Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.2.1

Top Ten Things to do in CLOS

FRANZ INC.

1. Define a Class

```
(defclass position ()
  ((x :initform 0 :initarg :x
      :accessor x-position)
   (y :initform 0 :initarg :y
      :accessor y-position)))
```

2. Make an Instance

```
(setq s1 (make-instance 'position))
(x-position s1)
⇒ 0
(setf (x-position s1) 5)
(x-position s1)
⇒ 5

(setq s2 (make-instance 'position :x 10 :y 10))
(x-position s2)
⇒ 10
```

3. Define a Subclass

```
(defclass aircraft (position)
  ((speed :initform 0 :initarg :speed
          :accessor speed)
   (flightno :initform "" :initarg :flightno
             :accessor flightno)))

(setq m1 (make-instance ‘aircraft
                         :x 5 :y 5 :speed 2
                         :flightno 1024))
```

4. Use Methods

```
(defmethod name ((a aircraft))
  (concatenate 'string "flight "
        (princ-to-string (flightno a))))

(defmethod draw ((a aircraft) stream)
  (draw-text stream (name a)
    (x-position a) (y-position a)))
```

5. Use Method Combination

```
(defclass aircraft-with-icon (aircraft)
  ())

(defmethod draw :AFTER
      ((a aircraft-with-icon) stream)
  "After drawing the name, draw the icon"
  (draw-icon stream *plane-icon*
    (x-position a) (y-position a)))
```

6. Initialize Instances

```
(defvar *all-aircraft* nil)

(defmethod initialize-instance :after
           ((a aircraft)
            &allow-other-keys)
  (push a *all-aircraft*))
```

7. Use Slot-Value

```
(defmethod set-position ((p position) x y)
  (setf (slot-value p 'x) x)
  (setf (slot-value p 'y) y)
  t)

(defmethod get-position ((p position))
  (values (slot-value p 'x)
          (slot-value p 'y)))
```

About Slot-Value

- You can always access a slot using `slot-value`
- The general rule is to prefer accessor methods (e.g. x-position and y-position) over raw `slot-value`.
- Exceptions:
  - When the accessor function has a lot of `:after`, `:before`, or `:around` methods, slot-value is faster
  - The accessor function may have `:after` methods that you want to avoid in some cases

8. Use SETF methods

```
(defmethod (setf x-position) :after
           (newvalue (a aircraft))
  (redraw-display *screen*))

;; This causes a redisplay when the
;; position changes

(setf (x-position myaircraft) 16)
```

9. Use Multiple Dispatch

```
(defmethod save ((p position) (stream file-stream))
  . . . )
(defmethod save ((a aircraft) (stream file-stream))
  . . . )
(defmethod save ((p position) (stream database))
  . . . )
(defmethod save ((a aircraft) (stream database))
  . . . )

;; The applicable method(s) depends on
;; multiple arguments.
```

10. Use Multiple Inheritance

```
(defclass boeing-747 (passengers-mixin
                      commercial-mixin
                      aircraft)
  ())

;; The class is (in most ways) the union
;; of the structure and behavior of the
;; components.
```

Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.2.2

CLOS Overview

FRANZ INC.

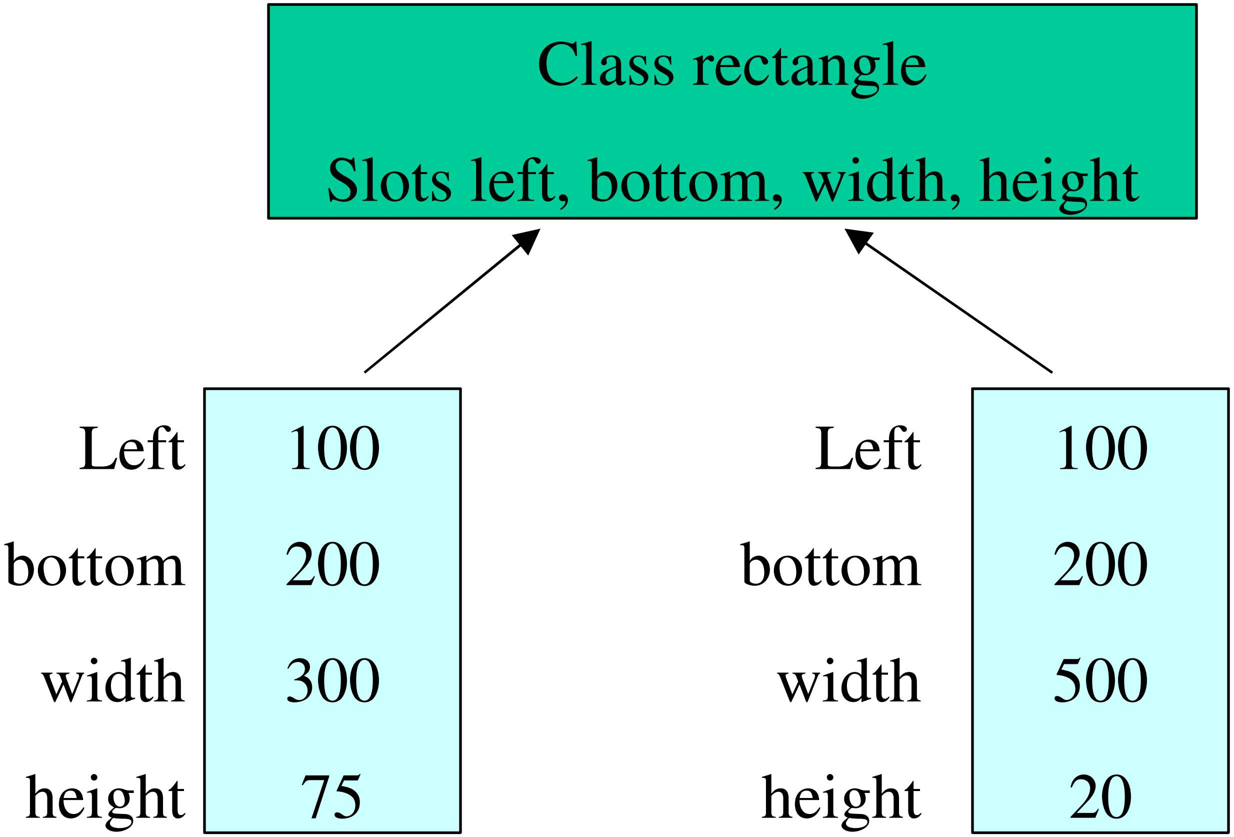
Common Lisp Object System

- Based on CommonLoops and New Flavors
- Integrates Common Lisp types with Classes
- Uses function calls, not messages
- Uses objects to implement Classes and other internals
  - Metaobject protocol
- Is part of ANSI Common Lisp standard

Terminology

- Define classes of objects (`defclass`)
- Make objects (instantiation)
- Instance variables (slots)
- Messages (generic function)
- Applicable behavior (methods)

Classes and Instances



Class rectangle

Slots left, bottom, width, height

Left	100
bottom	200
width	300
height	75

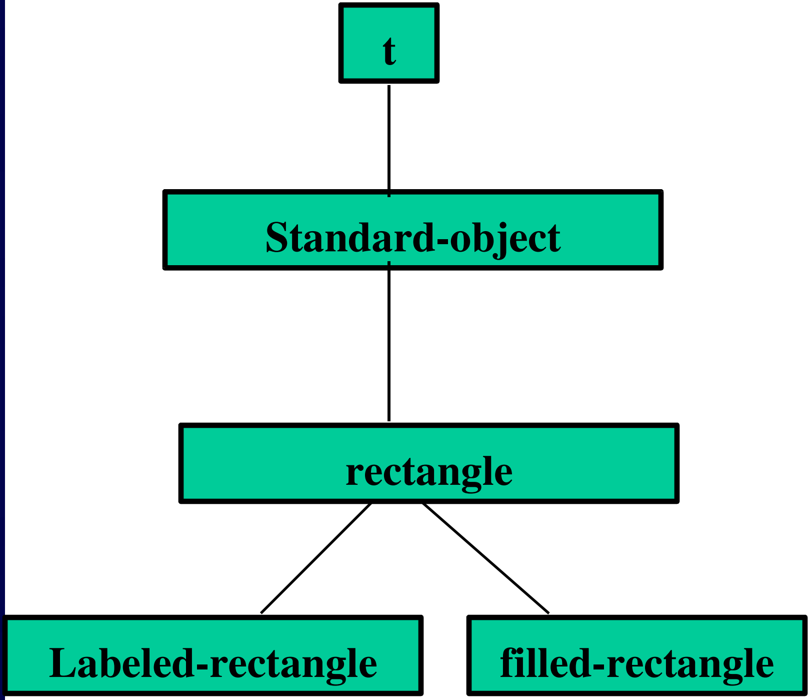
Left	100
bottom	200
width	500
height	20

Slot-names slot-values

Slots

- Instances of a class have slots
- Slots have name and value
- Two types of Slots
  - Local Slots (most common)
  - Shared Slots (more on this later)

Class Inheritance



Terminology

rectangle is a *direct superclass* of labeled-rectangle

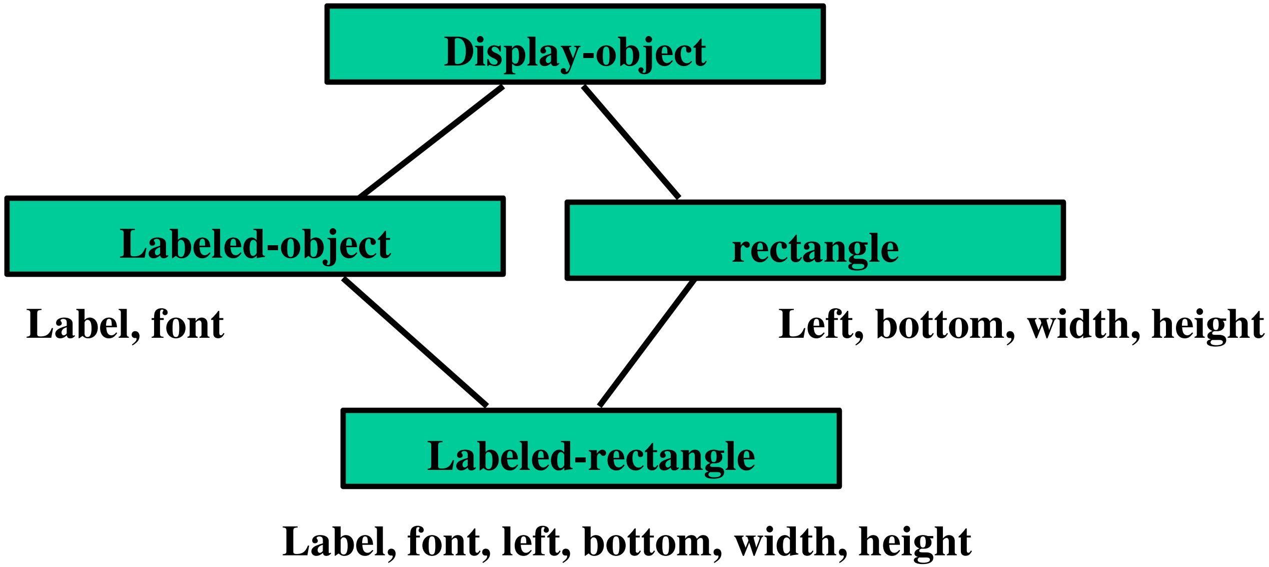
labeled-rectangle is a *direct subclass* of rectangle

labeled-rectangle is a *subclass* of standard-object

standard-object is a *superclass* of labeled-rectangle

t is a *superclass* of all classes

CLOS Supports Multiple Inheritance



Display-object

Labeled-object

rectangle

Label, font

Left, bottom, width, height

Labeled-rectangle

Label, font, left, bottom, width, height

Class inherits union of slot descriptions

Supporting Type-Specific Behavior

- In ordinary functions, a single definition must dispatch to the appropriate code

```
(defun area (shape)
  (ecase (type-of shape)
    (circle . . .)
    (rectangle . . .)
    (triangle . . .)))
```

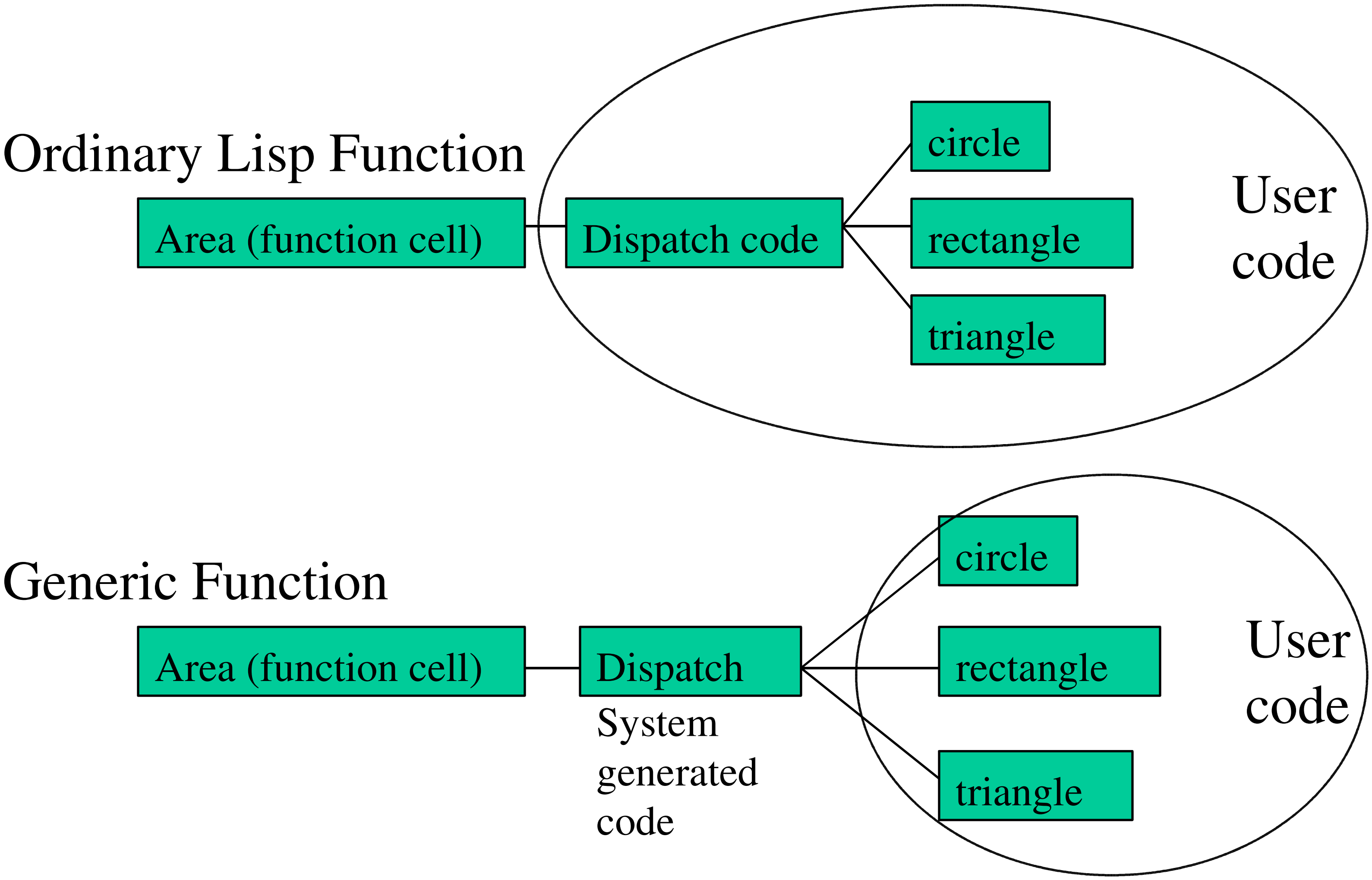
CLOS Generic Functions Support Modular Definitions

- Defgeneric to define the interface
- defmethod to define the implementations

```
(defgeneric area (shape) . . .)

(defmethod area ((shape circle)) . . .)
(defmethod area ((shape rectangle)) . . .)
(defmethod area ((shape triangle)) . . .)
```

CLOS Generates Dispatch Code



Ordinary Lisp Function

Area (function cell)

Dispatch code

circle

rectangle

triangle

User code

Generic Function

Area (function cell)

Dispatch

System generated code

circle

rectangle

triangle

User code

Dispatch on Multiple Arguments

```
(defmethod draw ((shape rectangle) (stream postscript-stream)) . . .)
(defmethod draw ((shape rectangle) (stream window-stream)) . . .)
(defmethod draw ((shape circle) (stream postscript-stream)) . . .)
(defmethod draw ((shape circle) (stream window-stream)) . . .)
```

- Because of multiple dispatch, methods do not "belong" to classes
  - They "belong" to a combination of one or more classes
  - Differs from message-passing systems where a class implements certain messages

- Methods are associated with the generic functions

Method Combination

- Each class in the list of superclasses can contribute a component of the *effective method*
  - Primary method performs the bulk of the work and returns values
  - Before methods do error checking and preparation
  - After methods perform side-effects and cleanup

Class Precedence Lists

- Class precedence list is list of superclasses
- For single inheritance, ordering is obvious (most-specific first)
- For multiple inheritance, class precedence list is computed according to local ordering constraints
- When two classes offer competing traits, CLOS resolves the conflict with precedence

Defining a class

- (defclass <class-name> (<superclass>…)
  (<slot-definition1>
  <slot-definition2>))

```
(defclass point (graphic-object)
   ((x :initarg :x :accessor point-x)
    (y :initarg :y :accessor point-y)))
```

Defining a slot

- Name
- Slot Options

– :initform	default value for initialization
– :initarg	argument for initialization
– :reader	define reader method only
– :writer	define writer method only
– :accessor	define both reader and writer

defclass Options

- Class Options
  - `:documentation` descriptive string
  - `:default-initargs` arguments for initialization

```
(defclass circle (point)
   ((radius :initform 5 :initarg :radius :accessor radius))
   (:documentation "A round thing")
   (:default-initargs :x 0 :y 0))
```

Defining the Interface

- defgeneric -- optional, defmethod will implicitly create

```
(defgeneric draw-part (part stream)
   (:documentation "Displays the part on a window"))
```

Defining the Implementation

- Specialized parameter - (part circle)
- Ordinary parameter - stream

```
(defmethod draw-part ((part circle) stream)
    (draw-circle stream (point-x part)
                        (point-y part) (radius part)))
```

Make-Instance

- Used to create object given a class
- You can specify initial slot values

- `(setq my-square (make-instance 'square :x 0 :y 0))`

Accessing and Changing Slot Values

- Retrieving current state
  - `(slot-value my-square 'x)`
    - 0
- Changing the state
  - `(setf (slot-value my-square 'x) 10)`
    - 10
- Syntactic sugar
  - `(with-slots (x) my-square (setq x 15) (print x))`

Example - Squares

- Define a simple graphical object

```
(defclass square ()
  ((x :initform 0 :initarg :x :accessor x-position)
   (y :initform 0 :initarg :y :accessor y-position)
   (width :initform 0 :initarg :width :accessor width)))

(setq my-square (make-instance 'square :x 5 :y 5
                                       :width 15))
```

Constructor Function

```
(defun make-square (x y width)
  (make-instance 'square :x x :y y :width width))
```

- Functional interface for instance creation
- Advantages
  - Checking of required arguments
  - Class name not advertised

Example - Rectangles

- Define a class using inheritance

```
(defclass rectangle (square)
   ((height :initform 0 :initarg :height
                 :accessor height)))

;; rectangle inherits from square

(setq my-rectangle
   (make-instance 'rectangle :x 10 :y 30
                            :width 10 :height 12))
```

Example - Method

- Compute area of graphical object

```
(defmethod area ((object square))
   (* (width object) (width object)))

(area my-square)   =>  225
```

Example - Method Inheritance

- To inherit or not to inherit

```
(area my-rectangle)  =>  100  ; wrong!

(defmethod area ((object rectangle))
   (* (width object) (height object)))

(area my-rectangle)   =>  120
```

Getting the class of an object

- Using CLASS-OF, CLASS-NAME, TYPEP, and TYPE-OF

```
> (class-of my-square)
#<standard-class square>
> (class-name (class-of my-square))
SQUARE
> (typep my-square ‘square)
T
> (type-of my-square)
SQUARE
```

DESCRIBE

- Objects are composed of slots

```
> (describe my-square)
#<SQUARE 31ab4> is an instance of class SQUARE
X            5
Y            5
WIDTH  15
```

SLOT-VALUE

- Gets the value of a slot

```
> (slot-value my-rectangle 'width)
15
> (slot-value my-rectangle 'height)
12
> (slot-value my-square 'height)
;; error!
> (setf (slot-value my-rectangle 'height) 15)
15
> (slot-value my-rectangle 'height)
15
```

Other slot functions

- slot-boundp
  - Determines if the slot has a value
- slot-exists-p
  - Determines if the object has a slot by that name
- slot-makunbound
  - Causes the slot to have no value

:ACCESSOR Slot Option

- Define a function for accessing the slot
- Advantage: Slot name not advertised
  - Accessor functions are a good idea

```
(defclass rectangle (square)
   ((height :initform 0 :initarg :height
                :accessor height)))
> (height my-rectangle)
12
> (setf (height my-rectangle) 15)
15
> (slot-value my-rectangle ‘height)
15
```

:INITFORM Slot Option

- Specifies default initial value

```
(defclass rectangle (square)
   ((height :initform 0 :initarg :height
                :accessor height)))
> (setq another (make-instance ‘rectangle :x 6 :y 6))
#<RECTANGLE 34a7>
> (height another)
0
```

:INITARG Slot Option

- Specifies keyword to use with make-instance

```
(defclass rectangle (square)
   ((height :initform 0 :initarg :height
                :accessor height)))
> (setq yet-another (make-instance ‘rectangle
                                      :height 14))
#<RECTANGLE @ #x6734a9>
> (height yet-another)
14
```

:ALLOCATION slot option

- Slots have two types of allocation:
  - :instance each instance gets its own slot value
  - :class all instances share the same slot value

```
(defclass triangle (basic-part)
   (…
    (number-of-sides       :reader number-of-sides
             :initform 3
             :allocation :class)))
```

Alternate approach

- Use methods instead of shared slots

```
(defmethod number-of-sides ((part triangle)) 3)
```

Methods

- Associate behavior with objects

```
(defclass point ()
   ((x :accessor point-x :initarg :x :initform 0)
    (y :accessor point-y :initarg :y :initform 0)))

(defmethod distance ((from point) (to point))
   (pythagonize (point-x from) (point-y from)
                         (point-x to) (point-y to)))
(defun pythagonize (x1 y1 x2 y2)
   (let ((dx (- x1 x2)) (dy (- y1 y2)))
       (sqrt (+ (* dx dx) (* dy dy)))))
```

Multiple Dispatch

- Method you get depends on all arguments

```
(defclass dot (point)
   ((size :accessor dot-size :initform 1 :initarg :size)))

(defmethod distance ((from point) (to dot))
   (- (pythagonize (point-x from) (point-y from)
                            (point-x to) (point-y to))
       (dot-size to)))
```

Dispatching on Class `T`

- Class `T` is the class of all objects

```
(defmethod distance ((from t) (to t))
   (error " Don’t know how to compute distance"))
```

OR

```
(defmethod distance (from to)
   (error " Don’t know how to compute distance"))
```

Dispatch Using EQL

- Applies to program constants

```
(defmethod distance ((from (eql :origin)) (to t))
   (distance (make-instance 'point :x 0 :y 0) to))

> (distance :origin (make-instance 'point :x 3 :y 4))
5
```

Dispatch Using EQL, cont'd.

- Also applies to instances

```
(defclass place ()())
(defmethod name ((x place)) "someplace")

(setq home (make-instance 'place))
(setq office (make-instance 'place))
(defmethod name ((x (eql home))) "my home")
(defmethod name ((x (eql office))) "my office")

(name (make-instance 'place)) --> "someplace"
(name office) --> "my office"
```

:BEFORE and :AFTER methods

- Before or after the “primary” method
- Return value is ignored

```
(defmethod area :before ((object square))
   (when (< (width object) 0)
      (error "Width is negative.")))
```

Order of Before and After

- All before-methods in most-specific-first order.
- The most specific primary method.
- All after-methods in most-specific-last order.

:AROUND methods

- An around method shadows all before, after, and primary methods
- Value returned from generic function is value of around method
- Nested around methods: most-specific first

```
(defmethod area :around ((object square-with-hole))
  (- (call-next-method)
     (area-of-hole object)))
```

Primary methods call-next-method

- Do it when you want to be “inside” all the :around, :before, and :after methods
- next-method-p can be useful in this context

```
(defmethod area ((object square-with-hole))
  (- (call-next-method)
     (area-of-hole object)))
```

Call-next-method with arguments

```
(defmethod draw-part ((part hidden-circle) stream)
   (declare (ignore stream))
   (call-next-method part *hidden-stream*))
```

call-next-method example

```
(defmethod ((a list) b)
  (format t "First arg ~S is a list .~%" a)
  (if (next-method-p) (call-next-method)))
(defmethod (a (b number))
  (format t "Second arg ~S is a number.~%" b)
  (if (next-method-p) (call-next-method)))
> (foo '(1 2 3) 'a)
First arg (1 2 3) is a list.
> (foo 'a 3)
Second arg 3 is a number.
> (foo '(1 2 3) 3)
First arg (1 2 3) is a list.
Second arg 3 is a number.
```

SETF Methods

- Example:

```
(defmethod (setf height) (newvalue (part square))
   (setf (width part) newvalue))
```

initialize-instance

- Never override the primary method!
- This is where you initialize the object

```
(defmethod initialize-instance :after
                          ((object square) &key
&allow-other-keys)
   (when (< (width object) 0)
      (error "Width is negative.")))

> (make-instance 'square :width -5)    ; errc
```

print-object

- Modify standard common lisp behavior

```
(defmethod print-object ((object point) stream)
   (let ((x (point-x object)) (y (point-y object)))
      (if *print-escape*
           (print-unreadable-object
                (object stream :identity t :type t)
                (format stream " ~S,~S " x y))
          (format stream " ~S ~S,~S " (type-of object) x y))))
> (setq p (make-instance ‘point :x 3 :y 2))
#<POINT 3,2  @ #x204d8452>
> (princ p)
POINT 3,2
```

print-object Support

- `print-unreadable-object` is a macro that helps you print
  - #<type stuffhere identity>
  - #<POINT 3,2 @ #x204d8452>
- `*print-escape*` is set by the pretty printer to indicate the desire for #< . . . >

Inheritance and Combining Methods

- Use Class Precedence List to determine methods that run
- Most specific applicable primary method runs
- All before methods run, most specific first
- All after methods run, most specific last

Class Precedence List

```
(defclass basic-part () …)
(defclass rectangle (basic-part) …)
```

- Rule1: A class always has precedence over its super classes
- Rectangle has precedence over basic-part
- Basic-part has precedence over standard-object
- Standard-object has precedence over T
- Precedence list that satisfies all these constraints:
  - (rectangle basic-part standard-object T)

Class Precedence Lists

- Complications when there is more than one direct super class
- Rule2: Each class definition sets the precedence order of its direct super classes
- Rule3: Classes appear only once in CPL

```
(defclass rectangle (selectable-part saveable-part
                     basic-part)
  (x y width height))
```

- Selectable-part has precedence over saveable-part
- Saveable-part has precedence over basic-part

Precendence example

```
(defclass bar () ())
(defclass baz () ())
(defmethod foo ((x bar)) (format t “I am a bar!~%”))
(defmethod foo ((x baz)) (format t “baz I am!~%”))

(defclass bsub1 (bar baz) ())
(defclass bsub2 (baz bar) ())
(setq b1 (make-instance ’bsub1))
(setq b2 (make-instance ’bsub2))

(foo b1)
I am a bar!
(foo b2)
baz I am!
```

Putting it all together

Developing a Simple CLOS Program

- Specify the problem
- Identify objects of interest
- Design a class hierarchy
- Design a client interface (API)
- Create the implementation
- Extend it (subclasses)

Some Guidelines on API

- Restrict access to internal data structures (encapsulation)
  - Specialize describe-object and print-object
  - Offer Accessor methods in the API
- Provide constructor functions
  - (Make-point) rather than (make-instance 'point)
- Define contracts for generic functions so client can extend them

Reasons to Use Class Hierarchies

- Subclasses inherit structure (via slots)
- Subclasses inherit behavior (via methods)
- Multiple inheritance supports modular reuse without copying
  - write labeled-object once and mix it in to labeled-circle and labeled-rectangle
- *Abstract* classes are classes in the hierarchy that you never instantiate
  - providing partial but not complete behavior (e.g. labeled-object)

Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.2.3

CLOS Advanced Features

FRANZ INC.

Congruence of Method Argument Lists

- All methods of a generic function must have congruent argument lists
- args are congruent when
  - there are the same number of required args
  - there are the same number of optional args
  - use of &rest and &key compatible
- CLOS signals error if you try to define a method whose arglist isn't congruent

Congruency examples

- (x y) is congruent with (height width)
- (n &optional inc) not congruent with (number incr)
- (thing &rest dims) is congruent with (box &key width height depth)

Keyword Congruency Examples

- Illegal:
  - (defmethod test (r1 r2) . . .)
  - (defmethod test (r1 r2 &key f2) . . .)

- Legal:
  - (defmethod test (r1 r2 &key f1 f2 f3) . . .)
  - (defmethod test (r1 r2 &key &allow-other-keys) . . .)
  - (defmethod test (r1 r2 &key f3 &rest key-args) . . .)

Specialization of Slots

```
(defclass labelled-rectangle (rectangle)
  ((label :initarg :label)
   (font :initform (make-font ‘(modern 10))
         :accessor rectangle-font)))

(defclass roman-rectangle (labelled-rectangle)
  ((font :initform (make-font ‘(times-roman 12)))))
```

- Most specific :initform is used.

Using a Shared Slot

- :allocation :class
- Use them as an alternative to global variables
- Shared slots are stored within the class
- Changes by one instance are visible to all instances

Inheritance of Shared Slots

- Shared slots are inherited
  - Instances of subclasses see the same value as instances of the class
- A subclass can shadow the slot value in a superclass by defining it as a direct slot definition
  - Instances of subclasses see a different value than do instances of the class

Specializing Shared Slots to Local Slots

- A subclass can change the slot allocation to : instance
- Instances of the subclass will use a local slot, whereas instances of the class will use a shared slot

defgeneric

- arglist normal, but no initial values or supplied-p allowed
- gf options
  - :declare -- declaration for whole gf, only optimize allowed by spec
  - :argument-precedence-order -- lists all required args in order for dispatch
  - also :documentation, :generic-function-class, : method-class, :method-combination

:argument-precedence-order

```
(defmethod foo ((a list) b)
   (format t "Arg 1 ~S is a list~%" a))
(defmethod foo (a (b number))
   (format t "Arg 2 ~S is a number~%" b))

(foo '(1 2 3) nil)
Arg 1 (1 2 3) is a list
(foo 'a 10)
Arg 2 10 is a number
(foo '(1 2 3) 10)
Arg 1 (1 2 3) is a list
(defgeneric foo (a b) (:argument-precedence-order b a))
(foo '(1 2 3) 10)
Arg 2 10 is a number
```

Changing Generic Functions

- Legal Changes
  - – any redefinition if there are no methods
  - – argument-precedence-order
  - – documentation
  - – default-method-class
- Illegal Changes
  - – lambda list (congruence rules not satisfied)
  - – method combination
  - – generic-function-class

Changing Methods

- Redefining a method with the same specializers and qualifiers replaces old
- If specializers and qualifiers change, a new method is added
- A method can be removed with remove-method or Emacs command `fi:kill-definition`

find-class

- given a class name, returns class object
- works for builtin types as well

class-name

- inverse of find-class
- given class object, returns name

class-of

- given instance of a class, returns class object
- returns special class objects for primitive types
- e.g. `(class-of "abc") -> #<BUILT-IN-CLASS STRING>`

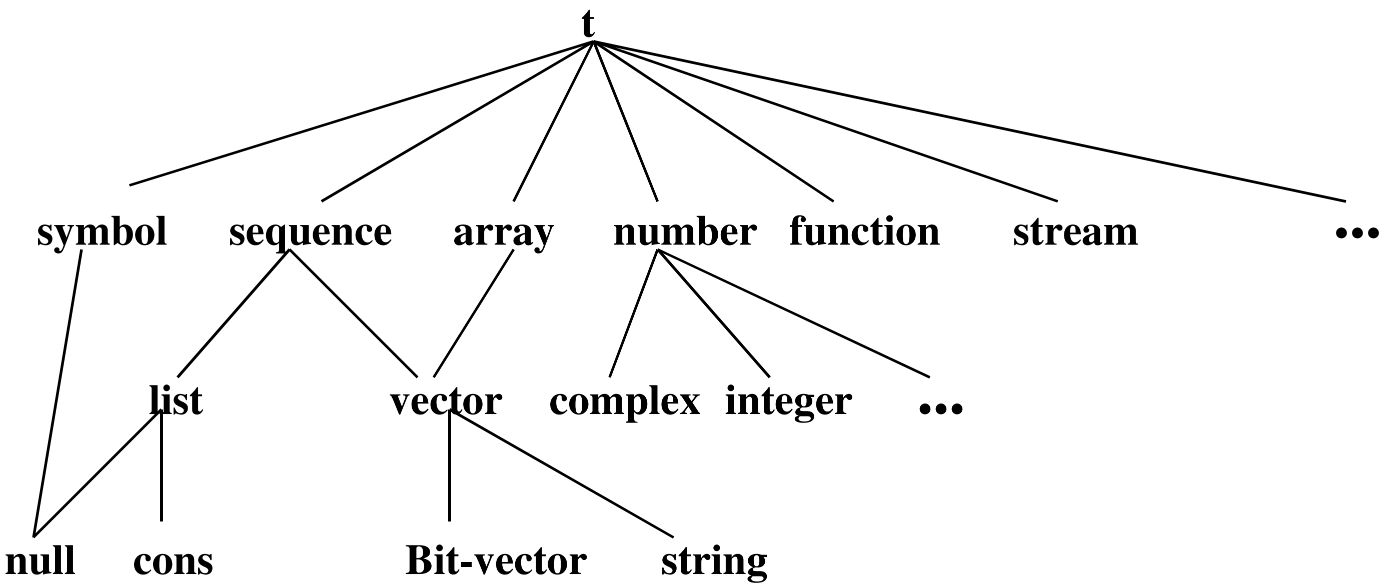
Almost All Built-in Types Have Corresponding Classes

All Classes with proper names have corresponding types

```
(find-class ‘string)
   #<BUILT-IN-CLASS STRING>
But there is no class named bit.

(defmethod pretty-type-name ((c cons)) "Cons")
(defmethod pretty-type-name ((c symbol)) "Symbol")
(defmethod pretty-type-name ((c rectangle)) "Rectangle")
```

Inheritance for Built-in Types



Defstruct defines classes

```
(defstruct s-rectangle
  (x 0)
  (y 0)
  width
  height)

(class-of (make-s-rectangle))
   => #<structure-class s-rectangle>

(defmethod area ((shape s-rectangle))
  (* (s-rectangle-width shape)
     (s-rectangle-height shape)))
```

But Structure Accessors are not Generic

- S-rectangle-width is an ordinary lisp function

- This is an error:

```
(defmethod s-rectangle-width :around ((shape s-rectangle))
   . . .)
```

Allegro CL Certification Program

Lisp Programming Series Level 2

Session 3.4

CLOS Elements of Style

Avoid typep

```
(if (typep x 'rectangle) …) ; bad
(if (rectangle-p x) …)   ; good
(defmethod rectangle-p ((object t)) nil)
(defmethod rectangle-p ((object
  rectangle)) t)
```

- Resulting code makes it easier to later adapt the code to new classes

Avoid Slot-value

```
(slot-value point 'x) ; bad
(point-x point) ; good
```

- Use accessor functions instead of slot-value
- Hide data structure decisions in case you change your mind

Avoid Multipurpose Slots

- Avoid using a slot for more than one purpose
- If you have to test the type of a slot value to know what is there, then consider adding more slots or defining more subclasses

Use Constructors

```
(defun make-circle (x y &key (radius 10))
   (make-instance 'circle
      :x x :y y :radius radius))
```

- It’s a good practice to write constructor fns
- You get better arg handling
- Hide data structure decisions in case you change your mind

Add Print-object Methods

- Printed representation should make concise statement about object state
  - point: x,y location
  - stream: input or output, open or closed
- Useful for debugging
- Especially useful when there are many instances in a big trace history

Use EQL Methods With Symbols

```
(defmethod handle-event ((event (eql 'redraw)) window)
   …)
(defmethod handle-event ((event (eql 'iconify)) window)
   …)
```

- Like a case statement but more modular and more easily extended
- The drawback is that method dispatch is a bit slower

Peter Norvig’s Lisp Style Maxims

- Be specific
  - SETQ is more specific than SETF
- Use abstractions
  - SECOND is more readable than CADR
- Be concise
- Use the provided tools, don’t reinvent them
- Don’t be obscure, avoid programming tricks
- Be consistent

The Goal

- Reduce a complicated problem to a collection of easy-to-understand procedures
- Good decomposition leads to
  - Faster implementation
  - Fewer bugs
  - Easily maintained source code

training@franz.com

http://www.franz.com/lab/