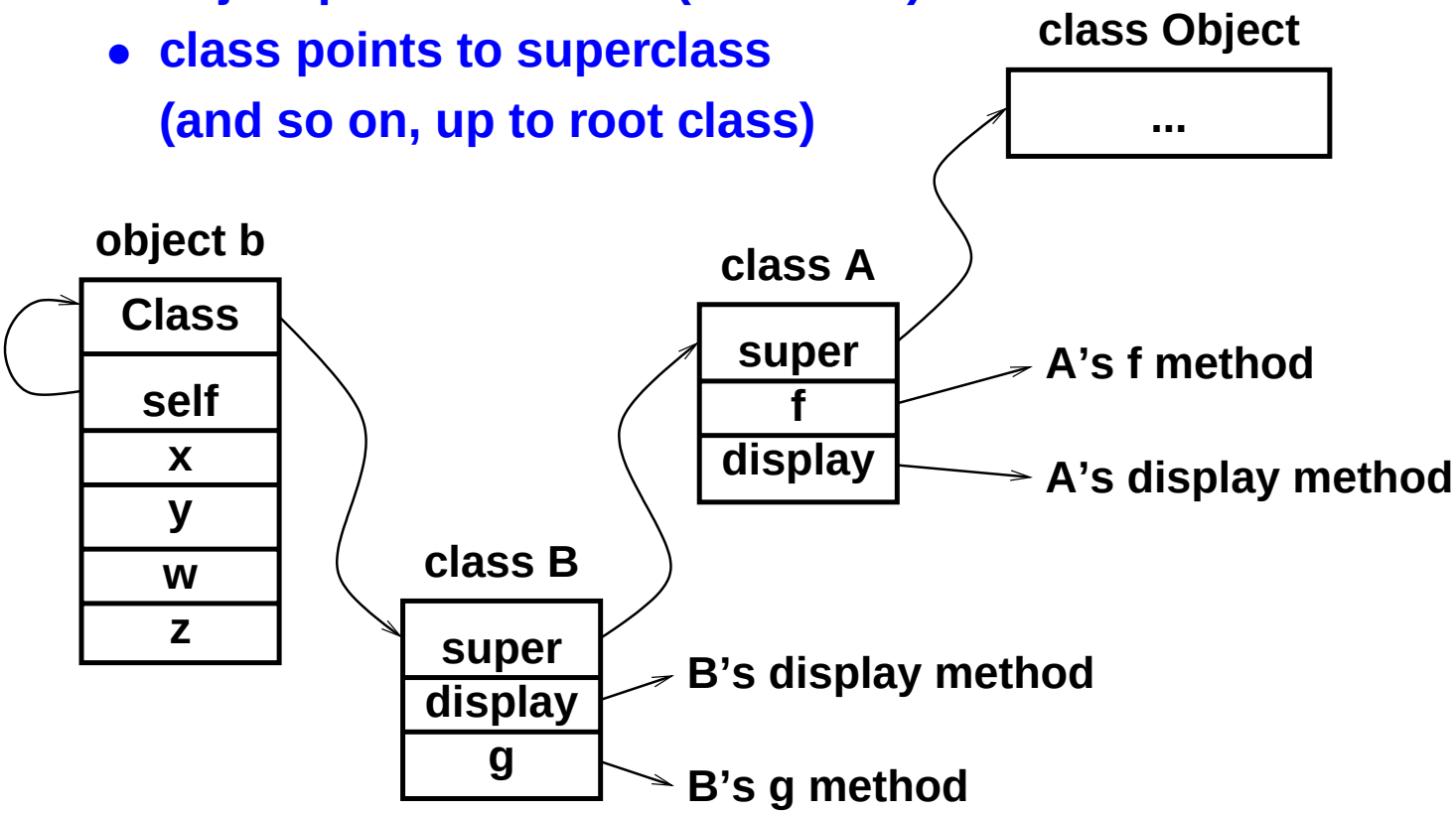
Implementing Smalltalk

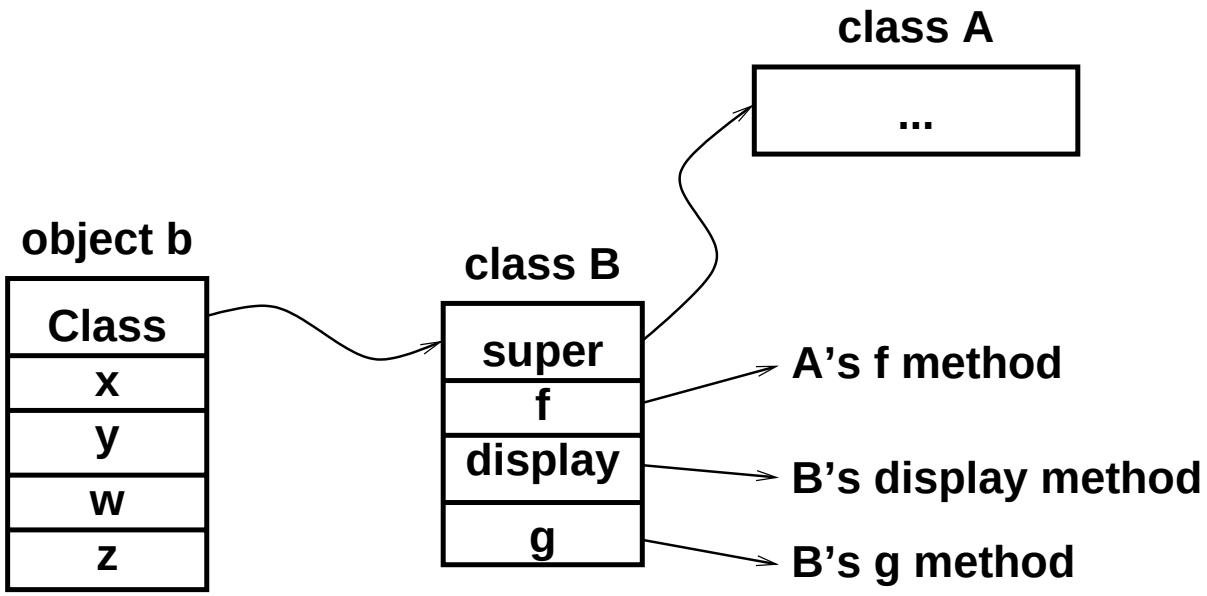
Basic idea:

- store state with objects
- store methods with class
- object points to class (its owner)
- class points to superclass
  (and so on, up to root class)



Compiling objects

In compiled system (Modula-3, C++), precompute method lookup (v-tables, or virtual tables):



All method offsets known at link time (or compile time)

Why not in Smalltalk?

- **Without static typing, any object can respond to any message: perhaps hundreds or thousands**
- **Deep belief in dynamic systems**
- **Clever compilation techniques outperform v-tables**

μSmalltalk implementation: Classes

Classes defined and constructed at compile time:

```
datatype class
  = CLASS of
    { name     : name
    , super    : class option
    , ivars    : name list      (* instance vars *)
    , methods  : method env     (* exported & private *)
    , id       : int            (* unique id *)
    }
(* to be continued *)
```

μSmalltalk implementation: Methods

Methods can be primitive or user-defined

```
and method
  = PRIM_METHOD of
      name * (value * value list -> value)
  | USER_METHOD of
      { name       : name
      , formals    : name list
      , temps      : name list
      , body       : exp
      , superclass : class (* to send to super *)
      }
(* to be continued *)
```

μSmalltalk implementation: Objects

Value is pair of (class, rep)

Basic rep is integer, symbol, array, or collection of instance variables.
Special rep is class, closure (block).

```
and rep
  = USER      of value ref env   (* instance vars *)
  | ARRAY     of value Array.array
  | NUM       of int
  | SYM       of name
  | CLASSREP  of class   (* classes are objects *)
  | CLOSURE   of    (* blocks are objects *)
        name list * exp * value ref env * class
withtype value = class * rep
```

Most objects are `USER` objects

Environments hold *mutable* cells: `value ref`.

μSmalltalk implementation: Environments

Operations:

```
emptyEnv : 'a env
bind     : name * 'a * 'a env -> 'a env
find     : name * 'a env -> 'a RAISES {NotFound}
```

Informal classification of operations for ADT:

- creators (emptyEnv)

  create value of type T, using values of other types
- producers (bind, +, -)

  produce value of T, using existing values of type T
- observers, also called "selectors" or "accessors" (find)

  take a T, produce something else
- mutators

  change a value of T

  sensible only with *mutable state*

μSmalltalk implementation: instance creation

New instance variables are all initialized to `nil`.

```
let
  fun mkIvars (CLASS { ivars, super, ... }) =
    let val supervars =
          case super
            of NONE   => emptyEnv
             | SOME c => mkIvars c
        fun add (n, rho) =  (* alloc new slot for n *)
          bind(n, ref nilValue, rho)
    in  foldl add supervars ivars
    end
(* to be continued *)
```

Note: Fields of `CLASS` record can be listed in any order, and ellipsis (`...`) matches all remaining unnamed fields.

μSmalltalk implementation: instance creation (continued)

```
in
  fun newUserObject (_, c) =
    let val ivars = mkIvars c
        val self = (c, USER ivars)
    in  (find("self", ivars) := self; self)
    end
end
```

Note that `new` assigns to `self`: makes it point to value itself!

μSmalltalk evaluator

Two environments: ρ and ξ (locals and globals)

`superclass` governs messages to `super`

```
fun eval(e, rho, superclass, xi) = let
  fun findMethod (name, class) = (* shown later *)
  fun evalMethod (m, receiver, actuals) = (* shown later *)
  fun evalClosure ((formals, body, rho,
                    superclass), actuals) = (* not shown *)
(* to be continued *)
```

μSmalltalk evaluator (continued)

```
  fun ev(VAR v) =
        !(find(v, rho)   (* ! produces the value to which reference refers *)
          handle NotFound _ => find(v, xi))
    | ev(SET (n, e)) =
        let val v = ev e
            val cell = find(n, rho)
              handle NotFound _ => find(n, xi)
        in  cell := v; v
        end
    | ev(VALUE v) = v
    | ev(LITERAL c) =
        case c of NUM n => mkInteger n
                | SYM n => mkSymbol n
                | _ => (* error *)
(* to be continued *)
```

μSmalltalk evaluator (continued)

```
    | ev(SUPER) = ev (VAR "self")
    | ev(BEGIN es) =
        let fun b(e::es, lastval) = b(es, ev e)
              | b(   [ ], lastval) = lastval
        in  b(es, nilValue)
        end
    | ev(BLOCK (formals, body)) =
        mkBlock (formals, body, rho, superclass)
(* to be continued *)
```

More evaluation: method dispatch

```
  | ev(SEND (message, receiver, args))  =
        let val obj as (class, rep) = ev receiver
            val args = map ev args
            val dispatchingClass =
              case receiver of SUPER => superclass
                             | _     => class
        in  case (message, rep)
              of ("value", CLOSURE clo) =>
                   evalClosure(clo, args)
               | _ =>
                 evalMethod (
                   findMethod(message, dispatchingClass),
                   obj, args))
        end
in ev e
end
```

Method dispatch (continued)

Function `findMethod` is defined as follows:

```
fun findMethod (name, class) =
  let fun fm (CLASS { methods, super, ... }) =
        find (name, methods)
        handle NotFound m =>
          case super
            of SOME c => fm c
             | NONE   => (* error *)
  in  fm class
  end
```

Evaluating a method

Function `evalMethod` is defined as follows:

```
fun evalMethod (PRIM_METHOD (name, f),
                receiver, actuals) =
      f (receiver, actuals)
  | evalMethod (USER_METHOD { name, superclass,
                              formals, temps, body },
                receiver, actuals) =
    let val rho = instanceVars receiver
        val rho = bindList(formals, map ref actuals, rho)
        val rho = bindList(temps,
                    map (fn _ => ref nilValue) temps, rho)
    in  eval(body, rho, superclass, xi)
    end
and instanceVars (_, USER rep) = rep
  | instanceVars self = bind("self", ref self, emptyEnv)
```

Note that `evalMethod` puts together the correct environment.