Common Lisp Object System Specification

2. Functions in the Programmer Interface

Authors: Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel,
Sonya E. Keene, Gregor Kiczales, and David A. Moon.

Draft Dated: June 15, 1988

For information about obtaining the sources for this document, send an Internet message to common-lisp-object-system-specification-request@sail.stanford.edu.

The authors wish to thank Patrick Dussud, Kenneth Kahn, Jim Kempf, Larry Masinter, Mark Stefik, Daniel L. Weinreb, and Jon L White for their contributions to this document.

At the X3J13 meeting on June 15, 1988, the following motion was adopted:

"The X3J13 Committee hereby accepts chapters 1 and 2 of the Common Lisp Object System, as defined in document 88-002R, for inclusion in the Common Lisp language being specified by this committee. Subsequent changes will be handled through the usual editorial and cleanup processes."

CONTENTS

Introduction

This chapter describes the functions, macros, special forms, and generic functions provided by the Common Lisp Object System Programmer Interface. The Programmer Interface comprises the functions and macros that are sufficient for writing most object-oriented programs.

This chapter is reference material that requires an understanding of the basic concepts of the Common Lisp Object System. The functions are arranged in alphabetical order for convenient reference.

The description of each function, macro, special form, and generic function includes its purpose, its syntax, the semantics of its arguments and returned values, and often an example and cross-references to related functions.

The syntax description for a function, macro, or special form describes its parameters. The following is an example of the format for the syntax description of a function:

Syntax:

F *x y* `&optional` *z* `&key` *k* [*Generic Function*]

This description indicates that the generic function **F** has two required parameters, *x* and *y*. In addition, there is an optional parameter *z* and a keyword parameter *k*.

The generic functions described in this chapter are all standard generic functions. They all use standard method combination.

The description of a generic function includes descriptions of the methods that are defined on that generic function by the Common Lisp Object System. A ***method signature*** is used to describe the parameters and parameter specializers for each method. The following is an example of the format for a method signature:

Method Signature:

F (*x class*) (*y* `t`) `&optional` *z* `&key` *k* [*Primary Method*]

This signature indicates that this method on the generic function **F** has two required parameters, *x*, which must be an instance of the class *class*, and *y*, which can be any object. In addition, there is an optional parameter *z* and a keyword parameter *k*. This signature also indicates that this method on **F** is a primary method and has no qualifiers.

The syntax description for a generic function describes the lambda-list of the generic function itself, while the method signatures describe the lambda-lists of the defined methods.

Any implementation of the Common Lisp Object System is allowed to provide additional methods on the generic functions described in this chapter.

It is useful to categorize the functions and macros according to their role in this standard:

- Tools used for simple object-oriented programming

  These tools allow for defining new classes, methods, and generic functions, and for making instances. Some tools used within method bodies are also listed here. Some of the macros listed here have a corresponding function that performs the same task at a lower level of abstraction.

  call-next-method
  change-class
  defclass
  defgeneric
  defmethod
  generic-flet
  generic-function
  generic-labels
  initialize-instance
  make-instance
  next-method-p
  slot-boundp
  slot-value
  with-accessors
  with-added-methods
  with-slots

- Functions underlying the commonly used macros

  add-method
  class-name
  compute-applicable-methods
  ensure-generic-function
  find-class
  find-method
  function-keywords
  make-instances-obsolete
  no-applicable-method
  no-next-method
  reinitialize-instance
  remove-method
  shared-initialize
  slot-exists-p
  slot-makunbound
  slot-missing
  slot-unbound
  update-instance-for-different-class

update-instance-for-redefined-class

- Tools for declarative method combination

  call-method
  define-method-combination
  invalid-method-error
  method-combination-error
  method-qualifiers

- General Common Lisp support tools

  class-of
  describe
  documentation
  print-object
  symbol-macrolet

Notation

This specification uses an extended Backus Normal Form (BNF) to describe the syntax of the Object System. This section discusses the syntax of BNF expressions. The primary extension used is the following:

$$[\![O]\!]$$

An expression of this form will appear whenever a list of elements is to be spliced into a larger structure and the elements can appear in any order. The symbol O represents a description of the syntax of some number of syntactic elements to be spliced; that description must be of the form

$$O_1 \mid \ldots \mid O_N$$

where each O_i can be either of the form S or of the form $S*$. The expression $[\![O]\!]$ means that a list of the form

$$(O_{i_1} \ldots O_{i_j}) \quad 1 \leq j$$

is spliced into the enclosing expression, such that if $n \neq m$ and $1 \leq n, m \leq j$, then either $O_{i_n} \neq O_{i_m}$ or $O_{i_n} = O_{i_m} = Q_k$, where for some $1 \leq k \leq N$, O_k is of the form Q_k*.

For example, the expression

$$(\texttt{x}\ [\![\ \texttt{A} \mid \texttt{B*} \mid \texttt{C}\ ]\!]\ \texttt{y})$$

means that at most one `A`, any number of `B`'s, and at most one `C` can occur in any order. It is a description of any of these:

```
(x y)
(x B A C y)
(x A B B B B B C y)
(x C B A B B B y)
```

but not any of these:

```
(x B B A A C C y)
(x C B C y)
```

In the first case, both `A` and `C` appear too often, and in the second case `C` appears too often.

A simple indirection extension is introduced in order to make this new syntax more readable:

$$\downarrow O$$

If O is a non-terminal symbol, the right-hand side of its definition is substituted for the entire expression $\downarrow O$. For example, the following BNF is equivalent to the BNF in the previous example:

$$(\texttt{x} \; [\![\downarrow O]\!] \; \texttt{y})$$

$$O ::= \texttt{A} \mid \texttt{B*} \mid \texttt{C}$$

add-method

Standard Generic Function

Purpose:

The generic function **add-method** adds a method to a generic function. It destructively modifies the generic function and returns the modified generic function as its result.

Syntax:

add-method *generic-function method* [*Generic Function*]

Method Signatures:

add-method (*generic-function* `standard-generic-function`) (*method* `method`) [*Primary Method*]

Arguments:

The *generic-function* argument is a generic function object.

The *method* argument is a method object. The lambda-list of the method function must be congruent with the lambda-list of the generic function, or an error is signaled.

Values:

The modified generic function is returned. The result of **add-method** is **eq** to the *generic-function* argument.

Remarks:

If the given method agrees with an existing method of the generic function on parameter specializers and qualifiers, the existing method is replaced. See the section "Agreement on Parameter Specializers and Qualifiers" for a definition of agreement in this context.

If the method object is a method object of another generic function, an error is signaled.

See Also:

"Agreement on Parameter Specializers and Qualifiers"

defmethod

defgeneric

find-method

remove-method

call-method *Macro*

Purpose:

The macro **call-method** is used in method combination. It hides the implementation-dependent details of how methods are called. The macro **call-method** has lexical scope and can only be used within an effective method form.

The macro **call-method** invokes the specified method, supplying it with arguments and with definitions for **call-next-method** and for **next-method-p**. The arguments are the arguments that were supplied to the effective method form containing the invocation of **call-method**. The definitions of **call-next-method** and **next-method-p** rely on the list of method objects given as the second argument to **call-method**.

The **call-next-method** function available to the method that is the first subform will call the first method in the list that is the second subform. The **call-next-method** function available in that method, in turn, will call the second method in the list that is the second subform, and so on, until the list of next methods is exhausted.

Syntax:

call-method *method next-method-list* [*Macro*]

Arguments:

The *method* argument is a method object; the *next-method-list* argument is a list of method objects.

A list whose first element is the symbol **make-method** and whose second element is a Lisp form can be used instead of a method object as the first subform of **call-method** or as an element of the second subform of **call-method**. Such a list specifies a method object whose method function has a body that is the given form.

Values:

The result of **call-method** is the value or values returned by the method invocation.

See Also:

call-next-method

define-method-combination

next-method-p

call-next-method *Function*

Purpose:

The function **call-next-method** can be used within the body of a method defined by a method-defining form to call the next method.

The function **call-next-method** returns the value or values returned by the method it calls. If there is no next method, the generic function **no-next-method** is called.

The type of method combination used determines which methods can invoke **call-next-method**. The standard method combination type allows **call-next-method** to be used within primary methods and **:around** methods. The standard method combination type defines the next method as follows:

- If **call-next-method** is used in an **:around** method, the next method is the next most specific **:around** method, if one is applicable.
- If there are no **:around** methods at all or if **call-next-method** is called by the least specific **:around** method, other methods are called as follows:
  - All the **:before** methods are called, in most-specific-first order. The function **call-next-method** cannot be used in **:before** methods.
  - The most specific primary method is called. Inside the body of a primary method, **call-next-method** may be used to pass control to the next most specific primary method. The generic function **no-next-method** is called if **call-next-method** is used and there are no more primary methods.
  - All the **:after** methods are called in most-specific-last order. The function **call-next-method** cannot be used in **:after** methods.

For further discussion of **call-next-method**, see the sections "Standard Method Combination" and "Built-in Method Combination Types."

Syntax:

call-next-method `&rest` *args* [*Function*]

Arguments:

When **call-next-method** is called with no arguments, it passes the current method's original arguments to the next method. Neither argument defaulting, nor using **setq**, nor rebinding variables with the same names as parameters of the method affects the values **call-next-method** passes to the method it calls.

When **call-next-method** is called with arguments, the next method is called with those arguments. When providing arguments to **call-next-method**, the following rule must be satisfied or an error is signaled: The ordered set of methods applicable for a changed set of arguments for **call-next-method** must be the same as the ordered set of applicable methods for the original arguments to the generic function. Optimizations of the error checking are possible, but they must not change the semantics of **call-next-method**.

If **call-next-method** is called with arguments but omits optional arguments, the next method called defaults those arguments.

Values:

The function **call-next-method** returns the value or values returned by the method it calls.

Remarks:

Further computation is possible after **call-next-method** returns.

The function **call-next-method** has lexical scope and indefinite extent.

For generic functions using a type of method combination defined by the short form of **define-method-combination**, **call-next-method** can be used in **:around** methods only.

The function **next-method-p** can be used to test whether there is a next method.

If **call-next-method** is used in methods that do not support it, an error is signaled.

See Also:

"Method Selection and Combination"

"Standard Method Combination"

"Built-in Method Combination Types"

define-method-combination

next-method-p

no-next-method

change-class

Standard Generic Function

Purpose:

The generic function **change-class** changes the class of an instance to a new class. It destructively modifies and returns the instance.

If in the old class there is any slot of the same name as a local slot in the new class, the value of that slot is retained. This means that if the slot has a value, the value returned by **slot-value** after **change-class** is invoked is **eql** to the value returned by **slot-value** before **change-class** is invoked. Similarly, if the slot was unbound, it remains unbound. The other slots are initialized as described in the section "Changing the Class of an Instance."

Syntax:

change-class *instance new-class* [*Generic Function*]

Method Signatures:

change-class (*instance* `standard-object`) (*new-class* `standard-class`) [*Primary Method*]

change-class (*instance* `t`) (*new-class* `symbol`) [*Primary Method*]

Arguments:

The *instance* argument is a Lisp object.

The *new-class* argument is a class object or a symbol that names a class.

If the second of the above methods is selected, that method invokes **change-class** on *instance* and (`find-class` *new-class*).

Values:

The modified instance is returned. The result of **change-class** is **eq** to the *instance* argument.

Examples:

```
(defclass position () ())

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

```
(defclass rho-theta-position (position)
    ((rho :initform 0)
     (theta :initform 0)))

(defmethod update-instance-for-different-class :before ((old x-y-position)
                                                        (new rho-theta-position)
                                                        &key)
  ;; Copy the position information from old to new to make new
  ;; be a rho-theta-position at the same position as old.
  (let ((x (slot-value old 'x))
        (y (slot-value old 'y)))
    (setf (slot-value new 'rho) (sqrt (+ (* x x) (* y y)))
          (slot-value new 'theta) (atan y x))))

;;; At this point an instance of the class x-y-position can be
;;; changed to be an instance of the class rho-theta-position using
;;; change-class:

(setq p1 (make-instance 'x-y-position :x 2 :y 0))

(change-class p1 'rho-theta-position)

;;; The result is that the instance bound to p1 is now an instance of
;;; the class rho-theta-position.   The update-instance-for-different-class
;;; method performed the initialization of the rho and theta slots based
;;; on the value of the x and y slots, which were maintained by
;;; the old instance.
```

Remarks:

After completing all other actions, **change-class** invokes the generic function **update-instance-for-different-class**. The generic function **update-instance-for-different-class** can be used to assign values to slots in the transformed instance.

The generic function **change-class** has several semantic difficulties. First, it performs a destructive operation that can be invoked within a method on an instance that was used to select that method. When multiple methods are involved because methods are being combined, the methods currently executing or about to be executed may no longer be applicable. Second, some implementations might use compiler optimizations of slot access, and when the class of an instance is changed the assumptions the compiler made might be violated. This implies that a programmer must not use **change-class** inside a method if any methods for that generic function access any slots, or the results are undefined.

change-class

See Also:

"Changing the Class of an Instance"

update-instance-for-different-class

class-name, (setf class-name) *Standard Generic Function*

Purpose:

The generic function **class-name** takes a class object and returns its name.

The generic function **(setf class-name)** takes a class object and sets its name.

Syntax:

class-name *class* [*Generic Function*]

(setf class-name) *new-value class* [*Generic Function*]

Method Signatures:

class-name (*class* `class`) [*Primary Method*]

(setf class-name) *new-value* (*class* `class`) [*Primary Method*]

Arguments:

The *class* argument is a class object.

The *new-value* argument is any object.

Values:

The name of the given class is returned.

Remarks:

The name of an anonymous class is **nil**.

If S is a symbol such that S =(`class-name` C) and C =(`find-class` S), then S is the proper name of C. For further discussion, see the section "Classes."

See Also:

"Classes"

find-class

class-of *Function*

Purpose:

The function **class-of** returns the class of which the given object is an instance.

Syntax:

class-of *object* [*Function*]

Arguments:

The argument to **class-of** may be any Common Lisp object.

Values:

The function **class-of** returns the class of which the argument is an instance.

compute-applicable-methods *Function*

Purpose:

Given a generic function and a set of arguments, the function **compute-applicable-methods** returns the set of methods that are applicable for those arguments. The methods are sorted according to precedence order. See the section "Method Selection and Combination."

Syntax:

compute-applicable-methods *generic-function function-arguments* [*Function*]

Arguments:

The *generic-function* argument is a generic function object. The *function-arguments* argument is a list of the arguments to that generic function.

Values:

The result is a list of the applicable methods in order of precedence.

See Also:

"Method Selection and Combination"

defclass *Macro*

Purpose:

The macro **defclass** defines a new named class. It returns the new class object as its result.

The syntax of **defclass** provides options for specifying initialization arguments for slots, for specifying default initialization values for slots, and for requesting that methods on specified generic functions be automatically generated for reading and writing the values of slots. No reader or writer functions are defined by default; their generation must be explicitly requested.

Defining a new class also causes a type of the same name to be defined. The predicate `(typep` *object class-name*`)` returns true if the class of the given object is *class-name* itself or a subclass of the class *class-name*. A class object can be used as a type specifier. Thus `(typep` *object class*`)` returns true if the class of the *object* is *class* itself or a subclass of *class*.

Syntax:

defclass *class-name* ({*superclass-name*}*) ({*slot-specifier*}*)[[↓ *class-option*]]

class-name::= *symbol*

superclass-name::= *symbol*

slot-specifier::= *slot-name* | (*slot-name* [[↓ *slot-option*]])

slot-name::= *symbol*

slot-option::= {`:reader` *reader-function-name*}* |
{`:writer` *writer-function-name*}* |
{`:accessor` *reader-function-name*}* |
{`:allocation` *allocation-type*} |
{`:initarg` *initarg-name*}* |
{`:initform` *form*} |
{`:type` *type-specifier*} |
{`:documentation` *string*}

reader-function-name::= *symbol*

writer-function-name::= *function-specifier*

function-specifier::= {*symbol* | (`setf` *symbol*)}

initarg-name::= *symbol*

allocation-type::= `:instance` | `:class`

class-option::= (`:default-initargs` *initarg-list*) |
(`:documentation` *string*) |
(`:metaclass` *class-name*)

initarg-list::= {*initarg-name default-initial-value-form*}*

Figure 2–1. Syntax for defclass

Arguments:

The *class-name* argument is a non-**nil** symbol. It becomes the proper name of the new class. If a class with the same proper name already exists and that class is an instance of **standard-class**, and if the **defclass** form for the definition of the new class specifies a class of class **standard-class**, the definition of the existing class is replaced.

Each *superclass-name* argument is a non-**nil** symbol that specifies a direct superclass of the new class. The new class will inherit slots and methods from each of its direct superclasses, from their direct superclasses, and so on. See the section "Inheritance" for a discussion of how slots and methods are inherited.

Each *slot-specifier* argument is the name of the slot or a list consisting of the slot name followed by zero or more slot options. The *slot-name* argument is a symbol that is syntactically valid for use as a Common Lisp variable name. If there are any duplicate slot names, an error is signaled.

The following slot options are available:

- The **:reader** slot option specifies that an unqualified method is to be defined on the generic function named *reader-function-name* to read the value of the given slot. The *reader-function-name* argument is a non-**nil** symbol. The **:reader** slot option may be specified more than once for a given slot.
- The **:writer** slot option specifies that an unqualified method is to be defined on the generic function named *writer-function-name* to write the value of the slot. The *writer-function-name* argument is a function specifier. The **:writer** slot option may be specified more than once for a given slot.
- The **:accessor** slot option specifies that an unqualified method is to be defined on the generic function named *reader-function-name* to read the value of the given slot and that an unqualified method is to be defined on the generic function named `(setf` *reader-function-name*`)` to be used with **setf** to modify the value of the slot. The *reader-function-name* argument is a non-**nil** symbol. The **:accessor** slot option may be specified more than once for a given slot.
- The **:allocation** slot option is used to specify where storage is to be allocated for the given slot. Storage for a slot may be located in each instance or in the class object itself. The value of the *allocation-type* argument can be either the keyword **:instance** or the keyword **:class**. The **:allocation** slot option may be specified once at most for a given slot. If the **:allocation** slot option is not specified, the effect is the same as specifying **:allocation :instance**.
  - If *allocation-type* is **:instance**, a local slot of the given name is allocated in each instance of the class.
  - If *allocation-type* is **:class**, a shared slot of the given name is allocated in the class object created by this **defclass** form. The value of the slot is shared by all instances of the class. If a class C_1 defines such a shared slot, any subclass C_2 of C_1 will share this single slot unless the **defclass** form for C_2 specifies a slot of the same name or there is a superclass

of C_2 that precedes C_1 in the class precedence list of C_2 and that defines a slot of the same name.

- The **:initform** slot option is used to provide a default initial value form to be used in the initialization of the slot. The **:initform** slot option may be specified once at most for a given slot. This form is evaluated every time it is used to initialize the slot. The lexical environment in which this form is evaluated is the lexical environment in which the **defclass** form was evaluated. Note that the lexical environment refers both to variables and to functions. For local slots, the dynamic environment is the dynamic environment in which **make-instance** was called; for shared slots, the dynamic environment is the dynamic environment in which the **defclass** form was evaluated. See the section "Object Creation and Initialization."

  No implementation is permitted to extend the syntax of **defclass** to allow (*slot-name form*) as an abbreviation for (*slot-name* `:initform` *form*).

- The **:initarg** slot option declares an initialization argument named *initarg-name* and specifies that this initialization argument initializes the given slot. If the initialization argument has a value in the call to **initialize-instance**, the value will be stored into the given slot, and the slot's **:initform** slot option, if any, is not evaluated. If none of the initialization arguments specified for a given slot has a value, the slot is initialized according to the **:initform** slot option, if specified. The **:initarg** slot option can be specified more than once for a given slot. The *initarg-name* argument can be any symbol.

- The **:type** slot option specifies that the contents of the slot will always be of the specified data type. It effectively declares the result type of the reader generic function when applied to an object of this class. The result of attempting to store in a slot a value that does not satisfy the type of the slot is undefined. The **:type** slot option may be specified once at most for a given slot. The **:type** slot option is further discussed in the section "Inheritance of Slots and Slot Options."

- The **:documentation** slot option provides a documentation string for the slot.

Each class option is an option that refers to the class as a whole or to all class slots. The following class options are available:

- The **:default-initargs** class option is followed by a list of alternating initialization argument names and default initial value forms. If any of these initialization arguments does not appear in the initialization argument list supplied to **make-instance**, the corresponding default initial value form is evaluated, and the initialization argument name and the form's value are added to the end of the initialization argument list before the instance is created (see the section "Object Creation and Initialization"). The default initial value form is evaluated each time it is used. The lexical environment in which this form is evaluated is the lexical environment in which the **defclass** form was evaluated. The dynamic environment is the dynamic environment in which **make-instance** was called. If an initialization argument name appears more than once in a **:default-initargs** class option, an error is signaled. The **:default-initargs** class option may be specified at most once.

- The **:documentation** class option causes a documentation string to be attached to the class name. The documentation type for this string is **type**. The form `(documentation` *class-name* `'type)` may be used to retrieve the documentation string. The **:documentation** class option may be specified once at most.
- The **:metaclass** class option is used to specify that instances of the class being defined are to have a different metaclass than the default provided by the system (the class **standard-class**). The *class-name* argument is the name of the desired metaclass. The **:metaclass** class option may be specified once at most.

Values:

The new class object is returned as the result.

Remarks:

If a class with the same proper name already exists and that class is an instance of **standard-class**, and if the **defclass** form for the definition of the new class specifies a class of class **standard-class**, the existing class is redefined, and instances of it (and its subclasses) are updated to the new definition at the time that they are next accessed. For details, see "Redefining Classes."

Note the following rules of **defclass** for standard classes:

- It is not required that the superclasses of a class be defined before the **defclass** form for that class is evaluated.
- All the superclasses of a class must be defined before an instance of the class can be made.
- A class must be defined before it can be used as a parameter specializer in a **defmethod** form.

The Object System may be extended to cover situations where these rules are not obeyed.

Some slot options are inherited by a class from its superclasses, and some can be shadowed or altered by providing a local slot description. No class options except **:default-initargs** are inherited. For a detailed description of how slots and slot options are inherited, see the section "Inheritance of Slots and Slot Options."

The options to **defclass** can be extended. It is required that all implementations signal an error if they observe a class option or a slot option that is not implemented locally.

It is valid to specify more than one reader, writer, accessor, or initialization argument for a slot. No other slot option may appear more than once in a single slot description, or an error is signaled.

If no reader, writer, or accessor is specified for a slot, the slot can only be accessed by the function **slot-value**.

See Also:

"Classes"

"Inheritance"

"Redefining Classes"

"Determining the Class Precedence List"

"Object Creation and Initialization"

slot-value

make-instance

initialize-instance

defgeneric *Macro*

Purpose:

The macro **defgeneric** is used to define a generic function or to specify options and declarations that pertain to a generic function as a whole.

If `(fboundp` *function-specifier*`)` is **nil**, a new generic function is created. If `(symbol-function` *function-specifier*`)` is a generic function, that generic function is modified. If *function-specifier* names a non-generic function, a macro, or a special form, an error is signaled.

Each *method-description* defines a method on the generic function. The lambda-list of each method must be congruent with the lambda-list specified by the *lambda-list* option. If this condition does not hold, an error is signaled. See the section "Congruent Lambda-Lists for All Methods of a Generic Function" for a definition of congruence in this context.

The macro **defgeneric** returns the generic function object as its result.

Syntax:

defgeneric *function-specifier lambda-list* [[↓ *option* | *method-description**]] [*Macro*]

function-specifier::= {*symbol* | `(setf` *symbol*`)`}

lambda-list::= ({*var*}*
`[&optional` {*var* | (*var*)}*`]`
`[&rest` *var*`]`
`[&key` {*var* | ({*var* | (*keyword var*)})}*
[`&allow-other-keys`]])

option::= `(:argument-precedence-order` {*parameter-name*}$^{+}$`)` |
`(declare` {*declaration*}$^{+}$`)` |
`(:documentation` *string*`)` |
`(:method-combination` *symbol* {*arg*}*`)` |
`(:generic-function-class` *class-name*`)` |
`(:method-class` *class-name*`)`

method-description::= `(:method` {*method-qualifier*}* *specialized-lambda-list*
{*declaration* | *documentation*}* {*form*}*`)`

method-qualifier::= *non-nil-atom*

specialized-lambda-list::= ({*var* | (*var parameter-specializer-name*)}*
`[&optional` {*var* | (*var* `[`*initform* [*supplied-p-parameter*] `]`)}*`]`
`[&rest` *var*`]`
`[&key` {*var* | ({*var* | (*keyword var*)} `[`*initform* [*supplied-p-parameter*] `]`)}*
`[&allow-other-keys]` `]`
`[&aux` {*var* | (*var* [*initform*])}*`]`)

parameter-specializer-name::= *symbol* | (`eql` *eql-specializer-form*)

Arguments:

The *function-specifier* argument is a non-**nil** symbol or a list of the form `(setf` *symbol*`)`.

The *lambda-list* argument is an ordinary function lambda-list with these exceptions:

- The use of **&aux** is not allowed.
- Optional and keyword arguments may not have default initial value forms nor use supplied-p parameters. The generic function passes to the method all the argument values passed to it, and only those; default values are not supported. Note that optional and keyword arguments in method definitions, however, can have default initial value forms and can use supplied-p parameters.

The following options are provided. A given option may occur only once, or an error is signaled.

- The **:argument-precedence-order** option is used to specify the order in which the required arguments in a call to the generic function are tested for specificity when selecting a particular method. Each required argument, as specified in the *lambda-list* argument, must be included exactly once as a *parameter-name* so that the full and unambiguous precedence order is supplied. If this condition is not met, an error is signaled.
- The **declare** option is used to specify declarations that pertain to the generic function. The following standard Common Lisp declaration is allowed:
  - An **optimize** declaration specifies whether method selection should be optimized for speed or space, but it has no effect on methods. To control how a method is optimized, an **optimize** declaration must be placed directly in the **defmethod** form or method description. The optimization qualities **speed** and **space** are the only qualities this standard requires, but an implementation can extend the Common Lisp Object System to recognize other qualities. A simple implementation that has only one method selection technique and ignores the **optimize** declaration is valid.

  The **special**, **ftype**, **function**, **inline**, **notinline**, and **declaration** declarations are not permitted. Individual implementations can extend the **declare** option to support additional declarations. If an implementation notices a declaration that it does not support and that has not been proclaimed as a non-standard declaration name in a **declaration** proclamation, it should issue a warning.

- The **:documentation** argument associates a documentation string with the generic function. The documentation type for this string is **function**. The form `(documentation` *function-specifier* `'function)` may be used to retrieve this string.
- The **:generic-function-class** option may be used to specify that the generic function is to have a different class than the default provided by the system (the class **standard-generic-function**). The *class-name* argument is the name of a class that can be the class of a generic function. If *function-specifier* specifies an existing generic function that has a different value for the **:generic-function-class** argument and the new generic function class is compatible with the old, **change-class** is called to change the class of the generic function; otherwise an error is signaled.
- The **:method-class** option is used to specify that all methods on this generic function are to have a different class from the default provided by the system (the class **standard-method**). The *class-name* argument is the name of a class that is capable of being the class of a method.
- The **:method-combination** option is followed by a symbol that names a type of method combination. The arguments (if any) that follow that symbol depend on the type of method combination. Note that the standard method combination type does not support any arguments. However, all types of method combination defined by the short form of **define-method-combination** accept an optional argument named *order*, defaulting to **:most-specific-first**, where a value of **:most-specific-last** reverses the order of the primary methods without affecting the order of the auxiliary methods.

The *method-description* arguments define methods that will be associated with the generic function. The *method-qualifier* and *specialized-lambda-list* arguments in a method description are the same as for **defmethod**.

The *form* arguments specify the method body. The body of the method is enclosed in an implicit block. If *function-specifier* is a symbol, this block bears the same name as the generic function. If *function-specifier* is a list of the form `(setf` *symbol*`)`, the name of the block is *symbol*.

Values:

The generic function object is returned as the result.

Remarks:

The effect of the **defgeneric** macro is as if the following three steps were performed: first, methods defined by previous **defgeneric** forms are removed; second, **ensure-generic-function** is called; and finally, methods specified by the current **defgeneric** form are added to the generic function.

If no method descriptions are specified and a generic function of the same name does not already exist, a generic function with no methods is created.

The *lambda-list* argument of **defgeneric** specifies the shape of lambda-lists for the methods on this generic function. All methods on the resulting generic function must have lambda-lists that are congruent with this shape. If a **defgeneric** form is evaluated and some methods for that generic function have lambda-lists that are not congruent with that given in the **defgeneric** form, an error is signaled. For further details on method congruence, see "Congruent Lambda-Lists for All Methods of a Generic Function"

Implementations can extend **defgeneric** to include other options. It is required that an implementation signal an error if it observes an option that is not implemented locally.

See Also:

"Congruent Lambda-Lists for All Methods of a Generic Function"

defmethod

ensure-generic-function

generic-function

define-method-combination *Macro*

Purpose:

The macro **define-method-combination** is used to define new types of method combination.

There are two forms of **define-method-combination**. The short form is a simple facility for the cases that are expected to be most commonly needed. The long form is more powerful but more verbose. It resembles **defmacro** in that the body is an expression, usually using backquote, that computes a Lisp form. Thus arbitrary control structures can be implemented. The long form also allows arbitrary processing of method qualifiers.

Syntax:

define-method-combination *name* [↓ *short-form-option*] [*Macro*]

short-form-option::= `:documentation` *string* |
`:identity-with-one-argument` *boolean* |
`:operator` *operator* |

define-method-combination *name lambda-list* [*Macro*]
({*method-group-specifier*}*)
[(`:arguments` . *lambda-list*)]
[(`:generic-function` *generic-function-symbol*)]
{*declaration* | *doc-string*}*
{*form*}*

method-group-specifier::= (*variable* {{*qualifier-pattern*}+ | *predicate*}
[↓ *long-form-option*])

long-form-option::= `:description` *format-string* |
`:order` *order* |
`:required` *boolean*

Arguments:

In both the short and long forms, *name* is a symbol. By convention, non-keyword, non-**nil** symbols are usually used.

Arguments of the Short Form:

The short form syntax of **define-method-combination** is recognized when the second subform is a non-**nil** symbol or is not present. When the short form is used, *name* is defined as a type of method combination that produces a Lisp form (*operator method-call method-call* ...). The *operator* is a symbol that can be the name of a function, macro, or special form. The *operator* can be specified by a keyword option; it defaults to *name*.

Keyword options for the short form are the following:

- The **:documentation** option is used to document the method-combination type.
- The **:identity-with-one-argument** option enables an optimization when *boolean* is true (the default is false). If there is exactly one applicable method and it is a primary method, that method serves as the effective method and *operator* is not called. This optimization avoids the need to create a new effective method and avoids the overhead of a function call. This option is designed to be used with operators such as **progn**, **and**, +, and **max**.
- The **:operator** option specifies the name of the operator. The *operator* argument is a symbol that can be the name of a function, macro, or special form. By convention, *name* and *operator* are often the same symbol. This is the default, but it is not required.

None of the subforms is evaluated.

These types of method combination require exactly one qualifier per method. An error is signaled if there are applicable methods with no qualifiers or with qualifiers that are not supported by the method combination type.

A method combination procedure defined in this way recognizes two roles for methods. A method whose one qualifier is the symbol naming this type of method combination is defined to be a primary method. At least one primary method must be applicable or an error is signaled. A method with **:around** as its one qualifier is an auxiliary method that behaves the same as a **:around** method in standard method combination. The function **call-next-method** can only be used in **:around** methods; it cannot be used in primary methods defined by the short form of the **define-method-combination** macro.

A method combination procedure defined in this way accepts an optional argument named *order*, which defaults to **:most-specific-first**. A value of **:most-specific-last** reverses the order of the primary methods without affecting the order of the auxiliary methods.

The short form automatically includes error checking and support for **:around** methods.

For a discussion of built-in method combination types, see the section "Built-in Method Combination Types."

Arguments of the Long Form:

The long form syntax of **define-method-combination** is recognized when the second subform is a list.

The *lambda-list* argument is an ordinary lambda-list. It receives any arguments provided after the name of the method combination type in the **:method-combination** option to **defgeneric**.

A list of method group specifiers follows. Each specifier selects a subset of the applicable methods to play a particular role, either by matching their qualifiers against some patterns or by testing their qualifiers with a predicate. These method group specifiers define all method qualifiers that can be used with this type of method combination. If an applicable method does not fall into any method group, the system signals the error that the method is invalid for the kind of method combination in use.

Each method group specifier names a variable. During the execution of the forms in the body of **define-method-combination**, this variable is bound to a list of the methods in the method group. The methods in this list occur in most-specific-first order.

A qualifier pattern is a list or the symbol ∗. A method matches a qualifier pattern if the method's list of qualifiers is **equal** to the qualifier pattern (except that the symbol ∗ in a qualifier pattern matches anything). Thus a qualifier pattern can be one of the following: the empty list **()**, which matches unqualified methods; the symbol ∗, which matches all methods; a true list, which matches methods with the same number of qualifiers as the length of the list when each qualifier matches the corresponding list element; or a dotted list that ends in the symbol ∗ (the ∗ matches any number of additional qualifiers).

Each applicable method is tested against the qualifier patterns and predicates in left-to-right order. As soon as a qualifier pattern matches or a predicate returns true, the method becomes a member of the corresponding method group and no further tests are made. Thus if a method could be a member of more than one method group, it joins only the first such group. If a method group has more than one qualifier pattern, a method need only satisfy one of the qualifier patterns to be a member of the group.

The name of a predicate function can appear instead of qualifier patterns in a method group specifier. The predicate is called for each method that has not been assigned to an earlier method group; it is called with one argument, the method's qualifier list. The predicate should return true if the method is to be a member of the method group. A predicate can be distinguished from a qualifier pattern because it is a symbol other than **nil** or ∗.

If there is an applicable method whose qualifiers are not valid for the method combination type, the function **invalid-method-error** is called.

Method group specifiers can have keyword options following the qualifier patterns or predicate. Keyword options can be distinguished from additional qualifier patterns because they are neither lists nor the symbol ∗. The keyword options are as follows:

- The **:description** option is used to provide a description of the role of methods in the

method group. Programming environment tools use `(apply #'format stream` *format-string* `(method-qualifiers` *method*`))` to print this description, which is expected to be concise. This keyword option allows the description of a method qualifier to be defined in the same module that defines the meaning of the method qualifier. In most cases, *format-string* will not contain any format directives, but they are available for generality. If **:description** is not specified, a default description is generated based on the variable name and the qualifier patterns and on whether this method group includes the unqualified methods. The argument *format-string* is not evaluated.

- The **:order** option specifies the order of methods. The *order* argument is a form that evaluates to **:most-specific-first** or **:most-specific-last**. If it evaluates to any other value, an error is signaled. This keyword option is a convenience and does not add any expressive power. If **:order** is not specified, it defaults to **:most-specific-first**.
- The **:required** option specifies whether at least one method in this method group is required. If the *boolean* argument is non-**nil** and the method group is empty (that is, no applicable methods match the qualifier patterns or satisfy the predicate), an error is signaled. This keyword option is a convenience and does not add any expressive power. If **:required** is not specified, it defaults to **nil**. The *boolean* argument is not evaluated.

The use of method group specifiers provides a convenient syntax to select methods, to divide them among the possible roles, and to perform the necessary error checking. It is possible to perform further filtering of methods in the body forms by using normal list-processing operations and the functions **method-qualifiers** and **invalid-method-error**. It is permissible to use **setq** on the variables named in the method group specifiers and to bind additional variables. It is also possible to bypass the method group specifier mechanism and do everything in the body forms. This is accomplished by writing a single method group with * as its only qualifier pattern; the variable is then bound to a list of all of the applicable methods, in most-specific-first order.

The body *forms* compute and return the Lisp form that specifies how the methods are combined, that is, the effective method. The effective method uses the macro **call-method**. This macro has lexical scope and is available only in an effective method form. Given a method object in one of the lists produced by the method group specifiers and a list of next methods, the macro **call-method** will invoke the method such that **call-next-method** has available the next methods.

When an effective method has no effect other than to call a single method, some implementations employ an optimization that uses the single method directly as the effective method, thus avoiding the need to create a new effective method. This optimization is active when the effective method form consists entirely of an invocation of the **call-method** macro whose first subform is a method object and whose second subform is **nil**. Each **define-method-combination** body is responsible for stripping off redundant invocations of **progn**, **and**, **multiple-value-prog1**, and the like, if this optimization is desired.

The list `(:arguments . lambda-list)` can appear before any declarations or documentation string. This form is useful when the method combination type performs some specific behavior as part of the combined method and that behavior needs access to the arguments to the generic function. Each parameter variable defined by *lambda-list* is bound to a form that can be inserted into the effective method. When this form is evaluated during execution of the effective method, its value is the corresponding argument to the generic function. If *lambda-list* is not congruent to the generic function's lambda-list, additional ignored parameters are automatically inserted until it is congruent. Thus it is permissible for *lambda-list* to receive fewer arguments than the number that the generic function expects.

Erroneous conditions detected by the body should be reported with **method-combination-error** or **invalid-method-error**; these functions add any necessary contextual information to the error message and will signal the appropriate error.

The body *forms* are evaluated inside of the bindings created by the lambda-list and method group specifiers. Declarations at the head of the body are positioned directly inside of bindings created by the lambda-list and outside of the bindings of the method group variables. Thus method group variables cannot be declared.

Within the body *forms*, *generic-function-symbol* is bound to the generic function object.

If a *doc-string* argument is present, it provides the documentation for the method-combination type.

The functions **method-combination-error** and **invalid-method-error** can be called from the body *forms* or from functions called by the body *forms*. The actions of these two functions can depend on implementation-dependent dynamic variables automatically bound before the generic function **compute-effective-method** is called.

Note that two methods with identical specializers, but with different qualifiers, are not ordered by the algorithm described in Step 2 of the method selection and combination process described in the section "Method Selection and Combination." Normally the two methods play different roles in the effective method because they have different qualifiers, and no matter how they are ordered in the result of Step 2, the effective method is the same. If the two methods play the same role and their order matters, an error is signaled. This happens as part of the qualifier pattern matching in **define-method-combination**.

Values:

The value returned by the **define-method-combination** macro is the new method combination object.

Examples:

Most examples of the long form of **define-method-combination** also illustrate the use of the related functions that are provided as part of the declarative method combination facility.

```
;;; Examples of the short form of define-method-combination

(define-method-combination and :identity-with-one-argument t)

(defmethod func and ((x class1) y) ...)

;;; The equivalent of this example in the long form is:

(define-method-combination and
        (&optional (order ':most-specific-first))
        ((around (:around))
         (primary (and) :order order :required t))
  (let ((form (if (rest primary)
                  `(and ,@(mapcar #'(lambda (method)
                                      `(call-method ,method ()))
                                  primary))
                  `(call-method ,(first primary) ()))))
    (if around
        `(call-method ,(first around)
                      (,@(rest around)
                       (make-method ,form)))
        form)))

;;; Examples of the long form of define-method-combination

;The default method-combination technique
(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))))

;A simple way to try several methods until one returns non-nil
(define-method-combination or ()
        ((methods (or)))
  `(or ,@(mapcar #'(lambda (method)
                     `(call-method ,method ()))
                 methods)))

;A more complete version of the preceding
(define-method-combination or
        (&optional (order ':most-specific-first))
        ((around (:around))
         (primary (or)))
  ;; Process the order argument
  (case order
    (:most-specific-first)
    (:most-specific-last (setq primary (reverse primary)))
    (otherwise (method-combination-error "~S is an invalid order.~@
    :most-specific-first and :most-specific-last are the possible values."
                                         order)))
  ;; Must have a primary method
  (unless primary
    (method-combination-error "A primary method is required."))
  ;; Construct the form that calls the primary methods
  (let ((form (if (rest primary)
                  `(or ,@(mapcar #'(lambda (method)
                                     `(call-method ,method ()))
                                 primary))
                  `(call-method ,(first primary) ()))))
    ;; Wrap the around methods around that form
    (if around
        `(call-method ,(first around)
                      (,@(rest around)
                       (make-method ,form)))
        form)))

;The same thing, using the :order and :required keyword options
(define-method-combination or
        (&optional (order ':most-specific-first))
```

```
        ((around (:around))
         (primary (or) :order order :required t))
  (let ((form (if (rest primary)
                  `(or ,@(mapcar #'(lambda (method)
                                     `(call-method ,method ()))
                                 primary))
                  `(call-method ,(first primary) ()))))
    (if around
        `(call-method ,(first around)
                      (,@(rest around)
                       (make-method ,form)))
        form)))

;This short-form call is behaviorally identical to the preceding
(define-method-combination or :identity-with-one-argument t)

;Order methods by positive integer qualifiers
;:around methods are disallowed to keep the example small
(define-method-combination example-method-combination ()
        ((methods positive-integer-qualifier-p))
  `(progn ,@(mapcar #'(lambda (method)
                        `(call-method ,method ()))
                    (stable-sort methods #'<
                      :key #'(lambda (method)
                               (first (method-qualifiers method)))))))

(defun positive-integer-qualifier-p (method-qualifiers)
  (and (= (length method-qualifiers) 1)
       (typep (first method-qualifiers) '(integer 0 *))))

;;; Example of the use of :arguments
(define-method-combination progn-with-lock ()
        ((methods ()))
  (:arguments object)
  `(unwind-protect
       (progn (lock (object-lock ,object))
              ,@(mapcar #'(lambda (method)
                            `(call-method ,method ()))
                        methods))
     (unlock (object-lock ,object))))
```

define-method-combination

Remarks:

The **:method-combination** option of **defgeneric** is used to specify that a generic function should use a particular method combination type. The argument to the **:method-combination** option is the name of a method combination type.

See Also:

"Method Selection and Combination"

"Built-in Method Combination Types"

call-method

method-qualifiers

method-combination-error

invalid-method-error

defgeneric

defmethod *Macro*

Purpose:

The macro **defmethod** defines a method on a generic function.

If `(fboundp` *function-specifier*`)` is **nil**, a generic function is created with default values for the argument precedence order (each argument is more specific than the arguments to its right in the argument list), for the generic function class (the class **standard-generic-function**), for the method class (the class **standard-method**), and for the method combination type (the standard method combination type). The lambda-list of the generic function is congruent with the lambda-list of the method being defined; if the **defmethod** form mentions keyword arguments, the lambda-list of the generic function will mention **&key** (but no keyword arguments). If *function-specifier* names a non-generic function, a macro, or a special form, an error is signaled.

If a generic function is currently named by *function-specifier*, where *function-specifier* is a symbol or a list of the form `(setf` *symbol*`)`, the lambda-list of the method must be congruent with the lambda-list of the generic function. If this condition does not hold, an error is signaled. See the section "Congruent Lambda-Lists for All Methods of a Generic Function" for a definition of congruence in this context.

Syntax:

defmethod *function-specifier* {*method-qualifier*}* *specialized-lambda-list* [*Macro*]
{*declaration* | *documentation*}* {*form*}*

function-specifier::= {*symbol* | `(setf` *symbol*`)`}

method-qualifier::= *non-nil-atom*

specialized-lambda-list::= ({*var* | (*var parameter-specializer-name*)}*
`[&optional` {*var* | (*var* `[`*initform* [*supplied-p-parameter*] `]`)}*`]`
`[&rest` *var*`]`
`[&key` {*var* | ({*var* | (*keyword var*)} `[`*initform* [*supplied-p-parameter*] `]`)}*
`[&allow-other-keys]` `]`
`[&aux` {*var* | (*var* [*initform*])}*`]`)

parameter-specializer-name::= *symbol* | (`eql` *eql-specializer-form*)

defmethod

Arguments:

The *function-specifier* argument is a non-**nil** symbol or a list of the form `(setf` *symbol*`)`. It names the generic function on which the method is defined.

Each *method-qualifier* argument is an object that is used by method combination to identify the given method. A method qualifier is a non-**nil** atom. The method combination type may further restrict what a method qualifier may be. The standard method combination type allows for unqualified methods or methods whose sole qualifier is the keyword **:before**, the keyword **:after**, or the keyword **:around**.

The *specialized-lambda-list* argument is like an ordinary function lambda-list except that the names of required parameters can be replaced by specialized parameters. A specialized parameter is a list of the form (*variable-name parameter-specializer-name*). Only required parameters may be specialized. A parameter specializer name is a symbol that names a class or `(eql` *eql-specializer-form*`)`. The parameter specializer name `(eql` *eql-specializer-form*`)` indicates that the corresponding argument must be **eql** to the object that is the value of *eql-specializer-form* for the method to be applicable. If no parameter specializer name is specified for a given required parameter, the parameter specializer defaults to the class named **t**. See the section "Introduction to Methods" for further discussion.

The *form* arguments specify the method body. The body of the method is enclosed in an implicit block. If *function-specifier* is a symbol, this block bears the same name as the generic function. If *function-specifier* is a list of the form `(setf` *symbol*`)`, the name of the block is *symbol*.

Values:

The result of **defmethod** is the method object.

Remarks:

The class of the method object that is created is that given by the method class option of the generic function on which the method is defined.

If the generic function already has a method that agrees with the method being defined on parameter specializers and qualifiers, **defmethod** replaces the existing method with the one now being defined. See the section "Agreement on Parameter Specializers and Qualifiers" for a definition of agreement in this context.

The parameter specializers are derived from the parameter specializer names as described in the section "Introduction to Methods."

The expansion of the **defmethod** macro "refers to" each specialized parameter (see the description of **ignore** in *Common Lisp: The Language*, p. 160). This includes parameters that have an explicit parameter specializer name of **t**. This means that a compiler warning does not occur if the body of the method does not refer to a specialized parameter. Note that a parameter that specializes on **t** is not synonymous with an unspecialized parameter in this context.

See Also:

"Introduction to Methods"

"Congruent Lambda-Lists for All Methods of a Generic Function"

"Agreement on Parameter Specializers and Qualifiers"

describe

Standard Generic Function

Purpose:

The Common Lisp function **describe** is replaced by a generic function. The generic function **describe** prints information about a given object on the standard output.

Each implementation is required to provide a method on the class **standard-object** and methods on enough other classes so as to ensure that there is always an applicable method. Implementations are free to add methods for other classes. Users can write methods for **describe** for their own classes if they do not wish to inherit an implementation-supplied method. These methods must conform to the definition of **describe** as specified in *Common Lisp: The Language*.

Syntax:

describe *object* [*Generic Function*]

Method Signatures:

describe (*object* `standard-object`) [*Primary Method*]

Arguments:

The *object* argument may be any Common Lisp object.

Values:

The generic function **describe** returns no values.

documentation, (setf documentation) *Standard Generic Function*

Purpose:

The Common Lisp function **documentation** is replaced by a generic function. The generic function **documentation** returns the documentation string associated with the given object if it is available; otherwise it returns **nil**.

The generic function **(setf documentation)** is used to update the documentation.

Syntax:

documentation *x* `&optional` *doc-type* [*Generic Function*]

(setf documentation) *new-value x* `&optional` *doc-type* [*Generic Function*]

Method Signatures:

documentation (*method* `standard-method`) `&optional` *doc-type* [*Primary Method*]

(setf documentation) *new-value* (*method* `standard-method`) `&optional` *doc-type* [*Primary Method*]

documentation (*generic-function* `standard-generic-function`) `&optional` *doc-type* [*Primary Method*]

(setf documentation) *new-value* (*generic-function* `standard-generic-function`) `&optional` *doc-type* [*Primary Method*]

documentation (*class* `standard-class`) `&optional` *doc-type* [*Primary Method*]

(setf documentation) *new-value* (*class* `standard-class`) `&optional` *doc-type* [*Primary Method*]

documentation (*method-combination* `method-combination`) `&optional` *doc-type* [*Primary Method*]

(setf documentation) *new-value* (*method-combination* `method-combination`) `&optional` *doc-type* [*Primary Method*]

documentation, (setf documentation)

documentation (*slot-description* `standard-slot-description`) `&optional` *doc-type* [*Primary Method*]

(setf documentation) *new-value* (*slot-description* `standard-slot-description`) `&optional` *doc-type* [*Primary Method*]

documentation (*symbol* `symbol`) `&optional` *doc-type* [*Primary Method*]

(setf documentation) *new-value* (*symbol* `symbol`) `&optional` *doc-type* [*Primary Method*]

documentation (*list* `list`) `&optional` *doc-type* [*Primary Method*]

(setf documentation) *new-value* (*list* `list`) `&optional` *doc-type* [*Primary Method*]

Arguments:

The first argument of **documentation** is either a symbol, a function specifier list of the form `(setf` *symbol*`)`, a method object, a class object, a generic function object, a method combination object, or a slot description object.

- If the first argument is a method object, a class object, a generic function object, a method combination object, or a slot description object, the second argument must not be supplied, or an error is signaled.
- If the first argument is a symbol or a list of the form `(setf` *symbol*`)`, the second argument must be supplied.
  - The forms `(documentation` *symbol* `'function)` and `(documentation '(setf` *symbol*`) 'function)` return the documentation string of the function, generic function, special form, or macro named by the symbol or list.
  - The form `(documentation` *symbol* `'variable)` returns the documentation string of the special variable or constant named by the symbol.
  - The form `(documentation` *symbol* `'structure)` returns the documentation string of the **defstruct** structure named by the symbol.
  - The form `(documentation` *symbol* `'type)` returns the documentation string of the class object named by the symbol, if there is such a class. If there is no such class, it returns the documentation string of the type specifier named by the symbol.
  - The form `(documentation` *symbol* `'setf)` returns the documentation string of the **defsetf**

or **define-setf-method** definition associated with the symbol.

– The form `(documentation` *symbol* `'method-combination)` returns the documentation string of the method combination type named by the symbol.

An implementation may extend the set of symbols that are acceptable as the second argument. If a symbol is not recognized as an acceptable argument by the implementation, an error must be signaled.

Values:

The documentation string associated with the given object is returned unless none is available, in which case **documentation** returns **nil**.

ensure-generic-function *Function*

Purpose:

The function **ensure-generic-function** is used to define a globally named generic function with no methods or to specify or modify options and declarations that pertain to a globally named generic function as a whole.

If `(fboundp` *function-specifier*`)` is **nil**, a new generic function is created. If `(symbol-function` *function-specifier*`)` is a non-generic function, a macro, or a special form, an error is signaled.

If *function-specifier* specifies a generic function that has a different value for any of the following arguments, the generic function is modified to have the new value: **:argument-precedence-order**, **:declare**, **:documentation**, **:method-combination**.

If *function-specifier* specifies a generic function that has a different value for the **:lambda-list** argument, and the new value is congruent with the lambda-lists of all existing methods or there are no methods, the value is changed; otherwise an error is signaled.

If *function-specifier* specifies a generic function that has a different value for the **:generic-function-class** argument and if the new generic function class is compatible with the old, **change-class** is called to change the class of the generic function; otherwise an error is signaled.

If *function-specifier* specifies a generic function that has a different value for the **:method-class** argument, the value is changed, but any existing methods are not changed.

Syntax:

ensure-generic-function *function-specifier* `&key` `:lambda-list` [*Function*]
`:argument-precedence-order`
`:declare`
`:documentation`
`:generic-function-class`
`:method-combination`
`:method-class`
`:environment`

function-specifier::= {*symbol* | `(setf` *symbol*`)`}

Arguments:

The *function-specifier* argument is a symbol or a list of the form `(setf` *symbol*`)`.

The keyword arguments correspond to the *option* arguments of **defgeneric**, except that the **:method-class** and **:generic-function-class** arguments can be class objects as well as names.

The **:environment** argument is the same as the **&environment** argument to macro expansion functions. It is typically used to distinguish between compile-time and run-time environments.

The **:method-combination** argument is a method combination object.

Values:

The generic function object is returned.

See Also:

defgeneric

find-class *Function*

Purpose:

The function **find-class** returns the class object named by the given symbol in the given environment.

Syntax:

find-class *symbol* `&optional` *errorp environment* [*Function*]

Arguments:

The first argument to **find-class** is a symbol.

If there is no such class and the *errorp* argument is not supplied or is non-**nil**, **find-class** signals an error. If there is no such class and the *errorp* argument is **nil**, **find-class** returns **nil**. The default value of *errorp* is **t**.

The optional *environment* argument is the same as the **&environment** argument to macro expansion functions. It is typically used to distinguish between compile-time and run-time environments.

Values:

The result of **find-class** is the class object named by the given symbol.

Remarks:

The class associated with a particular symbol can be changed by using **setf** with **find-class**. The results are undefined if the user attempts to change the class associated with a symbol that is defined as a type specifier by *Common Lisp: The Language*. See the section "Integrating Types and Classes."

find-method *Standard Generic Function*

Purpose:

The generic function **find-method** takes a generic function and returns the method object that agrees on method qualifiers and parameter specializers with the *method-qualifiers* and *specializers* arguments of **find-method**. See the section "Agreement on Parameter Specializers and Qualifiers" for a definition of agreement in this context.

Syntax:

find-method *generic-function method-qualifiers specializers* `&optional` *errorp* [*Generic Function*]

Method Signatures:

find-method (*generic-function* `standard-generic-function`) [*Primary Method*]
method-qualifiers specializers `&optional` *errorp*

Arguments:

The *generic-function* argument is a generic function.

The *method-qualifiers* argument is a list of the method qualifiers for the method. The order of the method qualifiers is significant.

The *specializers* argument is a list of the parameter specializers for the method. It must correspond in length to the number of required arguments of the generic function, or an error is signaled. This means that to obtain the default method on a given generic function, a list whose elements are the class named **t** must be given.

If there is no such method and the *errorp* argument is not supplied or is non-**nil**, **find-method** signals an error. If there is no such method and the *errorp* argument is **nil**, **find-method** returns **nil**. The default value of *errorp* is **t**.

Values:

The result of **find-method** is the method object with the given method qualifiers and parameter specializers.

See Also:

"Agreement on Parameter Specializers and Qualifiers"

function-keywords *Standard Generic Function*

Purpose:

The generic function **function-keywords** is used to return the keyword parameter specifiers for a given method.

Syntax:

function-keywords *method* [*Generic Function*]

Method Signatures:

function-keywords (*method* `standard-method`) [*Primary Method*]

Arguments:

The *method* argument is a method object.

Values:

The generic function **function-keywords** returns two values: a list of the explicitly named keywords and a boolean that states whether **&allow-other-keys** had been specified in the method definition.

generic-flet *Special Form*

Purpose:

The **generic-flet** special form is analogous to the Common Lisp **flet** special form. It produces new generic functions and establishes new lexical function definition bindings. Each generic function is created with the set of methods specified by its method descriptions.

The special form **generic-flet** is used to define functions whose names are meaningful only locally and to execute a series of forms with these function definition bindings. Any number of such local generic functions may be defined.

The names of functions defined by **generic-flet** have lexical scope; they retain their local definitions only within the body of the **generic-flet**. Any references within the body of the **generic-flet** to functions whose names are the same as those defined within the **generic-flet** are thus references to the local functions instead of to any global functions of the same names. The scope of these generic function definition bindings, however, includes only the body of **generic-flet**, not the definitions themselves. Within the method bodies, local function names that match those being defined refer to global functions defined outside the **generic-flet**. It is thus not possible to define recursive functions with **generic-flet**.

Syntax:

generic-flet ({(*function-specifier lambda-list* [[↓ *option* | *method-description**]])}*) [*Special Form*]
{*form*}*

function-specifier::= {*symbol* | `(setf` *symbol*`)`}

lambda-list::= ({*var*}*
`[&optional` {*var* | (*var*)}*`]`
`[&rest` *var*`]`
`[&key` {*var* | ({*var* | (*keyword var*)})}*
[`&allow-other-keys`] `]`)

option::= `(:argument-precedence-order` {*parameter-name*}$^{+}$`)` |
`(declare` {*declaration*}$^{+}$`)` |
`(:documentation` *string*`)` |
`(:method-combination` *symbol* {*arg*}*`)` |
`(:generic-function-class` *class-name*`)` |
`(:method-class` *class-name*`)`

generic-flet

method-description::= `(:method` {*method-qualifier*}* *specialized-lambda-list*
{*declaration* | *documentation*}* {*form*}*`)`

Arguments:

The *function-specifier*, *lambda-list*, *option*, *method-qualifier*, and *specialized-lambda-list* arguments are the same as for **defgeneric**.

A **generic-flet** local method definition is identical in form to the method definition part of a **defmethod**.

The body of each method is enclosed in an implicit block. If *function-specifier* is a symbol, this block bears the same name as the generic function. If *function-specifier* is a list of the form `(setf` *symbol*`)`, the name of the block is *symbol*.

Values:

The result returned by **generic-flet** is the value or values returned by the last form executed. If no forms are specified, **generic-flet** returns **nil**.

See Also:

generic-labels

defmethod

defgeneric

generic-function

generic-function *Macro*

Purpose:

The **generic-function** macro creates an anonymous generic function. The generic function is created with the set of methods specified by its method descriptions.

Syntax:

generic-function *lambda-list* [*Macro*]
⟦↓ *option* | *method-description** ⟧

lambda-list::= ({*var*}*
`[&optional` {*var* | (*var*)}*`]`
`[&rest` *var*`]`
`[&key` {*var* | ({*var* | (*keyword var*)})}*
[`&allow-other-keys`] `]`)

option::= `(:argument-precedence-order` {*parameter-name*}$^{+}$`)` |
`(declare` {*declaration*}$^{+}$`)` |
`(:documentation` *string*`)` |
`(:method-combination` *symbol* {*arg*}*`)` |
`(:generic-function-class` *class-name*`)` |
`(:method-class` *class-name*`)`

method-description::= `(:method` {*method-qualifier*}* *specialized-lambda-list*
{*declaration* | *documentation*}* {*form*}*`)`

Arguments:

The *option*, *method-qualifier*, and *specialized-lambda-list* arguments are the same as for **defgeneric**.

Values:

The generic function object is returned as the result.

Remarks:

If no method descriptions are specified, an anonymous generic function with no methods is created.

generic-function

See Also:

defgeneric

generic-flet

generic-labels

defmethod

generic-labels *Special Form*

Purpose:

The **generic-labels** special form is analogous to the Common Lisp **labels** special form. It produces new generic functions and establishes new lexical function definition bindings. Each generic function is created with the set of methods specified by its method descriptions.

The special form **generic-labels** is used to define functions whose names are meaningful only locally and to execute a series of forms with these function definition bindings. Any number of such local functions may be defined.

The names of functions defined by **generic-labels** have lexical scope; they retain their local definitions only within the body of the **generic-labels** construct. Any references within the body of the **generic-labels** construct to functions whose names are the same as those defined within the **generic-labels** form are thus references to the local functions instead of to any global functions of the same names. The scope of these generic function definition bindings includes the method bodies themselves as well as the body of the **generic-labels** construct.

Syntax:

generic-labels ({(*function-specifier lambda-list* [[↓ *option* | *method-description**]])}*) {*form*}* [*Special Form*]

function-specifier::= {*symbol* | `(setf` *symbol*`)`}

lambda-list::= ({*var*}*
`[&optional` {*var* | (*var*)}*`]`
`[&rest` *var*`]`
`[&key` {*var* | ({*var* | (*keyword var*)})}*
`[&allow-other-keys]` `]`)

option::= `(:argument-precedence-order` {*parameter-name*}$^{+}$`)` |
`(declare` {*declaration*}$^{+}$`)` |
`(:documentation` *string*`)` |
`(:method-combination` *symbol* {*arg*}*`)` |
`(:generic-function-class` *class-name*`)` |
`(:method-class` *class-name*`)`

method-description::= `(:method` {*method-qualifier*}* *specialized-lambda-list*
{*declaration* | *documentation*}* {*form*}*`)`

generic-labels

Arguments:

The *function-specifier*, *lambda-list*, *option*, *method-qualifier*, and *specialized-lambda-list* arguments are the same as for **defgeneric**.

A **generic-labels** local method definition is identical in form to the method definition part of a **defmethod**.

The body of each method is enclosed in an implicit block. If *function-specifier* is a symbol, this block bears the same name as the generic function. If *function-specifier* is a list of the form `(setf` *symbol*`)`, the name of the block is *symbol*.

Values:

The result returned by **generic-labels** is the value or values returned by the last form executed. If no forms are specified, **generic-labels** returns **nil**.

See Also:

generic-flet

defmethod

defgeneric

generic-function

initialize-instance *Standard Generic Function*

Purpose:

The generic function **initialize-instance** is called by **make-instance** to initialize a newly created instance. The generic function **initialize-instance** is called with the new instance and the defaulted initialization arguments.

The system-supplied primary method on **initialize-instance** initializes the slots of the instance with values according to the initialization arguments and the **:initform** forms of the slots. It does this by calling the generic function **shared-initialize** with the following arguments: the instance, **t** (this indicates that all slots for which no initialization arguments are provided should be initialized according to their **:initform** forms), and the defaulted initialization arguments.

Syntax:

initialize-instance *instance* `&rest` *initargs* [*Generic Function*]

Method Signatures:

initialize-instance (*instance* `standard-object`) `&rest` *initargs* [*Primary Method*]

Arguments:

The *instance* argument is the object to be initialized.

The *initargs* argument consists of alternating initialization argument names and values.

Values:

The modified instance is returned as the result.

Remarks:

Programmers can define methods for **initialize-instance** to specify actions to be taken when an instance is initialized. If only **:after** methods are defined, they will be run after the system-supplied primary method for initialization and therefore will not interfere with the default behavior of **initialize-instance**.

See Also:

"Object Creation and Initialization"

"Rules for Initialization Arguments"

"Declaring the Validity of Initialization Arguments"

initialize-instance

shared-initialize

make-instance

slot-boundp

slot-makunbound

invalid-method-error *Function*

Purpose:

The function **invalid-method-error** is used to signal an error when there is an applicable method whose qualifiers are not valid for the method combination type. The error message is constructed by using a format string and any arguments to it. Because an implementation may need to add additional contextual information to the error message, **invalid-method-error** should be called only within the dynamic extent of a method combination function.

The function **invalid-method-error** is called automatically when a method fails to satisfy every qualifier pattern and predicate in a **define-method-combination** form. A method combination function that imposes additional restrictions should call **invalid-method-error** explicitly if it encounters a method it cannot accept.

Syntax:

invalid-method-error *method format-string* `&rest` *args* [*Function*]

Arguments:

The *method* argument is the invalid method object.

The *format-string* argument is a control string that can be given to **format**, and *args* are any arguments required by that string.

Remarks:

Whether **invalid-method-error** returns to its caller or exits via **throw** is implementation dependent.

See Also:

define-method-combination

make-instance *Standard Generic Function*

Purpose:

The generic function **make-instance** creates and returns a new instance of the given class.

The generic function **make-instance** may be used as described in the section "Object Creation and Initialization."

Syntax:

make-instance *class* `&rest` *initargs* [*Generic Function*]

Method Signatures:

make-instance (*class* `standard-class`) `&rest` *initargs* [*Primary Method*]

make-instance (*class* `symbol`) `&rest` *initargs* [*Primary Method*]

Arguments:

The *class* argument is a class object or a symbol that names a class. The remaining arguments form a list of alternating initialization argument names and values.

If the second of the above methods is selected, that method invokes **make-instance** on the arguments (`find-class` *class*) and *initargs*.

The initialization arguments are checked within **make-instance**. See the section "Object Creation and Initialization."

Values:

The new instance is returned.

Remarks:

The meta-object protocol can be used to define new methods on **make-instance** to replace the object-creation protocol.

See Also:

"Object Creation and Initialization"

defclass

initialize-instance

class-of

make-instances-obsolete *Standard Generic Function*

Purpose:

The generic function **make-instances-obsolete** is invoked automatically by the system when **defclass** has been used to redefine an existing standard class and the set of local slots accessible in an instance is changed or the order of slots in storage is changed. It can also be explicitly invoked by the user.

The function **make-instances-obsolete** has the effect of initiating the process of updating the instances of the class. During updating, the generic function **update-instance-for-redefined-class** will be invoked.

Syntax:

make-instances-obsolete *class* [*Generic Function*]

Method Signatures:

make-instances-obsolete (*class* `standard-class`) [*Primary Method*]

make-instances-obsolete (*class* `symbol`) [*Primary Method*]

Arguments:

The *class* argument is a class object or a symbol that names the class whose instances are to be made obsolete.

If the second of the above methods is selected, that method invokes **make-instances-obsolete** on `(find-class` *class*`)`.

Values:

The modified class is returned. The result of **make-instances-obsolete** is **eq** to the *class* argument supplied to the first of the above methods.

See Also:

"Redefining Classes"

update-instance-for-redefined-class

method-combination-error *Function*

Purpose:

The function **method-combination-error** is used to signal an error in method combination. The error message is constructed by using a format string and any arguments to it. Because an implementation may need to add additional contextual information to the error message, **method-combination-error** should be called only within the dynamic extent of a method combination function.

Syntax:

method-combination-error *format-string* `&rest` *args* [*Function*]

Arguments:

The *format-string* argument is a control string that can be given to **format**, and *args* are any arguments required by that string.

Remarks:

Whether **method-combination-error** returns to its caller or exits via **throw** is implementation dependent.

See Also:

define-method-combination

method-qualifiers *Standard Generic Function*

Purpose:

The generic function **method-qualifiers** returns a list of the qualifiers of the given method.

Syntax:

method-qualifiers *method* [*Generic Function*]

Method Signatures:

method-qualifiers (*method* `standard-method`) [*Primary Method*]

Arguments:

The *method* argument is a method object.

Values:

A list of the qualifiers of the given method is returned.

Examples:

```
(setq methods (remove-duplicates methods
                                 :from-end t
                                 :key #'method-qualifiers
                                 :test #'equal))
```

See Also:

define-method-combination

next-method-p *Function*

Purpose:

The locally defined function **next-method-p** can be used within the body of a method defined by a method-defining form to determine whether a next method exists.

Syntax:

next-method-p [*Function*]

Arguments:

The function **next-method-p** takes no arguments.

Values:

The function **next-method-p** returns true or false.

Remarks:

Like **call-next-method**, the function **next-method-p** has lexical scope and indefinite extent.

See Also:

call-next-method

no-applicable-method *Standard Generic Function*

Purpose:

The generic function **no-applicable-method** is called when a generic function of the class **standard-generic-function** is invoked and no method on that generic function is applicable. The default method signals an error.

The generic function **no-applicable-method** is not intended to be called by programmers. Programmers may write methods for it.

Syntax:

no-applicable-method *generic-function* `&rest` *function-arguments* [*Generic Function*]

Method Signatures:

no-applicable-method (*generic-function* `t`) [*Primary Method*]
`&rest` *function-arguments*

Arguments:

The *generic-function* argument of **no-applicable-method** is the generic function object on which no applicable method was found.

The *function-arguments* argument is a list of the arguments to that generic function.

no-next-method *Standard Generic Function*

Purpose:

The generic function **no-next-method** is called by **call-next-method** when there is no next method. The system-supplied method on **no-next-method** signals an error.

The generic function **no-next-method** is not intended to be called by programmers. Programmers may write methods for it.

Syntax:

no-next-method *generic-function method* `&rest` *args* [*Generic Function*]

Method Signatures:

no-next-method (*generic-function* `standard-generic-function`) [*Primary Method*]
(*method* `standard-method`)
`&rest` *args*

Arguments:

The *generic-function* argument is the generic function object to which the method that is the second argument belongs.

The *method* argument is the method that contained the call to **call-next-method** for which there is no next method.

The *args* argument is a list of the arguments to **call-next-method**.

See Also:

call-next-method

print-object

Standard Generic Function

Purpose:

The generic function **print-object** writes the printed representation of an object to a stream. The function **print-object** is called by the print system; it should not be called by the user.

Each implementation is required to provide a method on the class **standard-object** and methods on enough other classes so as to ensure that there is always an applicable method. Implementations are free to add methods for other classes. Users can write methods for **print-object** for their own classes if they do not wish to inherit an implementation-supplied method.

Syntax:

print-object *object stream* [*Generic Function*]

Method Signatures:

print-object (*object* `standard-object`) *stream* [*Primary Method*]

Arguments:

The first argument is any Lisp object. The second argument is a stream; it cannot be **t** or **nil**.

Values:

The function **print-object** returns its first argument, the object.

Remarks:

Methods on **print-object** must obey the print control special variables described in *Common Lisp: The Language*. The specific details are the following:

- Each method must implement ***print-escape***.
- The ***print-pretty*** control variable can be ignored by most methods other than the one for lists.
- The ***print-circle*** control variable is handled by the printer and can be ignored by methods.
- The printer takes care of ***print-level*** automatically, provided that each method handles exactly one level of structure and calls **write** (or an equivalent function) recursively if there are more structural levels. The printer's decision of whether an object has components (and therefore should not be printed when the printing depth is not less than ***print-level***) is implementation dependent. In some implementations its **print-object** method is not called; in others the method is called, and the determination that the object has components is

based on what it tries to write to the stream.

- Methods that produce output of indefinite length must obey ***print-length***, but most methods other than the one for lists can ignore it.
- The ***print-base***, ***print-radix***, ***print-case***, ***print-gensym***, and ***print-array*** control variables apply to specific types of objects and are handled by the methods for those objects.

If these rules are not obeyed, the results are undefined.

In general, the printer and the **print-object** methods should not rebind the print control variables as they operate recursively through the structure, but this is implementation dependent.

In some implementations the stream argument passed to a **print-object** method is not the original stream, but is an intermediate stream that implements part of the printer. Methods should therefore not depend on the identity of this stream.

All of the existing printing functions (**write**, **prin1**, **print**, **princ**, **pprint**, **write-to-string**, **prin1-to-string**, **princ-to-string**, the **~S** and **~A** format operations, and the **~B**, **~D**, **~E**, **~F**, **~G**, **~$**, **~O**, **~R**, and **~X** format operations when they encounter a non-numeric value) are required to be changed to go through the **print-object** generic function. Each implementation is required to replace its former implementation of printing with one or more **print-object** methods. Exactly which classes have methods for **print-object** is not specified; it would be valid for an implementation to have one default method that is inherited by all system-defined classes.

reinitialize-instance *Standard Generic Function*

Purpose:

The generic function **reinitialize-instance** can be used to change the values of local slots according to initialization arguments. This generic function is called by the Meta-Object Protocol. It can also be called by users.

The system-supplied primary method for **reinitialize-instance** checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. The method then calls the generic function **shared-initialize** with the following arguments: the instance, **nil** (which means no slots should be initialized according to their initforms), and the initialization arguments it received.

Syntax:

reinitialize-instance *instance* `&rest` *initargs* [*Generic Function*]

Method Signatures:

reinitialize-instance (*instance* `standard-object`) `&rest` *initargs* [*Primary Method*]

Arguments:

The *instance* argument is the object to be initialized.

The *initargs* argument consists of alternating initialization argument names and values.

Values:

The modified instance is returned as the result.

Remarks:

Initialization arguments are declared as valid by using the **:initarg** option to **defclass**, or by defining methods for **reinitialize-instance** or **shared-initialize**. The keyword name of each keyword parameter specifier in the lambda-list of any method defined on **reinitialize-instance** or **shared-initialize** is declared as a valid initialization argument name for all classes for which that method is applicable.

See Also:

"Reinitializing an Instance"

"Rules for Initialization Arguments"

"Declaring the Validity of Initialization Arguments"

reinitialize-instance

initialize-instance

shared-initialize

update-instance-for-redefined-class

update-instance-for-different-class

slot-boundp

slot-makunbound

remove-method *Standard Generic Function*

Purpose:

The generic function **remove-method** removes a method from a generic function. It destructively modifies the specified generic function and returns the modified generic function as its result.

Syntax:

remove-method *generic-function method* [*Generic Function*]

Method Signatures:

remove-method (*generic-function* `standard-generic-function`) *method* [*Primary Method*]

Arguments:

The *generic-function* argument is a generic function object.

The *method* argument is a method object. The function **remove-method** does not signal an error if the method is not one of the methods on the generic function.

Values:

The modified generic function is returned. The result of **remove-method** is **eq** to the *generic-function* argument.

See Also:

find-method

shared-initialize *Standard Generic Function*

Purpose:

The generic function **shared-initialize** is used to fill the slots of an instance using initialization arguments and **:initform** forms. It is called when an instance is created, when an instance is re-initialized, when an instance is updated to conform to a redefined class, and when an instance is updated to conform to a different class. The generic function **shared-initialize** is called by the system-supplied primary method for **initialize-instance**, **reinitialize-instance**, **update-instance-for-redefined-class**, and **update-instance-for-different-class**.

The generic function **shared-initialize** takes the following arguments: the instance to be initialized, a specification of a set of names of slots accessible in that instance, and any number of initialization arguments. The arguments after the first two must form an initialization argument list. The system-supplied primary method on **shared-initialize** initializes the slots with values according to the initialization arguments and specified **:initform** forms. The second argument indicates which slots should be initialized according to their **:initform** forms if no initialization arguments are provided for those slots.

The system-supplied primary method behaves as follows, regardless of whether the slots are local or shared:

- If an initialization argument in the initialization argument list specifies a value for that slot, that value is stored into the slot, even if a value has already been stored in the slot before the method is run.
- Any slots indicated by the second argument that are still unbound at this point are initialized according to their **:initform** forms. For any such slot that has an **:initform** form, that form is evaluated in the lexical environment of its defining **defclass** form and the result is stored into the slot. For example, if a **:before** method stores a value in the slot, the **:initform** form will not be used to supply a value for the slot.
- The rules mentioned in the section "Rules for Initialization Arguments" are obeyed.

Syntax:

shared-initialize *instance slot-names* `&rest` *initargs* [*Generic Function*]

Method Signatures:

shared-initialize (*instance* `standard-object`) *slot-names* `&rest` *initargs* [*Primary Method*]

Arguments:

The *instance* argument is the object to be initialized.

The *slots-names* argument specifies the slots that are to be initialized according to their **:initform** forms if no initialization arguments apply. It is supplied in one of three forms as follows:

- It can be list of slot names, which specifies the set of those slot names.
- It can be **nil**, which specifies the empty set of slot names.
- It can be the symbol **t**, which specifies the set of all of the slots.

The *initargs* argument consists of alternating initialization argument names and values.

Values:

The modified instance is returned as the result.

Remarks:

Initialization arguments are declared as valid by using the **:initarg** option to **defclass**, or by defining methods for **shared-initialize**. The keyword name of each keyword parameter specifier in the lambda-list of any method defined on **shared-initialize** is declared as a valid initialization argument name for all classes for which that method is applicable.

Implementations are permitted to optimize **:initform** forms that neither produce nor depend on side effects, by evaluating these forms and storing them into slots before running any **initialize-instance** methods, rather than by handling them in the primary **initialize-instance** method. (This optimization might be implemented by having the **allocate-instance** method copy a prototype instance.)

Implementations are permitted to optimize default initial value forms for initialization arguments associated with slots by not actually creating the complete initialization argument list when the only method that would receive the complete list is the method on **standard-object**. In this case default initial value forms can be treated like **:initform** forms. This optimization has no visible effects other than a performance improvement.

See Also:

"Object Creation and Initialization"

"Rules for Initialization Arguments"

"Declaring the Validity of Initialization Arguments"

initialize-instance

reinitialize-instance

update-instance-for-redefined-class

update-instance-for-different-class

slot-boundp

slot-makunbound

slot-boundp *Function*

Purpose:

The function **slot-boundp** tests whether a specific slot in an instance is bound.

Syntax:

slot-boundp *instance slot-name* [*Function*]

Arguments:

The arguments are the instance and the name of the slot.

Values:

The function **slot-boundp** returns true or false.

Remarks:

The function **slot-boundp** allows for writing **:after** methods on **initialize-instance** in order to initialize only those slots that have not already been bound.

If no slot of the given name exists in the instance, **slot-missing** is called as follows:
`(slot-missing (class-of` *instance*`)` *instance slot-name* `'slot-boundp)`

The function **slot-boundp** is implemented using **slot-boundp-using-class**.

See Also:

slot-missing

slot-exists-p *Function*

Purpose:

The function **slot-exists-p** tests whether the specified object has a slot of the given name.

Syntax:

slot-exists-p *object slot-name* [*Function*]

Arguments:

The *object* argument is any object. The *slot-name* argument is a symbol.

Values:

The function **slot-exists-p** returns true or false.

Remarks:

The function **slot-exists-p** is implemented using **slot-exists-p-using-class**.

slot-makunbound *Function*

Purpose:

The function **slot-makunbound** restores a slot in an instance to the unbound state.

Syntax:

slot-makunbound *instance slot-name* [*Function*]

Arguments:

The arguments to **slot-makunbound** are the instance and the name of the slot.

Values:

The instance is returned as the result.

Remarks:

If no slot of the given name exists in the instance, **slot-missing** is called as follows:
`(slot-missing (class-of` *instance*`)` *instance slot-name* `'slot-makunbound)`

The function **slot-makunbound** is implemented using **slot-makunbound-using-class**.

See Also:

slot-missing

slot-missing *Standard Generic Function*

Purpose:

The generic function **slot-missing** is invoked when an attempt is made to access a slot in an object whose metaclass is **standard-class** and the name of the slot provided is not a name of a slot in that class. The default method signals an error.

The generic function **slot-missing** is not intended to be called by programmers. Programmers may write methods for it.

Syntax:

slot-missing *class object slot-name operation* `&optional` *new-value* [*Generic Function*]

Method Signatures:

slot-missing (*class* `t`) *object slot-name* [*Primary Method*]
operation `&optional` *new-value*

Arguments:

The required arguments to **slot-missing** are the class of the object that is being accessed, the object, the slot name, and a symbol that indicates the operation that caused **slot-missing** to be invoked. The optional argument to **slot-missing** is used when the operation is attempting to set the value of the slot.

Values:

If a method written for **slot-missing** returns values, these values get returned as the values of the original function invocation.

Remarks:

The generic function **slot-missing** may be called during evaluation of **slot-value**, **(setf slot-value)**, **slot-boundp**, and **slot-makunbound**. For each of these operations the corresponding symbol for the *operation* argument is **slot-value**, **setf**, **slot-boundp**, and **slot-makunbound** respectively.

The set of arguments (including the class of the instance) facilitates defining methods on the metaclass for **slot-missing**.

slot-unbound

Standard Generic Function

Purpose:

The generic function **slot-unbound** is called when an unbound slot is read in an instance whose metaclass is **standard-class**. The default method signals an error.

The generic function **slot-unbound** is not intended to be called by programmers. Programmers may write methods for it. The function **slot-unbound** is called only by the function **slot-value-using-class** and thus indirectly by **slot-value**.

Syntax:

slot-unbound *class instance slot-name* [*Generic Function*]

Method Signatures:

slot-unbound (*class* `t`) *instance slot-name* [*Primary Method*]

Arguments:

The arguments to **slot-unbound** are the class of the instance whose slot was accessed, the instance itself, and the name of the slot.

Values:

If a method written for **slot-unbound** returns values, these values get returned as the values of the original function invocation.

Remarks:

An unbound slot may occur if no **:initform** form was specified for the slot and the slot value has not been set, or if **slot-makunbound** has been called on the slot.

See Also:

slot-makunbound

slot-value *Function*

Purpose:

The function **slot-value** returns the value contained in the slot *slot-name* of the given object. If there is no slot with that name, **slot-missing** is called. If the slot is unbound, **slot-unbound** is called.

The macro **setf** can be used with **slot-value** to change the value of a slot.

Syntax:

slot-value *object slot-name* [*Function*]

Arguments:

The arguments are the object and the name of the given slot.

Values:

The result is the value contained in the given slot.

Remarks:

If an attempt is made to read a slot and no slot of the given name exists in the instance, **slot-missing** is called as follows: `(slot-missing (class-of` *instance*`)` *instance slot-name* `'slot-value)`

If an attempt is made to write a slot and no slot of the given name exists in the instance, **slot-missing** is called as follows: `(slot-missing (class-of` *instance*`)` *instance slot-name* `'setf` *new-value*`)`

The function **slot-value** is implemented using **slot-value-using-class**.

Implementations may optimize **slot-value** by compiling it inline.

See Also:

slot-missing

slot-unbound

symbol-macrolet *Macro*

Purpose:

The macro **symbol-macrolet** provides a mechanism for the substitution of forms for variable names within a lexical scope.

Syntax:

symbol-macrolet ((*symbol expansion*)*) `&body` *body* [*Macro*]

Arguments:

The *symbol* argument specifies the symbol with which the form specified by the *expansion* argument is to be associated.

Values:

The result returned is that obtained by executing the forms specified by the *body* argument.

Examples:

```
(symbol-macrolet ((x 'foo))
  (list x (let ((x 'bar)) x)))

;;; The result is (foo bar), not (foo foo).
;;; The expansion is (list 'foo (let ((x 'bar)) x)),
;;; not (list 'foo (let (('foo 'bar)) 'foo)).

(symbol-macrolet ((x (1+ x)))
  (print x))

;;; The expansion is (print (1+ x)),
;;; not (print (1+ (1+ (1+ ....
```

Remarks:

The lexical scope of **symbol-macrolet** is *body*; it does not include *expansion*.

Each reference to *symbol* as a variable within the lexical scope of **symbol-macrolet** is replaced by *expansion* (not the result of evaluating *expansion*).

The use of **symbol-macrolet** can be shadowed by **let**. In other words, **symbol-macrolet** only substitutes for occurrences of *symbol* that would be in the scope of a lexical binding of *symbol* surrounding the body.

symbol-macrolet

The macro **symbol-macrolet** is the basic mechanism that is used to implement **with-slots**.

When the body of the **symbol-macrolet** form is expanded, any use of **setq** to set the value of one of the specified variables is converted to a use of **setf**.

See Also:

with-slots

update-instance-for-different-class *Standard Generic Function*

Purpose:

The generic function **update-instance-for-different-class** is not intended to be called by programmers. Programmers may write methods for it. The function **update-instance-for-different-class** is called only by the function **change-class**.

The system-supplied primary method on **update-instance-for-different-class** checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. This method then initializes slots with values according to the initialization arguments, and initializes the newly added slots with values according to their **:initform** forms. It does this by calling the generic function **shared-initialize** with the following arguments: the instance, a list of names of the newly added slots, and the initialization arguments it received. Newly added slots are those local slots for which no slot of the same name exists in the previous class.

Methods for **update-instance-for-different-class** can be defined to specify actions to be taken when an instance is updated. If only **:after** methods for **update-instance-for-different-class** are defined, they will be run after the system-supplied primary method for initialization and therefore will not interfere with the default behavior of **update-instance-for-different-class**.

Syntax:

update-instance-for-different-class *previous current* `&rest` *initargs* [*Generic Function*]

Method Signatures:

update-instance-for-different-class (*previous* `standard-object`) (*current* `standard-object`) `&rest` *initargs* [*Primary Method*]

Arguments:

The arguments to **update-instance-for-different-class** are computed by **change-class**. When **change-class** is invoked on an instance, a copy of that instance is made; **change-class** then destructively alters the original instance. The first argument to **update-instance-for-different-class**, *previous*, is that copy; it holds the old slot values temporarily. This argument has dynamic extent within **change-class**; if it is referenced in any way once **update-instance-for-different-class** returns, the results are undefined. The second argument to **update-instance-for-different-class**, *current*, is the altered original instance.

The intended use of *previous* is to extract old slot values by using **slot-value** or **with-slots** or by invoking a reader generic function, or to run other methods that were applicable to instances of the original class.

The *initargs* argument consists of alternating initialization argument names and values.

Values:

The value returned by **update-instance-for-different-class** is ignored by **change-class**.

Examples:

See the example for the function **change-class**.

Remarks:

Initialization arguments are declared as valid by using the **:initarg** option to **defclass**, or by defining methods for **update-instance-for-different-class** or **shared-initialize**. The keyword name of each keyword parameter specifier in the lambda-list of any method defined on **update-instance-for-different-class** or **shared-initialize** is declared as a valid initialization argument name for all classes for which that method is applicable.

Methods on **update-instance-for-different-class** can be defined to initialize slots differently from **change-class**. The default behavior of **change-class** is described in "Changing the Class of an Instance."

See Also:

"Changing the Class of an Instance"

"Rules for Initialization Arguments"

"Declaring the Validity of Initialization Arguments"

change-class

shared-initialize

update-instance-for-redefined-class *Standard Generic Function*

Purpose:

The generic function **update-instance-for-redefined-class** is not intended to be called by programmers. Programmers may write methods for it. The generic function **update-instance-for-redefined-class** is called by the mechanism activated by **make-instances-obsolete**.

The system-supplied primary method on **update-instance-for-different-class** checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. This method then initializes slots with values according to the initialization arguments, and initializes the newly added slots with values according to their **:initform** forms. It does this by calling the generic function **shared-initialize** with the following arguments: the instance, a list of names of the newly added slots, and the initialization arguments it received. Newly added slots are those local slots for which no slot of the same name exists in the old version of the class.

Syntax:

update-instance-for-redefined-class *instance* [*Generic Function*]
added-slots discarded-slots
property-list
`&rest` *initargs*

Method Signatures:

update-instance-for-redefined-class (*instance* `standard-object`) [*Primary Method*]
added-slots discarded-slots
property-list
`&rest` *initargs*

Arguments:

When **make-instances-obsolete** is invoked or when a class has been redefined and an instance is being updated, a property list is created that captures the slot names and values of all the discarded slots with values in the original instance. The structure of the instance is transformed so that it conforms to the current class definition. The arguments to **update-instance-for-redefined-class** are this transformed instance, a list of the names of the new slots added to the instance, a list of the names of the old slots discarded from the instance, and the property list containing the slot names and values for slots that were discarded and had values. Included in this list of discarded slots are slots that were local in the old class and are shared in the new class.

The *initargs* argument consists of alternating initialization argument names and values.

update-instance-for-redefined-class

Values:

The value returned by **update-instance-for-redefined-class** is ignored.

Remarks:

Initialization arguments are declared as valid by using the **:initarg** option to **defclass**, or by defining methods for **update-instance-for-redefined-class** or **shared-initialize**. The keyword name of each keyword parameter specifier in the lambda-list of any method defined on **update-instance-for-redefined-class** or **shared-initialize** is declared as a valid initialization argument name for all classes for which that method is applicable.

Examples:

```
(defclass position () ())

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

;;; It turns out polar coordinates are used more than Cartesian
;;; coordinates, so the representation is altered and some new
;;; accessor methods are added.

(defmethod update-instance-for-redefined-class :before
   ((pos x-y-position) added deleted plist &key)
  ;; Transform the x-y coordinates to polar coordinates
  ;; and store into the new slots.
  (let ((x (getf plist 'x))
        (y (getf plist 'y)))
    (setf (position-rho pos) (sqrt (+ (* x x) (* y y)))
          (position-theta pos) (atan y x))))

(defclass x-y-position (position)
    ((rho :initform 0 :accessor position-rho)
     (theta :initform 0 :accessor position-theta)))

;;; All instances of the old x-y-position class will be updated
;;; automatically.

;;; The new representation is given the look and feel of the old one.

(defmethod position-x ((pos x-y-position))
   (with-slots (rho theta) pos (* rho (cos theta))))
```

```
(defmethod (setf position-x) (new-x (pos x-y-position))
   (with-slots (rho theta) pos
     (let ((y (position-y pos)))
       (setq rho (sqrt (+ (* new-x new-x) (* y y)))
             theta (atan y new-x))
       new-x)))

(defmethod position-y ((pos x-y-position))
   (with-slots (rho theta) pos (* rho (sin theta))))

(defmethod (setf position-y) (new-y (pos x-y-position))
   (with-slots (rho theta) pos
     (let ((x (position-x pos)))
       (setq rho (sqrt (+ (* x x) (* new-y new-y)))
             theta (atan new-y x))
       new-y)))
```

See Also:

"Redefining Classes"

"Rules for Initialization Arguments"

"Declaring the Validity of Initialization Arguments"

make-instances-obsolete

shared-initialize

with-accessors *Macro*

Purpose:

The macro **with-accessors** creates a lexical environment in which specified slots are lexically available through their accessors as if they were variables. The macro **with-accessors** invokes the appropriate accessors to access the specified slots. Both **setf** and **setq** can be used to set the value of the slot.

Syntax:

with-accessors ({*slot-entry*}*) *instance-form* `&body` *body* [*Macro*]

slot-entry::= (*variable-name accessor-name*)

Values:

The result returned is that obtained by executing the forms specified by the *body* argument.

Examples:

```
(with-accessors ((x position-x)
                 (y position-y))
                p1
  (setq x y))
```

Remarks:

A **with-accessors** expression of the form:

`(with-accessors` (*slot-entry*$_1$ ... *slot-entry*$_n$) *instance form*$_1$ ... *form*$_k$)

expands into the equivalent of

`(let ((`*in instance*`))`
`(symbol-macrolet (`$Q_1 \ldots Q_n$`)` *form*$_1$ ... *form*$_k$`))`

where Q_i is

`(`*variable-name*$_i$ `(`*accessor-name*$_i$ *in*`))`

See Also:

with-slots

symbol-macrolet

with-added-methods *Special Form*

Purpose:

The **with-added-methods** special form produces new generic functions and establishes new lexical function definition bindings. Each generic function is created by adding the set of methods specified by its method definitions to a copy of the lexically visible generic function of the same name and its methods. If such a generic function does not already exist, a new generic function is created; this generic function has lexical scope.

The special form **with-added-methods** is used to define functions whose names are meaningful only locally and to execute a series of forms with these function definition bindings.

The names of functions defined by **with-added-methods** have lexical scope; they retain their local definitions only within the body of the **with-added-methods** construct. Any references within the body of the **with-added-methods** construct to functions whose names are the same as those defined within the **with-added-methods** form are thus references to the local functions instead of to any global functions of the same names. The scope of these generic function definition bindings includes the method bodies themselves as well as the body of the **with-added-methods** construct.

Syntax:

with-added-methods (*function-specifier lambda-list* [[↓ *option* | *method-description**]]) {*form*}* [*Special Form*]

function-specifier::= {*symbol* | `(setf` *symbol*`)`}

option::= `(:argument-precedence-order` {*parameter-name*}$^{+}$`)` |
`(declare` {*declaration*}$^{+}$`)` |
`(:documentation` *string*`)` |
`(:method-combination` *symbol* {*arg*}*`)` |
`(:generic-function-class` *class-name*`)` |
`(:method-class` *class-name*`)`

method-description::= `(:method` {*method-qualifier*}* *specialized-lambda-list*
{*declaration* | *documentation*}* {*form*}*`)`

Arguments:

The *function-specifier*, *option*, *method-qualifier*, and *specialized-lambda-list* arguments are the same as for **defgeneric**.

The body of each method is enclosed in an implicit block. If *function-specifier* is a symbol, this block bears the same name as the generic function. If *function-specifier* is a list of the form `(setf` *symbol*`)`, the name of the block is *symbol*.

Values:

The result returned by **with-added-methods** is the value or values returned by the last form executed. If no forms are specified, **with-added-methods** returns **nil**.

Remarks:

If a generic function with the given name already exists, the lambda-list specified in the **with-added-methods** form must be congruent with the lambda-lists of all existing methods on that function as well as with the lambda-lists of all methods defined by the **with-added-methods** form; otherwise an error is signaled.

If *function-specifier* specifies an existing generic function that has a different value for any of the following *option* arguments, the copy of that generic function is modified to have the new value: **:argument-precedence-order**, **declare**, **:documentation**, **:generic-function-class**, **:method-combination**.

If *function-specifier* specifies an existing generic function that has a different value for the **:method-class** *option* argument, that value is changed in the copy of that generic function, but any methods copied from the existing generic function are not changed.

If a function of the given name already exists, that function is copied into the default method for a generic function of the given name. Note that this behavior differs from that of **defgeneric**.

If a macro or special form of the given name already exists, an error is signaled.

If there is no existing generic function, the *option* arguments have the same default values as the *option* arguments to **defgeneric**.

See Also:

generic-labels

generic-flet

defmethod

defgeneric

ensure-generic-function

with-slots *Macro*

Purpose:

The macro **with-slots** creates a lexical context for referring to specified slots as though they were variables. Within such a context the value of the slot can be specified by using its slot name, as if it were a lexically bound variable. Both **setf** and **setq** can be used to set the value of the slot.

The macro **with-slots** translates an appearance of the slot name as a variable into a call to **slot-value**.

Syntax:

with-slots ({*slot-entry*}*) *instance-form* `&body` *body* [*Macro*]

slot-entry::= *slot-name* | (*variable-name slot-name*)

Values:

The result returned is that obtained by executing the forms specified by the *body* argument.

Examples:

```
(with-slots (x y) position-1
  (sqrt (+ (* x x) (* y y))))


(with-slots ((x1 x) (y1 y)) position-1
  (with-slots ((x2 x) (y2 y)) position-2
     (psetf x1 x2
            y1 y2))))


  (with-slots (x y) position
    (setq x (1+ x)
          y (1+ y)))
```

Remarks:

A **with-slots** expression of the form:

`(with-slots` (*slot-entry*$_1$ ... *slot-entry*$_n$) *instance form*$_1$ ... *form*$_k$)

expands into the equivalent of

```
(let ((in instance))
 (symbol-macrolet (Q1 ... Qn) form1 ... formk))
```

where Q_i is

```
(slot-entry_i (slot-value in 'slot-entry_i))
```

if *slot-entry*$_i$ is a symbol and is

```
(variable-name_i (slot-value in 'slot-name_i))
```

if *slot-entry*$_i$ is of the form

```
(variable-name_i slot-name_i)
```

See Also:

with-accessors

symbol-macrolet