Object Oriented Programming

- Dominant paradigm in C++, Java
- Possible to do in Lisp as well
- This diverges from the traditional functional model of programming in some respects

Some Terminology

- Class
  - A definition of an object, "the theory," you might say.
- Instance
  - A initialized object, "the practice"
- Generic Function
  - A function that behaves in certain ways depending on the types of arguments it receives

More Terminology

- Methods
- Slots
- Fields in a class, the class attributes
- Similar to fields in a structure

Let's Do This By Example...

- Say we want some classes: shape, triangle, right-triangle
- Say we want some methods: area, move
- Let's implement this
Class-y Definitions

(defclass class-name () (slot1 slot2 slot3...))

Examples:
(defclass point () (x y))
(defclass shape () (position))

There's more to this story, as we will see shortly

Instantly Creating Instances

You guessed it: 
(make-instance 'class-name)

Examples:
(make-instance 'point)
(make-instance 'shape)

Setting Slots

Much as with arefs, or structure fields
(slot-value Instance Slot-name)
Use with setf to set a slot

A Slightly More Elaborate Example

(let ((pos (make-instance 'point))
      (my-shape (make-instance 'shape)))
  (setf (slot-value pos 'x) 3)
  (setf (slot-value pos 'y) 2)
  (setf (slot-value my-shape 'position) pos)
  (slot-value (slot-value my-shape 'position) 'x))

Slot Options

We can set several different things for each slot:
- Default value
- Initializers
- Access functions
- Class versus Instance Slots

Slot Default Value

:initform initial-value

Examples:
(defclass point ()
  ((x :initform 0)
   (y :initform 0))
)
Slot Initializer Arguments

- Tells make-instance to accept an argument that will be used as the initial value for a particular slot. Overides initform.
- :initarg Init-arg-name

Initarg Examples

```lisp
(defclass point ()
  ((x :initform 0 :initarg :x)
   (y :initform 0 :initarg :y))
)
> (slot-value (make-instance 'point :x 5) 'x)
5
```

Slot Accessors

- We can define alternatives to slot-value to read and write values to slots.
  - :accessor (read and write)
  - :reader
  - :writer
- Note that slot-value will always work.

Access Denied!

```lisp
(defclass shape ()
  ((position :initform (make-instance 'point :x 0 :y 0)
    :initarg :pos
    :reader shape-pos))
)
> (shape-pos (make-instance 'shape))
#<POINT #xE8E1A0>
> (setf (shape-pos (make-instance 'shape))
  (make-instance 'point :x 5 :y 5))
;; Error
```

The Case Of Class vs. Instance Allocation

- Instance Allocation: What we expect. Each instance has a slot with a value. Changing one instance leaves the others unaffected.
- Class Allocation: Each instance has a copy of the slot. Changing one instance changes them all.

Each Of Is Allocated...

```lisp
(defclass shape ()
  ((position :initform (make-instance 'point :x 0 :y 0)
    :initarg :pos
    :reader
    (color :initform 'BLUE
      :initarg :color
      :accessor cur-color
      :allocation :class)
  )
)
...But Four Score and Forty Slots

```lisp
> (setf x (make-instance 'shape))
<SHAPE #xE84A54>
> (cur-color x)
BLUE
> (setf y (make-instance 'shape :color 'Green))
<SHAPE #xE8A974>
> (cur-color y)
GREEN
> (cur-color x)
GREEN
```

Superclasses

- We can define a class to be a subclass of another. It will inherit slots, and functionality
- `(defclass class-name (super1 super2 ...)
(slot1 slot2 slot3...))`
- Precedence when multiple overlapping superclasses is given in a complex way. Suffice it to say that there is a precedence list for each class, in general going from most specific to least.

Superclass Example

(Leaping Tall Buildings...)

```lisp
(defclass triangle (shape)
  ((side1 :initarg :a
             :initform 1
             :accessor tri-a)
   (side2 :initarg :b
             :initform 1
             :accessor tri-b)
   (side3 :initarg :c
             :initform 1
             :accessor tri-c))
)
```

```lisp
(defclass right-triangle (triangle)
  ((side3 :initarg :hypo
             :accessor hypo))
)
```

```lisp
> (tri-c (make-instance 'right-triangle))
1
> (hypo (make-instance 'right-triangle))
1
```

...In A Single Bound

```lisp
(defclass right-triangle (triangle)
  ((side3 :initarg :hypo
             :accessor hypo))
)
```

```lisp
> (tri-c (make-instance 'right-triangle))
1
> (hypo (make-instance 'right-triangle))
1
```

Methods

- `(defmethod method-name
      ((param param-class) (param param-class) ... )
   Body)
- Similar to defun (key, optional, rest), except different versions can be defined for different classes. The most specific is called, based on param-classes, from left to right.
- Can only specialize required params.

Method Parameters

- ...must be "Congruent" for a method to be installed for a subclass. This means
  - They must have the same number of required params.
  - They must have the same number of optional params.
  - They must both either use key or rest params or not use key or rest params. (Key for one and rest for the other is ok. So is different numbers of key.)

Kenneth W. Flynn (19-24)
10/07/98
Watson, You Know My Methods
(Deitel / Deitel)

(defmethod radial-dist (p) 'Unknown)
(defmethod radial-dist ((p point))
  (sqrt (+ (* (point-x p) (point-x p))
            (* (point-y p) (point-y p))))
)

> (radial-dist 5)
UNKNOWN
> (radial-dist (make-instance 'point))
0.0

Before, After, and Around

We can define methods to run before, after, or instead of other methods

Standard Method Combination
Run most specific around-method
Otherwise
- All before methods from most specific to least
- The most specific primary method
- All after methods from least specific to most

Example
(defclass silly () ()
(defclass sillier (silly) ()
(defmethod met (c) (format t "0"))
(defmethod met ((c silly)) (format t "1"))
(defmethod met :before ((c silly)) (format t "~%0"))
(defmethod met :after ((c silly)) (format t "2~%"))
(defmethod met ((c sillier)) (format t "S"))
(defmethod met :around ((c sillier)) (format t "~%A") (call-next-method))

Example II

> (met (make-instance 'silly))
012
> (met (make-instance 'sillier))
A
0S2

That's It!

For the future
- Work on Project #2 (Friday)
- Work on Project #3 (10/23)
- Final Exam (10/16)
- Goodbye!

Kenneth W. Flynn (25-30)
10/07/98