Meta Object Protocols

António Menezes Leitão

March 17, 2021

1. CLOS
   - History
   - Generic Functions
   - Classes

CLOS-Common Lisp Object System

Roots

- 1980: Flavors - TI Explorer
- 1985: NewFlavors - Symbolics
- 1986: Loops (Lisp Object Oriented Programming System), CommonLoops - Xerox Lisp Machines
- 1986: ObjectLisp - LMI Lambda
- 1987: Common Objects - HP

Features

- Generic Functions, Multiple Dispatch.
- Classes, Multiple Inheritance.
- Meta-Objects, Protocols.

Function Call

Functional and Imperative Programming

```
(foo a b)
⇓
(call (function 'foo) a b)
```

Object-Oriented Programming - Single Dispatch

```
(foo a b) ⇔ a.foo(b)
⇓
(call (function 'foo (type-of a)) a b)
```

Object-Oriented Programming - Multiple Dispatch

```
(foo a b)
⇓
(call (function 'foo (type-of a) (type-of b)) a b)
```

Multiple Dispatch

Adding Entities

```
(defgeneric add (x y))
```

Multiple Dispatch

Adding Entities

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))
```

Multiple Dispatch

Adding Entities

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
```

Multiple Dispatch

Adding Entities

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
4
```

Multiple Dispatch

Adding Entities

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
4
> (add '(1 2) '(3 4))
```

Multiple Dispatch

Adding Entities

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
4
> (add '(1 2) '(3 4))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) (3 4)) of classes
(CONS CONS)
```

Multiple Dispatch

Adding Entities

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
4
> (add '(1 2) '(3 4))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) (3 4)) of classes
(CONS CONS)

(defmethod add ((x list) (y list))
  (mapcar #'add x y))
```

Multiple Dispatch

Adding Entities

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
4
> (add '(1 2) '(3 4))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) (3 4)) of classes
(CONS CONS)

(defmethod add ((x list) (y list))
  (mapcar #'add x y))

> (add '(1 2 3) '(4 5 6))
```

Multiple Dispatch

Adding Entities

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
4
> (add '(1 2) '(3 4))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) (3 4)) of classes
(CONS CONS)

(defmethod add ((x list) (y list))
  (mapcar #'add x y))

> (add '(1 2 3) '(4 5 6))
(5 7 9)
```

Multiple Dispatch

Adding Entities

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
4
> (add '(1 2) '(3 4))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) (3 4)) of classes
(CONS CONS)

(defmethod add ((x list) (y list))
  (mapcar #'add x y))

> (add '(1 2 3) '(4 5 6))
(5 7 9)
> (add '(1 (2 3)) '(4 (5 6)))
```

Multiple Dispatch

Adding Entities

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
4
> (add '(1 2) '(3 4))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) (3 4)) of classes
(CONS CONS)

(defmethod add ((x list) (y list))
  (mapcar #'add x y))

> (add '(1 2 3) '(4 5 6))
(5 7 9)
> (add '(1 (2 3)) '(4 (5 6)))
(5 (7 9))
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) 3) of classes
(CONS FIXNUM)
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod add ((x list) (y t))
  (add x (make-list (length x) :initial-element y)))
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod add ((x list) (y t))
  (add x (make-list (length x) :initial-element y)))

> (add '(1 2) 3)
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod add ((x list) (y t))
  (add x (make-list (length x) :initial-element y)))

> (add '(1 2) 3)
(4 5)
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod add ((x list) (y t))
  (add x (make-list (length x) :initial-element y)))

> (add '(1 2) 3)
(4 5)
> (add 1 '(2 3))
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod add ((x list) (y t))
  (add x (make-list (length x) :initial-element y)))

> (add '(1 2) 3)
(4 5)
> (add 1 '(2 3))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 (2 3)) of classes
(FIXNUM CONS)
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod add ((x list) (y t))
  (add x (make-list (length x) :initial-element y)))

> (add '(1 2) 3)
(4 5)
> (add 1 '(2 3))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 (2 3)) of classes
(FIXNUM CONS)

(defmethod add ((x t) (y list))
  (add (make-list (length y) :initial-element x) y))
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod add ((x list) (y t))
  (add x (make-list (length x) :initial-element y)))

> (add '(1 2) 3)
(4 5)
> (add 1 '(2 3))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 (2 3)) of classes
(FIXNUM CONS)

(defmethod add ((x t) (y list))
  (add (make-list (length y) :initial-element x) y))

> (add 1 '(2 3))
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod add ((x list) (y t))
  (add x (make-list (length x) :initial-element y)))

> (add '(1 2) 3)
(4 5)
> (add 1 '(2 3))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 (2 3)) of classes
(FIXNUM CONS)

(defmethod add ((x t) (y list))
  (add (make-list (length y) :initial-element x) y))

> (add 1 '(2 3))
(3 4)
```

Multiple Dispatch

Adding Entities

```
> (add #(1 2 3) #(4 5 6))
```

Multiple Dispatch

Adding Entities

```
> (add #(1 2 3) #(4 5 6))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2 3) #(4 5 6))
of classes (VECTOR VECTOR)
```

Multiple Dispatch

Adding Entities

```
> (add #(1 2 3) #(4 5 6))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2 3) #(4 5 6))
of classes (VECTOR VECTOR)

(defmethod add ((x vector) (y vector))
  (map 'vector #'add x y))
```

Multiple Dispatch

Adding Entities

```
> (add #(1 2 3) #(4 5 6))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2 3) #(4 5 6))
of classes (VECTOR VECTOR)

(defmethod add ((x vector) (y vector))
  (map 'vector #'add x y))

> (add #(1 2 3) #(4 5 6))
```

Multiple Dispatch

Adding Entities

```
> (add #(1 2 3) #(4 5 6))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2 3) #(4 5 6))
of classes (VECTOR VECTOR)

(defmethod add ((x vector) (y vector))
  (map 'vector #'add x y))

> (add #(1 2 3) #(4 5 6))
#(5 7 9)
```

Multiple Dispatch

Adding Entities

```
> (add #(1 2 3) #(4 5 6))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2 3) #(4 5 6))
of classes (VECTOR VECTOR)

(defmethod add ((x vector) (y vector))
  (map 'vector #'add x y))

> (add #(1 2 3) #(4 5 6))
#(5 7 9)
> (add #(1 2 3) 4)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2 3) 4) of
classes (VECTOR FIXNUM)
```

Multiple Dispatch

Adding Entities

```
> (add #(1 2 3) #(4 5 6))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2 3) #(4 5 6))
of classes (VECTOR VECTOR)

(defmethod add ((x vector) (y vector))
  (map 'vector #'add x y))

> (add #(1 2 3) #(4 5 6))
#(5 7 9)
> (add #(1 2 3) 4)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2 3) 4) of
classes (VECTOR FIXNUM)

(defmethod add ((x vector) (y number))
  (add x (make-array (list (length x)) :initial-element y)))
```

Multiple Dispatch

Adding Entities

```
> (add #(1 2 3) #(4 5 6))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2 3) #(4 5 6))
of classes (VECTOR VECTOR)

(defmethod add ((x vector) (y vector))
  (map 'vector #'add x y))

> (add #(1 2 3) #(4 5 6))
#(5 7 9)
> (add #(1 2 3) 4)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2 3) 4) of
classes (VECTOR FIXNUM)

(defmethod add ((x vector) (y number))
  (add x (make-array (list (length x)) :initial-element y)))

> (add #(1 2 3) 4)
```

Multiple Dispatch

Adding Entities

```
> (add #(1 2 3) #(4 5 6))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2 3) #(4 5 6))
of classes (VECTOR VECTOR)

(defmethod add ((x vector) (y vector))
  (map 'vector #'add x y))

> (add #(1 2 3) #(4 5 6))
#(5 7 9)
> (add #(1 2 3) 4)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2 3) 4) of
classes (VECTOR FIXNUM)

(defmethod add ((x vector) (y number))
  (add x (make-array (list (length x)) :initial-element y)))

> (add #(1 2 3) 4)
#(5 6 7)
```

Multiple Dispatch

Beware!

- Are you ready to repeat the entire set of methods for subtract, multiply, and divide?
- Multiple dispatch still requires a good design approach
- Instead of supporting all combinations of types for each possible arithmetic operation, we can use the same approach that is used in almost all programming languages: *promotion*

Multiple Dispatch

Beware!

- Are you ready to repeat the entire set of methods for subtract, multiply, and divide?
- Multiple dispatch still requires a good design approach
- Instead of supporting all combinations of types for each possible arithmetic operation, we can use the same approach that is used in almost all programming languages: *promotion*

Promotions

```
> (promote 1 1.5)
1.0
1.5
> (promote 1.5 1)
1.5
1.0
```

Multiple Dispatch

Adding Entities

```
(defun promoting-call (f x y)
  (multiple-value-bind (xp yp)
      (promote x y)
    (funcall f xp yp)))
```

Multiple Dispatch

Adding Entities

```
(defun promoting-call (f x y)
  (multiple-value-bind (xp yp)
      (promote x y)
    (funcall f xp yp)))

(defgeneric promote (x y)
  (:method ((x t) (y t))
    (error "No promotion for args (~S ~S) of classes (~S ~S)"
           x y
           (class-name (class-of x))
           (class-name (class-of y)))))
```

Multiple Dispatch

Adding Entities

```
(defun promoting-call (f x y)
  (multiple-value-bind (xp yp)
      (promote x y)
    (funcall f xp yp)))

(defgeneric promote (x y)
  (:method ((x t) (y t))
    (error "No promotion for args (~S ~S) of classes (~S ~S)"
           x y
           (class-name (class-of x))
           (class-name (class-of y)))))

(defgeneric add (x y)
  (:method ((x t) (y t))
    (promoting-call #'add x y)))
```

Multiple Dispatch

Adding Entities

```
(defun promoting-call (f x y)
  (multiple-value-bind (xp yp)
      (promote x y)
    (funcall f xp yp)))

(defgeneric promote (x y)
  (:method ((x t) (y t))
    (error "No promotion for args (~S ~S) of classes (~S ~S)"
           x y
           (class-name (class-of x))
           (class-name (class-of y)))))

(defgeneric add (x y)
  (:method ((x t) (y t))
    (promoting-call #'add x y)))


(defmethod add ((x number) (y number)) ...)
(defmethod add ((x list) (y list)) ...)
(defmethod add ((x vector) (y number)) ...)
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No promotion for args ((1 2) 3) of classes
(CONS FIXNUM)
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No promotion for args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod promote ((x list) (y t))
  (values x (make-list (length x) :initial-element y)))
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No promotion for args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod promote ((x list) (y t))
  (values x (make-list (length x) :initial-element y)))

(defmethod promote ((x t) (y list))
  (values (make-list (length y) :initial-element x) y))
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No promotion for args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod promote ((x list) (y t))
  (values x (make-list (length x) :initial-element y)))

(defmethod promote ((x t) (y list))
  (values (make-list (length y) :initial-element x) y))

(defmethod promote ((x vector) (y t))
  (values x (make-array (list (length x)) :initial-element y)))
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No promotion for args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod promote ((x list) (y t))
  (values x (make-list (length x) :initial-element y)))

(defmethod promote ((x t) (y list))
  (values (make-list (length y) :initial-element x) y))

(defmethod promote ((x vector) (y t))
  (values x (make-array (list (length x)) :initial-element y)))

(defmethod promote ((x t) (y vector))
  (values (make-array (list (length y)) :initial-element x) y))
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No promotion for args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod promote ((x list) (y t))
  (values x (make-list (length x) :initial-element y)))

(defmethod promote ((x t) (y list))
  (values (make-list (length y) :initial-element x) y))

(defmethod promote ((x vector) (y t))
  (values x (make-array (list (length x)) :initial-element y)))

(defmethod promote ((x t) (y vector))
  (values (make-array (list (length y)) :initial-element x) y))

> (add '(1 2) #(3 4)) ;;Can you guess the result?
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No promotion for args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod promote ((x list) (y t))
  (values x (make-list (length x) :initial-element y)))

(defmethod promote ((x t) (y list))
  (values (make-list (length y) :initial-element x) y))

(defmethod promote ((x vector) (y t))
  (values x (make-array (list (length x)) :initial-element y)))

(defmethod promote ((x t) (y vector))
  (values (make-array (list (length y)) :initial-element x) y))

> (add '(1 2) #(3 4)) ;;Can you guess the result?
(#(4 5) #(5 6))
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No promotion for args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod promote ((x list) (y t))
  (values x (make-list (length x) :initial-element y)))

(defmethod promote ((x t) (y list))
  (values (make-list (length y) :initial-element x) y))

(defmethod promote ((x vector) (y t))
  (values x (make-array (list (length x)) :initial-element y)))

(defmethod promote ((x t) (y vector))
  (values (make-array (list (length y)) :initial-element x) y))

> (add '(1 2) #(3 4)) ;;Can you guess the result?
(#(4 5) #(5 6))
> (add #(3 4) '(1 2)) ;;Can you guess the result?
```

Multiple Dispatch

Adding Entities

```
> (add '(1 2) 3)
No promotion for args ((1 2) 3) of classes
(CONS FIXNUM)

(defmethod promote ((x list) (y t))
  (values x (make-list (length x) :initial-element y)))

(defmethod promote ((x t) (y list))
  (values (make-list (length y) :initial-element x) y))

(defmethod promote ((x vector) (y t))
  (values x (make-array (list (length x)) :initial-element y)))

(defmethod promote ((x t) (y vector))
  (values (make-array (list (length y)) :initial-element x) y))

> (add '(1 2) #(3 4)) ;;Can you guess the result?
(#(4 5) #(5 6))
> (add #(3 4) '(1 2)) ;;Can you guess the result?
#((4 5) (5 6))
```

Multiple Dispatch

Improve the Design

- List and vectors are just different kinds of containers

Multiple Dispatch

Improve the Design

- List and vectors are just different kinds of containers

Adding Entities

```
(defmethod promote ((x vector) (y list))
  (values x (coerce y 'vector)))
```

Multiple Dispatch

Improve the Design

- List and vectors are just different kinds of containers

Adding Entities

```
(defmethod promote ((x vector) (y list))
  (values x (coerce y 'vector)))

(defmethod promote ((x list) (y vector))
  (values x (coerce y 'list)))
```

Multiple Dispatch

Improve the Design

- List and vectors are just different kinds of containers

Adding Entities

```
(defmethod promote ((x vector) (y list))
  (values x (coerce y 'vector)))

(defmethod promote ((x list) (y vector))
  (values x (coerce y 'list)))

> (add '(1 2) #(3 4)) ;;Can you guess?
```

Multiple Dispatch

Improve the Design

- List and vectors are just different kinds of containers

Adding Entities

```
(defmethod promote ((x vector) (y list))
  (values x (coerce y 'vector)))

(defmethod promote ((x list) (y vector))
  (values x (coerce y 'list)))

> (add '(1 2) #(3 4)) ;;Can you guess?
(4 6)
```

Multiple Dispatch

Improve the Design

- List and vectors are just different kinds of containers

Adding Entities

```
(defmethod promote ((x vector) (y list))
  (values x (coerce y 'vector)))

(defmethod promote ((x list) (y vector))
  (values x (coerce y 'list)))

> (add '(1 2) #(3 4)) ;;Can you guess?
(4 6)
> (add #(3 4) '(1 2)) ;;Can you guess?
```

Multiple Dispatch

Improve the Design

- List and vectors are just different kinds of containers

Adding Entities

```
(defmethod promote ((x vector) (y list))
  (values x (coerce y 'vector)))

(defmethod promote ((x list) (y vector))
  (values x (coerce y 'list)))

> (add '(1 2) #(3 4)) ;;Can you guess?
(4 6)
> (add #(3 4) '(1 2)) ;;Can you guess?
#(4 6)
```

Multiple Dispatch

Improve the Design

- List and vectors are just different kinds of containers

Adding Entities

```
(defmethod promote ((x vector) (y list))
  (values x (coerce y 'vector)))

(defmethod promote ((x list) (y vector))
  (values x (coerce y 'list)))

> (add '(1 2) #(3 4)) ;;Can you guess?
(4 6)
> (add #(3 4) '(1 2)) ;;Can you guess?
#(4 6)
> (add '(#(1 2) (3 4)) #((5 6) #(7 8))) ;;Can you guess?
```

Multiple Dispatch

Improve the Design

- List and vectors are just different kinds of containers

Adding Entities

```
(defmethod promote ((x vector) (y list))
  (values x (coerce y 'vector)))

(defmethod promote ((x list) (y vector))
  (values x (coerce y 'list)))

> (add '(1 2) #(3 4)) ;;Can you guess?
(4 6)
> (add #(3 4) '(1 2)) ;;Can you guess?
#(4 6)
> (add '(#(1 2) (3 4)) #((5 6) #(7 8))) ;;Can you guess?
(#(6 8) (10 12))
```

Multiple Dispatch

Improve the Design

- List and vectors are just different kinds of containers

Adding Entities

```
(defmethod promote ((x vector) (y list))
  (values x (coerce y 'vector)))

(defmethod promote ((x list) (y vector))
  (values x (coerce y 'list)))

> (add '(1 2) #(3 4)) ;;Can you guess?
(4 6)
> (add #(3 4) '(1 2)) ;;Can you guess?
#(4 6)
> (add '(#(1 2) (3 4)) #((5 6) #(7 8))) ;;Can you guess?
(#(6 8) (10 12))
> (add '(#(1 2) (3 4)) 5) ;;Can you guess?
```

Multiple Dispatch

Improve the Design

- List and vectors are just different kinds of containers

Adding Entities

```
(defmethod promote ((x vector) (y list))
  (values x (coerce y 'vector)))

(defmethod promote ((x list) (y vector))
  (values x (coerce y 'list)))

> (add '(1 2) #(3 4)) ;;Can you guess?
(4 6)
> (add #(3 4) '(1 2)) ;;Can you guess?
#(4 6)
> (add '(#(1 2) (3 4)) #((5 6) #(7 8))) ;;Can you guess?
(#(6 8) (10 12))
> (add '(#(1 2) (3 4)) 5) ;;Can you guess?
(#(6 7) (8 9))
```

Multiple Dispatch

Subtracting Entities

```
(defgeneric subtract (x y)
  (:method ((x t) (y t))
    (promoting-call #'subtract x y)))
```

Multiple Dispatch

Subtracting Entities

```
(defgeneric subtract (x y)
  (:method ((x t) (y t))
    (promoting-call #'subtract x y)))

(defmethod subtract ((x number) (y number))
  (- x y))
```

Multiple Dispatch

Subtracting Entities

```
(defgeneric subtract (x y)
  (:method ((x t) (y t))
    (promoting-call #'subtract x y)))

(defmethod subtract ((x number) (y number))
  (- x y))

(defmethod subtract ((x list) (y list))
  (mapcar #'subtract x y))
```

Multiple Dispatch

Subtracting Entities

```
(defgeneric subtract (x y)
  (:method ((x t) (y t))
    (promoting-call #'subtract x y)))

(defmethod subtract ((x number) (y number))
  (- x y))

(defmethod subtract ((x list) (y list))
  (mapcar #'subtract x y))

(defmethod subtract ((x vector) (y vector))
  (map 'vector #'subtract x y))
```

Multiple Dispatch

Subtracting Entities

```
(defgeneric subtract (x y)
  (:method ((x t) (y t))
    (promoting-call #'subtract x y)))

(defmethod subtract ((x number) (y number))
  (- x y))

(defmethod subtract ((x list) (y list))
  (mapcar #'subtract x y))

(defmethod subtract ((x vector) (y vector))
  (map 'vector #'subtract x y))

> (subtract 1 3)
```

Multiple Dispatch

Subtracting Entities

```
(defgeneric subtract (x y)
  (:method ((x t) (y t))
    (promoting-call #'subtract x y)))

(defmethod subtract ((x number) (y number))
  (- x y))

(defmethod subtract ((x list) (y list))
  (mapcar #'subtract x y))

(defmethod subtract ((x vector) (y vector))
  (map 'vector #'subtract x y))

> (subtract 1 3)
-2
```

Multiple Dispatch

Subtracting Entities

```
(defgeneric subtract (x y)
  (:method ((x t) (y t))
    (promoting-call #'subtract x y)))

(defmethod subtract ((x number) (y number))
  (- x y))

(defmethod subtract ((x list) (y list))
  (mapcar #'subtract x y))

(defmethod subtract ((x vector) (y vector))
  (map 'vector #'subtract x y))

> (subtract 1 3)
-2
> (subtract '(1 2 3) '(4 5 6))
```

Multiple Dispatch

Subtracting Entities

```
(defgeneric subtract (x y)
  (:method ((x t) (y t))
    (promoting-call #'subtract x y)))

(defmethod subtract ((x number) (y number))
  (- x y))

(defmethod subtract ((x list) (y list))
  (mapcar #'subtract x y))

(defmethod subtract ((x vector) (y vector))
  (map 'vector #'subtract x y))

> (subtract 1 3)
-2
> (subtract '(1 2 3) '(4 5 6))
(-3 -3 -3)
```

Multiple Dispatch

Subtracting Entities

```
(defgeneric subtract (x y)
  (:method ((x t) (y t))
    (promoting-call #'subtract x y)))

(defmethod subtract ((x number) (y number))
  (- x y))

(defmethod subtract ((x list) (y list))
  (mapcar #'subtract x y))

(defmethod subtract ((x vector) (y vector))
  (map 'vector #'subtract x y))

> (subtract 1 3)
-2
> (subtract '(1 2 3) '(4 5 6))
(-3 -3 -3)
> (subtract '(1 2) 3)
```

Multiple Dispatch

Subtracting Entities

```
(defgeneric subtract (x y)
  (:method ((x t) (y t))
    (promoting-call #'subtract x y)))

(defmethod subtract ((x number) (y number))
  (- x y))

(defmethod subtract ((x list) (y list))
  (mapcar #'subtract x y))

(defmethod subtract ((x vector) (y vector))
  (map 'vector #'subtract x y))

> (subtract 1 3)
-2
> (subtract '(1 2 3) '(4 5 6))
(-3 -3 -3)
> (subtract '(1 2) 3)
(-2 -1)
```

Multiple Dispatch

Subtracting Entities

```
(defgeneric subtract (x y)
  (:method ((x t) (y t))
    (promoting-call #'subtract x y)))

(defmethod subtract ((x number) (y number))
  (- x y))

(defmethod subtract ((x list) (y list))
  (mapcar #'subtract x y))

(defmethod subtract ((x vector) (y vector))
  (map 'vector #'subtract x y))

> (subtract 1 3)
-2
> (subtract '(1 2 3) '(4 5 6))
(-3 -3 -3)
> (subtract '(1 2) 3)
(-2 -1)
> (subtract 1 '(2 3))
```

Multiple Dispatch

Subtracting Entities

```
(defgeneric subtract (x y)
  (:method ((x t) (y t))
    (promoting-call #'subtract x y)))

(defmethod subtract ((x number) (y number))
  (- x y))

(defmethod subtract ((x list) (y list))
  (mapcar #'subtract x y))

(defmethod subtract ((x vector) (y vector))
  (map 'vector #'subtract x y))

> (subtract 1 3)
-2
> (subtract '(1 2 3) '(4 5 6))
(-3 -3 -3)
> (subtract '(1 2) 3)
(-2 -1)
> (subtract 1 '(2 3))
(-1 -2)
```

Multiple Dispatch

Subtracting Entities

```
> (subtract '(1 2) #(3 4))
```

Multiple Dispatch

Subtracting Entities

```
> (subtract '(1 2) #(3 4))
(-2 -2)
```

Multiple Dispatch

Subtracting Entities

```
> (subtract '(1 2) #(3 4))
(-2 -2)
> (subtract #(3 4) '(1 2))
```

Multiple Dispatch

Subtracting Entities

```
> (subtract '(1 2) #(3 4))
(-2 -2)
> (subtract #(3 4) '(1 2))
#(2 2)
```

Multiple Dispatch

Subtracting Entities

```
> (subtract '(1 2) #(3 4))
(-2 -2)
> (subtract #(3 4) '(1 2))
#(2 2)
> (subtract '(#(1 2) (3 4)) 5)
```

Multiple Dispatch

Subtracting Entities

```
> (subtract '(1 2) #(3 4))
(-2 -2)
> (subtract #(3 4) '(1 2))
#(2 2)
> (subtract '(#(1 2) (3 4)) 5)
(#(-4 -3) (-2 -1))
```

Multiple Dispatch

Subtracting Entities

```
> (subtract '(1 2) #(3 4))
(-2 -2)
> (subtract #(3 4) '(1 2))
#(2 2)
> (subtract '(#(1 2) (3 4)) 5)
(#(-4 -3) (-2 -1))
> (subtract 5 '(#(1 2) (3 4)))
```

Multiple Dispatch

Subtracting Entities

```
> (subtract '(1 2) #(3 4))
(-2 -2)
> (subtract #(3 4) '(1 2))
#(2 2)
> (subtract '(#(1 2) (3 4)) 5)
(#(-4 -3) (-2 -1))
> (subtract 5 '(#(1 2) (3 4)))
(#(4 3) (2 1))
```

Multiple Dispatch

Subtracting Entities

```
> (subtract '(1 2) #(3 4))
(-2 -2)
> (subtract #(3 4) '(1 2))
#(2 2)
> (subtract '(#(1 2) (3 4)) 5)
(#(-4 -3) (-2 -1))
> (subtract 5 '(#(1 2) (3 4)))
(#(4 3) (2 1))
```

Improve the Design

- Promotions seem to be symmetric
- We should only define one direction
- And the other should be automatically taken care

Instance Specialization

Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

Instance Specialization

Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

fact

```
(defgeneric fact (n))
```

Instance Specialization

Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

fact

```
(defgeneric fact (n))

(defmethod fact ((n integer)) ;;there is no class for n > 0
  (* n (fact (1- n))))
```

Instance Specialization

Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

fact

```
(defgeneric fact (n))

(defmethod fact ((n integer)) ;;there is no class for n > 0
  (* n (fact (1- n))))

(defmethod fact ((n (eql 0))) ;;but we can specialize on 0
  1)
```

Instance Specialization

Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

fact

```
(defgeneric fact (n))

(defmethod fact ((n integer)) ;;there is no class for n > 0
  (* n (fact (1- n))))

(defmethod fact ((n (eql 0))) ;;but we can specialize on 0
  1)

> (fact 5)
```

Instance Specialization

Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

fact

```
(defgeneric fact (n))

(defmethod fact ((n integer)) ;;there is no class for n > 0
  (* n (fact (1- n))))

(defmethod fact ((n (eql 0))) ;;but we can specialize on 0
  1)

> (fact 5)
120
```

Instance Specialization

foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{if } x = 5! \\ 0 & \text{otherwise} \end{cases}$$

Instance Specialization

foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{if } x = 5! \\ 0 & \text{otherwise} \end{cases}$$

foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)
```

Instance Specialization

foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{if } x = 5! \\ 0 & \text{otherwise} \end{cases}$$

foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)
```

Instance Specialization

foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{if } x = 5! \\ 0 & \text{otherwise} \end{cases}$$

foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)

> (foobar 34)
```

Instance Specialization

foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{if } x = 5! \\ 0 & \text{otherwise} \end{cases}$$

foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)

> (foobar 34)
0
```

Instance Specialization

foobar

$$\mathrm{foobar}(x) = \begin{cases} 1 & \text{if } x = 5! \\ 0 & \text{otherwise} \end{cases}$$

foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)

> (foobar 34)
0
> (foobar (fact 5))
```

Instance Specialization

foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{if } x = 5! \\ 0 & \text{otherwise} \end{cases}$$

foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)

> (foobar 34)
0
> (foobar (fact 5))
1
```

Instance Specialization

foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{if } x = 5! \\ 0 & \text{otherwise} \end{cases}$$

foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)

> (foobar 34)
0
> (foobar (fact 5))
1
> (foobar 120)
```

Instance Specialization

foobar

$$\mathrm{foobar}(x) = \begin{cases} 1 & \text{if } x = 5! \\ 0 & \text{otherwise} \end{cases}$$

foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)

> (foobar 34)
0
> (foobar (fact 5))
1
> (foobar 120)
1
```

Instance Specialization

Fibonacci

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{otherwise} \end{cases}$$

Instance Specialization

Fibonacci

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{otherwise} \end{cases}$$

fib

```
(defgeneric fib (n))

(defmethod fib ((n (eql 0)))
  0)

(defmethod fib ((n (eql 1)))
  1)

(defmethod fib ((n number))
  (+ (fib (- n 1)) (fib (- n 2))))
```

Method Combination

Example

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Method Combination

Example

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Memoization

Method Combination

Example

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Memoization

```
(let ((cached-results (make-hash-table)))
```

Method Combination

Example

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Memoization

```
(let ((cached-results (make-hash-table)))
  (defmethod fib :around ((n number))
```

Method Combination

Example

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Memoization

```
(let ((cached-results (make-hash-table)))
  (defmethod fib :around ((n number))
    (or (gethash n cached-results)
```

Method Combination

Example

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Memoization

```
(let ((cached-results (make-hash-table)))
  (defmethod fib :around ((n number))
    (or (gethash n cached-results)
      (setf (gethash n cached-results)
            (call-next-method)))))
```

Method Combination

Example

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Memoization

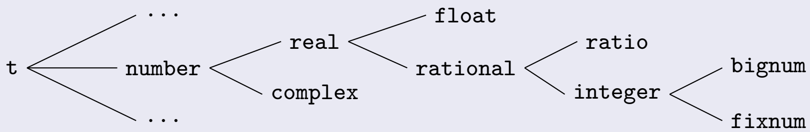
```
(let ((cached-results (make-hash-table)))
  (defmethod fib :around ((n number))
    (or (gethash n cached-results)
      (setf (gethash n cached-results)
            (call-next-method)))))
```

Example

```
CL-USER> (time (fib 40))
; real time  10 msec
102334155
```

Method Combination

Numerical Types Hierarchy

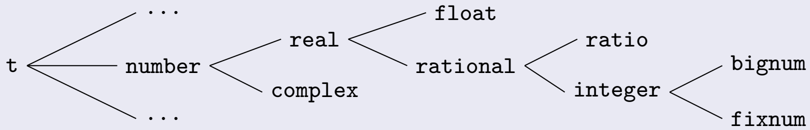


explain

```lisp
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))
```

Method Combination

Numerical Types Hierarchy



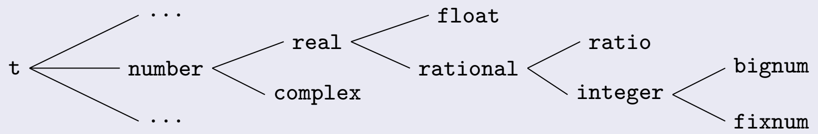
explain

```
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))

> (explain 123)
```

Method Combination

Numerical Types Hierarchy



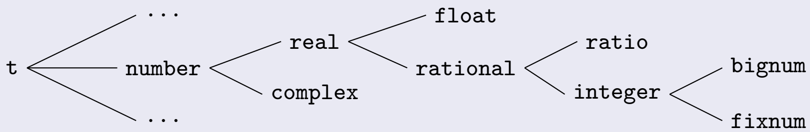
explain

```
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))

> (explain 123)
123 is a fixnum
```

Method Combination

Numerical Types Hierarchy



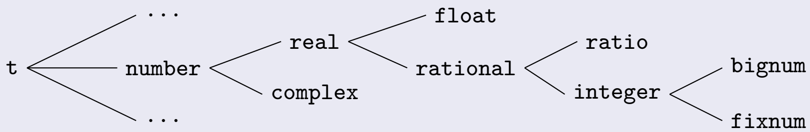
explain

```
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))

> (explain 123)
123 is a fixnum
> (explain "Hi")
```

Method Combination

Numerical Types Hierarchy



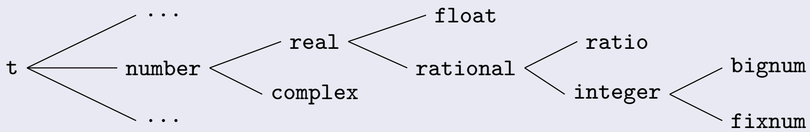
explain

```lisp
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))

> (explain 123)
123 is a fixnum
> (explain "Hi")
"Hi" is a string
```

Method Combination

Numerical Types Hierarchy



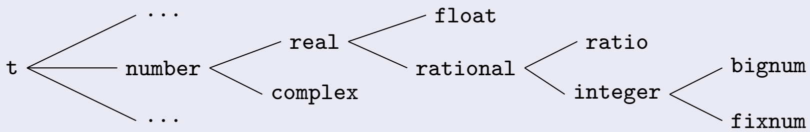
explain

```
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))

> (explain 123)
123 is a fixnum
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
```

Method Combination

Numerical Types Hierarchy



explain

```
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))

> (explain 123)
123 is a fixnum
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))

> (explain 123)
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))

> (explain 123)
The number 123 is a fixnum (in binary, is 1111011)
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))

> (explain 123)
The number 123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))

> (explain 123)
The number 123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))

> (explain 123)
The number 123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
```

Method Combination

explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))

> (explain 123)
The number 123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
The number 1/3 is a rational
```

Generic Functions

Generic Function Application

1. Computes the **effective method**.
2. If it exists, calls the **effective method** with the same arguments of the generic function call.
3. If it does not exist, calls `no-applicable-method` using, as arguments, the original generic function and the arguments of the original call.

Effective Method Computation

1. **Selects** the **applicable methods**.
2. **Sorts** applicable methods by precedence order.
3. **Combines** applicable methods, producing the **effective method**.

Generic Functions

Applicable Methods

- Given a generic function and required arguments $a_0, \ldots, a_n$, an **applicable method** is a method whose parameter specializers $p_0, \ldots, p_n$ are satisfied by their corresponding arguments.
- A parameter specializer p_i is satisfied by their corresponding argument a_i if `(typep` a_i `'`p_i`)`.

Applicable Methods for `(explain 123)`

```
(defmethod explain ((entity fixnum))
  (format t "~S is a fixnum" entity))
(defmethod explain ((entity rational))
  (format t "~S is a rational" entity))
(defmethod explain :before ((entity number))
  (format t "The number "))
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))
```

Generic Functions

Qualifiers

- Each method can have zero or more *qualifiers*.
- Each qualifier can be any object except a list (so that qualifiers can be distinguished from parameter lists).
- A ***standard* method combination** distinguishes:

  Primary Methods: non-qualified methods.

  Auxiliary Methods: methods qualified with the symbol `:before`, `:after`, or `:around`.
- Other method combinations can use other categories.
- New method combinations can be defined.

Generic Functions

Primary methods applicable to (explain 123)

```
(defmethod explain ((entity fixnum))
  (format t "~S is a fixnum" entity))

(defmethod explain ((entity rational))
  (format t "~S is a rational" entity))
```

Auxiliary methods applicable to (explain 123)

```
(defmethod explain :before ((entity number))
  (format t "The number "))

(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))
```

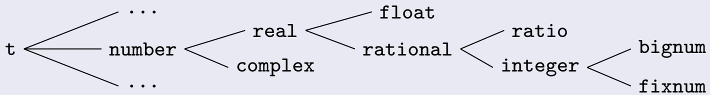
Generic Functions

Sorting the Applicable Methods

- Sorts applicable methods by precedence order from most specific to least specific.
- Given two applicable methods:
  1. Their parameter specializers are examined in order (by default, from left to right).
  2. When two specializers differ, the highest precedence method is the one whose parameter specializer occurs first in the **class precedence list** of the corresponding argument.
  3. When one specializer is an instance specializer (`(eql` *`object`*`)`), the highest precedence method is the one whose parameter contains that specializer.
  4. When all specializers are identical, the two methods must have different qualifiers and either one can be selected to precede the other.

Generic Functions

Class Precedence List for 123



fixnum, integer, rational, real, number, t

Sorted Applicable Methods for (explain 123)

```
(defmethod explain ((entity fixnum))
  (format t "~S is a fixnum" entity))

(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

(defmethod explain ((entity rational))
  (format t "~S is a rational" entity))

(defmethod explain :before ((entity number))
  (format t "The number "))
```

Generic Functions

Method Combination

- Happens after selecting and sorting applicable methods.
- Creates the **effective method** that will be applied to the generic function arguments.
- There are many pre-defined method combinations (known as *method combination types*):

Simple: `append`, `nconc`, `list`, `progn`, `max`, `min`, `+`, `and`, `or`
Requires using the same method combination type in the generic function and all methods of the generic function.

Standard: `standard`
Used by default when nothing is specified on the generic function. Implicitly used when the generic function is not specified.

Generic Functions

Standard Method Combination

- Primary methods define the main action of the effective method.
    - Only the most specific is (automatically) executed.
    - It can execute the next most specific method using `call-next-method`.
- Auxiliary methods modify that behavior:

`:before`	Methods called *before* primary methods.
`:after`	Methods called *after* primary methods.
`:around`	Method called *instead* of other applicable methods but that can call some of them by using `call-next-method`.

Generic Functions

Standard Method Combination

If there are no applicable `:around` methods:

1. All `:before` methods are called, from most specific to least specific, and their values are ignored.
2. The most specific primary method is called.
   - If that method calls `call-next-method`, the next most specific method is called and their values are returned to the caller.
   - The values returned by the most specific primary method become the values returned by the generic function call.
3. All `:after` methods are called, from least specific to most specific, and their values are ignored.

Generic Functions

Standard Method Combination

If there are applicable `:around` methods, the most specific one is called. If that method calls `call-next-method`:

1. If there are more applicable `:around` methods, the next most specific `:around` method is called.
2. If there are no more applicable `:around` methods:
   1. All `:before` methods are called, from most specific to least specific, and their values are ignored.
   2. The most specific primary method is called.
      - If that method calls `call-next-method`, the next most specific method is called and their values are returned to the caller.
      - The values returned by the most specific primary method become the values returned by the generic function call.
   3. All `:after` methods are called, from least specific to most specific, and their values are ignored.

Generic Functions

Standard Method Combination

- `call-next-method` might be called
  - without arguments: it uses the same arguments that were used in the method call.
  - with arguments: it uses the provided arguments but these should produce the same ordered sequence of applicable methods that was produced by the arguments used in the method call.
- If there are no more applicable methods, `call-next-method` calls the generic function `no-next-method` using, as arguments:
  - The generic function that contains the method that called `call-next-method`.
  - The method that called `call-next-method`.
  - The arguments that were used for calling `call-next-method`.
- `next-method-p` can be used in a method to determine whether a next applicable method exists.

Generic Functions

Applicable methods to (explain 123) according to standard combination

```
(defmethod explain :before ((entity number))
  (format t "The number "))

(defmethod explain ((entity fixnum))
  (format t "~S is a fixnum" entity))

(defmethod explain ((entity rational))
  (format t "~S is a rational" entity))

(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))
```

Effective method for (explain 123) (simplified)

```
(lambda (entity)
  (format t "The number ")
  (format t "~S is a fixnum" entity)
  (format t " (in binary, is ~B)" entity))
```

Generic Functions

Simple Method Combination

Primary Methods: methods qualified with the combination type (`append`, `nconc`, `list`, `progn`, `max`, `min`, `+`, `and`, `or`).

Auxiliary Methods: methods qualified with `:around`.

Simple Method Combination

If there are no applicable `:around` methods:

1. The effective method is the application of the combination type (the operator) to the results of calling all the applicable primary methods sorted by precedence order.

Generic Functions

Simple Method Combination

If there are applicable `:around` methods, the most specific one is called. If that method calls `call-next-method`:

1. If there are more applicable `:around` methods, the next most specific `:around` method is called.
2. If there are no more applicable `:around` methods:
   1. The effective method is the application of the combination type (the operator) to the results of calling all the applicable primary methods sorted by precedence order.

Generic Functions

Simple Method Combination

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))
```

Generic Functions

Simple Method Combination

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")
```

Generic Functions

Simple Method Combination

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")

> (what-are-you? 123)
```

Generic Functions

Simple Method Combination

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")
```

Generic Functions

Simple Method Combination

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
```

Generic Functions

Simple Method Combination

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")
```

Generic Functions

Simple Method Combination

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
```

Generic Functions

Simple Method Combination

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER")
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER" "I am a RATIO")
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER" "I am a RATIO")

(defmethod what-are-you? list ((obj (eql 1)))
  "I am THE SPECIAL ONE")
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER" "I am a RATIO")

(defmethod what-are-you? list ((obj (eql 1)))
  "I am THE SPECIAL ONE")

> (what-are-you? 0)
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER" "I am a RATIO")

(defmethod what-are-you? list ((obj (eql 1)))
  "I am THE SPECIAL ONE")

> (what-are-you? 0)
("I am a NUMBER" "I am a FIXNUM")
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER" "I am a RATIO")

(defmethod what-are-you? list ((obj (eql 1)))
  "I am THE SPECIAL ONE")

> (what-are-you? 0)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1)
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER" "I am a RATIO")

(defmethod what-are-you? list ((obj (eql 1)))
  "I am THE SPECIAL ONE")

> (what-are-you? 0)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1)
("I am a NUMBER" "I am a FIXNUM" "I am THE SPECIAL ONE")
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj null))
  "I am a NULL")

(defmethod what-are-you? list ((obj symbol))
  "I am a SYMBOL")

(defmethod what-are-you? list ((obj list))
  "I am a LIST")
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj null))
  "I am a NULL")

(defmethod what-are-you? list ((obj symbol))
  "I am a SYMBOL")

(defmethod what-are-you? list ((obj list))
  "I am a LIST")

> (what-are-you? 'hi)
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj null))
  "I am a NULL")

(defmethod what-are-you? list ((obj symbol))
  "I am a SYMBOL")

(defmethod what-are-you? list ((obj list))
  "I am a LIST")

> (what-are-you? 'hi)
("I am a SYMBOL")
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj null))
  "I am a NULL")

(defmethod what-are-you? list ((obj symbol))
  "I am a SYMBOL")

(defmethod what-are-you? list ((obj list))
  "I am a LIST")

> (what-are-you? 'hi)
("I am a SYMBOL")

> (what-are-you? '(1 2 3))
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj null))
  "I am a NULL")

(defmethod what-are-you? list ((obj symbol))
  "I am a SYMBOL")

(defmethod what-are-you? list ((obj list))
  "I am a LIST")

> (what-are-you? 'hi)
("I am a SYMBOL")

> (what-are-you? '(1 2 3))
("I am a LIST")
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj null))
  "I am a NULL")

(defmethod what-are-you? list ((obj symbol))
  "I am a SYMBOL")

(defmethod what-are-you? list ((obj list))
  "I am a LIST")

> (what-are-you? 'hi)
("I am a SYMBOL")

> (what-are-you? '(1 2 3))
("I am a LIST")

> (what-are-you? '())
```

Generic Functions

Simple Method Combination

```
(defmethod what-are-you? list ((obj null))
  "I am a NULL")

(defmethod what-are-you? list ((obj symbol))
  "I am a SYMBOL")

(defmethod what-are-you? list ((obj list))
  "I am a LIST")

> (what-are-you? 'hi)
("I am a SYMBOL")

> (what-are-you? '(1 2 3))
("I am a LIST")

> (what-are-you? '())
("I am a LIST" "I am a SYMBOL" "I am a NULL")
```

Generic Functions

User-Defined Method Combination

```
(define-method-combination list ()
         ((methods (list)))
    `(list ,@(mapcar (lambda (method)
                       `(call-method ,method))
                     methods)))
```

Generic Functions

User-Defined Method Combination

```
(define-method-combination list ()
        ((methods (list)))
   `(list ,@(mapcar (lambda (method)
                      `(call-method ,method))
                    methods)))
```

Definition

- Name of the method combination.

Generic Functions

User-Defined Method Combination

```
(define-method-combination list ()
        ((methods (list)))
   `(list ,@(mapcar (lambda (method)
                      `(call-method ,method))
                    methods)))
```

Definition

- Name of the method combination.
- Parameters of the method combination (e.g., sorting order for applicable methods).

Generic Functions

User-Defined Method Combination

```
(define-method-combination list ()
          ((methods (list)))
   `(list ,@(mapcar (lambda (method)
                       `(call-method ,method))
                    methods)))
```

Definition

- Name of the method combination.
- Parameters of the method combination (e.g., sorting order for applicable methods).
- Local variable to contain methods whose qualifiers ...

Generic Functions

User-Defined Method Combination

```
(define-method-combination list ()
         ((methods (list)))
    `(list ,@(mapcar (lambda (method)
                       `(call-method ,method))
                     methods)))
```

Definition

- Name of the method combination.
- Parameters of the method combination (e.g., sorting order for applicable methods).
- Local variable to contain methods whose qualifiers ...
- ...satisfy this pattern

Generic Functions

User-Defined Method Combination

```
(define-method-combination list ()
         ((methods (list)))
   `(list ,@(mapcar (lambda (method)
                      `(call-method ,method))
                    methods)))
```

Definition

- Name of the method combination.
- Parameters of the method combination (e.g., sorting order for applicable methods).
- Local variable to contain methods whose qualifiers ...
- ...satisfy this pattern
- Calls each applicable method in the effective method

Generic Functions

Standard Method Combination

```
(define-method-combination standard ()
         ((around (:around))
          (before (:before))
          (primary () :required t)
          (after (:after)))
   (flet ((call-methods (methods)
            (mapcar (lambda (method)
                      `(call-method ,method))
                    methods)))
     (let ((form (if (or before after (rest primary))
                     `(multiple-value-prog1
                        (progn ,@(call-methods before)
                               (call-method ,(first primary)
                                            ,(rest primary)))
                        ,@(call-methods (reverse after)))
                     `(call-method ,(first primary)))))
       (if around
           `(call-method ,(first around)
                         (,@(rest around)
                          (make-method ,form)))
           form))))
```

Classes

Class definition with defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))
```

Classes

Class definition with defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))
```

Classes

Class definition with defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))
```

Classes

Class definition with defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))
```

Classes

Class definition with defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))
```

Classes

Class definition with defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))
```

Classes

Class definition with defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))

(defmethod set-foo-slot1 ((obj foo) new-value)
  (setf (slot-value obj 'slot1) new-value))
```

Classes

Class definition with defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))

(defmethod set-foo-slot1 ((obj foo) new-value)
  (setf (slot-value obj 'slot1) new-value))
```

Classes

Class definition with defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))

(defmethod set-foo-slot1 ((obj foo) new-value)
  (setf (slot-value obj 'slot1) new-value))
```

Classes

Class definition with defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))

(defmethod set-foo-slot1 ((obj foo) new-value)
  (setf (slot-value obj 'slot1) new-value))

(defmethod foo-slot2 ((obj foo))
  (slot-value obj 'slot2))

(defmethod (setf foo-slot2) (new-value (obj foo))
  (setf (slot-value obj 'slot2) new-value))
```

Classes

Class definition with defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))

(defmethod set-foo-slot1 ((obj foo) new-value)
  (setf (slot-value obj 'slot1) new-value))

(defmethod foo-slot2 ((obj foo))
  (slot-value obj 'slot2))

(defmethod (setf foo-slot2) (new-value (obj foo))
  (setf (slot-value obj 'slot2) new-value))
```

Classes

Class definition with defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))

(defmethod set-foo-slot1 ((obj foo) new-value)
  (setf (slot-value obj 'slot1) new-value))

(defmethod foo-slot2 ((obj foo))
  (slot-value obj 'slot2))

(defmethod (setf foo-slot2) (new-value (obj foo))
  (setf (slot-value obj 'slot2) new-value))
```

CLOS-Common Lisp Object System

Example

```
(defclass shape ()
  ())

(defclass device ()
  ())
```

CLOS-Common Lisp Object System

Example

```
(defclass shape ()
  ())

(defclass device ()
  ())

(defgeneric draw (shape device))

(defmethod draw ((s shape) (d device))
  (format t "draw what where?~%"))
```

CLOS-Common Lisp Object System

Example

```
(defclass shape ()
  ())

(defclass device ()
  ())

(defgeneric draw (shape device))

(defmethod draw ((s shape) (d device))
  (format t "draw what where?~%"))

(defclass line (shape)
  ())

(defclass circle (shape)
  ())
```

CLOS-Common Lisp Object System

Example

```
(defclass shape ()
  ())

(defclass device ()
  ())

(defgeneric draw (shape device))

(defmethod draw ((s shape) (d device))
  (format t "draw what where?~%"))

(defclass line (shape)
  ())

(defclass circle (shape)
  ())

(defclass screen (device)
  ())

(defclass printer (device)
  ())
```

CLOS-Common Lisp Object System

Multiple Dispatch

```
(defmethod draw ((s line) (d device))
  (format t "draw a line where?~%"))

(defmethod draw ((s circle) (d device))
  (format t "draw a circle where?~%"))
```

CLOS-Common Lisp Object System

Multiple Dispatch

```
(defmethod draw ((s line) (d device))
  (format t "draw a line where?~%"))

(defmethod draw ((s circle) (d device))
  (format t "draw a circle where?~%"))

(defmethod draw ((s shape) (d screen))
  (format t "draw what on screen?~%"))

(defmethod draw ((s shape) (d printer))
  (format t "draw what on printer?~%"))
```

CLOS-Common Lisp Object System

Multiple Dispatch

```
(defmethod draw ((s line) (d screen))
  (format t "drawing a line on screen!~%"))

(defmethod draw ((s circle) (d screen))
  (format t "drawing a circle on screen!~%"))

(defmethod draw ((s line) (d printer))
  (format t "drawing a line on printer!~%"))

(defmethod draw ((s circle) (d printer))
  (format t "drawing a circle on printer!~%"))
```

CLOS-Common Lisp Object System

Multiple Dispatch

```
(let ((devices (list (make-instance 'screen)
                     (make-instance 'printer)))
```

CLOS-Common Lisp Object System

Multiple Dispatch

```
(let ((devices (list (make-instance 'screen)
                     (make-instance 'printer)))
      (shapes (list (make-instance 'line)
                    (make-instance 'circle))))
```

CLOS-Common Lisp Object System

Multiple Dispatch

```
(let ((devices (list (make-instance 'screen)
                     (make-instance 'printer)))
      (shapes (list (make-instance 'line)
                    (make-instance 'circle))))

  (dolist (device devices)
    (dolist (shape shapes)
      (draw shape device))))
```

CLOS-Common Lisp Object System

Multiple Dispatch

```
(let ((devices (list (make-instance 'screen)
                     (make-instance 'printer)))
      (shapes (list (make-instance 'line)
                    (make-instance 'circle))))

  (dolist (device devices)
    (dolist (shape shapes)
      (draw shape device))))

drawing a line on screen!
drawing a circle on screen!
drawing a line on printer!
drawing a circle on printer!
```

CLOS-Common Lisp Object System

Slots

```
(defclass 2d-position ()
  ((x :initarg :x)
   (y :initarg :y)))
```

CLOS-Common Lisp Object System

Slots

```
(defclass 2d-position ()
  ((x :initarg :x)
   (y :initarg :y)))

(defclass line (shape)
  ((origin :initarg :origin :accessor line-origin)
   (end :initarg :end :accessor line-end)))

(defclass circle (shape)
  ((center :initarg :center :accessor circle-center)
   (radius :initarg :radius :accessor circle-radius :initform 1)))
```

CLOS-Common Lisp Object System

Slots

```
(defclass 2d-position ()
  ((x :initarg :x)
   (y :initarg :y)))

(defclass line (shape)
  ((origin :initarg :origin :accessor line-origin)
   (end :initarg :end :accessor line-end)))

(defclass circle (shape)
  ((center :initarg :center :accessor circle-center)
   (radius :initarg :radius :accessor circle-radius :initform 1)))

> (make-instance 'circle
    :center (make-instance '2d-position :x 10 :y 30)
    :radius 5)
```

CLOS-Common Lisp Object System

Slots

```
(defclass 2d-position ()
  ((x :initarg :x)
   (y :initarg :y)))

(defclass line (shape)
  ((origin :initarg :origin :accessor line-origin)
   (end :initarg :end :accessor line-end)))

(defclass circle (shape)
  ((center :initarg :center :accessor circle-center)
   (radius :initarg :radius :accessor circle-radius :initform 1)))

> (make-instance 'circle
    :center (make-instance '2d-position :x 10 :y 30)
    :radius 5)
#<CIRCLE @ #x71641c1a>
```

CLOS-Common Lisp Object System

Slots

```
(defclass 2d-position ()
  ((x :initarg :x)
   (y :initarg :y)))

(defclass line (shape)
  ((origin :initarg :origin :accessor line-origin)
   (end :initarg :end :accessor line-end)))

(defclass circle (shape)
  ((center :initarg :center :accessor circle-center)
   (radius :initarg :radius :accessor circle-radius :initform 1)))

> (make-instance 'circle
    :center (make-instance '2d-position :x 10 :y 30)
    :radius 5)
#<CIRCLE @ #x71641c1a>

> (circle-radius (make-instance 'circle))
```

CLOS-Common Lisp Object System

Slots

```
(defclass 2d-position ()
  ((x :initarg :x)
   (y :initarg :y)))

(defclass line (shape)
  ((origin :initarg :origin :accessor line-origin)
   (end :initarg :end :accessor line-end)))

(defclass circle (shape)
  ((center :initarg :center :accessor circle-center)
   (radius :initarg :radius :accessor circle-radius :initform 1)))

> (make-instance 'circle
    :center (make-instance '2d-position :x 10 :y 30)
    :radius 5)
#<CIRCLE @ #x71641c1a>

> (circle-radius (make-instance 'circle))
1
```

CLOS-Common Lisp Object System

Mixins

```
(defclass color-mixin ()
  ((color :initarg :color :accessor color)))
```

CLOS-Common Lisp Object System

Mixins

```
(defclass color-mixin ()
  ((color :initarg :color :accessor color)))

(defmethod draw :around ((s color-mixin) (d device))
  (let ((previous-color (color d)))
    (setf (color d) (color s))
    (unwind-protect
       (call-next-method)
      (setf (color d) previous-color))))
```

CLOS-Common Lisp Object System

Mixins

```
(defclass color-mixin ()
  ((color :initarg :color :accessor color)))

(defmethod draw :around ((s color-mixin) (d device))
  (let ((previous-color (color d)))
    (setf (color d) (color s))
    (unwind-protect
       (call-next-method)
      (setf (color d) previous-color))))

(defclass colored-line (color-mixin line)
  ())

(defclass colored-circle (color-mixin circle)
  ())
```

CLOS-Common Lisp Object System

Mixins

```
(defclass colored-printer (printer)
  ((ink :initform :black :accessor color)))

(defmethod (setf color) :before (color (d colored-printer))
  (format t "changing printer ink color to ~A~%" color))
```

CLOS-Common Lisp Object System

Mixins

```
(defclass colored-printer (printer)
  ((ink :initform :black :accessor color)))

(defmethod (setf color) :before (color (d colored-printer))
  (format t "changing printer ink color to ~A~%" color))

(let ((shapes (list (make-instance 'line)
                    (make-instance 'colored-circle :color :red)
                    (make-instance 'colored-line :color :blue)))
      (printer (make-instance 'colored-printer)))
  (dolist (shape shapes)
    (draw shape printer)))
```

CLOS-Common Lisp Object System

Mixins

```
(defclass colored-printer (printer)
  ((ink :initform :black :accessor color)))

(defmethod (setf color) :before (color (d colored-printer))
  (format t "changing printer ink color to ~A~%" color))

(let ((shapes (list (make-instance 'line)
                    (make-instance 'colored-circle :color :red)
                    (make-instance 'colored-line :color :blue)))
      (printer (make-instance 'colored-printer)))
  (dolist (shape shapes)
    (draw shape printer)))

drawing a line on printer!
```

CLOS-Common Lisp Object System

Mixins

```
(defclass colored-printer (printer)
  ((ink :initform :black :accessor color)))

(defmethod (setf color) :before (color (d colored-printer))
  (format t "changing printer ink color to ~A~%" color))

(let ((shapes (list (make-instance 'line)
                    (make-instance 'colored-circle :color :red)
                    (make-instance 'colored-line :color :blue)))
      (printer (make-instance 'colored-printer)))
  (dolist (shape shapes)
    (draw shape printer)))

drawing a line on printer!
changing printer ink color to RED
drawing a circle on printer!
changing printer ink color to BLACK
```

CLOS-Common Lisp Object System

Mixins

```lisp
(defclass colored-printer (printer)
  ((ink :initform :black :accessor color)))

(defmethod (setf color) :before (color (d colored-printer))
  (format t "changing printer ink color to ~A~%" color))

(let ((shapes (list (make-instance 'line)
                    (make-instance 'colored-circle :color :red)
                    (make-instance 'colored-line :color :blue)))
      (printer (make-instance 'colored-printer)))
  (dolist (shape shapes)
    (draw shape printer)))

drawing a line on printer!
changing printer ink color to RED
drawing a circle on printer!
changing printer ink color to BLACK
changing printer ink color to BLUE
drawing a line on printer!
changing printer ink color to BLACK
```

Classes

Class Inheritance

- A class C_1 is a **direct subclass** of a class C_2 (class C_2 is a **direct superclass** of class C_1) if C_1 explicitly designates C_2 as a superclass in its definition.
- A class C_1 is a **subclass** of a class C_n (a class C_n is a **superclass** of class C_1) if there exists a sequence of classes $C_2, \ldots, C_{n-1}$ such that C_i is a **direct subclass** of C_{i+1}, $0 < i < n$.
- The **class precedence list** of class C is a total ordering of the set containing C and all its superclasses, from most specific to least specific.
- The ordering of the **class precedence list** of C is always consistent with the local ordering of the list of **direct superclasses** present in the definition of C.

Classes

Class Precedence List

Flavors Depth-first, from left to right, duplicates removed from the right (`standard-object` and `t` appended on the right).

Loops Identical but duplicates removed from the left.

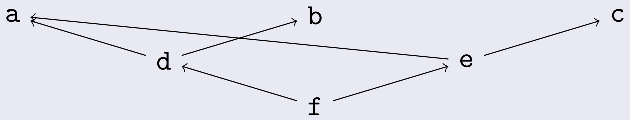
CLOS Topological sort of the inheritance graph using the local ordering of superclasses.

Class Hierarchy Example

```
(defclass a () ())
(defclass b () ())
(defclass c () ())
(defclass d (a b) ())
(defclass e (a c) ())
(defclass f (d e) ())
```

Classes

Class Inheritance Graph

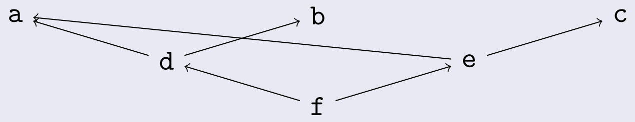


Class Precedence List of Class `f`

Flavors (1980) `f d a b e c`

Classes

Class Inheritance Graph



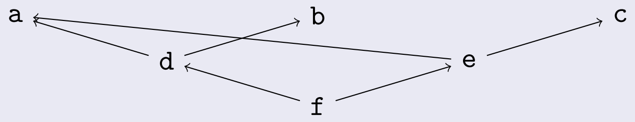
Class Precedence List of Class `f`

Flavors (1980) `f d a b e c`

Loops (1986) `f d b e a c`

Classes

Class Inheritance Graph



Class Precedence List of Class `f`

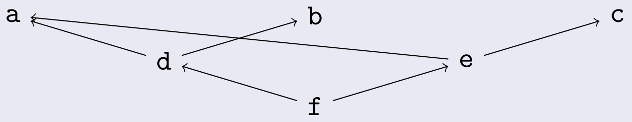
Flavors (1980) `f d a b e c`

Loops (1986) `f d b e a c`

CLOS (1991) `f d e a c b`

Classes

Class Inheritance Graph



Class Precedence List of Class `f`

Flavors (1980) `f d a b e c`

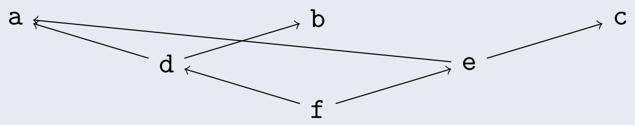
Loops (1986) `f d b e a c`

CLOS (1991) `f d e a c b`

Dylan (1996) `f d e a b c`

Classes

Class Inheritance Graph



Class Precedence List of Class `f`

Flavors (1980) `f d a b e c`

Loops (1986) `f d b e a c`

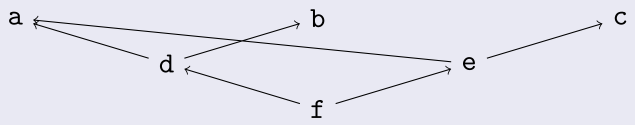
CLOS (1991) `f d e a c b`

Dylan (1996) `f d e a b c`

Python 2.1 (2001) `f d a b e c`

Classes

Class Inheritance Graph



Class Precedence List of Class `f`

Flavors (1980) `f d a b e c`

Loops (1986) `f d b e a c`

CLOS (1991) `f d e a c b`

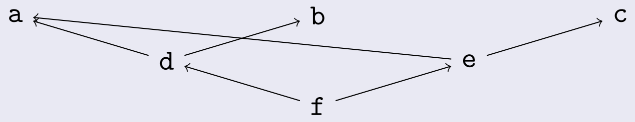
Dylan (1996) `f d e a b c`

Python 2.1 (2001) `f d a b e c`

Python 2.2 (2001) `f d b e a c`

Classes

Class Inheritance Graph



Class Precedence List of Class `f`

Flavors (1980) `f d a b e c`

Loops (1986) `f d b e a c`

CLOS (1991) `f d e a c b`

Dylan (1996) `f d e a b c`

Python 2.1 (2001) `f d a b e c`

Python 2.2 (2001) `f d b e a c`

Python 2.3 (2003) `f d e a b c`

Classes

MetaClasses

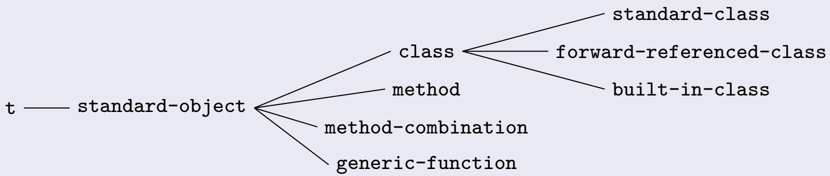
- Classes are represented by instances of other classes.
- A **metaclass** is a class whose instances are classes.
- The **metaclass** of an object is the class of the class of the object.

MetaClass Responsibilities

- Determines the inheritance process that is used by the classes that are its instances.
- Determines the representation of the instances of the classes that are its instances.
- Determines the access to the slots of the instances of the classes that are its instances.

Classes

Metaclass Hierarchy



Definition

- The `t` class does not have a superclass and is superclass of all classes except itself.
- The `standard-object` class is a direct subclass of class `t`, is an instance of class `standard-class` and is superclass of all classes that are instances of `standard-class` except itself.

Classes

Metaclass standard-class

```
> (defclass foo () ())                    ;A 'normal' class
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                    ;A 'normal' class
#<STANDARD-CLASS FOO>
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                    ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                    ;A 'normal' instance
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                    ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                    ;A 'normal' instance
#<FOO @ #x717910a2>                       ;Note #<class instance>
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                     ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                     ;A 'normal' instance
#<FOO @ #x717910a2>                        ;Note #<class instance>
> (class-of (make-instance 'foo))
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                ;A 'normal' instance
#<FOO @ #x717910a2>                   ;Note #<class instance>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                 ;Note #<metaclass class>
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                 ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                 ;A 'normal' instance
#<FOO @ #x717910a2>                    ;Note #<class instance>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                  ;Note #<metaclass class>
> (class-of (class-of (make-instance 'foo)))
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                  ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                  ;A 'normal' instance
#<FOO @ #x717910a2>                     ;Note #<class instance>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                   ;Note #<metaclass class>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>        ;STANDARD-CLASS metaclass
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                   ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                   ;A 'normal' instance
#<FOO @ #x717910a2>                      ;Note #<class instance>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                    ;Note #<metaclass class>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>         ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                  ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                  ;A 'normal' instance
#<FOO @ #x717910a2>                     ;Note #<class instance>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                   ;Note #<metaclass class>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>        ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS>        ;Looping
```

Metaclass built-in-class

Classes

Metaclass standard-class

```
> (defclass foo () ())                    ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                    ;A 'normal' instance
#<FOO @ #x717910a2>                       ;Note #<class instance>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                     ;Note #<metaclass class>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>          ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS>          ;Looping
```

Metaclass built-in-class

```
> (class-of 1)
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                    ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                    ;A 'normal' instance
#<FOO @ #x717910a2>                       ;Note #<class instance>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                     ;Note #<metaclass class>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>          ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS>          ;Looping
```

Metaclass built-in-class

```
> (class-of 1)
#<BUILT-IN-CLASS FIXNUM>                  ;Note #<metaclass class>
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                    ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                    ;A 'normal' instance
#<FOO @ #x717910a2>                       ;Note #<class instance>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                     ;Note #<metaclass class>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>          ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS>          ;Looping
```

Metaclass built-in-class

```
> (class-of 1)
#<BUILT-IN-CLASS FIXNUM>                  ;Note #<metaclass class>
> (class-of (class-of 1))                 ;The metaclass of 1
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                      ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                      ;A 'normal' instance
#<FOO @ #x717910a2>                         ;Note #<class instance>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                       ;Note #<metaclass class>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>            ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS>            ;Looping
```

Metaclass built-in-class

```
> (class-of 1)
#<BUILT-IN-CLASS FIXNUM>                    ;Note #<metaclass class>
> (class-of (class-of 1))                   ;The metaclass of 1
#<STANDARD-CLASS BUILT-IN-CLASS>            ; is BUILT-IN-CLASS
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                 ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                 ;A 'normal' instance
#<FOO @ #x717910a2>                    ;Note #<class instance>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                  ;Note #<metaclass class>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>       ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS>       ;Looping
```

Metaclass built-in-class

```
> (class-of 1)
#<BUILT-IN-CLASS FIXNUM>               ;Note #<metaclass class>
> (class-of (class-of 1))              ;The metaclass of 1
#<STANDARD-CLASS BUILT-IN-CLASS>       ; is BUILT-IN-CLASS
> (class-of (class-of (class-of 1)))   ;The metaclass of FIXNUM
```

Classes

Metaclass standard-class

```
> (defclass foo () ())                  ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                  ;A 'normal' instance
#<FOO @ #x717910a2>                     ;Note #<class instance>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                   ;Note #<metaclass class>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>        ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS>        ;Looping
```

Metaclass built-in-class

```
> (class-of 1)
#<BUILT-IN-CLASS FIXNUM>                ;Note #<metaclass class>
> (class-of (class-of 1))               ;The metaclass of 1
#<STANDARD-CLASS BUILT-IN-CLASS>        ; is BUILT-IN-CLASS
> (class-of (class-of (class-of 1)))    ;The metaclass of FIXNUM
#<STANDARD-CLASS STANDARD-CLASS>        ;is STANDARD-CLASS
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())        ;The class baz is not defined yet...
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())       ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())        ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())         ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())       ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
> (class-of (first bar-supers))
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())        ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
> (class-of (first bar-supers))
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())       ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
> (class-of (first bar-supers))
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())       ;We now define baz...
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())        ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
> (class-of (first bar-supers))
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())        ;We now define baz...
#<STANDARD-CLASS BAZ>
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())          ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
> (class-of (first bar-supers))
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())            ;We now define baz...
#<STANDARD-CLASS BAZ>
> bar-supers                      ;...and the saved class
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())          ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
> (class-of (first bar-supers))
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())          ;We now define baz...
#<STANDARD-CLASS BAZ>
> bar-supers                    ;...and the saved class
(#<STANDARD-CLASS BAZ>)         ;changes to a become a different thing
```

Function change-class

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())       ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
> (class-of (first bar-supers))
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())          ;We now define baz...
#<STANDARD-CLASS BAZ>
> bar-supers                    ;...and the saved class
(#<STANDARD-CLASS BAZ>)         ;changes to a become a different thing
```

Function change-class

```
> (setq foo-instance (make-instance 'foo)) ;;A normal instance
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())        ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
> (class-of (first bar-supers))
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())          ;We now define baz...
#<STANDARD-CLASS BAZ>
> bar-supers                    ;...and the saved class
(#<STANDARD-CLASS BAZ>)         ;changes to a become a different thing
```

Function change-class

```
> (setq foo-instance (make-instance 'foo)) ;;A normal instance
#<FOO @ #x717a0562>
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())       ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
> (class-of (first bar-supers))
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())         ;We now define baz...
#<STANDARD-CLASS BAZ>
> bar-supers                   ;...and the saved class
(#<STANDARD-CLASS BAZ>)        ;changes to a become a different thing
```

Function change-class

```
> (setq foo-instance (make-instance 'foo)) ;;A normal instance
#<FOO @ #x717a0562>
> (change-class foo-instance 'baz)        ;;Can we change its class?
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())        ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
> (class-of (first bar-supers))
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())          ;We now define baz...
#<STANDARD-CLASS BAZ>
> bar-supers                    ;...and the saved class
(#<STANDARD-CLASS BAZ>)         ;changes to a become a different thing
```

Function change-class

```
> (setq foo-instance (make-instance 'foo)) ;;A normal instance
#<FOO @ #x717a0562>
> (change-class foo-instance 'baz)      ;;Can we change its class?
#<BAZ @ #x717a0562>
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())          ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
> (class-of (first bar-supers))
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())          ;We now define baz...
#<STANDARD-CLASS BAZ>
> bar-supers                    ;...and the saved class
(#<STANDARD-CLASS BAZ>)          ;changes to a become a different thing
```

Function change-class

```
> (setq foo-instance (make-instance 'foo)) ;;A normal instance
#<FOO @ #x717a0562>
> (change-class foo-instance 'baz)      ;;Can we change its class?
#<BAZ @ #x717a0562>
> foo-instance
```

Classes

Metaclass forward-referenced-class

```
> (defclass bar (baz) ())      ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq bar-supers (class-direct-superclasses (find-class 'bar)))
(#<FORWARD-REFERENCED-CLASS BAZ>) ;...but it exists already.
> (class-of (first bar-supers))
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())         ;We now define baz...
#<STANDARD-CLASS BAZ>
> bar-supers                   ;...and the saved class
(#<STANDARD-CLASS BAZ>)        ;changes to a become a different thing
```

Function change-class

```
> (setq foo-instance (make-instance 'foo)) ;;A normal instance
#<FOO @ #x717a0562>
> (change-class foo-instance 'baz)       ;;Can we change its class?
#<BAZ @ #x717a0562>
> foo-instance
#<BAZ @ #x717a0562>                      ;;Yes, we can!
```

Classes

To obtain a class

- From an object *foo*:
  `(class-of foo)`
- From the name of a type `'bar`:
  `(find-class 'bar)`

Example

```
> (class-of "I am a string")
#<BUILT-IN-CLASS STRING>
> (find-class 'string)
#<BUILT-IN-CLASS STRING>

> (defclass foo () ())
#<STANDARD-CLASS FOO>
> (find-class 'foo)
#<STANDARD-CLASS FOO>
```

Classes

The generic function make-instance

```
(defgeneric make-instance (class &rest initargs))
```

Method specialization for **symbols**

```
(defmethod make-instance ((class symbol) &rest initargs)
  (apply #'make-instance (find-class class) initargs))
```

Method specialization for **classes**

```
(defmethod make-instance ((class class) &rest initargs)
  (let ((instance (apply #'allocate-instance class initargs)))
    (apply #'initialize-instance instance initargs)
    instance))
```

Optimizer

```
(define-compiler-macro make-instance (class-expr &rest init-exprs)
  (if (and (consp class-expr) (eq (first class-expr) 'quote))
    (make-instance->constructor-call (second class-expr) init-exprs)
    ...))
```

Classes

Slots

1. The expression `(slot-value` *obj name*`)` returns the value of slot *name* in *obj*.
2. If the slot does not exist, calls the generic function `slot-missing`:
   `(slot-missing (class-of` *obj*`)` *obj name* `'slot-value)`
3. If the slot exists but is unbound, calls the generic function `slot-unbound`:
   `(slot-unbound (class-of` *obj*`)` *obj name*`)`
4. The expression `(setf (slot-value` *obj name*`)` *new-value*`)` changes the value of slot *name* in *obj*.
5. If the slot does not exist, calls the generic function `slot-missing`:
   `(slot-missing (class-of` *obj*`)` *obj name* `'setf` *new-value*`)`

Classes

Missing Slots

```
(defclass foo ()
  ((slot1)))

> (setq foo1 (make-instance 'foo))
#<FOO @ #x716da0a2>
```

Classes

Missing Slots

```
(defclass foo ()
  ((slot1)))

> (setq foo1 (make-instance 'foo))
#<FOO @ #x716da0a2>
> (setf (slot-value foo1 'slot1) 1)    ;Updating slot1
1
```

Classes

Missing Slots

```
(defclass foo ()
  ((slot1)))

> (setq foo1 (make-instance 'foo))
#<FOO @ #x716da0a2>
> (setf (slot-value foo1 'slot1) 1)   ;Updating slot1
1
> (setf (slot-value foo1 'slot2) 2)   ;Error: slot2 does not exist!
```

Classes

Missing Slots

```
(defclass foo ()
  ((slot1)))

> (setq foo1 (make-instance 'foo))
#<FOO @ #x716da0a2>
> (setf (slot-value foo1 'slot1) 1)   ;Updating slot1
1
> (setf (slot-value foo1 'slot2) 2)   ;Error: slot2 does not exist!
The slot SLOT2 is missing from the object #<FOO @ #x716da0a2> of
class #<STANDARD-CLASS FOO> during operation SETF
   [Condition of type PROGRAM-ERROR]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [RETRY] Retry SLIME interactive evaluation request.
 3: [ABORT] Return to SLIME's top level.
 4: [ABORT] Abort entirely from this (lisp) process.
```

Classes

Dynamic Objects

A dynamic object can enlarge its set of slots.

Dynamic Objects

```
(defclass dynamic-object ()
  ((extra-slots :reader extra-slots
                :initform (make-hash-table :test #'eq))))
```

Classes

Dynamic Objects

A dynamic object can enlarge its set of slots.

Dynamic Objects

```
(defclass dynamic-object ()
  ((extra-slots :reader extra-slots
                :initform (make-hash-table :test #'eq))))

(defmethod slot-missing ((class t)
                         (object dynamic-object)
                         slot-name
                         operation
                         &optional new-value)
```

Classes

Dynamic Objects

A dynamic object can enlarge its set of slots.

Dynamic Objects

```
(defclass dynamic-object ()
  ((extra-slots :reader extra-slots
                :initform (make-hash-table :test #'eq))))

(defmethod slot-missing ((class t)
                         (object dynamic-object)
                         slot-name
                         operation
                         &optional new-value)
  (case operation
```

Classes

Dynamic Objects

A dynamic object can enlarge its set of slots.

Dynamic Objects

```
(defclass dynamic-object ()
  ((extra-slots :reader extra-slots
                :initform (make-hash-table :test #'eq))))

(defmethod slot-missing ((class t)
                         (object dynamic-object)
                         slot-name
                         operation
                         &optional new-value)
  (case operation
    (slot-value
```

Classes

Dynamic Objects

A dynamic object can enlarge its set of slots.

Dynamic Objects

```
(defclass dynamic-object ()
  ((extra-slots :reader extra-slots
                :initform (make-hash-table :test #'eq))))

(defmethod slot-missing ((class t)
                         (object dynamic-object)
                         slot-name
                         operation
                         &optional new-value)
  (case operation
    (slot-value
     (gethash slot-name (extra-slots object))
```

Classes

Dynamic Objects

A dynamic object can enlarge its set of slots.

Dynamic Objects

```
(defclass dynamic-object ()
  ((extra-slots :reader extra-slots
                :initform (make-hash-table :test #'eq))))

(defmethod slot-missing ((class t)
                         (object dynamic-object)
                         slot-name
                         operation
                         &optional new-value)
  (case operation
    (slot-value
     (gethash slot-name (extra-slots object)))
    (setf
```

Classes

Dynamic Objects

A dynamic object can enlarge its set of slots.

Dynamic Objects

```
(defclass dynamic-object ()
  ((extra-slots :reader extra-slots
                :initform (make-hash-table :test #'eq))))

(defmethod slot-missing ((class t)
                         (object dynamic-object)
                         slot-name
                         operation
                         &optional new-value)
  (case operation
    (slot-value
     (gethash slot-name (extra-slots object)))
    (setf
     (setf (gethash slot-name (extra-slots object))
           new-value))
```

Classes

Dynamic Objects

A dynamic object can enlarge its set of slots.

Dynamic Objects

```
(defclass dynamic-object ()
  ((extra-slots :reader extra-slots
                :initform (make-hash-table :test #'eq))))

(defmethod slot-missing ((class t)
                         (object dynamic-object)
                         slot-name
                         operation
                         &optional new-value)
  (case operation
    (slot-value
     (gethash slot-name (extra-slots object)))
    (setf
     (setf (gethash slot-name (extra-slots object))
           new-value))
    (t
```

Classes

Dynamic Objects

A dynamic object can enlarge its set of slots.

Dynamic Objects

```
(defclass dynamic-object ()
  ((extra-slots :reader extra-slots
                :initform (make-hash-table :test #'eq))))

(defmethod slot-missing ((class t)
                         (object dynamic-object)
                         slot-name
                         operation
                         &optional new-value)
  (case operation
    (slot-value
     (gethash slot-name (extra-slots object)))
    (setf
     (setf (gethash slot-name (extra-slots object))
           new-value))
    (t
     (call-next-method))))
```

Classes

Missing Slots

```
(defclass foo (dynamic-object)   ;Redefining class foo...
  ((slot1)))
```

Classes

Missing Slots

```
(defclass foo (dynamic-object)   ;Redefining class foo...
  ((slot1)))

> (slot-value foo1 'slot1)       ;...but keeping old instances alive
```

Classes

Missing Slots

```
(defclass foo (dynamic-object)   ;Redefining class foo...
  ((slot1)))

> (slot-value foo1 'slot1)       ;...but keeping old instances alive
1
```

Classes

Missing Slots

```
(defclass foo (dynamic-object)   ;Redefining class foo...
  ((slot1)))

> (slot-value foo1 'slot1)       ;...but keeping old instances alive
1
> (setf (slot-value foo1 'slot2) 2) ;Now, the slot exists...
```

Classes

Missing Slots

```
(defclass foo (dynamic-object)  ;Redefining class foo...
  ((slot1)))

> (slot-value foo1 'slot1)      ;...but keeping old instances alive
1
> (setf (slot-value foo1 'slot2) 2) ;Now, the slot exists...
2
```

Classes

Missing Slots

```
(defclass foo (dynamic-object)   ;Redefining class foo...
  ((slot1)))

> (slot-value foo1 'slot1)       ;...but keeping old instances alive
1
> (setf (slot-value foo1 'slot2) 2) ;Now, the slot exists...
2
> (slot-value foo1 'slot2)       ;...and it keeps its value
```

Classes

Missing Slots

```
(defclass foo (dynamic-object)  ;Redefining class foo...
  ((slot1)))

> (slot-value foo1 'slot1)      ;...but keeping old instances alive
1
> (setf (slot-value foo1 'slot2) 2) ;Now, the slot exists...
2
> (slot-value foo1 'slot2)      ;...and it keeps its value
2
```

Classes

Missing Slots

```
(defclass foo (dynamic-object)  ;Redefining class foo...
  ((slot1)))

> (slot-value foo1 'slot1)      ;...but keeping old instances alive
1
> (setf (slot-value foo1 'slot2) 2) ;Now, the slot exists...
2
> (slot-value foo1 'slot2)      ;...and it keeps its value
2
> (slot-value foo1 'slot3)
```

Classes

Missing Slots

```
(defclass foo (dynamic-object)   ;Redefining class foo...
  ((slot1)))

> (slot-value foo1 'slot1)       ;...but keeping old instances alive
1
> (setf (slot-value foo1 'slot2) 2) ;Now, the slot exists...
2
> (slot-value foo1 'slot2)       ;...and it keeps its value
2
> (slot-value foo1 'slot3)
NIL                              ;Humm, we should improve this
```

Problem

We should not return a value for "missing" slots that are nor present in the set of extra slots.

Classes

Dynamic Objects

```
(defmethod slot-missing ((class t)
                         (object dynamic-object)
                         slot-name
                         operation
                         &optional new-value)
  (case operation
    (slot-value
     (gethash slot-name (extra-slots object)))
    (setf
     (setf (gethash slot-name (extra-slots object))
           new-value))
    (t
     (call-next-method))))
```

Classes

Dynamic Objects

```
(defmethod slot-missing ((class t)
                         (object dynamic-object)
                         slot-name
                         operation
                         &optional new-value)
  (case operation
    (slot-value
     (multiple-value-bind (value found?)
         (gethash slot-name (extra-slots object))
       (if found?
         value
         (slot-unbound class object slot-name))))
    (setf
     (setf (gethash slot-name (extra-slots object))
           new-value))
    (slot-boundp
     (nth-value 1 (gethash slot-name (extra-slots object))))
    (slot-makunbound
     (remhash slot-name (extra-slots object)))))
```

Classes

Dynamic Objects

```
> (slot-value foo1 'slot1) ;A 'normal' slot
1
```

Classes

Dynamic Objects

```
> (slot-value foo1 'slot1) ;A 'normal' slot
1
> (slot-value foo1 'slot2) ;An 'added' slot
2
```

Classes

Dynamic Objects

```
> (slot-value foo1 'slot1) ;A 'normal' slot
1
> (slot-value foo1 'slot2) ;An 'added' slot
2
> (slot-value foo1 'slot3) ;A slot without an assigned value
```

Classes

Dynamic Objects

```
> (slot-value foo1 'slot1) ;A 'normal' slot
1
> (slot-value foo1 'slot2) ;An 'added' slot
2
> (slot-value foo1 'slot3) ;A slot without an assigned value
The slot SLOT3 is unbound in the object #<FOO @ #x714a1f0a> of
class #<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [RETRY] Retry SLIME interactive evaluation request.
 4: [ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.
```

Classes

Unbound Slots

```
> (setq foo2 (make-instance 'foo)) ;A new instance
#<FOO @ #x71648d6a>
> (* (slot-value foo2 'slot1) 2)   ;Error: slot1 is unbound in foo2
```

Classes

Unbound Slots

```
> (setq foo2 (make-instance 'foo)) ;A new instance
#<FOO @ #x71648d6a>
> (* (slot-value foo2 'slot1) 2)   ;Error: slot1 is unbound in foo2
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]
```

Classes

Unbound Slots

```
> (setq foo2 (make-instance 'foo)) ;A new instance
#<FOO @ #x71648d6a>
> (* (slot-value foo2 'slot1) 2)   ;Error: slot1 is unbound in foo2
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [RETRY] Retry SLIME interactive evaluation request.
 4: [ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.
```

Classes

Unbound Slots

```
> (setq foo2 (make-instance 'foo)) ;A new instance
#<FOO @ #x71648d6a>
> (* (slot-value foo2 'slot1) 2)   ;Error: slot1 is unbound in foo2
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [RETRY] Retry SLIME interactive evaluation request.
 4: [ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.

:C 2    ;Choose option 2: [STORE-VALUE]
```

Classes

Unbound Slots

```
> (setq foo2 (make-instance 'foo)) ;A new instance
#<FOO @ #x71648d6a>
> (* (slot-value foo2 'slot1) 2)   ;Error: slot1 is unbound in foo2
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [RETRY] Retry SLIME interactive evaluation request.
 4: [ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.

:C 2    ;Choose option 2: [STORE-VALUE]
enter expression which will evaluate to a value to use: 25
```

Classes

Unbound Slots

```
> (setq foo2 (make-instance 'foo)) ;A new instance
#<FOO @ #x71648d6a>
> (* (slot-value foo2 'slot1) 2)   ;Error: slot1 is unbound in foo2
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [RETRY] Retry SLIME interactive evaluation request.
 4: [ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.

:C 2    ;Choose option 2: [STORE-VALUE]
enter expression which will evaluate to a value to use: 25
```

Classes

Unbound Slots

```
> (setq foo2 (make-instance 'foo)) ;A new instance
#<FOO @ #x71648d6a>
> (* (slot-value foo2 'slot1) 2)   ;Error: slot1 is unbound in foo2
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [RETRY] Retry SLIME interactive evaluation request.
 4: [ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.

:C 2    ;Choose option 2: [STORE-VALUE]
enter expression which will evaluate to a value to use: 25
50
```

Classes

Unbound Slots

```
> (setq foo2 (make-instance 'foo)) ;A new instance
#<FOO @ #x71648d6a>
> (* (slot-value foo2 'slot1) 2)   ;Error: slot1 is unbound in foo2
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [RETRY] Retry SLIME interactive evaluation request.
 4: [ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.

:C 2    ;Choose option 2: [STORE-VALUE]
enter expression which will evaluate to a value to use: 25
50
CL-USER> (slot-value foo2 'slot1)
```

Classes

Unbound Slots

```
> (setq foo2 (make-instance 'foo)) ;A new instance
#<FOO @ #x71648d6a>
> (* (slot-value foo2 'slot1) 2)   ;Error: slot1 is unbound in foo2
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [RETRY] Retry SLIME interactive evaluation request.
 4: [ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.

:C 2    ;Choose option 2: [STORE-VALUE]
enter expression which will evaluate to a value to use: 25
50
CL-USER> (slot-value foo2 'slot1)
25
```

Classes

Slots

- `slot-value` and `(setf slot-value)` are **functions**...
- ... but **not generic functions**.
- In all implementations that include the MOP (all of them, in practice), they call the generic functions `slot-value-using-class` and `(setf slot-value-using-class)`

The (non-generic) function `slot-value`

```
(defun slot-value (object slot-name)
  (let* ((class (class-of object))
         (slot-definition (find-slot-definition class slot-name)))
    (if (null slot-definition)
      (slot-missing class object slot-name 'slot-value)
      (slot-value-using-class class object slot-definition))))
```

Classes

The Generic Function slot-value-using-class

```
(defmethod slot-value-using-class
    ((class standard-class)
     (object standard-object)
     (slotdef standard-effective-slot-definition))
  (if ...
    (slot-unbound class object (slot-definition-name slotdef))
    ...))
```

The Generic Function slot-unbound

```
(defmethod slot-unbound ((class t) instance slot-name)
  (restart-case
      (error 'unbound-slot :name slot-name :instance instance)
    (use-value (value)
      ...)
    (store-value (new-value)
      ...)))
```

Object Protocol

Protocol

Abstract Model of a *behavior*.

Protocol

Set of generic functions that implement a *behavior*.

Object Protocols in CLOS

- Instance creation and initialization
- Instance reinitialization
- Changing the class of an instance
- Class redefinition
- Slot accessing
- Generic function calling

Object Protocol

Instance Creation

1. Combine explicit initializations (`make-instance`) with default values (`:default-initargs` and `:initforms`).
2. Validate initializations.
3. Allocate memory space for the instance (`allocate-instance`).
4. Filling in the slots using initialization values (`initialize-instance` and `shared-initialize`).

Object Protocol

Instance Creation

- `make-instance` calls `allocate-instance` and `initialize-instance`.
- `allocate-instance` allocates space for the instance.
- `initialize-instance` calls `shared-initialize`.
- `shared-initialize` assigns slots using `:initargs`, `:default-initargs`, and `:initforms`.

Object Protocol

Instance Creation - make-instance

```
(defmethod make-instance ((class class) &rest initargs)

  ;; Verify initialization validity

  (let ((instance (apply #'allocate-instance class initargs)))
    (apply #'initialize-instance instance initargs)
    instance))
```

Instance Creation - initialize-instance

```
(defmethod initialize-instance ((instance standard-object)
                                &rest initargs &key)
  (apply #'shared-initialize instance t initargs))
```

Object Protocol

Changing the Class of an Instance

1. Modifies the instance to conform to the new class, adding new slots and discarding slots that are not defined in the new class.
2. Initializes the new slots.

Changing the Class of an Instance

- `change-class` modifies an object to become an instance of a different class.
- `change-class` calls `update-instance-for-different-class`.
- `update-instance-for-different-class` calls `shared-initialize`.

Object Protocol

Changing the Class of an Instance

```
(defmethod change-class ((instance standard-object)
                         (new-class standard-class)
                         &rest initargs &key)
  (let* ((old-class (class-of instance))
         (new-instance (allocate-instance new-class))
         (old-slots (get-slots instance))
         (new-slots (get-slots new-instance)))
    ;; Copy shared slots
    ;; Make the old instance point to the new storage.
    (apply #'update-instance-for-different-class
           new-instance
           instance
           initargs)
    instance))

(defmethod update-instance-for-different-class
    ((previous standard-object) (current standard-object)
     &rest initargs &key)
    ...
    (apply #'shared-initialize current added-slots initargs))
```

Object Protocol

Class Redefinition

1. Modifies the structure of the existing class.
2. If there are added or removed slots or changes in their order, already existent instances are updated (in an undetermined moment but always before a slot access).
3. For each instance,
   1. Modifies the structure of the instance to conform to the redefined class, adding new slots and discarding slots that are not defined in the redefined class.
   2. Initializes the new slots (`update-instance-for-redefined-class` and `shared-initialize`).

Object Protocol

Class Redefinition

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))
```

Object Protocol

Class Redefinition

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
```

Object Protocol

Class Redefinition

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
```

Object Protocol

Class Redefinition

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
> (slot-value 1+2i 'real)
1
```

Object Protocol

Class Redefinition

```
(defclass complex-number ()   ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
> (slot-value 1+2i 'real)
1

(defclass complex-number ()   ;Redefine (polar) complex-number
  ((rho :initarg :rho)
   (theta :initarg :theta)))
```

Object Protocol

Class Redefinition

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
> (slot-value 1+2i 'real)
1

(defclass complex-number ()    ;Redefine (polar) complex-number
  ((rho :initarg :rho)
   (theta :initarg :theta)))

> (slot-value 1+2i 'real)      ;The slot 'real' is gone
```

Object Protocol

Class Redefinition

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
> (slot-value 1+2i 'real)
1

(defclass complex-number ()    ;Redefine (polar) complex-number
  ((rho :initarg :rho)
   (theta :initarg :theta)))

> (slot-value 1+2i 'real)      ;The slot 'real' is gone
The slot REAL is missing in the object #<COMPLEX-NUMBER @ #x717705a2>
```

Object Protocol

Class Redefinition

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
> (slot-value 1+2i 'real)
1

(defclass complex-number ()    ;Redefine (polar) complex-number
  ((rho :initarg :rho)
   (theta :initarg :theta)))

> (slot-value 1+2i 'real)      ;The slot 'real' is gone
The slot REAL is missing in the object #<COMPLEX-NUMBER @ #x717705a2>

> (slot-value 1+2i 'rho)       ;The slot 'rho' is unbound
```

Object Protocol

Class Redefinition

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
> (slot-value 1+2i 'real)
1

(defclass complex-number ()    ;Redefine (polar) complex-number
  ((rho :initarg :rho)
   (theta :initarg :theta)))

> (slot-value 1+2i 'real)      ;The slot 'real' is gone
The slot REAL is missing in the object #<COMPLEX-NUMBER @ #x717705a2>

> (slot-value 1+2i 'rho)       ;The slot 'rho' is unbound
The slot RHO is unbound in the object #<COMPLEX-NUMBER @ #x717705a2>
```

Object Protocol

Class Redefinition

```
(defmethod update-instance-for-redefined-class :before
    ((c complex-number)
     added-slots
     discarded-slots
     property-list                    ;(real 3 imag 4)
     &rest args
     &key &allow-other-keys)
```

Object Protocol

Class Redefinition

```
(defmethod update-instance-for-redefined-class :before
    ((c complex-number)
     added-slots
     discarded-slots
     property-list                    ;(real 3 imag 4)
     &rest args
     &key &allow-other-keys)
  (let ((r (getf property-list 'real))    ;3
        (i (getf property-list 'imag)))   ;4
    (setf (slot-value c 'rho)
          (sqrt (+ (* r r) (* i i)))
          (slot-value c 'theta)
          (atan i r))))
```

Object Protocol

Class Redefinition

```
(defmethod update-instance-for-redefined-class :before
    ((c complex-number)
     added-slots
     discarded-slots
     property-list                ;(real 3 imag 4)
     &rest args
     &key &allow-other-keys)
  (let ((r (getf property-list 'real))    ;3
        (i (getf property-list 'imag)))   ;4
    (setf (slot-value c 'rho)
          (sqrt (+ (* r r) (* i i)))
          (slot-value c 'theta)
          (atan i r))))

> (slot-value 1+2i 'rho) ;Too late for the first instance
```

Object Protocol

Class Redefinition

```
(defmethod update-instance-for-redefined-class :before
    ((c complex-number)
     added-slots
     discarded-slots
     property-list                ;(real 3 imag 4)
     &rest args
     &key &allow-other-keys)
  (let ((r (getf property-list 'real))   ;3
        (i (getf property-list 'imag)))  ;4
    (setf (slot-value c 'rho)
          (sqrt (+ (* r r) (* i i)))
          (slot-value c 'theta)
          (atan i r))))

> (slot-value 1+2i 'rho) ;Too late for the first instance
The slot RHO is unbound in the object #<COMPLEX-NUMBER @ #x717705a2>
```

Object Protocol

Class Redefinition

```
(defmethod update-instance-for-redefined-class :before
    ((c complex-number)
     added-slots
     discarded-slots
     property-list                  ;(real 3 imag 4)
     &rest args
     &key &allow-other-keys)
  (let ((r (getf property-list 'real))    ;3
        (i (getf property-list 'imag)))   ;4
    (setf (slot-value c 'rho)
            (sqrt (+ (* r r) (* i i)))
          (slot-value c 'theta)
            (atan i r))))

> (slot-value 1+2i 'rho) ;Too late for the first instance
The slot RHO is unbound in the object #<COMPLEX-NUMBER @ #x717705a2>

> (slot-value 3+4i 'rho) ;But on time for the second one
```

Object Protocol

Class Redefinition

```
(defmethod update-instance-for-redefined-class :before
    ((c complex-number)
     added-slots
     discarded-slots
     property-list                 ;(real 3 imag 4)
     &rest args
     &key &allow-other-keys)
  (let ((r (getf property-list 'real))   ;3
        (i (getf property-list 'imag)))  ;4
    (setf (slot-value c 'rho)
            (sqrt (+ (* r r) (* i i)))
          (slot-value c 'theta)
            (atan i r))))

> (slot-value 1+2i 'rho) ;Too late for the first instance
The slot RHO is unbound in the object #<COMPLEX-NUMBER @ #x717705a2>

> (slot-value 3+4i 'rho) ;But on time for the second one
5.0
```

Meta Object Protocol

Hypothetical Example: Slot Access

- An instance is represented by an *array*.
- The first element of the *array* is the class of the instance.
- The remaining elements of the *array* are the values of the slots.

Function `slot-value`

```
(defun slot-value (instance slot-name)
  (let ((class (aref instance 0)))
    (let ((slots (class-slots class)))
      (aref instance
            (1+ (position slot-name slots))))))
```

Problem

An array is not the best representation for all possible cases. For instances with many slots, an hash table might be more efficient.

Meta Object Protocol

Hypothetical Example: Slot Access

- Solution: delegate the *implementation* of slot access.
- Candidate: the class of the class of the instance (i.e., the instance's metaclass).
- The metaclass *intermediates* the access to the instance.

Function slot-value

```
(defun slot-value (instance slot-name)
  (slot-value-using-class (class-of instance)
                          instance
                          slot-name))
```

Meta Object Protocol

Hypothetical Example: Slot Access

- For the default metaclass (e.g., `default-class`), an instance is represented by an *array*.

Function `slot-value`

```
(defmethod slot-value-using-class ((class default-class)
                                   instance
                                   slot-name)
  (let ((slots (class-slots class)))
    (aref instance
          (1+ (position slot-name slots)))))
```

Meta Object Protocol

Hypothetical Example: Slot Access

- For a different metaclass (e.g., `hash-table-class`), an instance is represented by an *hash-table*.

Function `slot-value`

```
(defmethod slot-value-using-class ((class hash-table-class)
                                    instance
                                    slot-name)
  (gethash instance slot-name))
```

Meta Object Protocol

Real Example: Slot Access

- For the default metaclass `standard-class`:

Function `slot-value`

```
(defun slot-value (object slot-name)
  (let* ((class (class-of object))
         (slot-definition (find-slot-definition class slot-name)))
    (if (null slot-definition)
      (slot-missing class object slot-name 'slot-value)
      (slot-value-using-class class object slot-definition))))
```

Function `slot-value-using-class`

```
(defmethod slot-value-using-class
    ((class standard-class)
     (object standard-object)
     (slotdef standard-effective-slot-definition))
  (if ...
    (slot-unbound class object (slot-definition-name slotdef))
    ...))
```

Undoable Programs

Problem

- We want to be able to *undo* the execution of CLOS programs.
- We want to be able to create *checkpoints* representing the execution state of a CLOS program.
- We want to be able to force a program to go back in time until it reaches a given *checkpoint*.

Undoable Programs

A person has a name, an age, and a friend - Java

```
class Person {
    String name;
    int age;
    Person friend;

    public String toString() {
        return "[" + name + "," + age +
            ((friend == null) ? "" : " with friend " + friend) +
            "]";
    }
}
```

Yes, I know:

- Missing constructor.
- Missing *getters* and *setters*.
- They are not relevant for the example.

Undoable Programs

A person has a name, an age, and a friend - CLOS

```
(defclass person ()
  ((name :accessor name :initarg :name)
   (age :accessor age :initarg :age)
   (friend :initform nil :accessor friend :initarg :friend))
  (:metaclass undoable-class))

(defmethod print-object ((p person) stream)
  (format stream
          "[~A,~A~@[ with friend ~A~]]"
          (name p) (age p) (friend p)))
```

Yes, I know:

- Defined constructor.
- Defined *readers* and *writers*.
- Nothing else is needed.

Paul, John and Mary - Java

```
Person p0 = new Person() {{ name = "John"; age = 21; }};
Person p1 = new Person() {{ name = "Paul"; age = 23; }};
//Paul has friend named John
p1.friend = p0;
println(p1);//[Paul,23 with friend [John,21]]
```

Paul, John and Mary - Java

```
Person p0 = new Person() {{ name = "John"; age = 21; }};
Person p1 = new Person() {{ name = "Paul"; age = 23; }};
//Paul has friend named John
p1.friend = p0;
println(p1);//[Paul,23 with friend [John,21]]
int state0 = History.currentState();
//32 years later, John changed his name to 'Louis' and got a friend
p0.age = 53;
p1.age = 55;
p0.name = "Louis";
p0.friend = new Person() {{ name = "Mary"; age = 19; }};
println(p1);//[Paul,55 with friend [Louis,53 with friend [Mary,19]]]
```

Paul, John and Mary - Java

```
Person p0 = new Person() {{ name = "John"; age = 21; }};
Person p1 = new Person() {{ name = "Paul"; age = 23; }};
//Paul has friend named John
p1.friend = p0;
println(p1);//[Paul,23 with friend [John,21]]
int state0 = History.currentState();
//32 years later, John changed his name to 'Louis' and got a friend
p0.age = 53;
p1.age = 55;
p0.name = "Louis";
p0.friend = new Person() {{ name = "Mary"; age = 19; }};
println(p1);//[Paul,55 with friend [Louis,53 with friend [Mary,19]]]
int state1 = History.currentState();
//15 years later, John (hum, I mean 'Louis') died
p1.age = 70;
p1.friend = null;
println(p1);//[Paul,70]
```

Paul, John and Mary - Java

```
Person p0 = new Person() {{ name = "John"; age = 21; }};
Person p1 = new Person() {{ name = "Paul"; age = 23; }};
//Paul has friend named John
p1.friend = p0;
println(p1);//[Paul,23 with friend [John,21]]
int state0 = History.currentState();
//32 years later, John changed his name to 'Louis' and got a friend
p0.age = 53;
p1.age = 55;
p0.name = "Louis";
p0.friend = new Person() {{ name = "Mary"; age = 19; }};
println(p1);//[Paul,55 with friend [Louis,53 with friend [Mary,19]]]
int state1 = History.currentState();
//15 years later, John (hum, I mean 'Louis') died
p1.age = 70;
p1.friend = null;
println(p1);//[Paul,70]
//Let's go back in time
History.restoreState(state1);
println(p1);//[Paul,55 with friend [Louis,53 with friend [Mary,19]]]
```

Paul, John and Mary - Java

```java
Person p0 = new Person() {{ name = "John"; age = 21; }};
Person p1 = new Person() {{ name = "Paul"; age = 23; }};
//Paul has friend named John
p1.friend = p0;
println(p1);//[Paul,23 with friend [John,21]]
int state0 = History.currentState();
//32 years later, John changed his name to 'Louis' and got a friend
p0.age = 53;
p1.age = 55;
p0.name = "Louis";
p0.friend = new Person() {{ name = "Mary"; age = 19; }};
println(p1);//[Paul,55 with friend [Louis,53 with friend [Mary,19]]]
int state1 = History.currentState();
//15 years later, John (hum, I mean 'Louis') died
p1.age = 70;
p1.friend = null;
println(p1);//[Paul,70]
//Let's go back in time
History.restoreState(state1);
println(p1);//[Paul,55 with friend [Louis,53 with friend [Mary,19]]]
//and even earlier
History.restoreState(state0);
println(p1);//[Paul,23 with friend [John,21]]
```

Paul, John and Mary - CLOS

```
(setf p0 (make-instance 'person :name "John" :age 21)
      p1 (make-instance 'person :name "Paul" :age 23))
;;Paul has friend named John
(setf (friend p1) p0)
(print p1)  ;;[Paul,23 with friend [John,21]]
(setf state0 (current-state))
;;32 years later, John changed his name to 'Louis' and got a friend
(setf (age p0) 53
      (age p1) 55
      (name p0) "Louis"
      (friend p0) (make-instance 'person :name "Mary" :age 19))
(print p1)  ;;[Paul,55 with friend [Louis,53 with friend [Mary,19]]]
(setf state1 (current-state))
;;25 years later, John (hum, I mean 'Louis') died
(setf (age p1) 70
      (friend p1) nil)
(print p1)  ;;[Paul,70]
;;Let's go back in time
(restore-state state1)
(print p1)  ;;[Paul,55 with friend [Louis,53 with friend [Mary,19]]]
;;and even earlier
(restore-state state0)
(print p1)  ;;[Paul,23 with friend [John,21]]
```

Undoable Programs

Save Program State - Java

```
import java.util.Stack;
import java.lang.reflect.*;

public class History {

    static Stack<ObjectFieldValue> undoTrail =
        new Stack<ObjectFieldValue>();

    public static void storePrevious(Object object,
                                     String className,
                                     String fieldName,
                                     Object value) {
        undoTrail.push(new ObjectFieldValue(object,
                                            className,
                                            fieldName,
                                            value));
    }

    ...
}
```

Undoable Programs

Save Program State - CLOS

```
(defparameter *undo-trail* (list))

(defun store-previous (object slot value)
  (push (list object slot value) *undo-trail*))
```

Undoable Programs

Save Program State - Java

```
import java.util.Stack;
import java.lang.reflect.*;

public class History {

    ...

    public static int currentState() {
        return undoTrail.size();
    }

    public static void restoreState(int state) {
        //undo all actions until size == state
        while (undoTrail.size() != state) {
            undoTrail.pop().restore();
        }
    }
}
```

Undoable Programs

Save Program State - CLOS

```
(defun current-state ()
  *undo-trail*)

(defun restore-state (trail)
  (loop
     until (eq *undo-trail* trail)
     do (apply #'restore (pop *undo-trail*))))
```

Undoable Programs

Save Program State - Java

```
class ObjectFieldValue {
    Object object;
    String className;
    String fieldName;
    Object value;

    ObjectFieldValue(Object object,
                     String className,
                     String fieldName,
                     Object value) {
        this.object = object;
        this.className = className;
        this.fieldName = fieldName;
        this.value = value;
    }

    …
}
```

Undoable Programs

Save Program State - CLOS

Undoable Programs

Save Program State - Java

```
class ObjectFieldValue {

    …

    void restore() {
        try {
            Field field =
                Class.forName(className).
                getDeclaredField(fieldName);
            field.setAccessible(true);
            field.set(object, value);
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        } catch (NoSuchFieldException e) {
            throw new RuntimeException(e);
        } catch (IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Undoable Programs

Save Program State - CLOS

```
(defparameter *save-previous-value* t)

(defun restore (object slot value)
  (let ((*save-previous-value* nil))
    (setf (slot-value object slot) value)))
```

Undoable Programs

Javassist

```
import javassist.*;
import javassist.expr.*;
import java.io.*;
import java.lang.reflect.*;

public class Undoable {

    public static void main(String[] args) throws … {
        if (args.length < 1) {
            …
        } else {
            Translator translator = new UndoableTranslator();
            ClassPool pool = ClassPool.getDefault();
            Loader classLoader = new Loader();
            classLoader.addTranslator(pool, translator);
            String[] restArgs = new String[args.length - 1];
            System.arraycopy(args, 1, restArgs, 0, restArgs.length);
            classLoader.run(args[0], restArgs);
        }
    }
}
```

Undoable Programs

CLOS MOP

Undoable Programs

Javassist

```
class UndoableTranslator implements Translator {

    public void start(ClassPool pool)
        throws NotFoundException, CannotCompileException {
    }

    public void onLoad(ClassPool pool, String className)
        throws NotFoundException, CannotCompileException {
        CtClass ctClass = pool.get(className);
        makeUndoable(ctClass);
    }

    void makeUndoable(CtClass ctClass) {
        …
    }
}
```

Undoable Programs

CLOS MOP

```
(defclass undoable-class (standard-class)
  ())

(defmethod validate-superclass ((class undoable-class)
                                (superclass standard-class))
  t)
```

Metaclass Compatibility

- Except for `standard-object`, all classes have at least one superclass.
- What happens when a class is instantiated from a metaclass and one of its superclasses is an instance from a different metaclass?
- The MOP requires that we validate metaclass compatibility.

Undoable Programs

Javassist

```
void makeUndoable(CtClass ctClass)
    throws NotFoundException, CannotCompileException {
    final String template =
        "{" +
        "  History.storePrevious($0, \"%s\",\"%s\", ($w)$0.%s);" +
        "  $0.%s = $1;" +
        "}";
    for (CtMethod ctMethod : ctClass.getDeclaredMethods()) {
        ctMethod.instrument(new ExprEditor() {
                public void edit(FieldAccess fa)
                    throws CannotCompileException {
                    if (fa.isWriter()) {
                        String name = fa.getFieldName();
                        fa.replace(String.format(template,
                                                 fa.getClassName(),
                                                 name, name, name));
                    }
                }
            });
    }
}
```

Undoable Programs

CLOS MOP

```
(defmethod (setf slot-value-using-class) :before
    (new-value (class undoable-class) object slot)
  (when *save-previous-value*
    (when (slot-boundp-using-class class object slot)
      (store-previous object
                      (slot-definition-name slot)
                      (slot-value-using-class class object slot)))))
```

Meta Object Protocol

Basic MetaObject Classes

- `class`
- `slot-definition`
- `generic-function`
- `method`

Standard MetaObject Classes

- `standard-class`
- `standard-direct-slot-definition`, `standard-effective-slot-definition`
- `standard-generic-function`
- `standard-method`, `standard-reader-method`, `standard-writer-method`
- `funcallable-standard-class`

Meta Object Protocol

Funcallable Instances

- Instances of classes which are themselves instances of `funcallable-standard-class` or one of its subclasses.
- Like standard instances, funcallable instances can have slots.
- They differ from standard instances in that they can be used as functions as well:
  1. They can be passed to funcall and apply
  2. They can be stored as the definition of a function name.
- Associated with each funcallable instance is the function which it runs when it is called. This function can be changed with `set-funcallable-instance-function`.

Meta Object Protocol

Pure Functions are Maps

Meta Object Protocol

Pure Functions are Maps

```
(defclass funcallable-map ()
  ((map :initform (make-hash-table) :reader function-map))
  (:metaclass funcallable-standard-class))
```

Meta Object Protocol

Pure Functions are Maps

```
(defclass funcallable-map ()
  ((map :initform (make-hash-table) :reader function-map))
  (:metaclass funcallable-standard-class))

(defmethod get-value ((f funcallable-map) arg)
  (gethash arg (function-map f)))

(defmethod set-value ((f funcallable-map) arg result)
  (setf (gethash arg (function-map f)) result))
```

Meta Object Protocol

Pure Functions are Maps

```
(defclass funcallable-map ()
  ((map :initform (make-hash-table) :reader function-map))
  (:metaclass funcallable-standard-class))

(defmethod get-value ((f funcallable-map) arg)
  (gethash arg (function-map f)))

(defmethod set-value ((f funcallable-map) arg result)
  (setf (gethash arg (function-map f)) result))

(defmethod initialize-instance :after ((f funcallable-map) &key)
  (set-funcallable-instance-function
   f
   (lambda (arg)
     (get-value f arg))))
```

Meta Object Protocol

Pure Functions are Maps

```
> (setf (symbol-function 'car-prices)     ;Let's define a function to
        (make-instance 'funcallable-map)) ;compute the price of a car
#<FUNCALLABLE-MAP @ #x71670612>
```

Meta Object Protocol

Pure Functions are Maps

```
> (setf (symbol-function 'car-prices)       ;Let's define a function to
        (make-instance 'funcallable-map))   ;compute the price of a car
#<FUNCALLABLE-MAP @ #x71670612>
> (car-prices 'audi)                         ;We don't have a price
NIL
NIL
```

Meta Object Protocol

Pure Functions are Maps

```
> (setf (symbol-function 'car-prices)        ;Let's define a function to
        (make-instance 'funcallable-map)) ;compute the price of a car
#<FUNCALLABLE-MAP @ #x71670612>
> (car-prices 'audi)                         ;We don't have a price
NIL
NIL
> (set-value #'car-prices 'audi 40000)       ;Let's give it a price
40000
```

Meta Object Protocol

Pure Functions are Maps

```
> (setf (symbol-function 'car-prices)            ;Let's define a function to
        (make-instance 'funcallable-map))         ;compute the price of a car
#<FUNCALLABLE-MAP @ #x71670612>
> (car-prices 'audi)                              ;We don't have a price
NIL
NIL
> (set-value #'car-prices 'audi 40000)            ;Let's give it a price
40000
> (car-prices 'audi)                              ;OK, now it's working
40000
T
```

Meta Object Protocol

Pure Functions are Maps

```
> (setf (symbol-function 'car-prices)          ;Let's define a function to
        (make-instance 'funcallable-map))      ;compute the price of a car
#<FUNCALLABLE-MAP @ #x71670612>
> (car-prices 'audi)                           ;We don't have a price
NIL
NIL
> (set-value #'car-prices 'audi 40000)         ;Let's give it a price
40000
> (car-prices 'audi)                           ;OK, now it's working
40000
T
> (set-value #'car-prices 'bmw 50000)          ;Let's define more prices
50000
> (set-value #'car-prices 'bentley 150000)
150000
> (set-value #'car-prices 'audi 35000)         ;and redefine some
35000
```

Meta Object Protocol

Pure Functions are Maps

```
> (setf (symbol-function 'car-prices)          ;Let's define a function to
        (make-instance 'funcallable-map))      ;compute the price of a car
#<FUNCALLABLE-MAP @ #x71670612>
> (car-prices 'audi)                           ;We don't have a price
NIL
NIL
> (set-value #'car-prices 'audi 40000)         ;Let's give it a price
40000
> (car-prices 'audi)                           ;OK, now it's working
40000
T
> (set-value #'car-prices 'bmw 50000)          ;Let's define more prices
50000
> (set-value #'car-prices 'bentley 150000)
150000
> (set-value #'car-prices 'audi 35000)         ;and redefine some
35000
> (mapcar #'car-prices
          '(bmw audi bentley))                 ;Let's see some prices
(50000 35000 150000)
```

Meta Object Protocol

CLOS Introspection

- CLOS allows extensive introspection:
  - `class-name`, `class-direct-superclasses`, `class-precedence-list`, `class-slots`, `class-direct-slots`, etc.
  - `slot-definition-name`, `slot-definition-type`, `slot-definition-initform`, `slot-definition-readers`, etc.
  - `generic-function-name`, `generic-function-methods`, `generic-function-method-combination`, `generic-function-lambda-list`, etc.
  - `method-qualifiers`, `method-specializers`, `method-generic-function`, `method-function`, `method-lambda-list`, etc.
- However, “normal” functions are relatively opaque.

Meta Object Protocol

Introspectable Functions

```
(defclass introspectable-function ()
  ((name :initarg :name :reader function-name)
   (arglist :initarg :arglist :reader function-arglist)
   (body :initarg :body :reader function-body))
  (:metaclass funcallable-standard-class))
```

Meta Object Protocol

Introspectable Functions

```
(defclass introspectable-function ()
  ((name :initarg :name :reader function-name)
   (arglist :initarg :arglist :reader function-arglist)
   (body :initarg :body :reader function-body))
  (:metaclass funcallable-standard-class))
```

Meta Object Protocol

Introspectable Functions

```
(defclass introspectable-function ()
  ((name :initarg :name :reader function-name)
   (arglist :initarg :arglist :reader function-arglist)
   (body :initarg :body :reader function-body))
  (:metaclass funcallable-standard-class))
```

Meta Object Protocol

Introspectable Functions

```
(defclass introspectable-function ()
  ((name :initarg :name :reader function-name)
   (arglist :initarg :arglist :reader function-arglist)
   (body :initarg :body :reader function-body))
  (:metaclass funcallable-standard-class))

(defmethod initialize-instance :after ((f introspectable-function)
                                        &key base-function)
  (set-funcallable-instance-function
   f
   base-function))
```

Meta Object Protocol

Introspectable Functions

```
(defclass introspectable-function ()
  ((name :initarg :name :reader function-name)
   (arglist :initarg :arglist :reader function-arglist)
   (body :initarg :body :reader function-body))
  (:metaclass funcallable-standard-class))

(defmethod initialize-instance :after ((f introspectable-function)
                                        &key base-function)
  (set-funcallable-instance-function
   f
   base-function))
```

Fahrenheit from Centigrade

```
(setf (symbol-function 'fahrenheit<-centigrade)
      (make-instance 'introspectable-function
        :name 'fahrenheit<-centigrade
        :arglist '(c)
        :body '(+ (* c 9/5) 32.0)
        :base-function (lambda (c) (+ (* c 9/5) 32.0))))
```

What about Julia?

Introspectable Functions

```
> (fahrenheit<-centigrade 10)
```

What about Julia?

Introspectable Functions

```
> (fahrenheit<-centigrade 10)
50.0
```

What about Julia?

Introspectable Functions

```
> (fahrenheit<-centigrade 10)
50.0
> (function-arglist #'fahrenheit<-centigrade)
```

What about Julia?

Introspectable Functions

```
> (fahrenheit<-centigrade 10)
50.0
> (function-arglist #'fahrenheit<-centigrade)
(C)
```

What about Julia?

Introspectable Functions

```
> (fahrenheit<-centigrade 10)
50.0
> (function-arglist #'fahrenheit<-centigrade)
(C)
> (function-body #'fahrenheit<-centigrade)
```

What about Julia?

Introspectable Functions

```
> (fahrenheit<-centigrade 10)
50.0
> (function-arglist #'fahrenheit<-centigrade)
(C)
> (function-body #'fahrenheit<-centigrade)
(+ (* C 9/5) 32.0)
```

What about Julia?

Introspectable Functions

```
> (fahrenheit<-centigrade 10)
50.0
> (function-arglist #'fahrenheit<-centigrade)
(C)
> (function-body #'fahrenheit<-centigrade)
(+ (* C 9/5) 32.0)
```

Introspectable Functions

```
;To define the fahrenheit<-centigrade function (long version)
   (setf (symbol-function 'fahrenheit<-centigrade)
         (make-instance 'introspectable-function
           :name 'fahrenheit<-centigrade
           :arglist '(c)
           :body '(+ (* c 9/5) 32.0)
           :base-function (lambda (c) (+ (* c 9/5) 32.0))))
```

What about Julia?

Introspectable Functions

```
> (fahrenheit<-centigrade 10)
50.0
> (function-arglist #'fahrenheit<-centigrade)
(C)
> (function-body #'fahrenheit<-centigrade)
(+ (* C 9/5) 32.0)
```

Introspectable Functions

```
;To define the fahrenheit<-centigrade function (long version)
   (setf (symbol-function 'fahrenheit<-centigrade)
         (make-instance 'introspectable-function
           :name 'fahrenheit<-centigrade
           :arglist '(c)
           :body '(+ (* c 9/5) 32.0)
           :base-function (lambda (c) (+ (* c 9/5) 32.0))))

;To define the fahrenheit<-centigrade function (short version)
(define fahrenheit<-centigrade (c)
  (+ (* c 9/5) 32.0))
```

What about Julia?

Introspectable Functions

```
> (fahrenheit<-centigrade 10)
50.0
> (function-arglist #'fahrenheit<-centigrade)
(C)
> (function-body #'fahrenheit<-centigrade)
(+ (* C 9/5) 32.0)
```

Introspectable Functions

```
(defmacro define (name arglist body)
  `(setf (symbol-function ',name)
         (make-instance 'introspectable-function
           :name ',name
           :arglist ',arglist
           :body ',body
           :base-function (lambda ,arglist ,body))))

;To define the fahrenheit<-centigrade function (short version)
(define fahrenheit<-centigrade (c)
  (+ (* c 9/5) 32.0))
```

What about Julia?

Introspectable Functions

```
> (fahrenheit<-centigrade 10)
50.0
> (function-arglist #'fahrenheit<-centigrade)
(C)
> (function-body #'fahrenheit<-centigrade)
(+ (* C 9/5) 32.0)
```

Introspectable Functions

```
(defmacro define (name arglist body)
  `(setf (symbol-function ',name)
         (make-instance 'introspectable-function
           :name ',name
           :arglist ',arglist
           :body ',body
           :base-function (lambda ,arglist ,body))))

;To define the fahrenheit<-centigrade function (short version)
(define fahrenheit<-centigrade (c)
  (+ (* c 9/5) 32.0))
```

What about Julia?

Introspectable Functions

```
> (fahrenheit<-centigrade 10)
50.0
> (function-arglist #'fahrenheit<-centigrade)
(C)
> (function-body #'fahrenheit<-centigrade)
(+ (* C 9/5) 32.0)
```

Introspectable Functions

```
(defmacro define (name arglist body)
  `(setf (symbol-function ',name)
         (make-instance 'introspectable-function
           :name ',name
           :arglist ',arglist
           :body ',body
           :base-function (lambda ,arglist ,body))))

;To define the fahrenheit<-centigrade function (short version)
(define fahrenheit<-centigrade (c)
  (+ (* c 9/5) 32.0))
```

What about Julia?

Introspectable Functions

```
> (fahrenheit<-centigrade 10)
50.0
> (function-arglist #'fahrenheit<-centigrade)
(C)
> (function-body #'fahrenheit<-centigrade)
(+ (* C 9/5) 32.0)
```

Introspectable Functions

```
(defmacro define (name arglist body)
  `(setf (symbol-function ',name)
         (make-instance 'introspectable-function
           :name ',name
           :arglist ',arglist
           :body ',body
           :base-function (lambda ,arglist ,body))))

;To define the fahrenheit<-centigrade function (short version)
(define fahrenheit<-centigrade (c)
  (+ (* c 9/5) 32.0))
```

What about Julia?

Introspectable Functions

```
struct IntrospectableFunction
  name
  parameters
  body
  native_function
end
```

What about Julia?

Introspectable Functions

```
struct IntrospectableFunction
  name
  parameters
  body
  native_function
end

(f::IntrospectableFunction)(x...) = f.native_function(x...)
```

What about Julia?

Introspectable Functions

```
struct IntrospectableFunction
  name
  parameters
  body
  native_function
end

(f::IntrospectableFunction)(x...) = f.native_function(x...)
```

Fahrenheit from Centigrade

```
fahrenheit_from_centigrade =
  IntrospectableFunction(
    :fahrenheit_from_centigrade,
    (:c,),
    :(c*9/5 + 32.0),
    c -> c*9/5 + 32.0)
```

What about Julia?

Introspectable Functions

```
> fahrenheit_from_centigrade(10)
```

What about Julia?

Introspectable Functions

```
> fahrenheit_from_centigrade(10)
50.0
```

What about Julia?

Introspectable Functions

```
> fahrenheit_from_centigrade(10)
50.0
> fahrenheit_from_centigrade.parameters
```

What about Julia?

Introspectable Functions

```
> fahrenheit_from_centigrade(10)
50.0
> fahrenheit_from_centigrade.parameters
(:c,)
```

What about Julia?

Introspectable Functions

```
> fahrenheit_from_centigrade(10)
50.0
> fahrenheit_from_centigrade.parameters
(:c,)
> fahrenheit_from_centigrade.body
```

What about Julia?

Introspectable Functions

```
> fahrenheit_from_centigrade(10)
50.0
> fahrenheit_from_centigrade.parameters
(:c,)
> fahrenheit_from_centigrade.body
:((c * 9) / 5 + 32.0)
```

What about Julia?

Introspectable Functions

```
#To define the fahrenheit<-centigrade function (long version)



fahrenheit_from_centigrade =
      IntrospectableFunction(
        :fahrenheit_from_centigrade,
        (:c,),
        :(c*9/5 + 32.0),
        c -> c*9/5 + 32.0)
```

What about Julia?

Introspectable Functions

```
#To define the fahrenheit<-centigrade function (long version)



fahrenheit_from_centigrade =
      IntrospectableFunction(
        :fahrenheit_from_centigrade,
        (:c,),
        :(c*9/5 + 32.0),
        c -> c*9/5 + 32.0)



#To define the fahrenheit<-centigrade function (short version)
@introspectable fahrenheit_from_centigrade(c) = c*9/5 + 32.0
```

What about Julia?

Introspectable Functions

```
macro introspectable(form)
  let name = form.args[1].args[1],
      parameters = form.args[1].args[2:end],
      body = form.args[2]
    esc(:($(name) =
      IntrospectableFunction(
        $(QuoteNode(name)),
        $((parameters...,)),
        $(QuoteNode(body)),
        ($(parameters...),) -> $body)))
  end
end

#To define the fahrenheit<-centigrade function (short version)
@introspectable fahrenheit_from_centigrade(c) = c*9/5 + 32.0
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5) :reader foo-slot1 :writer set-foo-slot1)
   (slot2 :type string :initarg :slot2 :accessor foo-slot2))
  (:metaclass special-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots
    (list
      (list :name 'slot1
            :initform '(fact 5)
            :initfunction (lambda () (fact 5))
            :readers '(foo-slot1) :writers '(set-foo-slot1))
      (list :name 'slot2
            :type 'string
            :initargs '(:slot2)
            :readers '(foo-slot2) :writers '((setf foo-slot2))))
  :metaclass 'special-class)
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5) :reader foo-slot1 :writer set-foo-slot1)
   (slot2 :type string :initarg :slot2 :accessor foo-slot2))
  (:metaclass special-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots
    (list
      (list :name 'slot1
            :initform '(fact 5)
            :initfunction (lambda () (fact 5))
            :readers '(foo-slot1) :writers '(set-foo-slot1))
      (list :name 'slot2
            :type 'string
            :initargs '(:slot2)
            :readers '(foo-slot2) :writers '((setf foo-slot2))))
  :metaclass 'special-class)
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5) :reader foo-slot1 :writer set-foo-slot1)
   (slot2 :type string :initarg :slot2 :accessor foo-slot2))
  (:metaclass special-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots
    (list
      (list :name 'slot1
            :initform '(fact 5)
            :initfunction (lambda () (fact 5))
            :readers '(foo-slot1) :writers '(set-foo-slot1))
      (list :name 'slot2
            :type 'string
            :initargs '(:slot2)
            :readers '(foo-slot2) :writers '((setf foo-slot2))))
  :metaclass 'special-class)
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5) :reader foo-slot1 :writer set-foo-slot1)
   (slot2 :type string :initarg :slot2 :accessor foo-slot2))
  (:metaclass special-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots
    (list
      (list :name 'slot1
            :initform '(fact 5)
            :initfunction (lambda () (fact 5))
            :readers '(foo-slot1) :writers '(set-foo-slot1))
      (list :name 'slot2
            :type 'string
            :initargs '(:slot2)
            :readers '(foo-slot2) :writers '((setf foo-slot2))))
  :metaclass 'special-class)
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5) :reader foo-slot1 :writer set-foo-slot1)
   (slot2 :type string :initarg :slot2 :accessor foo-slot2))
  (:metaclass special-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots
    (list
      (list :name 'slot1
            :initform '(fact 5)
            :initfunction (lambda () (fact 5))
            :readers '(foo-slot1) :writers '(set-foo-slot1))
      (list :name 'slot2
            :type 'string
            :initargs '(:slot2)
            :readers '(foo-slot2) :writers '((setf foo-slot2))))
  :metaclass 'special-class)
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5) :reader foo-slot1 :writer set-foo-slot1)
   (slot2 :type string :initarg :slot2 :accessor foo-slot2))
  (:metaclass special-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots
    (list
      (list :name 'slot1
            :initform '(fact 5)
            :initfunction (lambda () (fact 5))
            :readers '(foo-slot1) :writers '(set-foo-slot1))
      (list :name 'slot2
            :type 'string
            :initargs '(:slot2)
            :readers '(foo-slot2) :writers '((setf foo-slot2))))
  :metaclass 'special-class)
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5) :reader foo-slot1 :writer set-foo-slot1)
   (slot2 :type string :initarg :slot2 :accessor foo-slot2))
  (:metaclass special-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots
    (list
      (list :name 'slot1
            :initform '(fact 5)
            :initfunction (lambda () (fact 5))
            :readers '(foo-slot1) :writers '(set-foo-slot1))
      (list :name 'slot2
            :type 'string
            :initargs '(:slot2)
            :readers '(foo-slot2) :writers '((setf foo-slot2))))
  :metaclass 'special-class)
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5) :reader foo-slot1 :writer set-foo-slot1)
   (slot2 :type string :initarg :slot2 :accessor foo-slot2))
  (:metaclass special-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots
    (list
      (list :name 'slot1
            :initform '(fact 5)
            :initfunction (lambda () (fact 5))
            :readers '(foo-slot1) :writers '(set-foo-slot1))
      (list :name 'slot2
            :type 'string
            :initargs '(:slot2)
            :readers '(foo-slot2) :writers '((setf foo-slot2))))
  :metaclass 'special-class)
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5) :reader foo-slot1 :writer set-foo-slot1)
   (slot2 :type string :initarg :slot2 :accessor foo-slot2))
  (:metaclass special-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots
    (list
      (list :name 'slot1
            :initform '(fact 5)
            :initfunction (lambda () (fact 5))
            :readers '(foo-slot1) :writers '(set-foo-slot1))
      (list :name 'slot2
            :type 'string
            :initargs '(:slot2)
            :readers '(foo-slot2) :writers '((setf foo-slot2))))
  :metaclass 'special-class)
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5) :reader foo-slot1 :writer set-foo-slot1)
   (slot2 :type string :initarg :slot2 :accessor foo-slot2))
  (:metaclass special-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots
    (list
      (list :name 'slot1
            :initform '(fact 5)
            :initfunction (lambda () (fact 5))
            :readers '(foo-slot1) :writers '(set-foo-slot1))
      (list :name 'slot2
            :type 'string
            :initargs '(:slot2)
            :readers '(foo-slot2) :writers '((setf foo-slot2))))
  :metaclass 'special-class)
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5) :reader foo-slot1 :writer set-foo-slot1)
   (slot2 :type string :initarg :slot2 :accessor foo-slot2))
  (:metaclass special-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots
    (list
      (list :name 'slot1
            :initform '(fact 5)
            :initfunction (lambda () (fact 5))
            :readers '(foo-slot1) :writers '(set-foo-slot1))
      (list :name 'slot2
            :type 'string
            :initargs '(:slot2)
            :readers '(foo-slot2) :writers '((setf foo-slot2))))
  :metaclass 'special-class)
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5) :reader foo-slot1 :writer set-foo-slot1)
   (slot2 :type string :initarg :slot2 :accessor foo-slot2))
  (:metaclass special-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots
    (list
      (list :name 'slot1
            :initform '(fact 5)
            :initfunction (lambda () (fact 5))
            :readers '(foo-slot1) :writers '(set-foo-slot1))
      (list :name 'slot2
            :type 'string
            :initargs '(:slot2)
            :readers '(foo-slot2) :writers '((setf foo-slot2))))
  :metaclass 'special-class)
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :my-special-initform (fact 5) :your-reader foo-slot1)
   (slot2 :his-own-type string our-initarg :slot2))
  (their-option 1 2 3)
  (:metaclass strange-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots (list
                 (list :name 'slot1
                       :my-special-initform '(fact 5)
                       :your-reader 'foo-slot1)
                 (list :name 'slot2
                       :his-own-type 'string
                       'our-initarg ':slot2))
  :metaclass 'strange-class
  'their-option '(1 2 3))
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :my-special-initform (fact 5) :your-reader foo-slot1)
   (slot2 :his-own-type string our-initarg :slot2))
  (their-option 1 2 3)
  (:metaclass strange-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots (list
                 (list :name 'slot1
                       :my-special-initform '(fact 5)
                       :your-reader 'foo-slot1)
                 (list :name 'slot2
                       :his-own-type 'string
                       'our-initarg ':slot2))
  :metaclass 'strange-class
  'their-option '(1 2 3))
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :my-special-initform (fact 5) :your-reader foo-slot1)
   (slot2 :his-own-type string our-initarg :slot2))
  (their-option 1 2 3)
  (:metaclass strange-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots (list
                  (list :name 'slot1
                        :my-special-initform '(fact 5)
                        :your-reader 'foo-slot1)
                  (list :name 'slot2
                        :his-own-type 'string
                        'our-initarg ':slot2))
  :metaclass 'strange-class
  'their-option '(1 2 3))
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :my-special-initform (fact 5) :your-reader foo-slot1)
   (slot2 :his-own-type string our-initarg :slot2))
  (their-option 1 2 3)
  (:metaclass strange-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots (list
                 (list :name 'slot1
                       :my-special-initform '(fact 5)
                       :your-reader 'foo-slot1)
                 (list :name 'slot2
                       :his-own-type 'string
                       'our-initarg ':slot2))
  :metaclass 'strange-class
  'their-option '(1 2 3))
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :my-special-initform (fact 5) :your-reader foo-slot1)
   (slot2 :his-own-type string our-initarg :slot2))
  (their-option 1 2 3)
  (:metaclass strange-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots (list
                    (list :name 'slot1
                          :my-special-initform '(fact 5)
                          :your-reader 'foo-slot1)
                    (list :name 'slot2
                          :his-own-type 'string
                          'our-initarg ':slot2))
  :metaclass 'strange-class
  'their-option '(1 2 3))
```

User Interface Macros

Macro Call

```
(defclass foo (bar baz)
  ((slot1 :my-special-initform (fact 5) :your-reader foo-slot1)
   (slot2 :his-own-type string our-initarg :slot2))
  (their-option 1 2 3)
  (:metaclass strange-class))
```

Macro Expansion

```
(ensure-class 'foo
  :direct-superclasses '(bar baz)
  :direct-slots (list
                  (list :name 'slot1
                        :my-special-initform '(fact 5)
                        :your-reader 'foo-slot1)
                  (list :name 'slot2
                        :his-own-type 'string
                        'our-initarg ':slot2))
  :metaclass 'strange-class
  'their-option '(1 2 3))
```

User Interface Macros

Macro Call

```
(defgeneric xpto (x y &optional z)
  (:argument-precedence-order y x)
  (:generic-function-class special-generic-function)
  (:method-class special-method)
  (:method-combination special-combination))
```

Macro Expansion

```
(ensure-generic-function 'xpto
  :lambda-list '(x y &optional z)
  :initial-methods nil
  :method-combination '(special-combination)
  :method-class 'special-method
  :generic-function-class 'special-generic-function
  :argument-precedence-order '(y x))
```

User Interface Macros

Macro Call

```
(defgeneric xpto (x y &optional z)
  (:argument-precedence-order y x)
  (:generic-function-class special-generic-function)
  (:method-class special-method)
  (:method-combination special-combination))
```

Macro Expansion

```
(ensure-generic-function 'xpto
  :lambda-list '(x y &optional z)
  :initial-methods nil
  :method-combination '(special-combination)
  :method-class 'special-method
  :generic-function-class 'special-generic-function
  :argument-precedence-order '(y x))
```

User Interface Macros

Macro Call

```
(defgeneric xpto (x y &optional z)
  (:argument-precedence-order y x)
  (:generic-function-class special-generic-function)
  (:method-class special-method)
  (:method-combination special-combination))
```

Macro Expansion

```
(ensure-generic-function 'xpto
  :lambda-list '(x y &optional z)
  :initial-methods nil
  :method-combination '(special-combination)
  :method-class 'special-method
  :generic-function-class 'special-generic-function
  :argument-precedence-order '(y x))
```

User Interface Macros

Macro Call

```
(defgeneric xpto (x y &optional z)
  (:argument-precedence-order y x)
  (:generic-function-class special-generic-function)
  (:method-class special-method)
  (:method-combination special-combination))
```

Macro Expansion

```
(ensure-generic-function 'xpto
  :lambda-list '(x y &optional z)
  :initial-methods nil
  :method-combination '(special-combination)
  :method-class 'special-method
  :generic-function-class 'special-generic-function
  :argument-precedence-order '(y x))
```

User Interface Macros

Macro Call

```
(defgeneric xpto (x y &optional z)
  (:argument-precedence-order y x)
  (:generic-function-class special-generic-function)
  (:method-class special-method)
  (:method-combination special-combination))
```

Macro Expansion

```
(ensure-generic-function 'xpto
  :lambda-list '(x y &optional z)
  :initial-methods nil
  :method-combination '(special-combination)
  :method-class 'special-method
  :generic-function-class 'special-generic-function
  :argument-precedence-order '(y x))
```

User Interface Macros

Macro Call

```
(defgeneric xpto (x y &optional z)
  (:argument-precedence-order y x)
  (:generic-function-class special-generic-function)
  (:method-class special-method)
  (:method-combination special-combination))
```

Macro Expansion

```
(ensure-generic-function 'xpto
  :lambda-list '(x y &optional z)
  :initial-methods nil
  :method-combination '(special-combination)
  :method-class 'special-method
  :generic-function-class 'special-generic-function
  :argument-precedence-order '(y x))
```

User Interface Macros

Macro Call

```
(defgeneric xpto (x y &optional z)
  (:argument-precedence-order y x)
  (:generic-function-class special-generic-function)
  (:method-class special-method)
  (:method-combination special-combination))
```

Macro Expansion

```
(ensure-generic-function 'xpto
  :lambda-list '(x y &optional z)
  :initial-methods nil
  :method-combination '(special-combination)
  :method-class 'special-method
  :generic-function-class 'special-generic-function
  :argument-precedence-order '(y x))
```

User Interface Macros

Macro Call

```
(defmethod xpto special-combination ((x foo) (y bar) &optional (z 1))
  (some-special-computation x y z))
```

Macro Expansion

```
(let ((gf (ensure-generic-function 'xpto)))
  (add-method gf
              (make-instance (generic-function-method-class gf)
                :qualifiers '(special-combination)
                :specializers (list (find-class 'foo)
                                    (find-class 'bar))
                :lambda-list '((x foo) (y bar) &optional (z 1))
                :function (lambda (x y &optional (z 1))
                            (block xpto
                              (some-special-computation x y z))))))
```

User Interface Macros

Macro Call

```
(defmethod xpto special-combination ((x foo) (y bar) &optional (z 1))
  (some-special-computation x y z))
```

Macro Expansion

```
(let ((gf (ensure-generic-function 'xpto)))
  (add-method gf
              (make-instance (generic-function-method-class gf)
                :qualifiers '(special-combination)
                :specializers (list (find-class 'foo)
                                    (find-class 'bar))
                :lambda-list '((x foo) (y bar) &optional (z 1))
                :function (lambda (x y &optional (z 1))
                            (block xpto
                              (some-special-computation x y z))))))
```

User Interface Macros

Macro Call

```
(defmethod xpto special-combination ((x foo) (y bar) &optional (z 1))
  (some-special-computation x y z))
```

Macro Expansion

```
(let ((gf (ensure-generic-function 'xpto)))
  (add-method gf
              (make-instance (generic-function-method-class gf)
                :qualifiers '(special-combination)
                :specializers (list (find-class 'foo)
                                    (find-class 'bar))
                :lambda-list '((x foo) (y bar) &optional (z 1))
                :function (lambda (x y &optional (z 1))
                              (block xpto
                                (some-special-computation x y z))))))
```

User Interface Macros

Macro Call

```
(defmethod xpto special-combination ((x foo) (y bar) &optional (z 1))
  (some-special-computation x y z))
```

Macro Expansion

```
(let ((gf (ensure-generic-function 'xpto)))
  (add-method gf
              (make-instance (generic-function-method-class gf)
                :qualifiers '(special-combination)
                :specializers (list (find-class 'foo)
                                    (find-class 'bar))
                :lambda-list '((x foo) (y bar) &optional (z 1))
                :function (lambda (x y &optional (z 1))
                              (block xpto
                                (some-special-computation x y z))))))
```

User Interface Macros

Macro Call

```
(defmethod xpto special-combination ((x foo) (y bar) &optional (z 1))
  (some-special-computation x y z))
```

Macro Expansion

```
(let ((gf (ensure-generic-function 'xpto)))
  (add-method gf
              (make-instance (generic-function-method-class gf)
                :qualifiers '(special-combination)
                :specializers (list (find-class 'foo)
                                    (find-class 'bar))
                :lambda-list '((x foo) (y bar) &optional (z 1))
                :function (lambda (x y &optional (z 1))
                              (block xpto
                                (some-special-computation x y z))))))
```

User Interface Macros

Macro Call

```
(defmethod xpto special-combination ((x foo) (y bar) &optional (z 1))
  (some-special-computation x y z))
```

Macro Expansion

```
(let ((gf (ensure-generic-function 'xpto)))
  (add-method gf
              (make-instance (generic-function-method-class gf)
                :qualifiers '(special-combination)
                :specializers (list (find-class 'foo)
                                    (find-class 'bar))
                :lambda-list '((x foo) (y bar) &optional (z 1))
                :function (lambda (x y &optional (z 1))
                              (block xpto
                                (some-special-computation x y z))))))
```

User Interface Macros

Macro Call

```
(defmethod xpto special-combination ((x foo) (y bar) &optional (z 1))
  (some-special-computation x y z))
```

Macro Expansion

```
(let ((gf (ensure-generic-function 'xpto)))
  (add-method gf
              (make-instance (generic-function-method-class gf)
                :qualifiers '(special-combination)
                :specializers (list (find-class 'foo)
                                    (find-class 'bar))
                :lambda-list '((x foo) (y bar) &optional (z 1))
                :function (lambda (x y &optional (z 1))
                            (block xpto
                              (some-special-computation x y z))))))
```

Giuseppe Attardi, Cinzia Bonini, Maria Rosario Boscotrecase, Tito Flagella, and Mauro Gaspari.
Metalevel Programming in CLOS.
In S. Cook, editor, *Proceedings of the ECOOP '89 European Conference on Object-oriented Programming*, pages 243–256, Nottingham, July 1989. Cambridge University Press.

D. G. Bobrow, R. P. Gabriel, and J. L. White.
CLOS in context — the shape of the design.
In A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, pages 29–61. MIT Press, 1993.

Sonya E. Keene.
Object-Oriented Programming in Common Lisp: A programmer's guide to CLOS.
Addison-Wesley Publishing Company, Cambridge, MA, 1989.

Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow.
The Art of the Metaobject Protocol.

MIT Press, 1991.

Andreas Paepcke.
PCLOS: Stress testing CLOS experiencing the metaobject protocol.
In *OOPSLA/ECOOP*, pages 194–211, 1990.

Andreas Paepcke.
User-level language crafting.
In *Object-Oriented Programming: the CLOS perspective*, pages 66–99. MIT Press, 1993.