# Robot Programming with Lisp

## 3. Object-Oriented Programming and Failure Handling

Gayane Kazhoyan

Institute for Artificial Intelligence
University of Bremen

1st of November, 2018

# Contents

# Structures

## Handling Structs

```
CL-USER> (defstruct player
           id
           (name "mysterious stranger" :type string)
           (hp 10 :type integer)
           (mp 0 :type integer)
           and-so-on)
CL-USER> (defvar *player* (make-player :name "Turtle" :and-so-on '123))
         *player*
#S(PLAYER :ID NIL :NAME "Turtle" :HP 10 :MP 0 :AND-SO-ON 123)
CL-USER> (player-name *)
"Turtle"
CL-USER> (defvar *player-copy* (copy-player *player*))
         (setf (player-name *player-copy*) "Cat")
         *player-copy*
#S(PLAYER :ID NIL :NAME "Cat" :HP 10 :MP 0 :AND-SO-ON SOME-DATA)
CL-USER> *player*
#S(PLAYER :ID NIL :NAME "Turtle" :HP 10 :MP 0 :AND-SO-ON 123)
```

# Hash Tables

## Handling Hash Tables

```
CL-USER> (defvar *table* (make-hash-table :test 'equal))
*TABLE*
CL-USER> *table*
#<HASH-TABLE :TEST EQUAL :COUNT 0 {100A84AF03}>

CL-USER> (setf (gethash "MZH" *table*) "Bibliothekstrasse 3"
               (gethash "TAB" *table*) "Am Fallturm 1")
"Am Fallturm 1"
CL-USER> (gethash "MZH" *table*)
"Bibliothekstrasse 3"
T
```

# Contents

# Classes

## Handling Classes

```
CL-USER> (defclass shape ()
           ((color :accessor get-shape-color
                   :initarg :set-color)
            (center :accessor shape-center
                    :initarg :center
                    :initform '(0 . 0))))
#<STANDARD-CLASS SHAPE>
CL-USER> (defvar *red-shape* (make-instance 'shape :set-color 'red))
*RED-SHAPE*
CL-USER> (describe *red-shape*)
#<SHAPE {100536B6A3}>
  [standard-object]

Slots with :INSTANCE allocation:
  COLOR    = RED
  CENTER   = (0 . 0)
CL-USER> (get-shape-color *red-shape*)
RED
```

# Classes [2]

## Inheritance

```
CL-USER> (defclass circle (shape)
           ((radius :initarg :radius)))
#<STANDARD-CLASS CIRCLE>
CL-USER> (defvar *circle*
           (make-instance 'circle :set-color 'green :radius 10))
*CIRCLE*
CL-USER> (describe *circle*)
#<CIRCLE {1005F61973}>
  [standard-object]

Slots with :INSTANCE allocation:
  COLOR    = GREEN
  CENTER   = (0 . 0)
  RADIUS   = 10
CL-USER> (slot-value *circle* 'radius)
10
```

# Lisp class vs. Java class

Lisp classes have / support:

- attributes
- getter-setter methods
- multiple inheritance

Lisp classes don't have:

- attribute access specifications (managed with package namespaces)
- methods

# Contents

# Function Overloading: Generic Programming

## Defining Generic Functions

```
CL-USER> (defgeneric area (x)
           (:documentation "Calculates area of object of type SHAPE."))
CL-USER> (area 1)
; #<SIMPLE-ERROR "~@<There is no applicable method for ..."
CL-USER> (defmethod area (x)
           (error "AREA is only applicable to SHAPE instances"))
CL-USER> (defmethod area ((obj shape))
           (error "We need more information about OBJ to know its area"))
CL-USER> (defmethod area ((obj circle))
           (* pi (expt (slot-value obj 'radius) 2)))
CL-USER> (area 1)
; #<SIMPLE-ERROR "AREA is only applicable to SHAPE instances">
CL-USER> (area *red-shape*)
; #<SIMPLE-ERROR "We need more information about OBJ to know its area"
CL-USER> (area *circle*)
314.1592653589793d0
```

# Function Overloading: Generic Programming [2]

## Method combinations: :before, :after, :around

```
CL-USER> (defmethod area :before (obj)
           (format t "Before area. "))
CL-USER> (area *circle*)
Before area.
314.1592653589793d0
CL-USER> (defmethod area :around ((obj shape))
           (format t "Taking over shape area. "))
CL-USER> (area *red-shape*)
Taking over shape area.
CL-USER> (defmethod area :around ((obj shape))
           (format t "Taking over shape area. ")
           (call-next-method))
CL-USER> (area *red-shape*)
Taking over shape area. Before area. ; #<SIMPLE-ERROR "We need ..."
CL-USER> (defmethod area :around ((obj shape))
           (round (call-next-method)))
CL-USER> (area *circle*)
Before area. 314
```

# Function Overloading: Generic Programming [3]

## Custom :method-combination

```
CL-USER> (defgeneric awesome-function (x)
           (:method-combination +))
#<STANDARD-GENERIC-FUNCTION AWESOME-FUNCTION (0)>
CL-USER> (defmethod awesome-function + ((x number))
           x)
#<STANDARD-METHOD AWESOME-FUNCTION + (NUMBER) {1006E16443}>
CL-USER> (awesome-function 2)
2
CL-USER> (typep 2 'number)
T
CL-USER> (typep 2 'integer)
T
CL-USER> (defmethod awesome-function + ((x integer))
           x)
#<STANDARD-METHOD AWESOME-FUNCTION + (INTEGER) {10072D6323}>
CL-USER> (awesome-function 2)
4
```

# OOP in Lisp

## Summary

OOP:

- Everything is an object.
- Objects interact with each other.
- Methods "belong" to objects.

Functional programming:

- Everything is a function.
- Functions interact with each other.
- Objects "belong" to (generic) functions.

OOP principles in Lisp:

- inheritance (`defclass`)
- encapsulation (`closures`)
- subtyping polymorphism (`defclass`)
- parametric polymorphism (generic functions)

# Contents

# Invoking Conditions

## define-condition, error

```
CL-USER> (error "oops, something went wrong...")
; #<COMMON-LISP:SIMPLE-ERROR "oops, something went wrong...">.
CL-USER> (define-condition input-not-a-number (simple-error)
           ((actual-input :initarg :actual-input
                          :reader actual-input
                          :initform nil))
           (:report (lambda (condition stream)
                      (format stream "~a is not a number!"
                              (actual-input condition)))))
INPUT-NOT-A-NUMBER
CL-USER> (let ((input (read)))
           (if (numberp input)
               input
               (error (make-condition 'input-not-a-number
                                      :actual-input input))))
asdf
; Evaluation aborted on #<COMMON-LISP-USER::INPUT-NOT-A-NUMBER>.
```

# Catching Conditions

## handler-case

```
CL-USER> (defparameter *result* nil)
         (let ((x (random 3)))
           (setf *result* (/ 123 x))
           (format t "new result is: ~a~%" *result*)
           (setf *result* 0)
           (format t "cleaned up: ~a~%" *result*))
; Evaluation aborted on #<DIVISION-BY-ZERO {1008D6E5B3}>.
CL-USER> (defparameter *result* nil)
         (let ((x (random 3)))
           (handler-case
               (progn (setf *result* (/ 123 x))
                      (format t "new result is: ~a~%" *result*)
                      (setf *result* 0)
                      (format t "cleaned up: ~a~%" *result*))
             (division-by-zero (error)
               (format t "~a~%" error)))
           (format t "Final result: ~a~%" *result*))
arithmetic error DIVISION-BY-ZERO signalled  Final result: NIL.
```

# Catching Conditions [2]

## unwind-protect

```
CL-USER> (defparameter *result* nil)
         (let ((x (random 3)))
           (handler-case
               (unwind-protect
                    (progn
                      (setf *result* (/ 123 x))
                      (format t "new result is: ~a~%" *result*))
                 (setf *result* 0)
                 (format t "cleaned up: ~a~%" *result*))
             (division-by-zero (error)
               (format t "~a~%" error)))
           (format t "final result: ~a~%" *result*))
cleaned up: 0
arithmetic error DIVISION-BY-ZERO signalled
final result: 0
```

# Contents

# Links

- Cool article by Paul Graham on programming languages:
  `http://www.paulgraham.com/avg.html`
- "Practical Common Lisp" failure handling chapter:
  `http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html`

# Organizational Info

- Assignment due: 07.11, Wednesday, 23:59 German time.
- Assignment points: 10 points.
- Next class: 08.11, 14:15.

# Q & A

Thanks for your attention!