

COMPUTER SCIENCE 51

Spring 2009

cs51.seas.harvard.edu

Prof. Greg Morrisett**Prof. Ramin Zabih****Notes**

Midterms:

- With Susan Welby in Maxwell Dworkin 239

Last Time:

- mutation & side effects
see VII of *HTDP*, 4.9 & 5.8 of *PLT Guide*
- bottom line: difficult to reason about, minimize at all costs, but necessary in some cases for asymptotic space and/or time efficiency.

Today

- key concepts of OO programming
see 5.3 & 13 of *PLT Guide*

OO

“Object-Oriented” means a whole bunch of different things in different languages.

All OO languages have these ideas:

- an *object* is like a struct or record:
 - a named collection of values
 - including (possibly mutable) state, and *functions*
 - the functions are often called *methods*
 - like a *first-class* module
- want to *encapsulate* values within an object.
 - same idea as with modules: information hiding, minimal interface, easy to change internals, etc.
 - outside world doesn't need to know the details

Example Encoding

```
(define-struct stack (push pop))

(define (new-stack)
  (let* ([s empty]
         [push (λ (v) (set! s (cons v s)))]
         [pop
          (λ () (let ([v (car s)])
                  (begin (set! s (cdr s))
                          v)))]])
    (make-stack push pop)))
```

Additional OO Concept #1:

Sub-typing:

- an object that has a “bigger” interface can be used anywhere an object with a “smaller” interface is expected.
- not really an OO concept, rather a math concept:
 - e.g., integers are a subtype of rationals
 - but OO is pointless without subtyping
- a form of *polymorphism*
 - a way for code to be re-used in multiple contexts
 - e.g., matrix multiply algorithm is the same regardless as to whether the elements are integers or rationals.

Example

```
(define-struct (sized-stack stack) (size))

(define (new-sized-stack)
  (let* ([s empty]
         [push (λ (v) (set! s (cons v s)))]
         [pop
          (λ () (let ([v (car s)])
                  (begin (set! s (cdr s))
                          v)))]])
    [size (λ () (length s))]
    (make-sized-stack push pop size)))
```

Subtyping on Structs

```
(define-struct (sized-stack stack) (size))
```

is almost the same as writing:

```
(define-struct sized-stack (push pop size))
```

However:

- the accessors are still `stack-push` and `stack-pop` instead of `sized-stack-push` and `sized-stack-pop`.
- (but we use `sized-stack-size`)
- the predicate `stack?` will return `true` for a `sized-stack`.
- upshot: you can use `sized-stack` anywhere a `stack` is expected.
- i.e., `sized-stack` is a *sub-type* of `stack`.

Additional OO Concept #2

Inheritance:

- a technique for avoiding code duplication.
- in our definition of `new-sized-stack`, we would like to *inherit* the code for `push` and `pop`, instead of writing the code twice.
 - remember, good programmers are lazy!

An Attempt:

```
(define-struct (sized-stack stack) (size))

(define (new-sized-stack)
  (let* ([stck (new-stack)]
        [push (stack-push stck)]
        [pop (stack-pop stck)]
        [size ???])
    (make-sized-stack push pop size)))
```

Alas, we can't get to the list `s` that is encapsulated in `stck`!

An Alternative...

```
(define-struct (sized-stack stack) (size))

(define (new-sized-stack)
  (let* ([stck (new-stack)]
        [count 0]
        [push (λ (v)
                (begin (set! count (+ 1 count))
                       ((stack-push stck) v)))]
        [pop (λ ()
              (begin (set! count (- count 1))
                     ((stack-pop stck))))])
    [size (λ () count)])
    (make-sized-stack push pop size)))
```

Ugh!

- This doesn't seem to save much code.
- We do this sort of extension a lot:
 - add new state (e.g., `count`)
 - add new methods (e.g., `size`)
 - re-define a method in terms of an existing method.
- It would be nice to have better linguistic support for this.

Additional OO Concept #3:

Classes:

- a collection of definitions which can be used to *generate* objects.
 - Conceptually, ties together a struct declaration
 - with definitions for variables and methods
- we can *inherit* definitions from a super-class.
 - allows us to avoid code duplication
 - often (but not always), objects generated from a sub-class are sub-types of objects generated from a super-class.
 - provides convenient access to inherited methods.

A Stack Class in PLT

```
(define stack%
  (class object% ; inherits from object%
    (define s empty) ; a private variable
    (super-new) ; invoke super-class upon new
    (define/public (push x) ; public method
      (set! s (cons x s)))
    (define/public (pop) ; public method
      (let ([v (car s)])
        (begin (set! s (cdr s)) v)))
  )
)
```

Creating Instances

- Given a class definition like `stack%`:


```
(define stack%
  (class object%
    ...
    (define/public (push x) ...)
    (define/public (pop) ...)
  ))
```
- we can create an *instance* through `new`:


```
(define my-stack (new stack%))
```
- and **send** the instance messages:


```
(send my-stack push 3)
(send my-stack push 4)
(send my-stack pop) ==> 4
(send my-stack pop) ==> 3
```

Inheritance in PLT

```
(define sized-stack%
  (class stack% ; inherits from stack
    (define count 0)
    (super-new)
    (define/public (size) count) ; new method
    (define/override (push v) ; redefine push
      (set! count (+ 1 count))
      (super push v)) ; call stack's push here
    (define/override (pop) ; redefine pop
      (set! count (- count 1))
      (super pop v)) ; call stack's pop here
  )
)
```

Another Example

```
(define multi-stack%
  (class stack%
    (super-new)
    ; inherit push & pop directly
    (inherit push pop)
    ; within the class this refers to the object
    ; we could write (push v) but this is short-
    ; hand for (send this push v).
    (define/public (push-list xs)
      (map (lambda (v) (send this push v)) xs))
  )
)
```

In Java

```
class Stack extends Object {
  // protected variables are accessible by sub-
  // classes.
  protected List s = null;

  public void push(Object x) {
    s = new List(x,s);
  }

  public Object pop() {
    Object v = s.car;
    s = s.cdr;
    return v;
  }
}
```

Java Continued

```
class CountedStack extends Stack {
  public int size() {
    int c = 0;
    List t = s;
    while (t != null) {
      c = c + 1;
      t = t.cdr;
    }
    return c;
  }
}
```

Initialization

```
(define point%
  (class object%
    ; x & y must be passed in via new
    (init x y)
    ; so we can use them here
    (define current-x x)
    (define current-y y)
    (super-new)
    (define/public (move-x new-x)
      (set! current-x new-x))
    (define/public (move-y new-y)
      (set! current-y new-y))
  ))
; initialization arguments are named
(define origin (new point% [x 0] [y 0]))
```

3D points

```
(define point3d%
  (class point%
    (init x y z)
    (define current-z z)
    ; invoke point's new, passing in
    ; x and y
    (super-new x y)
    (define/public (move-z new-z)
      (set! current-z new-z))
  ))

(define x (new point3d% [x 0] [y 0] [z 0]))
```

GUI's

- Inheritance works particularly well in some settings like Graphical User Interfaces.
 - Lots of “default behaviors”
 - e.g., similar menus across apps
 - Occasionally, want to modify/specialize
 - e.g., make sure user saves before quitting
 - see <http://docs.plt-scheme.org/gui>
 - (small demo here)

Issues with Inheritance

Control flow is tricky:

```
(define C%
  (class object%
    (define/public (foo) ...)
    (define/public (bar)
      (... (foo) ...))
  ))

(define D%
  (class C%
    (inherit bar)
    (define/override (foo) (do-evil-to-bar))
  ))
```

Uncontrolled overriding leads to spaghetti.

Inheritance <> Subtyping

- Kingdom, Phylum, Class, Order, ...
 - These are types
 - Grouping things with similar attributes
 - But strictly hierarchal (a tree)
- Duckbilled Platypus
 - it lays eggs!
 - bird? reptile?
 - it feeds its young milk!
 - mammals?
 - leads to much confusion
 - A tree isn't always sufficient.



Ideally

- We should be able to assemble a new class by using many other classes.
 - e.g., a car using a GM chassis and Ford engine.
 - This is known as *multiple-inheritance*.
 - PLT supports this through Mixins & Traits. (but we won't be using this much.)
 - Java doesn't.
 - what if we inherit from classes C & D that both export the method foo?
 - should we get C's foo or D's foo?
 - (it's also difficult to implement as efficiently)

The Expression Problem

The style of OO we've seen here is oriented around sending a known message to an object.

- e.g., print, rename, move-to-trash, etc.
- each object gets to decide how it will accomplish the action.
- makes it easy to add new kinds of objects to the system.
- but makes it difficult to add new messages *unless* we can code them in terms of the old.
- e.g., add “convert to XML”

Quick Recap

Object-Oriented Programming consists of many orthogonal constructs:

- 1st class modules
 - encapsulation, information hiding
 - or a struct with lambdas in it!
- Sub-typing
 - can write code that operates over points and it will also work on 3d-points.
- Classes & Inheritance
 - class: collection of definitions that can be used to generate objects (aka instances).
 - one class can inherit methods from another
 - many tricky corner issues.