_1 \mapsto l_1, \ldots, f_k \mapsto l_k] \\ \zeta, \rho, \sigma \vdash E \Rightarrow v, \sigma', \quad l_1, \ldots, l_k, l' \notin \mathrm{Dom}(\sigma') \\ \textbf{lookup}(\zeta, C, \texttt{initialize}) = (x, E_b, C_h) \\ \rho_b = \rho_f[x \mapsto l', \texttt{self} \mapsto (C, \rho_f), \texttt{host} \mapsto C_h] \\ \zeta, \rho_b, \sigma'[l' \mapsto v] \vdash E_b \Rightarrow \_, \sigma_b\end{array}}{\zeta, \rho, \sigma \vdash \texttt{new}\ C(E) \Rightarrow (C, \rho_f), \sigma_b}$$

1. Collect all fields of $C$: $(f_1, \ldots, f_k)$.
2. Allocate fresh locations and form $\rho_f = [f_i \mapsto l_i]$
3. Evaluate the argument $E$ to value $v$ (memory becomes $\sigma'$).
4. Look up `initialize`, recording its host class $C_h$.
5. Build the body environment $\rho_b$ (in which $E_b$ runs): fields via $\rho_f$, parameter $x$ at fresh $l' \mapsto v$, `self` at the new object, `host` at the host class.
6. Run the body $E_b$; the value is discarded.
7. Return $(C, \rho_f)$ and the final memory $\sigma_b$.

# Rule: Method Call ($E_0.m(E_1)$)

$$\frac{\begin{array}{c}\zeta, \rho, \sigma \vdash E_0 \Rightarrow (C, \rho_f),\ \sigma_1 \\ \zeta, \rho, \sigma_1 \vdash E_1 \Rightarrow v_1,\ \sigma_2, \quad l \notin \mathrm{Dom}(\sigma_2) \\ \textbf{lookup}(\zeta, C, m) = (x, E_b, C_h) \\ \rho_b = \rho_f[x \mapsto l,\ \texttt{self} \mapsto (C, \rho_f),\ \texttt{host} \mapsto C_h] \\ \zeta,\ \rho_b,\ \sigma_2[l \mapsto v_1] \vdash E_b \Rightarrow v, \sigma_3\end{array}}{\zeta, \rho, \sigma \vdash E_0.m(E_1) \Rightarrow v, \sigma_3}$$

1. Evaluate the receiver $E_0$ to an object $(C, \rho_f)$.
2. Evaluate the argument $E_1$ to $v_1$ (memory becomes $\sigma_2$).
3. Look up $m$ starting at $C$ (*dynamic dispatch*: the receiver's actual class drives the lookup); record the host class $C_h$.
4. Build $\rho_b$: fields via $\rho_f$, parameter $x$ at fresh $l \mapsto v_1$, `self` at the receiver, `host` at $C_h$.
5. Run the body $E_b$; its value $v$ is the value of $E_0.m(E_1)$.

# Rules: `self` and `super`

$$\frac{}{\zeta, \rho, \sigma \vdash \texttt{self} \Rightarrow \rho(\texttt{self}), \sigma}$$ (`self` is just looked up in the environment)

$$\frac{\begin{array}{c}\rho(\texttt{self}) = (C, \rho_f), \quad \rho(\texttt{host}) = C_h \\ \zeta(C_h) = (C_p, \_, \_), \quad \textbf{lookup}(\zeta, C_p, m) = (x, E_b, C'_h) \\ \zeta, \rho, \sigma \vdash E_1 \Rightarrow v_1, \sigma_1, \quad l \notin \mathrm{Dom}(\sigma_1) \\ \rho_b = \rho_f[x \mapsto l, \texttt{self} \mapsto (C, \rho_f), \texttt{host} \mapsto C'_h] \\ \zeta,\ \rho_b,\ \sigma_1[l \mapsto v_1] \vdash E_b \Rightarrow v, \sigma_2\end{array}}{\zeta, \rho, \sigma \vdash \texttt{super}.m(E_1) \Rightarrow v, \sigma_2}$$

1. Fetch the current receiver $(C, \rho_f)$ and the current host class $C_h$.
2. Find $C_h$'s parent $C_p$ and look up $m$ starting at $C_p$ (*not* at $C$).
3. Evaluate the argument $E_1$; allocate fresh $l$.
4. Build $\rho_b$: `self` stays the original receiver $(C, \rho_f)$; `host` becomes $C'_h$.
5. Run the body $E_b$; its value $v$ is the value of $\texttt{super}.m(E_1)$.

# Beyond Classes

We have designed the bare minimum. Real OO languages add features such as:

- Access control (`public`, `private`, `protected`).
- Distinct constructors (and destructors).
- Interfaces and abstract classes (signatures without bodies).
- Class variables (`static` fields) shared by all instances.
- Multiple inheritance (e.g. C++).

Each can be layered on top of the core semantics we built today.

# Summary

We extended IMPLICIT-REFS with:

- Classes, objects, fields, methods
- Inheritance and method overriding
- Dynamic dispatch via `self`
- Static dispatch via `super`