```
;;;;;
;; 6.037 − Object System

;; Class data abstraction
(define (class? obj)
  (tagged−list? obj 'class))
(define (class−type class)
  (second class))
(define (class−state class)
  (third class))
(define (class−parent class)
  (fourth class))
(define (class−methods class)
  (fifth class))

(define (make−class type state parent methods)
  (list 'class
        type
        state
        parent
        methods))

;; used to produce a methods alist, given a list of arguments
(define (make−methods . args)
  (define (mhelper lst result)
    (cond ((null? lst) result)
          ; error catching
          ((null? (cdr lst))
           (error "unmatched method (name,proc) pair"))
          ((not (symbol? (car lst)))
           (if (procedure? (car lst))
               (display (car lst))
               (void))
           (error "invalid method name" (car lst)))
          ((not (procedure? (cadr lst)))
           (error "invalid method procedure" (cadr lst)))
          ;; end error catching
          (else
           (mhelper (cddr lst) (cons (list (car lst) (cadr lst)) result)))))
  (mhelper args '()))
(define (find−class−method methodname class)
  (let ((result (assq methodname (class−methods class))))
    (if result
        (second result)
        #f)))

;; Instance data abstraction
(define (instance? obj)
  (tagged−list? obj 'instance))
(define (instance−state inst)
  (second inst))
(define (instance−class inst)
  (third inst))

(define (collect−type class)
  (if (class? class)
      (cons (class−type class)
            (collect−type (class−parent class)))
      '()))

(define (collect−state class)
  (if (class? class)
      (append (class−state class)
              (collect−state (class−parent class)))
      '()))

(define (make−instance class . args)
  (let ((inst
         (list 'instance
               (map (lambda (x) (list x #f)) (collect−state class))
               class)))
    (if (has−method? inst 'CONSTRUCTOR)  ;; if it has a constructor, invoke it
        (apply invoke inst 'CONSTRUCTOR args)
        (void))
    inst))

(define (has−method? instance method)
  (define (helper class)
    (cond ((not (class? class))
           #f)
          ((find−class−method method class)
           #t)
          (else (helper (class−parent class)))))
  (helper (instance−class instance)))

;; global variables − used for dynamic scoping
(define self #f)
(define super #f)
```

```
; relies on self dynamically bound to current instance
(define (read−state varname . default)
  (let ((result (assq varname (instance−state self))))
    (if result
        (cadr result)
        (if (not (null? default))
            (car default)
            (error "no state named" varname)))))

(define (write−state! varname value)
  (let ((result (assq varname (instance−state self))))
    (if result
        (set−car! (cdr result) value)
        (error "no state named" varname))))

; builds a procedure that invokes a method on the class' parent
(define (make−super class)
  (lambda (method . args)
    (method−call (class−parent class)
                 method
                 args)))

; walks up class hierarchy looking for method
(define (method−call class method args)
  (if (class? class)
      (let ((proc (find−class−method method class)))
        (if proc
            ; dynamically bind super
            (fluid−let ((super (make−super class)))
              (apply proc args))
            (method−call (class−parent class) method args)))
      (error "no such method" method)))

(define (invoke instance method . args)
  (fluid−let ((self instance))  ; dynamically bind self
    (method−call (instance−class instance) method args)))

; handy for
(define (is−a type)
  (lambda (obj)
    (if (memq type (collect−type (instance−class obj)))
        #t
        #f)))
```