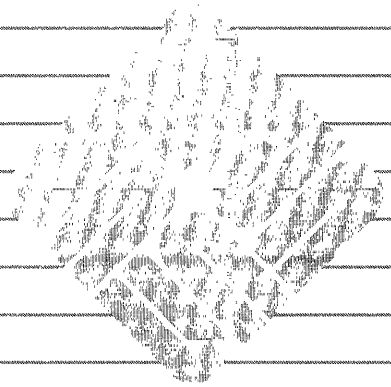
sun microsystems

Sun™ 2.1 Common Lisp Object System



Part Number: 800-2151-10
Revision A of 25 September 1987

Sun™ 2.1 Common Lisp Object System

Part Number: 800-2151-10
Revision A of 25 September 1987

This document is a draft specification from the XJ313 Common Lisp Object System sub-committee. This specification will change in October after the committee has their next formal meeting.

The specification is similar to the otherwise-undocumented Portable Common Loops (PCL) software, however it should not be mistaken as a manual for PCL. There are some significant differences between the specification and the current version of PCL, at the moment the best PCL documentation is the source code itself. A subsequent version of PCL is expected to be a complete implementation of the CLOS specification.

Special Message — Please Read First

Unsupported Product Supplements:

Draft of the Proposed CommonLisp Object System (CLOS) Specification and Source Code for Portable Common Loops (PCL)

Sun Common Lisp Users:

As you know it is Sun's policy to adopt and promote industry standards as they develop. The Sun Common Lisp language as it exists today represents one such standard. There are, in addition, a number of standardization efforts underway in areas relating to Common Lisp, including window toolkits and object-oriented programming systems. Sun is an active participant in these standards organizations and we plan to incorporate these important new capabilities into the Sun Common Lisp product as soon as technically feasible.

We feel it is important for you to be kept up to date on these standardization activities and as a result we have included a recent draft of the proposed standard Common Lisp Object System (CLOS) Specification. This was generated by the ANSI X3J13 standard committee — **it is not a Sun document**. We have also included on the distribution media an experimental object system that is evolving towards CLOS called Portable Common Loops (PCL), developed by `XEROX PARC`.

Both of these items are included **for your information only. The CLOS draft and PCL are NOT part of Sun Common Lisp and are COMPLETELY UNSUPPORTED.** These have simply been included to give you some insight into our general development directions in the area of object-oriented programming systems.

If you choose to experiment with PCL and want to participate in the ongoing discussions of it, contact `CommonLoops-Coordinator.pa@Xerox.com` This is for contributions only, however, and is not a means of receiving support.

Special Message — Please Read First

87–002

Common Lisp Object System Specification

1. Programmer Interface Concepts

This document was written by Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya Keene, Gregor Kiczales, and David A. Moon.

Contributors to this document include Patrick Dussud, Kenneth Kahn, Larry Masinter, Mark Stefik, Daniel L. Weinreb, and Jon L White.

87-002

CONTENTS

87–002

Introduction

This proposal presents a description of the standard Programmer Interface for object-oriented programming in the Common Lisp Object System. The first chapter of this document describes the concepts of the Common Lisp Object System, and the second gives an alphabetic list of the functions and macros that comprise the Common Lisp Object System Programmer Interface. A third chapter, "The Common Lisp Object System Meta-Object Protocol," will describe how the Common Lisp Object System can be customized to support other existing object-oriented paradigms and to define new ones.

The fundamental objects of the Common Lisp Object System are classes, instances, generic functions, and methods.

A ***class*** object determines the structure and behavior of a set of other objects, which are called its ***instances***. It is an important feature of the Object System that every Common Lisp object is an ***instance*** of a class. The class of an object determines the set of operations that can be performed on the object.

A ***generic function*** is a function whose behavior depends on the classes or identities of the arguments supplied to it. A generic function object comprises a set of methods, a lambda-list, a method combination type, and other information. The methods define the class-specific behavior and operations of the generic function. Thus, generic functions are objects that may be ***specialized*** by the definition of methods to provide class-specific operations. A generic function chooses one or more of the set of its methods based on the classes of its arguments.

A generic function can be used in the same ways that an ordinary function can be used in Common Lisp; in particular, a generic function can be used as an argument to **funcall** and **apply** and stored in the function cell of a symbol.

The class-specific operations provided by generic functions are themselves defined and implemented by ***methods***. A method object contains a method function, an ordered set of ***parameter specializers*** that specify when the given method is applicable, and an ordered set of ***qualifiers*** that are used by the method combination facility to distinguish among methods. Each required formal parameter of each method has an associated parameter specializer, and the method is expected to be invoked only on arguments that satisfy its parameter specializers.

To summarize, a generic function is a function that contains or encompasses a number of methods. When a generic function is invoked, the classes of its required arguments determine which methods might be invoked. The behavior of the generic function results from which methods are selected for execution, the order in which the selected methods are called, and how their values are combined to produce the value or values of the generic function. The ***method combination*** facility controls the selection of methods, the order in which they are run, and the values that are returned by the generic function. The Common Lisp Object System offers a default method combination type that is appropriate for most user programs. The Common Lisp Object System also provides a facility for declaring new types of method combination for programs that require them.

87–002

Classes

A ***class*** is an object that determines the structure and behavior of a set of other objects, which are called its ***instances***.

A class can inherit structure and behavior from other classes. A class whose definition refers to other classes for the purpose of inheriting from them is said to be a ***subclass*** of each of those classes. The classes that are designated for purposes of inheritance are said to be ***superclasses*** of the inheriting class.

We say that a class, C_1, is a ***direct superclass*** of a class, C_2, if C_2 explicitly designates C_1 as a superclass in its definition. We say that C_2 is a ***direct subclass*** of C_1. We will say that a class C_n is a ***superclass*** of a class C_1 if there exists a series of classes $C_2, \ldots, C_{n-1}$ such that C_{i+1} is a direct superclass of C_i for $1 \leq i < n$. In this case, we say that C_1 is a ***subclass*** of C_n. A class is considered neither a superclass nor a subclass of itself. That is, if C_1 is a superclass of C_2, then $C_1 \neq C_2$. We refer to the set of classes consisting of some given class C along with all of its superclasses as "C and its superclasses."

When a class is defined, the order in which its direct superclasses are mentioned in the defining form is important. Each class has a ***local precedence order***, which is a list consisting of the class followed by its direct superclasses in the order mentioned in the defining form.

Each class has a ***class precedence list***, which is a total ordering on the set of the given class and its superclasses. The total ordering is expressed as a list ordered from most specific to least specific. The class precedence list is used in several ways. In general, more specific classes can ***shadow***, or override, features that would otherwise be inherited from less specific classes. The method selection and combination process uses the class precedence list to order methods from most specific to least specific.

A class precedence list is always consistent with the local precedence order of each class in the list. The classes in each local precedence order appear within the class precedence list in the same order. If the local precedence orders are inconsistent with each other, no class precedence list can be constructed, and an error will be signaled. The class precedence list and its computation is discussed at length in the section "Determining the Class Precedence List."

Classes are organized into a ***directed acyclic graph***. There is a distinguished class named t. The class t has no superclasses. It is a superclass of every class except itself.

There is a mapping from the Common Lisp type space into the Common Lisp Object System class space. Many of the standard Common Lisp types specified in *Common Lisp: The Language* by Guy L. Steele Jr. have a corresponding class that has the same name as the type. Some Common Lisp types do not have a corresponding class. The integration of the type and class system is discussed later in this chapter.

Classes are represented by first-class objects that are themselves instances of classes. The class of the class of an object is termed the ***metaclass*** of that object. When no misinterpretation is possible, we will also use the term ***metaclass*** to refer to a class that has instances that are

themselves classes. The metaclass determines the form of inheritance used by the classes that are its instances and the representation of the instances of those classes. The Common Lisp Object System provides a default metaclass that is appropriate for most programs. The meta-object protocol will allow for defining and using new metaclasses.

Defining Classes

The macro **defclass** is used to define a new class. The syntax for **defclass** is given in Figure 2-1.

The definition of a class includes:

- The name of the new class.
- The list of the direct superclasses of the new class.
- A set of slot specifiers. Each slot specifier includes the name of the slot and zero or more slot options. A slot option pertains only to a single slot.
- A set of class options. Each class option pertains to the class as a whole.

The slot options and class options of the **defclass** form allow for the following:

- Providing a default initial value form for a given slot.
- Requesting that methods for appropriately named generic functions be automatically generated for reading or writing one or more slots.
- Controlling whether one copy of a given slot is shared by all instances or whether each instance has its own copy of that slot.
- Requesting that a constructor function be automatically generated for making instances of the new class.
- Indicating that the instances of the class are to have a metaclass other than the default.

Creating Instances of Classes

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

The initialization protocol of **make-instance** is not yet specified.

Slots

An object has zero or more named slots. The slots of an object are determined by the class of the object. Each slot can hold one value. The name of a slot is a symbol that can be used as a Common Lisp variable name.

There are two kinds of slots: slots that are local to an individual instance and slots that are shared by all instances of a given class. The **:allocation** slot option controls the kind of slot that is defined. If the value of the **:allocation** slot option is **:instance**, a ***local slot*** is created. This is the most commonly used kind of slot. If the value of **:allocation** is **:class**, a ***shared slot*** is created.

In general, slots are inherited by subclasses. That is, a slot defined by a class is also a slot implicitly defined by any subclass of that class unless the subclass explicitly shadows the slot definition. A class can also shadow some of the slot options declared in the **defclass** form of one of its superclasses by providing its own description for that slot. A detailed explanation of the inheritance of slot options is given in the section "Inheritance of Slots and Slot Options."

We say that a slot is ***accessible*** in an instance of a class if the slot is defined by the class of the instance or is inherited from a superclass of that class. At most, one slot of a given name can be accessible in an instance.

Accessing Slots

Slots can be accessed in two ways: by use of generic functions defined by the **defclass** form and by use of the primitive function **slot-value**.

The **defclass** syntax allows for generating methods to read and write slots. If an ***accessor*** is requested, a method is automatically generated for reading the value of the slot, and a **setf** method for it is also generated. If a ***reader*** is requested, a method is automatically generated for reading the value of the slot, but no **setf** method for it is generated. These methods are added to the appropriate generic functions. Readers and accessors are implemented by using **slot-value**.

Readers and accessors can be specified for individual slots or for all slots. When a reader or accessor is specified for an individual slot, the name of the generic function is directly specified. When readers and accessors are requested for all slots, the names of the generic functions are determined by combining a prefix and the individual slot names. It is possible to modify the behavior of these generic functions by writing methods for them.

The function **slot-value** can be used with any of the slot names specified in the **defclass** form to access a specific slot in an object of the given class. Note that **slot-value** can be used to read or write the value of a slot whether or not accessor functions exist for that slot. When **slot-value** is used, no methods for the accessors are called.

Sometimes it is convenient to access slots from within the body of a method or a function. The macro **with-slots** can be used to set up a lexical environment in which certain slots are lexically available as if they were variables. The macro **with-slots** enables one to specify whether the accessors or the function **slot-value** is to be used to access the slots.

87–002

Inheritance

Inheritance is the key to program modularity within the Common Lisp Object System. A typical object-oriented program consists of several classes, each of which defines some aspect of behavior. New classes are defined by including the appropriate classes as superclasses, thus gathering desired aspects of behavior into one class.

A class can inherit slots and some **defclass** options from its superclasses. This section describes what is inherited from superclasses and how a class can shadow an aspect of inherited behavior.

Inheritance of Methods

A subclass inherits methods in the sense that any method applicable to an instance of a class is also applicable to instances of any subclass of that class (all other arguments to the method being the same).

The inheritance of methods acts the same way regardless of whether the method was created by using **defmethod** or by using one of the **defclass** options that cause methods to be generated automatically.

The inheritance of methods is described in detail in the section "Method Selection and Combination."

Inheritance of Slots and Slot Options

In general, slot descriptions are inherited by subclasses; that is, slots defined by a class are usually slots implicitly defined by any subclass of that class.

In the simplest case, only one class in the class precedence list provides a slot description with a given slot name. If it is a local slot, then each instance of the class and all of its subclasses allocate storage for it. If it is a shared slot, the storage for the slot is allocated by the class that provided the slot description, and the single slot is accessible in instances of that class and all of its subclasses.

More than one class in the class precedence list can provide a slot description with a given slot name. In such cases, at most one slot with a given name is accessible in any instance, and the characteristics of that slot involve some combination of the several slot descriptions.

The following is a simple description of how the characteristics of a slot named S in a class C are determined. The characteristics of a slot are the effective values of the slot options **:allocation**, **:initform**, and **:type**. The **defclass** form for C might not mention the slot S, and whether there is a slot named S in C will also be determined.

Let the class precedence list of C be $(C_1 \ldots C_{n+1})$, $C = C_1$ and $C_{n+1} = \mathbf{t}$. We produce a list of slot option values

$$D = \{(E_1, A_1, T_1, I_1), \ldots, (E_n, A_n, T_n, I_n)\}$$

derived from the class precedence list.

The valid values of these slot option values are the following:

- E_i can be **t** or **nil**. This will indicate whether or not a slot named S was mentioned in the **defclass** form for C_i.
- A_i can be **:instance**, **:class**, or (**:class** j), where (**:class** j) indicates that the class C_i shares a shared slot defined in the **defclass** form for C_j. This will indicate the effective value of the **:allocation** option.
- T_i can be a Common Lisp type. This will indicate the effective value of the **:type** option.
- I_i can be a Common Lisp expression or **unsupplied**. This will indicate the effective value for the **:initform** option.

In overview, the list D is a representation of the effective values of the slot options for the slot named S in the class precedence list for C. The intuitive inheritance rules for these effective values are captured by a simple recursive process of calculating this representation. In brief, this process computes the values for (E_i, A_i, T_i, I_i) from the slot options supplied in the **defclass** form for each class in the class precedence list and from other elements of the list D. If the **defclass** form for C_i supplies a value for one of the options, that value is copied into the corresponding element in the list D. If some options are not specified for some class C_i, the option value in the corresponding element of D is defaulted.

The precise process is as follows: For each C_i in the class precedence list for C, consider the **defclass** form for C_i; let E_i be t if the **defclass** form specifies the slot named S; let A_i be the value supplied for the **:allocation** slot option for the slot S; let T_i be the value supplied for the **:type** slot option for the slot S; and let I_i be the value supplied for the **:initform** slot option for the slot S. If any of these options for the slot S is not explicitly specified by the **defclass** form for C_i, they are defaulted as follows:

- E_i defaults to **nil**.
- For $1 \leq i < n$: If A_{i+1} is **:class** and E_i is **nil**, then A_i defaults to (**:class** $i+1$); if A_{i+1} is (**:class** j) and E_i is **nil**, A_i defaults to (**:class** j); otherwise A_i defaults to **:instance**. A_n defaults to **:instance**.
- T_i defaults to **t**.
- I_i defaults to the value I_{i+1}, $0 \leq i < n$. I_n defaults to **unsupplied**.

If there exists an i, $1 \leq i \leq n$, such E_i is **t**, then the class C has a slot named S. The slot option values for **:allocation**, **:type**, and **:initform** for the slot S in C are derived from the set D as follows:

- The **:allocation** option for C is determined by the value A_1. If it is **:class**, it is a shared slot of the class C; if it is **:instance**, it is a local slot; and if it is (**:class** j), it is a shared slot shared with the class C_j.
- The **:type** option for C is the value T_1, and the contents of the slot S will always be of type (**and** $T_1 \ldots T_n$).
- The **:initform** option for C is the value I_1. If the value is **unsupplied**, the slot is not initialized.

Methods that access slots know only the name of the slot and the type of the slot's value. Suppose a superclass provides a method that expects to access a shared slot of a given name and a subclass provides a local description of a local slot with the same name. If the method provided by the superclass is used on an instance of the subclass, the method accesses the local slot.

The **:reader** and **:accessor** options are not inherited. Reader and accessor methods are inherited in the sense described in the section "Inheritance of Methods."

Inheritance of Class Options

Class options are not inherited. Reader and accessor methods defined by class options are inherited in the sense described in the section "Inheritance of Methods."

Examples of Inheritance

```
(defclass C1 ()
    ((S1 :initform 5.4 :type number)
     (S2 :allocation :class))

(defclass C2 (C1)
    ((S1 :initform 5 :type integer)
     (S2 :allocation :instance)
     (S3 :accessor C2-S3))
  (:reader-prefix C2-))
```

Instances of the class `C1` have a local slot named `S1`, whose default initial value is 5.4 and whose value will always be a number. `C1` also has a shared slot named `S2`.

There is a local slot named `S1` in instances of `C2`. The default initial value of `S1` is 5. The value of `S1` will be of type `(and integer number)`. There are also local slots named `S2` and `S3` in instances

of C2. C2 has methods defined for reading the value of its slots; these methods are for the generic functions named C2-S1, C2-S2, and C2-S3. There is also a method for **setf** of C2-S3 that writes the value of S3.

Redefining Classes

When a **defclass** form is evaluated and a class with the given name already exists, the existing class is redefined. Redefining a class modifies the existing class object to reflect the new class definition; it does not create a new class object for the class. Any method created by an **:accessor**, **:reader**, **:accessor-prefix**, or **:reader-prefix** option specified by the old **defclass** form is removed from the corresponding generic function unless that same method is specified by the new **defclass** form; any function specified by the **:constructor** option of the old **defclass** form is removed from the corresponding symbol function cell. Let C be the class being redefined, and let M_C be the set of methods removed. When C is redefined, an obsolete copy of the old class is made for use during the process of updating the instances of C. Call the obsolete copy C_o. The class C_o has no name, and it has no instances except during a particular part of the process of updating instances.

When the class C is redefined, changes are propagated to instances of it and to instances of any of its subclasses. The updating of an instance whose class has been redefined (or any of whose superclasses have been redefined) occurs at an implementation-dependent time, but will usually be upon the next access to that instance or the next time that a generic function is applied to that instance. In this context, an instance is said to be ***accessed*** when a generic function is called with the instance as an argument that is used for method selection or when **slot-value** is called with the instance as its first argument. Updating an instance does not change its identity as defined by the **eq** function. The updating process may change the slots of that particular instance, but it does not create a new instance. Whether updating an instance consumes storage is implementation dependent.

If a class is redefined in such a way that the set of local slots accessible in an instance of the class is changed or the order of slots in storage is changed, the function **change-class** is called whenever an instance of the class is updated. Implementations may choose to invoke **change-class** under other circumstances as well. The function **change-class** always calls **class-changed** to complete the transformation from the old representation of the instance to the new representation. The generic function **class-changed** is provided to allow programmers to specify actions to be taken when an instance is updated.

Because the class C_o does not have a name, any method for **class-changed** that is specialized to a particular C_o must be written using the functional interface, that is, by using **add-method** rather than by using **defmethod**. See the section "Introduction to Methods."

Note that redefining a class may cause slots to be added or deleted. When an instance is updated, new slots are initialized and the values of deleted slots are discarded. Each local slot of the new version of the class with no slot by the same name in the old version of the class is initialized to the value of the corresponding **:initform** option of the new class or remains uninitialized if the new version of the class does not specify or inherit the **:initform** option for that slot. This is the same as the initialization done by **make-instance** except that no initialization methods are invoked and no **make-instance** initialization arguments are present.

If in the new version of the class there is a local slot of the same name as any slot in the old version of the class, the value of that slot is unchanged. 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 uninitialized, it remains uninitialized. If in the new version of the class there is a shared slot of the same name as any shared slot in the old version of the class, the value of that slot will be the same in both. If in the new version of the class there is a shared slot of the same name as any local slot in the old version of the class, that shared slot is initialized to the value of the corresponding **:initform** option of the new class or remains uninitialized if the new version of the class does not specify or inherit the **:initform** option for that slot.

After **change-class** has completed the above operations, it invokes the generic function **class-changed** on two arguments computed by **change-class**. The first argument passed is a copy of the instance being updated and is an instance of the obsolete class C_o; call this instance I_o. The second argument is the instance as updated so far by **change-class** and is an instance of the class C. The methods M_C, which were removed by the redefinition of the class C, apply to instances of C_o; for example, they apply to I_o. The methods defined on **class-changed** can use the methods in M_C—by invoking the appropriate generic functions—to obtain information that is not directly stored in I_o. I_o is constrained to have dynamic extent, so referencing it outside of the dynamic extent of **class-changed** is an error. Note that C_o has an instance within the dynamic extent of **class-changed**; this is the only time that any instances of C_o will exist.

The Common Lisp Object System guarantees that **defclass** can be used to change the definition of an existing class that was previously defined by **defclass** as long as the **:metaclass** option is not used in either the old or the new definition. Whether **defclass** is allowed to change the metaclass and whether redefining a class causes existing instances to be updated is up to the implementor of the particular metaclass. "The Common Lisp Object System Meta-Object Protocol" will describe how to control this.

87–002

Integrating Types and Classes

The Common Lisp Object System maps the Common Lisp type space into the space of classes. Many but not all of the predefined Common Lisp type specifiers have a class associated with them that has the same name as the type. For example, an array is of type **array** and of class **array**.

A class that corresponds to a predefined Common Lisp type is called a ***standard type class***. Each standard type class has the class **standard-type-class** as a metaclass. Users can write methods that discriminate on any primitive Common Lisp type that has a corresponding class. However, it is not allowed to make an instance of a standard type class with **make-instance** or to include a standard type class as a superclass of a class.

Which Common Lisp types will have corresponding classes is still under discussion.

Creating a type by means of **defstruct** creates a class in the space of Common Lisp classes. Such a class is an instance of **structure-class** and a direct subclass of the class that corresponds to the type given as its **:includes** argument, if any.

Every class that has a name has a corresponding type with the same name. In addition, every class object is a valid type specifier. Thus the expression (`typep` *object class*) evaluates to true if the class of *object* is *class* itself or a subclass of *class*. The evaluation of the expression (`subtypep` *class1 class2*) returns the values **t t** if *class1* is a subclass of *class2* or if they are the same class; otherwise it returns the values **nil t**.

87–002

Determining the Class Precedence List

The **defclass** form for a class provides a total ordering on that class and its direct superclasses. This ordering is called the ***local precedence order***. It is an ordered list of the class and its direct superclasses. A class precedes its direct superclasses, and a direct superclass precedes all other direct superclasses specified to its right in the superclasses list of the **defclass** form. For every class C we define

$$R_C = \{(C, C_1), (C_1, C_2), \ldots, (C_{n-1}, C_n)\}$$

where $C_1, \ldots, C_n$ are the direct superclasses of C in the order in which they are mentioned in the **defclass** form. These ordered pairs generate the total ordering on the class C and its direct superclasses.

Let S_C be the set of C and its superclasses. Let R be

$$R = \bigcup_{c \in S_C} R_c$$

R may or may not generate a partial ordering, depending on whether the R_c, $c \in S_C$, are consistent; we assume they are consistent and that R generates a partial ordering. When the R_c are not consistent we say that R is inconsistent. This partial ordering is the transitive closure of R. When $(C_1, C_2) \in R$, we say that C_1 ***precedes*** C_2.

To compute the class precedence list at C, we topologically sort the elements of S_C with respect to the partial ordering generated by R. When the topological sort must select a class from a set of two or more classes, none of which are preceded by other classes with respect to R, the class selected is chosen deterministically, as described below.

We require that an implementation of Common Lisp Object System signal an error if R is inconsistent, that is, if the class precedence list cannot be computed.

Topological Sorting

Topological sorting proceeds by finding a class C in S_C such that no other class precedes that element according to the elements in R. C is placed first in the result. We remove C from S_C, and we remove all pairs of the form (C, D), $D \in S_C$, from R. We repeat the process, adding classes with no predecessors at the end of the result. We stop when no element can be found that has no predecessor.

If S_C is not empty and the process has stopped, the set R is inconsistent: if every class in the finite set of classes is preceded by another, then R contains a loop, and there are two classes, C_1 and C_2, such that C_1 precedes C_2 and C_2 precedes C_1.

Sometimes there are several classes from S_C with no predecessors. In this case we select the one that has a direct subclass rightmost in the class precedence list computed so far. Because a direct superclass precedes all other direct superclasses to its right, there can be only one such candidate class. If there are no such candidate classes, R does not generate a partial ordering—the R_c, $c \in S_C$, are inconsistent.

In more precise terms, let $\{N_1, \ldots, N_m\}$, $2 \leq m$, be the classes from S_C with no predecessors. Let $(C_1 \ldots C_n)$, $1 \leq n$, be the class precedence list constructed so far. C_1 is the most specific class and C_n is the least specific. Let $1 \leq j \leq n$ be the largest number such that $\exists i$ where $1 \leq i \leq m$ and N_i is a direct superclass of C_j; N_i is placed next.

The effect of this rule for selecting from a set of classes with no predecessors is that simple superclass chains and relatively separated subgraphs are kept together in the class precedence list. For example, let T_1 and T_2 be subgraphs whose only element in common is the class t; let C_1 be the bottom of T_1; and let C_2 be the bottom of T_2. Suppose C is a class whose direct superclasses are C_1 and C_2 in that order, then the class precedence list for C will start with C and will be followed by all classes in T_1; after all the classes of T_1 will be all classes in T_2.

Examples

Here is an example of determining a class precedence list. The classes are defined:

```
(defclass pie (apple cinnamon) ())

(defclass apple (fruit) ())

(defclass cinnamon (spice) ())

(defclass fruit (food) ())

(defclass spice (food) ())

(defclass food () ())
```

The set S = {`pie, apple, cinnamon, fruit, spice, food, t`}. The set R = {`(pie, apple), (pie, cinnamon), (apple, cinnamon), (apple, fruit), (cinnamon, spice), (fruit, food), (spice, food), (food t)`}.

The class `pie` is not preceded by anything, so it comes first; the result so far is `(pie)`. We remove `pie` from S and pairs mentioning `pie` from R and get S = {`apple, cinnamon, fruit, spice, food, t`} and R = {`(apple, cinnamon), (apple, fruit), (cinnamon, spice), (fruit, food), (spice, food), (food t)`}.

The class apple is not preceded by anything, so it is next; the result is (pie apple). Removing apple and the relevant pairs we get $S = \{$cinnamon, fruit, spice, food, t$\}$ and $R = \{$(cinnamon, spice), (fruit, food), (spice, food), (food t)$\}$.

The classes cinnamon and fruit are not preceded by anything, so we look at which of these two has a direct subclass rightmost in the class precedence list computed so far. The class apple is a direct subclass of fruit and is rightmost in the precedence list. Therefore, we select fruit next, and the result so far is (pie apple fruit). $S = \{$cinnamon, spice, food, t$\}$; $R = \{$(cinnamon, spice), (spice, food), (food t)$\}$.

The class cinnamon is next, giving the result so far as (pie apple fruit cinnamon). $S = \{$spice, food, t$\}$; $R = \{$(spice, food), (food t)$\}$.

The classes spice, food, and t are added in that order, and the class precedence list is (pie apple fruit cinnamon spice food t).

It is possible to write a set of class definitions that cannot be ordered. For example:

```
(defclass new-class (fruit apple) ())

(defclass apple (fruit) ())
```

The class fruit must precede apple because the local ordering of superclasses is preserved. The class apple must precede fruit because a class always precedes its own superclasses. When this situation occurs, an error is signaled when the system tries to compute the class precedence list.

Note the following example, which appears at first glance to be a conflicting set of definitions:

```
(defclass pie (apple cinnamon) ())

(defclass pastry (cinnamon apple) ())

(defclass apple () ())

(defclass cinnamon () ())
```

The class precedence list for pie is (pie apple cinnamon t).

The class precedence list for pastry is (pastry cinnamon apple t).

There is no problem with the fact that apple precedes cinnamon in the ordering of the superclasses of pie but does not in the ordering for pastry. However, it is not possible to build a new class that has both pie and pastry as superclasses.

87–002

Generic Functions and Methods

Introduction to Generic Functions

A generic function object comprises a set of methods, a lambda-list, a method combination type, and other information.

Like an ordinary Lisp function, a generic function takes arguments, performs a series of operations, and perhaps returns useful values. An ordinary function has a single body of code that is always executed when the function is called. A generic function might perform a different series of operations and combine the results of the operations in different ways, depending on the class or identity of one or more of its arguments. A generic function can have several methods associated with it, and the class or identity of each argument to the generic function indicates which method or methods to use.

Ordinary functions and generic functions are called with identical syntax.

Generic functions are true functions that can be passed as arguments and used as the first argument to **funcall** and **apply**.

Ordinary functions have a definition that is in one place; generic functions have a distributed definition. The definition of a generic function can be found in a set of **defmethod** forms, possibly along with a **defgeneric-options** form that provides information about the behavior of the generic function. Evaluating these forms produces a generic function object.

In Common Lisp, a name can be given to a function in one of two ways: a ***global*** name can be given to a function using the **defun** construct; a ***local*** name can be given using the **flet** or **labels** special forms. Generic functions can be given a global name using the **defmethod** or **defgeneric-options** constructs. It is currently under discussion whether to provide constructs for giving generic functions local names.

Both **defmethod** and **defgeneric-options** use **symbol-function** to find the generic function that they affect. When a generic function is associated with a symbol, that name is in a certain package and can be exported if it is part of an external interface.

When a new **defgeneric-options** form is evaluated and a generic function of the given name already exists, the existing generic function object is modified. This does not modify any of the methods associated with the generic function. When a **defgeneric-options** form is evaluated and no generic function of the given name exists, a generic function with no methods is created.

When a **defmethod** form is evaluated and a generic function of the given name already exists, the existing generic function object is modified to contain the new method. The lambda-list of the new method must be congruent with the lambda-list of the generic function.

When a **defmethod** form is evaluated and no generic function with that name exists, a generic function is created with default values for the argument precedence order, the generic function class, the method class, and the method combination type. The lambda-list of the generic func-

tion is congruent with the lambda-list of the new method.

For a discussion of ***congruence***, see the section "Congruent Lambda-lists for All Methods of a Generic Function."

Introduction to setf Generic Functions

A ***setf generic function*** is called in an expression such as the following:
`(setf (`*symbol arguments*`)` *new-value*`)`.

One example of a setf generic function is the automatically generated setf function created when the **:accessor** option is given to **defclass**. The **:accessor** option defines a reader generic function of a given name. It also defines a generic function that is invoked when **setf** is used with the reader generic function. If the reader generic function is named *ship-color*, the corresponding setf generic function is invoked by means of the expression `(setf (ship-color` *ship*`)` *new-value*`)`.

The macro **defgeneric-options-setf** can be used to define a setf generic function. The macro **defmethod-setf** can be used to define a method for a setf generic function.

Note that unlike other generic functions, setf generic functions do not have names. The macros **defmethod-setf** and **defgeneric-options-setf** use **get-setf-generic-function** to find the generic function they affect.

Introduction to Methods

A method object contains a method function, an ordered set of ***parameter specializers*** that specify when the given method is applicable, and an ordered set of ***qualifiers*** that are used by the method combination facility to distinguish among methods.

The macro **defmethod** is used to create a method object. A **defmethod** form contains the code that is to be run when the arguments to the generic function cause the method that it defines to be selected. If a **defmethod** form is evaluated and a method object corresponding to the given generic function name, parameter specializers, and qualifiers already exists, the new definition replaces the old.

Each method has a ***specialized lambda-list***, which determines when that method can be selected. A specialized lambda-list is like an ordinary lambda-list except that a ***specialized parameter*** may occur instead of the name of a parameter. A specialized parameter is a list, (*variable-name parameter-specializer-name*), where *parameter-specializer-name* is a parameter specializer name. Every parameter specializer name is a Common Lisp type specifier, but the only Common Lisp type specifiers that are valid as parameter specializer names are the following:

- The name of any class
- **(quote** *object*)

A parameter specializer name denotes a parameter specializer as follows: Let N be a parameter specializer name and P be the corresponding parameter specializer; if N is a class name, then P is the class with that name; otherwise N equals P.

Parameter specializer names are used in macros intended as the user-level interface (**defmethod** and **defmethod-setf**), while parameter specializers are used in the functional interface (**make-method** and **get-method**).

Only required parameters can be specialized, and there must be a parameter specializer for each required parameter. For notational simplicity, if some required parameter in a specialized lambda-list is simply a variable name, its parameter specializer defaults to the class named **t**.

A method can be selected for a set of arguments when each required argument satisfies its corresponding parameter specializer. The following is a definition of what it means for an argument to satisfy a parameter specializer.

Let $\langle A_1, \ldots, A_n \rangle$ be the required arguments to a generic function in order. Let $\langle P_1, \ldots, P_n \rangle$ be the parameter specializers corresponding to the required parameters of the method M in order. The method M can be selected when each A_i satisfies P_i. If P_i is a class, and if A_i is an instance of a class C, then we say that A_i satisfies P_i when $C = P_i$ or when C is a subclass of P_i. If P_i is **(quote** *object*), then we say that A_i satisfies P_i when A_i is **eql** to *object*.

This proposal requires that both parameter specializers and parameter specializer names be Common Lisp type specifiers.

Because a parameter specializer is a type specifier, the function **typep** can be used to determine whether an argument satisfies a parameter specializer during method selection. Thus, this standard includes an upward-compatible extension of the Common Lisp type system and does not create another type system. Note that in general a parameter specializer cannot be a type specifier list, such as **(vector single-float)**. The only parameter specializer that can be a list is **(quote** *object*). This proposal requires that Common Lisp be modified to include **(deftype quote** (*object*) '**(member** ,*object*)).

A future extension might allow optional and keyword parameters to be specialized.

A method all of whose parameter specializers are the class named **t** is a ***default method***; it is always applicable but often shadowed by a more specific method.

Methods can have ***qualifiers***, which give the method combination procedure a way to distinguish between methods. A method that has one or more qualifiers is called a ***qualified*** method. A method with no qualifiers is called an ***unqualified method***. A qualifier is any object other than a list, that is, any non-**nil** atom. By convention, qualifiers are usually keyword symbols—it is rare to find a vector used as a qualifier.

In this specification, the terms ***primary method*** and ***auxiliary method*** are used to partition methods within a method combination type according to their intended use. In standard method combination, primary methods are unqualified methods and auxiliary methods are methods with a single qualifier that is **:around**, **:before**, or **:after**. When a method combination type is defined using the short form of **define-method-combination**, primary methods are defined to include not only the unqualified methods but certain others as well.

Thus, the terms ***primary method*** and ***auxiliary method*** have only a relative definition within a given method combination type.

Congruent Lambda-lists for All Methods of a Generic Function

The *lambda-list* argument of **defgeneric-options** specifies the ***shape*** of lambda-lists for the methods of that generic function. All methods for the given generic function must have lambda-lists that are congruent with this shape; this implies that the system can determine whether a call is syntactically correct. The shape of a lambda-list is defined by the number of required arguments, the number of optional arguments, whether or not **&allow-other-keys** appears, the number and names of keyword arguments, and whether or not **&rest** is used.

The rules for congruence are the following:

1. Each method must have the same number of required arguments.
2. Each method must have the same number of optional arguments, but methods can supply different defaults for optional arguments.
3. If **&allow-other-keys** is used by one method, all methods must use it.
4. If **&allow-other-keys** is not used, each method must allow the same keyword arguments, if any.
5. If **&rest** is used by one method, all methods must use it.
6. The use of **&aux** need not be consistent across methods.

87–002

Method Selection and Combination

When a generic function is called with particular arguments, it must decide what code to execute. We call this code the ***effective method*** for those arguments. The effective method can be one of the methods for the generic function or a combination of several of them.

When the effective method has been determined, it is called with the same arguments that were passed to the generic function. Whatever values it returns are returned as the values of the generic function.

Choosing the effective method involves the following decisions:

- Which method or methods to call
- The order in which to call the methods
- Which method to call when **call-next-method** is invoked
- Which value or values to return

Determining the Effective Method

The effective method is determined by the following three-step procedure:

1. Select the set of applicable methods.
2. Sort the applicable methods by precedence order, putting the most specific method first.
3. Apply method combination to the sorted list of applicable methods, producing the effective method.

Selecting the Set of Applicable Methods

Given a generic function and a set of arguments, the ***applicable methods*** are all methods for that generic function whose parameter specializers are satisfied by their corresponding arguments.

Sorting the Applicable Methods by Precedence Order

To compare the precedence of two methods, examine their parameter specializers in order. The default examination order is from left to right, but an alternative order may be specified by the **:argument-precedence-order** option to **defgeneric-options** or **defgeneric-options-setf**.

Compare the corresponding parameter specializers from each method. When a pair of parameter specializers are equal, proceed to the next pair and compare them for equality. If all corresponding parameter specializers are equal, the two methods must have different qualifiers; in this case, either method can be selected to precede the other. If some corresponding parameter specializers are not equal, the first pair of parameter specializers that are not equal determines the

precedence.

If both parameter specializers are classes, consider the class precedence list of the class of the argument. The more specific of the two methods is the method whose parameter specializer appears earlier in the class precedence list. Because of the way in which the set of applicable methods is chosen, the parameter specializers are guaranteed to be present in the class precedence list of the class of the argument.

If just one parameter specializer is **(quote** *object*), the method with that parameter specializer precedes the other method. If both parameter specializers are quoted objects, the specializers must be equal (otherwise the two methods would not both have been applicable for this argument).

The resulting list of applicable methods has the most specific method first and the least specific method last.

Applying Method Combination to the Sorted List of Applicable Methods

In the simple case—if standard method combination is used and all applicable methods are primary methods—the effective method is the most specific method. That method can call the next most specific method by using the function **call-next-method**. The method that **call-next-method** will call is referred to as the ***next method***.

In general, the effective method is some combination of the applicable methods. It is defined by a Lisp form that contains calls to some or all of the applicable methods, returns the value or values to be returned by the generic function, and optionally makes some of the methods accessible by means of **call-next-method**. This Lisp form is the body of the effective method; it is augmented with an appropriate lambda-list to make it a function.

Method qualifiers determine the role of each method in the effective method. The meaning of the qualifiers of a method is defined entirely by this step of the procedure. If an applicable method has an unrecognized qualifier, this step reports an error and does not include that method in the effective method.

When standard method combination is used together with qualified methods, the effective method is produced as described in the section "Standard Method Combination."

The programmer can select another type of method combination by using the **:method-combination** option of **defgeneric-options**. This allows the programmer to customize this step of this procedure without having to consider what happens in the other steps.

New types of method combination can be defined using the **define-method-combination** macro. The body of the **define-method-combination** returns the Lisp form that defines the effective method.

The meta-object level also offers a mechanism for defining new types of method combination. The generic function **compute-effective-method** receives as arguments the generic function, the sorted list of applicable methods, the name of the method combination type, and the

list of options specified in the **:method-combination** option of **defgeneric-options**. It returns the Lisp form that defines the effective method. The programmer can define a method for **compute-effective-method** directly by using **defmethod** or indirectly by using **define-method-combination**.

Implementation Note:
In the simplest implementation, the generic function would compute the effective method each time it was called. In practice, this might be too inefficient for most implementations. Instead, these implementations might employ a variety of optimizations of the three-step procedure, such as precomputation into tables, compilation, and/or caching to speed things up. Some illustrative examples of such optimizations are the following:

- Use a hash table keyed by the class of the arguments to store the effective method.
- Compile the effective method and save the resulting compiled function in a table.
- Recognize the Lisp form as an instance of a pattern of control structure and substitute a closure that implements that structure.
- Examine the parameter specializers of all methods for the generic function and enumerate all possible effective methods. Combine the effective methods, together with code to select from among them, into a single function and compile that function. Call that function whenever the generic function is called.

The Lisp form computed by Step 3 as the body of the effective method serves as a more general interface. For example, a tool that determines which methods are called and presents them to the user works by going through the first three steps of the above procedure and then by analyzing the form to determine which methods it calls instead of by evaluating it.

Separating the procedure of determining the effective method from the procedure of invoking methods and combining their results, and using a Lisp form as the interface between these procedures, allows for more optimizations to the speed of the code and also enables more powerful programming tools to be written.

Standard Method Combination

Standard method combination is used if no other type of method combination is specified. Standard method combination recognizes four roles for methods, as determined by method qualifiers.

Primary methods define the main action of the effective method, while ***auxiliary methods*** modify that action in one of three ways. A primary method has no method qualifiers.

The auxiliary methods are **:before**, **:after**, and **:around** methods.

- A **:before** method has the keyword **:before** as its only qualifier. A **:before** method specifies code that is to be run before the primary method.
- An **:after** method has the keyword **:after** as its only qualifier. An **:after** method specifies code that is to be run after the primary method.
- An **:around** method has the keyword **:around** as its only qualifier.

The semantics of standard method combination are:

- If there are any **:around** methods, the most specific **:around** method is called. It supplies the value or values of the generic function.
- Inside the body of an **:around** method, **call-next-method** can be used to immediately call the next method. When the next method returns, the **:around** method can execute more code, perhaps based on the returned value or values. By convention, **:around** methods almost always use **call-next-method**.
- If an **:around** method invokes **call-next-method**, the next most specific **:around** method is called, if one is applicable. If there are no **:around** methods or if **call-next-method** is called by the least specific **:around** method, the other methods are called as follows:
  - All the **:before** methods are called, in most specific first order. Their values are ignored.
  - 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. When that method returns, the first primary method can execute more code, perhaps based on the returned value or values. An error is signaled if **call-next-method** is used and there is no applicable primary method to call. If **call-next-method** is not used, only the most specific primary method is called.
  - All the **:after** methods are called in most specific last order. Their values are ignored.
- If no **:around** methods were invoked, the most specific primary method supplies the value or values returned by the generic function. Otherwise, the value or values returned by the most specific primary method are those returned by the invocation of **call-next-method** in the least specific **:around** method.

In standard method combination, if there are any applicable methods at all, then there must be an applicable primary method. In cases where there are applicable methods, but no primary method, an error is signaled.

An error is signaled if **call-next-method** is used and there is no next method remaining.

An error is signaled if **call-next-method** is used in a **:before** or **:after** method.

Standard method combination allows no more than one qualifier per method.

Running **:before** methods in most specific first order while running **:after** methods in least specific first order provides a degree of transparency. If class C_1 modifies the behavior of its superclass, C_2, by adding an auxiliary method, the partitioning of C_2's behavior into primary, **:before**, and **:after** methods is transparent. Whether class C_2 defines these methods directly or inherits them from its superclasses is transparent. Class C_1's **:before** method runs before all of class C_2's methods. Class C_1's **:after** method runs after all of class C_2's methods.

The **:around** methods are an exception to this rule; they do not combine transparently. All **:around** methods run before any other methods run. Thus, a less specific **:around** method runs before a more specific primary method.

If only primary methods are used, standard method combination behaves like CommonLoops. If **call-next-method** is not used, only the most specific method is invoked; that is, more general methods are shadowed by more specific ones. If **call-next-method** is used, the effect is the same as **run-super** in CommonLoops.

If **call-next-method** is not used, standard method combination behaves like **:daemon** method combination of New Flavors, with **:around** methods playing the role of whoppers, except that the ability to reverse the order of the primary methods has been removed.

Declarative Method Combination

The programmer can define new forms of method combination by using the **define-method-combination** macro. This allows customization of Step 3 of the method combination procedure described in the section "Determining the Effective Method." There are two forms of **define-method-combination**. The short form is a simple facility for the cases that have been found to be most commonly needed. The long form is more powerful but more verbose. It resembles **defmacro** in that the body is an expression that computes a Lisp form, usually using backquote. Thus, arbitrary control structures can be implemented. The long form also allows arbitrary processing of method qualifiers. The syntax and use of both forms of **define-method-combination** is explained in the second chapter of this document.

87–002

Meta Objects

Metaclasses

The ***metaclass*** of an object is the class of its class. The metaclass determines the form of inheritance used by its classes and the representation of the instances of its classes. The metaclass mechanism can be used to provide particular forms of optimization or to tailor the Common Lisp Object System for particular uses (such as the implementation of other object languages like Smalltalk-80, Loops, and CommonObjects).

Any new metaclass must define the structure of its instances, how their storage is allocated, how their slots are accessed, and how slots and methods are inherited. The protocol for defining metaclasses will be discussed in Chapter 3, "The Common Lisp Object System Meta-Object Protocol."

Standard Metaclasses

The Common Lisp Object System provides a number of predefined metaclasses. These include the following: **standard-class**, **standard-type-class**, and **structure-class**.

- The class **standard-class** is the default class of classes defined by **defclass**.
- The class **standard-type-class** is a metaclass of all the classes that correspond to the standard Common Lisp types specified in *Common Lisp: The Language* by Guy L. Steele Jr. It is not allowed to make an instance of a standard type class by using **make-instance**, or to use a standard type class as a superclass of a user-defined class. It is still under discussion which Common Lisp types will have corresponding classes.
- The class **structure-class** is a subclass of **standard-type-class**. All classes defined by means of **defstruct** are instances of **structure-class** or a subclass of **structure-class**.

Standard Meta-Objects

The Common Lisp Object System provides the predefined meta-objects **standard-method** and **standard-generic-function**.

- The class **standard-method** is the default class of methods defined by **defmethod** or **defmethod-setf**.
- The class **standard-generic-function** is the default class of generic functions defined by **defmethod**, **defmethod-setf**, **defgeneric-options**, or **defgeneric-options-setf**.

The Common Lisp Object System also provides the standard method combination type, which is not implemented as a meta-object, but as a method.

87–002

Common Lisp Object System Specification

2. Functions in the Programmer Interface

This document was written by Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya Keene, Gregor Kiczales, and David A. Moon.

Contributors to this document include Patrick Dussud, Kenneth Kahn, Larry Masinter, Mark Stefik, Daniel L. Weinreb, and Jon L White.

87–002

CONTENTS

87–002

Introduction

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

The description of each function includes its purpose, its syntax, the semantics of its arguments and returned values, and often an example and cross-references to related functions. This chapter is reference material that requires an understanding of the basic concepts of the Common Lisp Object System. The functions are arranged in alphabetic order for convenient reference.

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 the body of methods 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.

  defclass
  make-instance
  defmethod
  defmethod-setf
  defgeneric-options
  defgeneric-options-setf
  call-next-method
  slot-value
  with-slots
  change-class
  class-changed

- Functions underlying the commonly-used macros

  add-method
  get-method
  get-setf-generic-function
  make-generic-function
  make-method
  remove-method

- Tools for declarative method combination

  define-method-combination
  make-method-call
  method-qualifiers
  multiple-value-prog2
  method-combination-error
  invalid-method-error

- General Common Lisp support tools

  class-of
  describe
  documentation
  print-object

87–002

add-method *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*]

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-lists of any other methods associated with the generic function and with the lambda-list of the generic function.

Values:

The generic function **add-method** returns the modified generic function.

Remarks:

If the given method is already one of the methods of the generic function or if the method corresponds in parameter specializers and method qualifiers to an existing method of the generic function, an error is signaled.

See Also:

defmethod

make-method

make-generic-function

defgeneric-options

87–002

call-next-method *Function*

Purpose:

The binding for the local function variable **call-next-method** has lexical scope and dynamic extent. The function **call-next-method** is used within the body of a method to call the next method. The next method is called with the same arguments that were received by the method in which **call-next-method** is invoked. The function **call-next-method** returns the value or values returned by the method it calls. If there is no next method, an error is signaled.

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. It 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. Their values are ignored.
  - 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. An error is signaled if **call-next-method** is used and there is no applicable primary method.
  - All the **:after** methods are called in most specific last order. Their values are ignored.

Syntax:

call-next-method [*Function*]

Arguments:

The function **call-next-method** is used with no arguments.

Values:

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

Remarks:

The function **call-next-method** 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.

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

If the short form of **define-method-combination** is used to define a new type of method combination, **call-next-method** can be used in **:around** methods only.

A proposed extension is to allow **call-next-method** to accept arguments. When arguments are given, **call-next-method** invokes the next method on those arguments instead of the arguments that were received by the method from which it is called.

See Also:

"Method Selection and Combination"

"Standard Method Combination"

define-method-combination

87–002

change-class *Function*

Purpose:

The function **change-class** changes the class of an instance to a new class. It destructively modifies and returns the instance. The values of local slots held in common between the old and new class are preserved in the new instance. The other slots are initialized as described in the section "Redefining Classes." The generic function **class-changed** is run.

The generic function **change-class** is invoked automatically by the system after **defclass** has been used to redefine an existing class. It can also be explicitly invoked by the user.

Syntax:

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

Arguments:

The *instance* argument is a Lisp object, although not all objects are required to allow **change-class**.

The *new-class* argument is a class object or a symbol that names a 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)
     (y :initform 0))
  (:accessor-prefix position-))

(defclass rho-theta-position (position)
    ((rho :initform 0)
     (theta :initform 0))
  (:accessor-prefix position-))

(defmethod class-changed ((old x-y-position)
                          (new rho-theta-position))
  ;; Copy the position information from old to new to make new
  ;; be a rho-theta-position at the same position as old.
```

```
  (let ((x (position-x old))
        (y (position-y old)))
    (setf (position-rho new) (atan y x)
          (position-theta new) (sqrt (+ (* x x) (* y y))))))

;;; 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 method for class-changed
;;; performed the initializion of the rho and theta slots based
;;; on the value of the x and y slots, which were maintained by
;;; the old instance.
```

Remarks:

The Common Lisp Object System requires **change-class** to apply in only the following case: let C_1 and C_2 be classes that are defined by **defclass** without using the **:metaclass** option in either case; let X be an instance of C_1. Then, the class of X can be changed from C_1 to C_2. Both before and after the call to **change-class**, the metaclass of X is the default metaclass, namely **standard-class**.

Implementors can choose to support **change-class** in additional cases. For example, this standard does not require that **change-class** be able to accept an instance of a standard type class as its first argument or a standard type class as its second argument; however, it is valid for an implementation to support this for some standard type classes.

If **change-class** is applied to arguments that are not supported by the implementation, an error is signaled.

After completing all other actions, **change-class** invokes the generic function **class-changed**. The generic function **class-changed** can be used to reinitialize slots.

The 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 problem could be compounded. 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 an application programmer must not use **change-class** inside a method if any methods for that generic function access any slots.

87–002

change-class

See Also:

"Redefining Classes"

class-changed

87–002

class-changed *Generic Function*

Purpose:

The generic function **class-changed** is not intended to be called by programmers. Programmers are expected to write methods for it. The function **class-changed** is called only by the function **change-class**. It can also be explicitly invoked by the user.

Syntax:

class-changed *previous current* [*Generic function*]

Arguments:

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 **class-changed**, *previous*, is that copy, and the second argument, *current*, is the altered original instance.

The typical use of *previous* is to extract old slot values by using **slot-value** or **with-slots** or by invoking an accessor generic function.

Values:

The value returned by **class-changed** is ignored by **change-class**.

Examples:

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

Remarks:

The arguments to **class-changed** are computed by **change-class**. The first argument is an instance of the original class created to hold the old slot values temporarily. This argument has dynamic extent within **class-changed**, and therefore it is an error to reference it in any way once **class-changed** returns.

The default method for **class-changed** does nothing. Methods on **class-changed** can be defined to initialize slots differently from **change-class**. In this way **class-changed** methods can alter the default behavior of **change-class** with respect to slot value preservation. The default behavior is described in "Redefining Classes."

The generic function **class-changed** uses standard method combination; thus **:before**, **:after**, **:around**, and unqualified methods are allowed.

87–002

class-changed

See Also:

"Redefining Classes"

change-class

add-method

make-method

87–002

class-named *Function*

Purpose:

The function **class-named** returns the class named by a given symbol.

Syntax:

class-named *name* `&optional` *errorp* [*Function*]

Arguments:

The first argument to **class-named** is a symbol. If there is no class named by the symbol and the *errorp* argument is unsupplied or is non-**nil**, **class-named** signals an error. If there is no class named by the symbol and the second argument is **nil**, **class-named** returns **nil**. The default value of *errorp* is **t**.

Values:

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

87–002

class-of *Function*

Purpose:

The function **class-of** returns the class object for the most specific class of which the given object is an instance. Every Common Lisp object has a class.

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 object that represents the most specific class of which the argument is an instance.

Remarks:

Which Common Lisp types will have corresponding classes is still under discussion. That decision will affect the behavior of **class-of**.

87–002

defclass *Macro*

Purpose:

The macro **defclass** defines a new class. It returns the name of the new class as its result.

The syntax of **defclass** provides options for specifying default initialization values for slots, for requesting that methods for appropriately named generic functions be automatically generated for reading and writing the values of slots, and for requesting that a constructor function be automatically generated for making instances of the new class. No accessors, readers, or constructor 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*) is 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*) is true if the class of the *object* is *class* itself or a subclass of *class*.

87–002

defclass

Syntax:

defclass *class-name* ({*superclass-name*}*) ({*slot-spec*}*) {*class-option*}*

class-name::= *symbol*

superclass-name::= *symbol*

slot-spec::= *slot-name* | (*slot-name* {*slot-option*}*)

slot-name::= *symbol*

slot-option::= :**accessor** *generic-function-name* |
:**reader** *generic-function-name* |
:**allocation** *allocation-type* |
:**initform** *form* |
:**type** *type-specifier*

generic-function-name::= *symbol*

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

class-option::= (:**accessor-prefix** *symbol*) |
(:**reader-prefix** *symbol*) |
(:**constructor** *symbol* [*boa-arglist*]) |
(:**documentation** *string*) |
(:**metaclass** *class-name*)

boa-arglist::= ({*symbol*}*
[**&optional** {*var* | (*var* [*initform*])}*]
[**&rest** *var*]
[**&aux** {*var* | (*var* [*initform*])}*])

Figure 2–1. Syntax for defclass

Arguments:

The *class-name* argument is a non-**nil** symbol. It becomes the name of the new class. If a class with the same name already exists, the definition of that class is replaced.

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

Each *slot-spec* 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 can be used as a Common Lisp variable name. If there are any duplicate slot names, an error is signaled.

The following slot options are available:

- The **:accessor** option specifies that an unqualified method is to be defined on the generic function named *generic-function-name* to read the value of the given slot and that an unqualified method is to be defined on the setf generic function named *generic-function-name* to be used with **setf** to modify the value of the slot. The *generic-function-name* argument is a non-**nil** symbol. The **:accessor** option may be specified more than once for a given slot.
- The **:reader** option specifies that an unqualified method is to be defined on the generic function named *generic-function-name* to read the value of the given slot. The *generic-function-name* argument is a non-**nil** symbol. The **:reader** option may be specified more than once for a given slot.
- The **:allocation** 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 one of the following keywords: **:instance** or **:class**. The **:allocation** option may be specified once at most for a given slot. If the **:allocation** option is not specified, a local slot of the given name is allocated in each instance of the class.
  - 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. Any subclass of this class will share this single slot unless the **defclass** form for that subclass specifies a slot of the same name.
- The **:initform** option is used to provide a default initial value form to be used in the initialization of the slot. The **:initform** option may be specified once at most for a given slot. This form is evaluated every time it is used. The lexical environment in which this form is evaluated is the lexical environment in which **defclass** was evaluated. Note that the lexical environment refers both to variables and to functions. The dynamic environment is the one in effect at the time the form is evaluated. This is the same behavior specified for **defstruct** slot initialization forms in *Common Lisp: The Language*.

- The **:type** option specifies the type of the slot contents. This 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. An implementation may or may not choose to check the type of the new value when initializing or assigning to a slot. The expression (typep *value type-specifier*) will be true for the value stored in the slot. The **:type** option may be specified once at most for a given slot. The **:type** option is further discussed in the section "Inheritance of Slots and Slot Options."

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 **:accessor-prefix** option specifies that, for each slot, an unqualified method to read the value of the slot is to be defined on the generic function named *symbol* followed by the name of the slot. Similarly, a method to be used with the macro **setf** to modify the value of the slot is to be defined on the setf generic function named *symbol* followed by the name of the slot. The names of these accessor functions are interned in the package that is current at the time the **defclass** form is macro-expanded. If the prefix is **nil**, the names of the accessor functions are the symbols that are used as the slot names. The **:accessor-prefix** option may be specified more than once.

- The **:reader-prefix** option specifies that, for each slot, an unqualified method to read the value of the slot is to be defined on the generic function named *symbol* followed by the name of the slot. The names of these reader functions are interned in the package that is current at the time the **defclass** form is macro-expanded. If the prefix is **nil**, the names of the reader functions are the symbols that are used as the slot names. The **:reader-prefix** option may be specified more than once.

- The **:constructor** option causes a constructor function to be generated automatically. The constructor function is used to make new instances of the class. The *symbol* argument is a non-**nil** symbol that specifies the name of the constructor function. If the *boa-arglist* argument is present, it describes the arguments to the constructor. The *boa-arglist* argument of **defclass** is the same as that of **defstruct**. The **:constructor** option may be specified more than once.

- The **:documentation** 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** option may be specified once at most.

- The **:metaclass** 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** option may be specified once at most.

Values:

The name of the new class is returned as the result.

Remarks:

If a class of the same name already exists, that class is redefined and instances of the class (and subclasses of it) are updated to the new definition at the time that they are next accessed. For details, see "Redefining Classes" and **change-class**. Redefining a standard type class is not allowed.

Note the following rules of **defclass**:

- 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.
- All the superclasses of a class must be defined before a **with-slots** form that uses that class can be evaluated or compiled.

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 are inherited. For a detailed description of how slots and slot options are inherited, see the section "Inheritance of Slots and Slot Options."

Some implementations might add other options to **defclass**. Therefore, it is required that all implementations signal an error if they observe a class option or a slot option that is not implemented locally.

If no default value for a slot is specified in either a **defclass** or a **make-instance** form, the initial value of the slot is unspecified.

It is valid to specify more than one accessor or reader for a slot. No other slot option may appear more than once in a single slot description.

The **:accessor-prefix**, **:constructor**, and **:reader-prefix** class options may appear more than once. No other class option may appear more than once in a single **defclass** form.

If neither a reader nor an accessor is specified for a slot, the slot can only be accessed by the function **slot-value** or by **with-slots** using **:use-accessors nil**.

87–002

defclass

See Also:

slot-value

with-slots

"Classes"

"Inheritance"

"Redefining Classes"

"Determining the Class Precedence List"

87–002

defgeneric-options *Macro*

Purpose:

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

The generic function is stored in the function cell of the symbol *name*. If (`fboundp` *name*) is **nil**, a new generic function is created. If (`symbol-function` *name*) is a generic function, that generic function is modified. If neither of these conditions holds, **defgeneric-options** signals an error.

The macro **defgeneric-options** returns *name* as its result.

Syntax:

defgeneric-options *name lambda-list* {*option*}* [*Macro*]

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

Arguments:

The *name* argument is a non-**nil** symbol.

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

- No **&aux** variables are 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:

- 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. By default, all required arguments are considered from left to right; each required argument has precedence over those to its right. Each required argument must be included exactly once as a *parameter-name* so that the full and unambiguous precedence order

defgeneric-options

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. The optimization qualities **speed** and **space** are the only qualities this standard requires, but other qualities may be recognized by particular implementations. 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 support their own additional declarations. If an implementation notices a declaration that it does not support and that has not been proclaimed as a nonstandard declaration name, 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` *generic-function-name* '`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.

- The **:method-class** option is used to specify that all methods for this generic function are to have a different class than 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.

Values:

The macro **defgeneric-options** returns *name* as its result.

Remarks:

The *lambda-list* argument of **defgeneric-options** specifies the shape of lambda-lists for the methods of this generic function. All methods for the generic function must have lambda-lists that are congruent with this shape. If a **defgeneric-options** form is evaluated and some methods for

that generic function have lambda-lists that are not congruent with that given in the **defgeneric-options** form, an error is signaled. For further details on method congruence, see "Congruent Lambda-lists for all Methods of a Generic Function"

Some implementations might add other options to **defgeneric-options**. Therefore, it is required that all implementations signal an error if they observe an option that is not implemented locally.

See Also:

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

defgeneric-options-setf *Macro*

Purpose:

The macro **defgeneric-options-setf** is used to define a setf generic function. A setf generic function is called in an expression such as (**setf** (*name arguments*) *new-value*).

The macro **defgeneric-options-setf** returns *name* as its result.

Syntax:

defgeneric-options-setf *name lambda-list setf-lambda-list* {*option*}* [*Macro*]

setf-lambda-list::= (*variable-name*)

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

Arguments:

The *name* argument is a non-**nil** symbol.

The *lambda-list* argument is identical to the *lambda-list* argument of **defgeneric-options**.

The *setf-lambda-list* argument is (*variable-name*). It describes the parameter that receives the *new-value* argument to **setf**.

The options are the same as for **defgeneric-options**.

Values:

The macro **defgeneric-options-setf** returns *name* as its result.

Remarks:

The *lambda-list* argument of **defgeneric-options-setf** specifies the shape of lambda-lists for the methods of this setf generic function. All methods for the setf generic function must have lambda-lists that are congruent with this shape. For further details on method congruence, see "Congruent Lambda-lists for All Methods of a Generic Function"

87–002

See Also:

defgeneric-options

defmethod

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

87–002

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 have been found 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*}*)
{*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, but this is not required.

None of the subforms is evaluated.

A method combination procedure defined in this way recognizes two roles for methods. An unqualified method is a primary method. A method with the keyword symbol with the same name as *name* as its one qualifier is also defined to be a primary method. Attaching this qualifier to a primary method documents that this method is intended for use with an unusual form of method combination and can make programs easier to understand.

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, not in primary methods.

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.

A large fraction of the types of method combination needed by most programmers can be implemented with this short form, which is provided for convenience. The short form automatically includes error checking and support for **:around** methods and avoids the need for the use of the backquote and comma.

87–002

define-method-combination

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-options** or **defgeneric-options-setf**.

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 reports 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 *.

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, that

is, one or two words. This keyword option allows the description of a method qualifier to be defined in the same module that defines the semantic 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 body of **define-method-combination** resembles the body of **defmacro** and uses backquote in a similar way. The function **make-method-call** is also used in constructing the Lisp form; it hides the implementation-dependent details of how methods are called. Programmers always use **make-method-call** to translate from the lists of method objects produced by the method-group specifiers to Lisp forms that invoke those methods.

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*, the lexical variable **generic-function** is bound to the generic-function object.

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

87–002

define-method-combination

The functions **make-method-call**, **method-combination-error**, and **invalid-method-error** can be called from the body *forms* or from functions called by the body *forms*. The action of these three functions can depend on dynamic variables automatically bound before the method combination function is called. These variables might contain the parameter list of the effective method or other implementation-dependent information.

Note that two methods with identical specializers, but 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, implementations are encouraged to signal an error (this would happen as part of qualifier pattern matching in **define-method-combination**).

Values:

The name of the new method combination type is returned.

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))
  (make-method-call `(,@around
                      ,(make-method-call primary
                                         :operator 'and
                                         :identity-with-one-argument t))
                    :operator :call-next-method))

;;; 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)))
  (make-method-call `(,@around
                      (multiple-value-prog2
                        ,(make-method-call before)
                        ,(make-method-call primary
                                           :operator :call-next-method)
                        ,(make-method-call (reverse after))))
                    :operator :call-next-method))

;A simple way to try several methods until one returns non-nil
(define-method-combination and ()
        ((methods () (:and)))
  (make-method-call methods :operator 'and))

;A more complete version of the preceding
(define-method-combination and
        (&optional (order ':most-specific-first))
        ((around (:around))
         (primary () (:and)))
  ;; 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."))
  (make-method-call `(,@around
                      ,(make-method-call primary
                                         :operator 'and
                                         :identity-with-one-argument t))
                    :operator :call-next-method))

;The same thing, using the :order and :required keyword options
(define-method-combination and
        (&optional (order ':most-specific-first))
        ((around (:around))
         (primary () (:and) :order order :required t))
  (make-method-call `(,@around
                      ,(make-method-call primary
                                         :operator 'and
                                         :identity-with-one-argument t))
```

87–002

define-method-combination

```
                    :operator :call-next-method))

;This short-form call is behaviorally identical to the preceding
(define-method-combination and :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))
  (make-method-call (stable-sort methods #'<
                      :key #'(lambda (method)
                               (first (method-qualifiers method))))))

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

Remarks:

The **:method-combination** option of **defgeneric-options** and **defgeneric-options-setf** 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.

Individual implementations might support other keyword options. Therefore, it is required that all implementations signal an error if they observe a keyword option that is not implemented locally.

See Also:

make-method-call

method-qualifiers

multiple-value-prog2

method-combination-error

invalid-method-error

defgeneric-options

defgeneric-options-setf

87–002

defmethod *Macro*

Purpose:

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

If a generic function is currently named by the symbol *name*, 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.

If (`fboundp` *name*) 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 symbol *name* names a non-generic function, a macro, or a special form, an error is signaled.

Syntax:

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

name::= *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* | (`quote` *datum*)

Arguments:

The *name* argument is a non-**nil** symbol that names the generic function on which the method is defined.

87–002

defmethod

Each method qualifier 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 one of the following keywords: **:before**, **:after**, or **: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 user-defined class, a structure defined by **defstruct** if the **:type** option was not used, or a class that corresponds to a Common Lisp type specifier. Note that not all Common Lisp types have a corresponding class. A parameter specializer name can also be (quote *object*). Such a parameter specializer name indicates that the corresponding argument must be **eql** to the quoted object for the method to be applicable. If no parameter specializer name is specified for a given required parameter, the parameter specializer name for that parameter defaults to **t**. A method whose required parameters all have **t** parameter specializers is termed a ***default method***. Such a method is selected when no more specific method for the generic function is applicable. See the section "Introduction to Methods" for further discussion.

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 a method already exists on the given generic function with the same parameter specializers and the same qualifiers, **defmethod** replaces the existing method with the one now being defined.

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

See Also:

add-method

"Introduction to Methods"

"Congruent Lambda-lists for all Methods of a Generic Function"

87–002

defmethod-setf *Macro*

Purpose:

The macro **defmethod-setf** defines a method for a setf generic function. A setf generic function is called in an expression such as `(setf (name arguments) new-value)`.

Syntax:

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

name::= *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*])}*])

specialized-setf-lambda-list::= ({*var* | (*var parameter-specializer-name*))}

parameter-specializer-name::= *symbol* | (**quote** *datum*)

Arguments:

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

The *specialized-setf-lambda-list* argument is the same as *specialized-lambda-list* except that for now there can be only one parameter. In other words, *specialized-setf-lambda-list* is a lambda-list containing exactly one required parameter, which may be specialized. It describes the parameter that receives the *new-value* argument to **setf** in the expression `(setf place new-value)`.

The *var* argument in the *specialized-setf-lambda-list* argument is the name of the variable that gets bound to the value of the *new-value* form in the expression `(setf place new-value)`.

Values:

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

87–002

defmethod-setf

Remarks:

If a method already exists on the given generic function with the same parameter specializers and the same qualifiers, **defmethod-setf** replaces the existing method with the one now being defined.

See Also:

defmethod

87–002

describe

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 default method for **describe**, that is, a method for the class **t**. Implementations are free to add methods for specific 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*]

Arguments:

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

Values:

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

87–002

documentation *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**.

Syntax:

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

Arguments:

The first argument is a generic function object, a method object, a class object, or a symbol.

If the first argument is not a symbol, the second argument must not be supplied. If the first argument is a generic function object, method object, or class object, **documentation** returns the documentation string for that object.

If the first argument is a symbol, the second argument must be supplied. The generic function **documentation** returns the documentation string of the given type. The *doc-type* argument is a symbol. It can be one of the following types: **variable**, **function**, **structure**, **type**, and **setf**.

If the first argument is a symbol and the second argument is **type**, **documentation** returns the documentation string of the class object named by the symbol.

If the first argument is a symbol and the second argument is **function**, **documentation** returns the documentation string of the function or generic function named by the symbol.

If the first argument is a symbol and the second argument is **setf**, **documentation** returns the documentation string of the setf generic function associated with that name.

Values:

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

Remarks:

The macro **setf** can be used with **documentation** to update the documentation for a symbol, generic function object, method object, or class object.

87–002

get-method

Generic function

Purpose:

The generic function **get-method** takes a generic function and returns the method object that has the given method qualifiers and parameter specializers.

Syntax:

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

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. This means that to obtain the default method for a given generic function, a list of t's corresponding in length to the number of required arguments of that generic function must be given.

If there is no such method and the *errorp* argument is unsupplied or is non-**nil**, **get-method** signals an error. If there is no such method and the *errorp* argument is **nil**, **get-method** returns **nil**. The default value of *errorp* is **t**.

Values:

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

87–002

get-setf-generic-function *Function*

Purpose:

The function **get-setf-generic-function** takes a symbol for which a setf generic function has been defined by either **defmethod-setf** or **defgeneric-options-setf** and returns a generic function object. This object is the generic function that is called when the form (`setf` (*name arguments*) *new-value*) is evaluated.

Syntax:

get-setf-generic-function *name* [*Function*]

Arguments:

The *name* argument is a symbol for which a setf generic function has been defined. If no such setf generic function has been defined, an error is signaled.

Values:

The function **get-setf-generic-function** returns a generic function object. This object is the generic function that is called when the form (`setf` (*name arguments*) *new-value*) is evaluated.

87–002

invalid-method-error *Function*

Purpose:

The function **invalid-method-error** reports 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.

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

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:

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.

The function **invalid-method-error** will use the condition-signaling system when and if it is incorporated into Common Lisp.

See Also:

define-method-combination

87–002

make-generic-function *Function*

Purpose:

The function **make-generic-function** creates and returns a generic function. This resulting function can be used an argument to **funcall** and **apply**.

Syntax:

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

Arguments:

The **:lambda-list** argument is a lambda-list of the type that may be given to **defgeneric-options**.

The following arguments have the same semantics as the corresponding arguments of **defgeneric-options**, although their syntax may differ:

- The **:argument-precedence-order** argument is a list containing the parameter names for all required arguments. Each required argument must be included exactly once so that the full and unambiguous precedence order is supplied. If this condition is not met, an error is signaled.
- The **:method-combination** argument is a symbol or a list. If it is a symbol, that symbol names a type of method combination. If it is a list, its first element is a symbol that names a type of method combination, and its remaining elements are any arguments accepted by the method combination type. Any arguments 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*, which defaults 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 **:documentation** argument is a string.
- The **:declare** argument is a list of declaration specifiers.
- The **:generic-function-class** argument is a class or the name of a class.
- The **:method-class** argument is a class or the name of a class.

87–002

make-generic-function

Values:

The result of **make-generic-function** is a generic function object.

Remarks:

The function **make-generic-function** is part of the programmatic interface to **defgeneric-options**.

See Also:

defgeneric-options

87–002

make-instance *Function*

Purpose:

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

Syntax:

make-instance *class* **&rest** *initialize-keywords-and-values* [*Function*]

Arguments:

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

Values:

The new instance is returned.

Remarks:

The initialization protocol of **make-instance** has not been specified.

It is not possible to make an instance of a class whose class is **standard-type-class** by using the function **make-instance**. If *class* is an instance of **standard-type-class**, **make-instance** signals an error.

The function **class-of** can be used to determine the class of the instance that is returned.

See Also:

class-of

87–002

make-method *Function*

Purpose:

The function **make-method** creates and returns a method object.

Syntax:

make-method *method-qualifiers specializers function* [*Function*]

Arguments:

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

The *specializers* argument is a list of the parameter specializers for the method.

The *function* argument is the method function.

The length of the list of specializers must be equal to the number of required arguments of the method function.

Values:

The function **make-method** returns the resulting method object.

See Also:

defmethod

add-method

87–002

make-method-call *Function*

Purpose:

The function **make-method-call** is used in method combination. It has dynamic scope within the body of a **define-method-combination** form.

The function **make-method-call** returns a form whose effect is the same as a form whose first element is the operator specified by the **:operator** keyword argument (the default is **progn**) and the rest of which is a list of forms that call the methods in the given method list. Each method receives the same arguments that the generic function received. The function **make-method-call** hides the implementation-dependent details of how methods are called.

Syntax:

make-method-call *method-list* `&key :operator :identity-with-one-argument` [*Function*]

Arguments:

Each element of *method-list* can be either a method object or a list. When a list is given, it is regarded as a form and converted when necessary into a method whose body is that form.

If the value of **:identity-with-one-argument** is true and *method-list* contains exactly one element, the result is simply a form that calls that single method and does not invoke the operator. If **:operator** is **progn**, the default for **:identity-with-one-argument** is true; otherwise the default for this option is false. This option is to be used with operators that are identity operators when applied to one argument, that is, such operators as **progn**, **and**, +, and **max**. This optimization can enable the use of an existing method as the effective method, thus avoiding the need to create a new effective method.

If *method-list* is **nil**, the result is a call to the specified operator with no arguments or a form with the same effect.

If **:operator** is **:call-next-method**, the methods are combined in a different way, rather than calling a function named **:call-next-method**. The result is a form that calls the first method and arranges for **call-next-method** to reach the rest of the methods in the order in which they appear in *method-list*. If there is only one method in *method-list*, the result is a form that calls that method, and if the method calls **call-next-method**, an error is signaled.

As a convenience, if *method-list* is a method object, it is automatically converted to a one-element list of that method.

If **call-next-method** is extended as noted, additional keyword arguments will be needed for **make-method-call**.

87–002

make-method-call

Values:

The result is a form whose effect is the same as a form whose first element is the operator specified by the **:operator** keyword argument and the rest of which is a list of forms that call the methods in *method-list*.

See Also:

define-method-combination

87–002

method-combination-error *Function*

Purpose:

The function **method-combination-error** reports a problem 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.

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

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:

The function **method-combination-error** will use the condition signaling system when and if it is incorporated into Common Lisp.

See Also:

define-method-combination

87–002

method-qualifiers *Function*

Purpose:

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

Syntax:

method-qualifiers *method* [*Function*]

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

87–002

multiple-value-prog2 *Macro*

Purpose:

The macro **multiple-value-prog2** is similar to **multiple-value-prog1** except that it returns the values of its *second* form.

Syntax:

multiple-value-prog2 *first second* {*form*}* [*Macro*]

Values:

All the values of *second* are returned.

See Also:

define-method-combination

87–002

print-object

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 default method for **print-object**, that is, a method for the class **t**. Implementations are free to add methods for specific 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*]

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 for **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.

print-object

- 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.

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.

87–002

remove-method

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*]

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 no such method is part of the generic function.

Values:

The function **remove-method** returns the modified generic function.

See Also:

get-method

add-method

87–002

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, an error is signaled.

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

Syntax:

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

87–002

with-slots *Macro*

Purpose:

The macro **with-slots** creates a lexical context for referring to 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** can be used inside a method or inside any function.

The macro **with-slots** translates an appearance of the slot name as a variable into a call to the accessor generated by **defclass** unless the **:use-accessors** argument is **nil**, in which case **slot-value** is used instead of the accessor.

Syntax:

with-slots ({*instance-form* | (*instance-form option**)}*) {*form*}* [*Macro*]

option::= `:use-accessors` *flag* |
`:class` *class-name* |
`:prefix` *symbol*

Arguments:

The *instance-form* should evaluate to an object that has slots, such as an instance of a user-defined class. The *instance-form* should not evaluate to an instance of a standard type class.

Each *instance-form* is evaluated exactly once, upon entry to the body of the **with-slots** form. The *instance-form* forms are evaluated in the order in which they appear.

It is necessary that the class of the instance can be determined lexically (at compile-time). Either *instance-form* must be the name of a specialized parameter in the lambda-list of a method that lexically contains this **with-slots** form, or the **:class** option must be used to indicate the class of the instance.

The keyword options in this special form are not evaluated.

- The **:class** option is used to specify the class of the instance. Its argument is the name of a class. This option is necessary if the class of the instance cannot be determined from the lambda-list of a method that lexically contains the **with-slots** form or if the **with-slots** form does not occur within a method body. If the **:class** option is used, an error is signaled at runtime if the class of the instance is not the specified class or a subclass of the specified class. Note that if a superclass of the actual class of the instance is given here, only those slots accessible in instances of the superclass are accessible; this might be a smaller set of slots than those accessible in the instance itself.

with-slots

- The **:prefix** option is used to generate a variable name by which the given slot may be referenced. This name is given by the name of *symbol* followed by the slot name. It causes a symbol of the given name to be created and interned in the package that is current at the time the **with-slots** form is macro-expanded. This option can be used to keep separate two instances whose slot names overlap, such as two instances of the same class or two instances that have a common superclass.
- The **:use-accessors** option is used to specify whether accessing a slot is performed by calling the accessor function generated by **defclass** or by calling the function **slot-value**. If **:use-accessors** is **t**, the accessor function generated by **defclass** is called to access the slots; this means that any methods written for the accessor are also run. If **:use-accessors** is **nil**, **with-slots** accesses the slots by calling **slot-value** instead of the accessor. If accessors for the slot were not specified in the relevant **defclass** form, then the value of **:use-accessors** should be specified as **nil**. The default value of **:use-accessors** is **t**.

Values:

The values of the **with-slots** form are the values returned by the last form in its body.

Examples:

```
(defclass point () ((x 0) (y 0))
  (:accessor-prefix point-))

(defmethod move ((p point) dx dy)
  (with-slots (p)  ; p is known as a point from the method args
    (setf x (+ x dx) y (+ y dy))))       ; use accessor functions

(defmethod move ((p point) dx dy)
  (with-slots ((p :use-accessors nil))
    (setf x (+ x dx) y (+ y dy))))       ; use slot-value

(defmethod make-same-height ((p1 point) (p2 point))
  ;; use :prefix to make distinction between the two points
  (with-slots ((p1 :prefix p1-) (p2 :prefix p2-))
    (setf p1-y p2-y)))

(defmethod make-horizontal ((l line))
  ;; it is necessary to specify the class of point explicitly,
  ;; because there is no lexical way to determine it
  (with-slots (((left-point l) :class point :prefix left-)
               ((right-point l) :class point :prefix right-))
    (setf left-y right-y)))
```

Remarks:

The examples have used **setf** to change the value of instance variables; **setq** is also allowed.

An error is signaled if the class of *instance-form* cannot be inferred from the lexical context in which it occurs and the **:class** option is not specified.

An error is signaled if there is any conflict between variable names.

87–002

Systems for Open Computing™

Corporate Headquarters
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
TLX 37-29639

For U.S. Sales Office locations, call:
800 821-4643
In CA: 800 821-4642

European Headquarters
Sun Microsystems Europe, Inc.
Bagshot Manor, Green Lane
Bagshot, Surrey GU19 5NL
England
0276 51440
TLX 859017

Australia: (02) 413 2666
Canada: 416 477-6745
France: (1) 40 94 80 00
Germany: (089) 95094-0
Hong Kong: 852 5-8651688
Italy: (39) 6056337
Japan: (03) 221-7021
Korea: 2-7802255
Nordic Countries: + 46 (0)8 7647810
PRC: 1-8315568
Singapore: 224 3388
Spain: (1) 2532003
Switzerland: (1) 8289555
The Netherlands: 02155 24888
Taiwan: 2-7213257
UK: 0276 62111

Europe, Middle East, and Africa, call European Headquarters:
0276 51440

Elsewhere in the world, call Corporate Headquarters:
415 960-1300
Intercontinental Sales