The Art of the Metaobject Protocol
Chapters 5 and 6

Gregor Kiczales, Jim des Rivières, and Daniel G Bobrow
Xerox Palo Alto Research Center

Contents

In this part of the book, we provide the detailed specification of a metaobject protocol for CLOS. Our work with this protocol has always been rooted in our own implementation of CLOS, PCL. This has made it possible for us to have a user community, which in turn has provided us with feedback on this protocol as it has evolved. As a result, much of the design presented here is well-tested and stable. As this is being written, those parts have been implemented not only in PCL, but in at least three other CLOS implementations we know of. Other parts of the protocol, even though they have been implemented in one form or another in PCL and other implementations, are less well worked out. Work remains to improve not only the ease of use of these protocols, but also the balance they provide between user extensibility and implementor freedom.

In preparing this specification, it is our hope that it will provide a basis for the users and implementors who wish to work with a metaobject protocol for CLOS. This document should not be construed as any sort of final word or standard, but rather only as documentation of what has been done so far. We look forward to seeing the improvements, both small and large, which we hope this publication will catalyze.

To this end, for Part II only (chapters 5 and 6), we grant permission to prepare revisions or other derivative works including any amount of the original text. We ask only that you properly acknowledge the source of the original text and explicitly allow subsequent revisions and derivative works under the same terms. To further facilitate improvements in this work, we have made the electronic source for these chapters publicly available; it can be accessed by anonymous FTP from the `/pcl/mop` directory on `arisia.xerox.com`.

In addition to the valuable feedback from users of PCL, the protocol design presented here has benefited from detailed comments and suggestions by the following people: Kim Barrett, Scott Cyphers, Harley Davis, Patrick Dussud, John Foderaro, Richard P. Gabriel, David Gray, David A. Moon, Andreas Paepcke, Chris Richardson, Walter van Roggen, and Jon L. White. We are very grateful to each of them. Any remaining errors, inconsistencies or design deficiencies are the responsibility of the authors alone.

Chapter 5

Concepts

Introduction

The CLOS Specification [X3J13, CLtLII] describes the standard Programmer Interface for the Common Lisp Object System (CLOS). This document extends that specification by defining a metaobject protocol for CLOS—that is, a description of CLOS itself as an extensible CLOS program. In this description, the fundamental elements of CLOS programs (classes, slot definitions, generic functions, methods, specializers and method combinations) are represented by first-class objects. The behavior of CLOS is provided by these objects, or, more precisely, by methods specialized to the classes of these objects.

Because these objects represent pieces of CLOS programs, and because their behavior provides the behavior of the CLOS language itself, they are considered meta-level objects or metaobjects. The protocol followed by the metaobjects to provide the behavior of CLOS is called the CLOS Metaobject Protocol (MOP).

Metaobjects

For each kind of program element there is a corresponding *basic metaobject class*. These are the classes: **class**, **slot-definition**, **generic-function**, **method** and **method-combination**. A *metaobject class* is a subclass of exactly one of these classes. The results are undefined if an attempt is made to define a class that is a subclass of more than one basic metaobject class. A *metaobject* is an instance of a metaobject class.

Each metaobject represents one program element. Associated with each metaobject is the information required to serve its role. This includes information that might be provided directly in a user interface macro such as **defclass** or **defmethod**. It also includes information computed indirectly from other metaobjects such as that computed from class inheritance or the full set of methods associated with a generic function.

Much of the information associated with a metaobject is in the form of connections to other metaobjects. This interconnection means that the role of a metaobject is always based on that of other metaobjects. As an introduction to this interconnected structure, this section presents a partial enumeration of the kinds of information associated with each kind of metaobject. More detailed information is presented later.

Classes

A *class metaobject* determines the structure and the default behavior of its instances. The following information is associated with class metaobjects:

- The name, if there is one, is available as an object.
- The direct subclasses, direct superclasses and class precedence list are available as lists of class metaobjects.
- The slots defined directly in the class are available as a list of direct slot definition metaobjects. The slots which are accessible in instances of the class are available as a list of effective slot definition metaobjects.
- The documentation is available as a string or **nil**.
- The methods which use the class as a specializer, and the generic functions associated with those methods are available as lists of method and generic function metaobjects respectively.

Slot Definitions

A *slot definition metaobject* contains information about the definition of a slot. There are two kinds of slot definition metaobjects. A direct slot definition metaobject is used to represent the direct definition of a slot in a class. This corresponds roughly to the slot specifiers found in **defclass** forms. An effective slot definition metaobject is used to represent information, including inherited information, about a slot which is accessible in instances of a particular class.

Associated with each class metaobject is a list of direct slot definition metaobjects representing the slots defined directly in the class. Also associated with each class metaobject is a list of effective slot definition metaobjects representing the set of slots accessible in instances of that class.

The following information is associated with both direct and effective slot definitions metaobjects:

- The name, allocation, and type are available as forms that could appear in a **defclass** form

- The initialization form, if there is one, is available as a form that could appear in a **defclass** form. The initialization form together with its lexical environment is available as a function of no arguments which, when called, returns the result of evaluating the initialization form in its lexical environment. This is called the *initfunction* of the slot.
- The slot filling initialization arguments are available as a list of symbols.
- The documentation is available as a string or **nil**.

Certain other information is only associated with direct slot definition metaobjects. This information applies only to the direct definition of the slot in the class (it is not inherited).

- The function names of those generic functions for which there are automatically generated reader and writer methods. This information is available as lists of function names. Any accessors specified in the **defclass** form are broken down into their equivalent readers and writers in the direct slot definition.

Information, including inherited information, which applies to the definition of a slot in a particular class in which it is accessible is associated only with effective slot definition metaobjects.

- For certain slots, the location of the slot in instances of the class is available.

Generic Functions

A *generic function metaobject* contains information about a generic function over and above the information associated with each of the generic function's methods.

- The name is available as a function name.
- The methods associated with the generic function are available as a list of method metaobjects.
- The default class for this generic function's method metaobjects is available as a class metaobject.
- The lambda list is available as a list.
- The method combination is available as a method combination metaobject.
- The documentation is available as a string or **nil**.
- The argument precedence order is available as a permutation of those symbols from the lambda list which name the required arguments of the generic function.

- The declarations are available as a list of declarations.

  Terminology Note:
  There is some ambiguity in Common Lisp about the terms used to identify the various parts of **declare** special forms. In this document, the term *declaration* is used to refer to an object that could be an argument to a **declare** special form. For example, in the special form **(declare (special *g1*))**, the list **(special *g1*)** is a declaration.

Methods

A *method metaobject* contains information about a specific method.

- The qualifiers are available as a list of of non-null atoms.
- The lambda list is available as a list.
- The specializers are available as a list of specializer metaobjects.
- The function is available as a function. This function can be applied to arguments and a list of next methods using **apply** or **funcall**.
- When the method is associated with a generic function, that generic function metaobject is available. A method can be associated with at most one generic function at a time.
- The documentation is available as a string or **nil**.

Specializers

A specializer metaobject represents the specializers of a method. Class metaobjects are themselves specializer metaobjects. A special kind of specializer metaobject is used for **eql** specializers.

Method Combinations

A *method combination metaobject* represents the information about the method combination being used by a generic function.

Note:
This document does not specify the structure of method combination metaobjects.

Inheritance Structure of Metaobject Classes

The inheritance structure of the specified metaobject classes is shown in Table 5.1.

Each class marked with a "∗" is an *abstract class* and is not intended to be instantiated. The results are undefined if an attempt is made to make an instance of one of these classes with **make-instance**.

	Metaobject Class	Direct Superclasses
	standard-object	(t)
	funcallable-standard-object	(standard-object function)
*	metaobject	(standard-object)
*	generic-function	(metaobject funcallable-standard-object)
	standard-generic-function	(generic-function)
*	method	(metaobject)
	standard-method	(method)
*	standard-accessor-method	(standard-method)
	standard-reader-method	(standard-accessor-method)
	standard-writer-method	(standard-accessor-method)
*	method-combination	(metaobject)
*	slot-definition	(metaobject)
*	direct-slot-definition	(slot-definition)
*	effective-slot-definition	(slot-definition)
*	standard-slot-definition	(slot-definition)
	standard-direct-slot-definition	(standard-slot-definition direct-slot-definition)
	standard-effective-slot-definition	(standard-slot-definition effective-slot-definition)
*	specializer	(metaobject)
	eql-specializer	(specializer)
*	class	(specializer)
	built-in-class	(class)
	forward-referenced-class	(class)
	standard-class	(class)
	funcallable-standard-class	(class)

Table 5.1 Direct superclass relationships among the specified metaobject classes. The class of every class shown is **standard-class** except for the class **t** which is an instance of the class **built-in-class** and the classes **generic-function** and **standard-generic-function** which are instances of the class **funcallable-standard-class**.

The classes **standard-class**, **standard-direct-slot-definition**, **standard-effective-slot-definition**, **standard-method**, **standard-reader-method**, **standard-writer-method** and **standard-generic-function** are called *standard metaobject classes*. For each kind of metaobject, this is the class the user interface macros presented in the CLOS Specification use by default. These are also the classes on which user specializations are normally based.

The classes **built-in-class**, **funcallable-standard-class** and **forward-referenced-class** are special-purpose class metaobject classes. Built-in classes are instances of the class **built-in-class**. The class **funcallable-standard-class** provides a special kind of instances described in the section called "Funcallable Instances." When the definition of a class references another class which has not yet been defined, an instance of **forward-referenced-class** is used as a stand-in until the class is actually defined.

The class **standard-object** is the *default direct superclass* of the class **standard-class**. When an instance of the class **standard-class** is created, and no direct superclasses are explicitly specified, it defaults to the class **standard-object**. In this way, any behavior associated with the class **standard-object** will be inherited, directly or indirectly, by all instances of the class **standard-class**. A subclass of **standard-class** may have a different class as its default direct superclass, but that class must be a subclass of the class **standard-object**.

The same is true for **funcallable-standard-class** and **funcallable-standard-object**.

The class **specializer** captures only the most basic behavior of method specializers, and is not itself intended to be instantiated. The class **class** is a direct subclass of **specializer** reflecting the property that classes by themselves can be used as method specializers. The class **eql-specializer** is used for **eql** specializers.

Implementation and User Specialization

The purpose of the Metaobject Protocol is to provide users with a powerful mechanism for extending and customizing the basic behavior of the Common Lisp Object System. As an object-oriented description of the basic CLOS behavior, the Metaobject Protocol makes it possible to create these extensions by defining specialized subclasses of existing metaobject classes.

The Metaobject Protocol provides this capability without interfering with the implementor's ability to develop high-performance implementations. This balance between user extensibility and implementor freedom is mediated by placing explicit restrictions on each. Some of these restrictions are general—they apply to the entire class graph and the applicability of all methods. These are presented in this section.

The following additional terminology is used to present these restrictions:

- Metaobjects are divided into three categories. Those defined in this document are called *specified*; those defined by an implementation but not mentioned in this document are called *implementation-specific*; and those defined by a portable program are called *portable*.

- A class I is *interposed* between two other classes C_1 and C_2 if and only if there is some path, following direct superclasses, from the class C_1 to the class C_2 which includes I.

- A method is *specialized to* a class if and only if that class is in the list of specializers associated with the method; and the method is in the list of methods associated with some generic function.

- In a given implementation, a specified method is said to have been *promoted* if and only if the specializers of the method, $S_1 \ldots S_n$, are defined in this specification as the classes $C_1 \ldots C_n$, but in the implementation, one or more of the specializers S_i is a superclass of the class given in the specification C_i.

- For a given generic function and set of arguments, a method M_2 *extends* a method M_1 if and only if:

  (i) M_1 and M_2 are both associated with the given generic function,
  (ii) M_1 and M_2 are both applicable to the given arguments,
  (iii) the specializers and qualifiers of the methods are such that when the generic function is called, M_2 is executed before M_1,
  (iv) M_1 will be executed if and only if **call-next-method** is invoked from within the body of M_2 and
  (v) **call-next-method** is invoked from within the body of M_2, thereby causing M_1 to be executed.

- For a given generic function and set of arguments, a method M_2 *overrides* a method M_1 if and only if conditions i through iv above hold and

  (v') **call-next-method** is not invoked from within the body of M_2, thereby preventing M_1 from being executed.

Restrictions on Implementations

Implementations are allowed latitude to modify the structure of specified classes and methods. This includes: the interposition of implementation-specific classes; the promotion of specified methods; and the consolidation of two or more specified methods into a single method specialized to interposed classes.

Any such modifications are permitted only so long as for any portable class C_p that is a subclass of one or more specified classes $C_0 \ldots C_i$, the following conditions are met:

- In the actual class precedence list of C_p, the classes $C_0 \ldots C_i$ must appear in the same order as they would have if no implementation-specific modifications had been made.

- The method applicability of any specified generic function must be the same in terms of behavior as it would have been had no implementation-specific changes been made. This includes specified generic functions that have had portable methods added. In this context, the expression "the same in terms of behavior" means that methods with the same behavior as those specified are applicable, and in the same order.

- No portable class C_p may inherit, by virtue of being a direct or indirect subclass of a specified class, any slot for which the name is a symbol accessible in the **common-lisp-user** package or exported by any package defined in the ANSI Common Lisp standard.

- Implementations are free to define implementation-specific before- and after-methods on specified generic functions. Implementations are also free to define implementation-specific around-methods with extending behavior.

Restrictions on Portable Programs

Portable programs are allowed to define subclasses of specified classes, and are permitted to define methods on specified generic functions, with the following restrictions. The results are undefined if any of these restrictions is violated.

- Portable programs must not redefine any specified classes, generic functions, methods or method combinations. Any method defined by a portable program on a specified generic function must have at least one specializer that is neither a specified class nor an **eql** specializer whose associated value is an instance of a specified class.

- Portable programs may define methods that extend specified methods unless the description of the specified method explicitly prohibits this. Unless there is a specific statement to the contrary, these extending methods must return whatever value was returned by the call to **call-next-method**.

- Portable programs may define methods that override specified methods only when the description of the specified method explicitly allows this. Typically, when a method is allowed to be overridden, a small number of related methods will need to be overridden as well.

  An example of this is the specified methods on the generic functions **add-dependent**, **remove-dependent** and **map-dependents**. Overriding a specified method on one of these generic functions requires that the corresponding method on the other two generic functions be overridden as well.

- Portable methods on specified generic functions specialized to portable metaobject classes must be defined before any instances of those classes (or any subclasses) are created, either directly or indirectly by a call to **make-instance**. Methods can be defined after instances are created by **allocate-instance** however. Portable metaobject classes cannot be redefined.

> **Implementation Note:**
> The purpose of this last restriction is to permit implementations to provide performance optimizations by analyzing, at the time the first instance of a metaobject class is initialized, what portable methods will be applicable to it. This can make it possible to optimize calls to those specified generic functions which would have no applicable portable methods.

> **Note:**
> The specification technology used in this document needs further development. The concepts of object-oriented protocols and subclass specialization are intuitively familiar to programmers of object-oriented systems; the protocols presented here fit quite naturally into this framework. Nonetheless, in preparing this document, we have found it difficult to give specification-quality descriptions of the protocols in a way that makes it clear what extensions users can and cannot write. Object-oriented protocol specification is inherently about specifying leeway, and this seems difficult using current technology.

Processing of the User Interface Macros

A list in which the first element is one of the symbols **defclass**, **defmethod**, **defgeneric**, **define-method-combination**, **generic-function**, **generic-flet** or **generic-labels**, and which has proper syntax for that macro is called a *user interface macro form*. This document provides an extended specification of the **defclass**, **defmethod** and **defgeneric** macros.

The user interface macros **defclass**, **defgeneric** and **defmethod** can be used not only to define metaobjects that are instances of the corresponding standard metaobject class, but also to define metaobjects that are instances of appropriate portable metaobject classes. To make it possible for portable metaobject classes to properly process the information appearing in the macro form, this document provides a limited specification of the processing of these macro forms.

User interface macro forms can be *evaluated* or *compiled* and later *executed*. The effect of evaluating or executing a user interface macro form is specified in terms of calls to specified functions and generic functions which provide the actual behavior of the macro. The arguments received by these functions and generic functions are derived in a specified way from the macro form.

Converting a user interface macro form into the arguments to the appropriate functions and generic functions has two major aspects: the conversion of the macro argument syntax into a form more suitable for later processing, and the processing of macro arguments which are forms to be evaluated (including method bodies).

In the syntax of the **defclass** macro, the *initform* and *default-initarg-initial-value-form* arguments are forms which will be evaluated one or more times after the macro form is evaluated or executed. Special processing must be done on these arguments to ensure that the lexical scope of the forms is captured properly. This is done by building a function of zero arguments which, when called, returns the result of evaluating the form in the proper lexical environment.

In the syntax of the **defmethod** macro the *form** argument is a list of forms that comprise the body of the method definition. This list of forms must be processed specially to capture the lexical scope of the macro form. In addition, the lexical functions available only in the body of methods must be introduced. To allow this and any other special processing (such as slot access optimization), a specializable protocol is used for processing the body of methods. This is discussed in the section "Processing Method Bodies."

Compile-file Processing of the User Interface Macros

It is common practice for Common Lisp compilers, while processing a file or set of files, to maintain information about the definitions that have been compiled so far. Among other things, this makes it possible to ensure that a global macro definition (**defmacro** form) which appears in a file will affect uses of the macro later in that file. This information about the state of the compilation is called the *compile-file environment*.

When compiling files containing CLOS definitions, it is useful to maintain certain additional information in the compile-file environment. This can make it possible to issue various kinds of warnings (e.g., lambda list congruence) and to do various performance optimizations that would not otherwise be possible.

At this time, there is such significant variance in the way existing Common Lisp implementations handle compile-file environments that it would be premature to specify this mechanism. Consequently, this document specifies only the behavior of evaluating or executing user interface macro forms. What functions and generic functions are called during compile-file processing of a user interface macro form is not specified. Implementations are free to define and document their own behavior. Users may need to check implementation-specific behavior before attempting to compile certain portable programs.

The defclass Macro

The evaluation or execution of a **defclass** form results in a call to the **ensure-class** function. The arguments received by **ensure-class** are derived from the **defclass** form in a defined way. The exact macro-expansion of the **defclass** form is not defined, only the relationship between the arguments to the **defclass** macro and the arguments received by the **ensure-class** function. Examples of typical **defclass** forms and sample expansions are shown in Figures 5.1 and 5.2.

```
(defclass plane (moving-object graphics-object)
     ((altitude :initform 0 :accessor plane-altitude)
      (speed))
  (:default-initargs :engine *jet*))

(ensure-class 'plane
  ':direct-superclasses '(moving-object graphics-object)
  ':direct-slots (list (list ':name 'altitude
                              ':initform '0
                              ':initfunction #'(lambda () 0)
                              ':readers '(plane-altitude)
                              ':writers '((setf plane-altitude)))
                        (list ':name 'speed))
  ':direct-default-initargs (list (list ':engine
                                        '*jet*
                                        #'(lambda () *jet*))))
```

Figure 5.1 A **defclass** form with standard slot and class options and an expansion of it that would result in the proper call to **ensure-class**.

- The *name* argument to **defclass** becomes the value of the first argument to **ensure-class**. This is the only positional argument accepted by **ensure-class**; all other arguments are keyword arguments.

- The *direct-superclasses* argument to **defclass** becomes the value of the **:direct-superclasses** keyword argument to **ensure-class**.

- The *direct slots* argument to **defclass** becomes the value of the **:direct-slots** keyword argument to **ensure-class**. Special processing of this value is done to regularize the form of each slot specification and to properly capture the lexical scope of the initialization forms. This is done by converting each slot specification to a property list called a *canonicalized slot specification*. The resulting list of canonicalized slot specifications is the value of the **:direct-slots** keyword argument.

  Canonicalized slot specifications are later used as the keyword arguments to a generic function which will, in turn, pass them to **make-instance** for use as a set of initialization arguments. Each canonicalized slot specification is formed from the corresponding slot specification as follows:

  - The name of the slot is the value of the **:name** property. This property appears in every canonicalized slot specification.

```
(defclass sst (plane)
     ((mach mag-step 2
            locator sst-mach
            locator mach-location
            :reader mach-speed
            :reader mach))
  (:metaclass faster-class)
  (another-option foo bar))

(ensure-class 'sst
  ':direct-superclasses '(plane)
  ':direct-slots (list (list ':name 'mach
                             ':readers '(mach-speed mach)
                             'mag-step '2
                             'locator '(sst-mach mach-location)))
  ':metaclass 'faster-class
  'another-option '(foo bar))
```

Figure 5. 2 A **defclass** form with non-standard class and slot options, and an expansion of it which results in the proper call to **ensure- class**. Note that the order of the slot options has not affected the order of the properties in the canonicalized slot specification, but has affected the order of the elements in the lists which are the values of those properties.

- When the **:initform** slot option is present in the slot specification, then both the **:initform** and **:initfunction** properties are present in the canonicalized slot specification. The value of the **:initform** property is the initialization form. The value of the **:initfunction** property is a function of zero arguments which, when called, returns the result of evaluating the initialization form in its proper lexical environment.

  If the **:initform** slot option is not present in the slot specification, then either the **:initfunction** property will not appear, or its value will be false. In such cases, the value of the **:initform** property, or whether it appears, is unspecified.
- The value of the **:initargs** property is a list of the values of each **:initarg** slot option. If there are no **:initarg** slot options, then either the **:initargs** property will not appear or its value will be the empty list.
- The value of the **:readers** property is a list of the values of each **:reader** and **:accessor** slot option. If there are no **:reader** or **:accessor** slot options, then either the **:readers** property will not appear or its value will be the empty list.
- The value of the **:writers** property is a list of the values specified by each **:writer** and **:accessor** slot option. The value specified by a **:writer** slot option is just the value of the slot option. The value specified by an **:accessor** slot option is a two element list: the first element is the symbol **setf**, the second element is the value of the slot option. If there are no **:writer** or **:accessor** slot options, then either the **:writers** property will not appear or its value will be the empty list.
- The value of the **:documentation** property is the value of the **:documentation** slot option. If there is no **:documentation** slot option, then either the **:documentation** property will not appear or its value will be false.
- All other slot options appear as the values of properties with the same name as the slot option. Note that this includes not only the remaining standard slot options (**:allocation** and **:type**), but also any other options and values appearing in the slot specification. If one of these slot options appears more than once, the value of the property will be a list of the specified values.
- An implementation is free to add additional properties to the canonicalized slot specification provided these are not symbols accessible in the **common-lisp-user** package, or exported by any package defined in the ANSI Common Lisp standard.

Returning to the correspondence between arguments to the **defclass** macro and the arguments received by the **ensure-class** function:

- The *default initargs* class option, if it is present in the **defclass** form, becomes the value of the **:direct-default-initargs** keyword argument to **ensure-class**. Special processing of this value is done to properly capture the lexical scope of the default value forms. This is done by converting each default initarg in the class option into a *canonicalized default initarg*. The resulting list of canonicalized default initargs is the value of the **:direct-default-initargs** keyword argument to **ensure-class**.

A canonicalized default initarg is a list of three elements. The first element is the name; the second is the actual form itself; and the third is a function of zero arguments which, when called, returns the result of evaluating the default value form in its proper lexical environment.

- The *metaclass* class option, if it is present in the **defclass** form, becomes the value of the **:metaclass** keyword argument to **ensure-class**.
- The *documentation* class option, if it is present in the **defclass** form, becomes the value of the **:documentation** keyword argument to **ensure-class**.
- Any other class options become the value of keyword arguments with the same name. The value of the keyword argument is the tail of the class option. An error is signaled if any class option appears more than once in the **defclass** form.

In the call to **ensure-class**, every element of its arguments appears in the same left-to-right order as the corresponding element of the **defclass** form, except that the order of the properties of canonicalized slot specifications is unspecified. The values of properties in canonicalized slot specifications do follow this ordering requirement. Other ordering relationships in the keyword arguments to **ensure-class** are unspecified.

The result of the call to **ensure-class** is returned as the result of evaluating or executing the **defclass** form.

The defmethod Macro

The evaluation or execution of a **defmethod** form requires first that the body of the method be converted to a method function. This process is described in the next section. The result of this process is a method function and a set of additional initialization arguments to be used when creating the new method. Given these two values, the evaluation or execution of a **defmethod** form proceeds in three steps.

The first step ensures the existence of a generic function with the specified name. This is done by calling the function **ensure-generic-function**. The first argument in this call is the generic function name specified in the **defmethod** form.

The second step is the creation of the new method metaobject by calling **make-instance**. The class of the new method metaobject is determined by calling **generic-function-method-class** on the result of the call to **ensure-generic-function** from the first step.

The initialization arguments received by the call to **make-instance** are as follows:

- The value of the **:qualifiers** initialization argument is a list of the qualifiers which appeared in the **defmethod** form. No special processing is done on these values. The order of the elements of this list is the same as in the **defmethod** form.
- The value of the **:lambda-list** initialization argument is the unspecialized lambda list from the **defmethod** form.

- The value of the **:specializers** initialization argument is a list of the specializers for the method. For specializers which are classes, the specializer is the class metaobject itself. In the case of **eql** specializers, it will be an **eql-specializer** metaobject obtained by calling **intern-eql-specializer** on the result of evaluating the **eql** specializer form in the lexical environment of the **defmethod** form.

- The value of the **:function** initialization argument is the method function.

- The value of the **:declarations** initialization argument is a list of the declarations from the **defmethod** form. If there are no declarations in the macro form, this initialization argument either doesn't appear, or appears with a value of the empty list.

- The value of the **:documentation** initialization argument is the documentation string from the **defmethod** form. If there is no documentation string in the macro form this initialization argument either doesn't appear, or appears with a value of false.

- Any other initialization argument produced in conjunction with the method function are also included.

- The implementation is free to include additional initialization arguments provided these are not symbols accessible in the **common-lisp-user** package, or exported by any package defined in the ANSI Common Lisp standard.

In the third step, **add-method** is called to add the newly created method to the set of methods associated with the generic function metaobject.

The result of the call to **add-method** is returned as the result of evaluating or executing the **defmethod** form.

An example showing a typical **defmethod** form and a sample expansion is shown in Figure 5.3. The processing of the method body for this method is shown in Figure 5.4.

Processing Method Bodies

Before a method can be created, the list of forms comprising the method body must be converted to a method function. This conversion is a two step process.

Note:

The body of methods can also appear in the **:initial-methods** option of **defgeneric** forms. Initial methods are not considered by any of the protocols specified in this document.

The first step occurs during macro-expansion of the macro form. In this step, the method lambda list, declarations and body are converted to a lambda expression called a *method lambda*. This conversion is based on information associated with the generic function definition in effect at the time the macro form is expanded.

```
(defmethod move :before ((p position) (l (eql 0))
                         &optional (visiblyp t)
                         &key color)
  (set-to-origin p)
  (when visiblyp (show-move p 0 color)))

(let ((#:g001 (ensure-generic-function 'move)))
  (add-method #:g001
    (make-instance (generic-function-method-class #:g001)
                   ':qualifiers '(:before)
                   ':specializers (list (find-class 'position)
                                        (intern-eql-specializer 0))
                   ':lambda-list '(p l &optional (visiblyp t)
                                       &key color)
                   ':function (function method-lambda)
                   'additional-initarg-1 't
                   'additional-initarg-2 '39)))
```

Figure 5. 3 An example **defmethod** form and one possible correct expansion. In the expansion, *method-lambda* is the result of calling **make-method-lambda** as described in the section "Processing Method Bodies". The initargs appearing after **:function** are assumed to be additional initargs returned from the call to **make-method-lambda**.

```
(let ((gf (ensure-generic-function 'move)))
  (make-method-lambda
    gf
    (class-prototype (generic-function-method-class gf))
    '(lambda (p l &optional (visiblyp t) &key color)
       (set-to-origin p)
       (when visiblyp (show-move p 0 color)))
    environment))
```

Figure 5. 4 During macro-expansion of the **defmethod** macro shown in Figure 5.3, code similar to this would be run to produce the method lambda and additional initargs. In this example, *environment* is the macroexpansion environment of the **defmethod** macro form.

The generic function definition is obtained by calling **ensure-generic-function** with a first argument of the generic function name specified in the macro form. The **:lambda-list** keyword argument is not passed in this call.

Given the generic function, production of the method lambda proceeds by calling **make-method-lambda**. The first argument in this call is the generic function obtained as described above. The second argument is the result of calling **class-prototype** on the result of calling **generic-function-method-class** on the generic function. The third argument is a lambda expression formed from the method lambda list, declarations and body. The fourth argument is the macro-expansion environment of the macro form; this is the value of the **&environment** argument to the **defmethod** macro.

The generic function **make-method-lambda** returns two values. The first is the method lambda itself. The second is a list of initialization arguments and values. These are included in the initialization arguments when the method is created.

In the second step, the method lambda is converted to a function which properly captures the lexical scope of the macro form. This is done by having the method lambda appear in the macro-expansion as the argument of the **function** special form. During the subsequent evaluation of the macro-expansion, the result of the **function** special form is the method function.

The defgeneric Macro

The evaluation or execution of a **defgeneric** form results in a call to the **ensure-generic-function** function. The arguments received by **ensure-generic-function** are derived from the **defgeneric** form in a defined way. As with **defclass** and **defmethod**, the exact macro-expansion of the **defgeneric** form is not defined, only the relationship between the arguments to the macro and the arguments received by **ensure-generic-function**.

- The *function-name* argument to **defgeneric** becomes the first argument to **ensure-generic-function**. This is the only positional argument accepted by **ensure-generic-function**; all other arguments are keyword arguments.

- The *lambda-list* argument to **defgeneric** becomes the value of the **:lambda-list** keyword argument to **ensure-generic-function**.

- For each of the options **:argument-precedence-order**, **:documentation**, **:generic-function-class** and **:method-class**, the value of the option becomes the value of the keyword argument with the same name. If the option does not appear in the macro form, the keyword argument does not appear in the resulting call to **ensure-generic-function**.

- For the option **declare**, the list of declarations becomes the value of the **:declarations** keyword argument. If the **declare** option does not appear in the macro form, the **:declarations** keyword argument does not appear in the call to **ensure-generic-function**.

- The handling of the **:method-combination** option is not specified.

The result of the call to **ensure-generic-function** is returned as the result of evaluating or executing the **defgeneric** form

Subprotocols

This section provides an overview of the Metaobject Protocols. The detailed behavior of each function, generic function and macro in the Metaobject Protocol is presented in Chapter 6. The remainder of this chapter is intended to emphasize connections among the parts of the Metaobject Protocol, and to provide some examples of the kinds of specializations and extensions the protocols are designed to support.

Metaobject Initialization Protocols

Like other objects, metaobjects can be created by calling **make-instance**. The initialization arguments passed to **make-instance** are used to initialize the metaobject in the usual way. The set of legal initialization arguments, and their interpretation, depends on the kind of metaobject being created. Implementations and portable programs are free to extend the set of legal initialization arguments. Detailed information about the initialization of each kind of metaobject are provided in Chapter 6; this section provides an overview and examples of this behavior.

Initialization of Class Metaobjects

Class metaobjects created with **make-instance** are usually *anonymous*; that is, they have no proper name. An anonymous class metaobject can be given a proper name using **(setf find-class)** and **(setf class-name)**.

When a class metaobject is created with **make-instance**, it is initialized in the usual way. The initialization arguments passed to **make-instance** are use to establish the definition of the class. Each initialization argument is checked for errors and associated with the class metaobject. The initialization arguments correspond roughly to the arguments accepted by the **defclass** macro, and more closely to the arguments accepted by the **ensure-class** function.

Some class metaobject classes allow their instances to be redefined. When permissible, this is done by calling **reinitialize-instance**. This is discussed in the next section.

An example of creating an anonymous class directly using **make-instance** follows:

```
(flet ((zero () 0)
       (propellor () *propellor*))
  (make-instance 'standard-class
    :name '(my-class foo)
    :direct-superclasses (list (find-class 'plane)
                               another-anonymous-class)
    :direct-slots `((:name x
                     :initform 0
                     :initfunction ,#'zero
                     :initargs (:x)
                     :readers (position-x)
                     :writers ((setf position-x)))
                    (:name y
                     :initform 0
                     :initfunction ,#'zero
                     :initargs (:y)
                     :readers (position-y)
                     :writers ((setf position-y))))
    :direct-default-initargs `((:engine *propellor* ,#'propellor))))
```

Reinitialization of Class Metaobjects

Some class metaobject classes allow their instances to be reinitialized. This is done by calling **reinitialize-instance**. The initialization arguments have the same interpretation as in class initialization.

If the class metaobject was finalized before the call to **reinitialize-instance**, **finalize-inheritance** will be called again once all the initialization arguments have been processed and associated with the class metaobject. In addition, once finalization is complete, any dependents of the class metaobject will be updated by calling **update-dependent**.

Initialization of Generic Function and Method Metaobjects

An example of creating a generic function and a method metaobject, and then adding the method to the generic function is shown below. This example is comparable to the method definition shown in Figure 5.3.

```
(let* ((gf (make-instance 'standard-generic-function
                          :lambda-list '(p l &optional visiblyp &key)))
       (method-class (generic-function-method-class gf)))
  (multiple-value-bind (lambda initargs)
       (make-method-lambda
         gf
         (class-prototype method-class)
         '(lambda (p l &optional (visiblyp t) &key color)
            (set-to-origin p)
            (when visiblyp (show-move p 0 color)))
         nil)
(add-method gf
            (apply #'make-instance method-class
                   :function (compile nil lambda)
                   :specializers (list (find-class 'position)
                                       (intern-eql-specializer 0))
                   :qualifiers ()
                   :lambda-list '(p l &optional (visiblyp t)
                                      &key color)
                   initargs))))
```

Class Finalization Protocol

Class *finalization* is the process of computing the information a class inherits from its superclasses and preparing to actually allocate instances of the class. The class finalization process includes computing the class's class precedence list, the full set of slots accessible in instances of the class and the full set of default initialization arguments for the class. These values are associated with the class metaobject and can be accessed by calling the appropriate reader. In addition, the class finalization process makes decisions about how instances of the class will be implemented.

To support forward-referenced superclasses, and to account for the fact that not all classes are actually instantiated, class finalization is not done as part of the initialization of the class metaobject. Instead, finalization is done as a separate protocol, invoked by calling the generic function **finalize-inheritance**. The exact point at which **finalize-inheritance** is called depends on the class of the class metaobject; for **standard-class** it is called sometime after all the classes superclasses are defined, but no later than when the first instance of the class is allocated (by **allocate-instance**).

The first step of class finalization is computing the class precedence list. Doing this first allows subsequent steps to access the class precedence list. This step is performed by calling the generic function **compute-class-precedence-list**. The value returned from this call is associated with the class metaobject and can be accessed by calling the **class-precedence-list** generic function.

The second step is computing the full set of slots that will be accessible in instances of the class. This step is performed by calling the generic function **compute-slots**. The result of this call is a list of effective slot definition metaobjects. This value is associated with the class metaobject and can be accessed by calling the **class-slots** generic function.

The behavior of **compute-slots** is itself layered, consisting of calls to **effective-slot-definition-class** and **compute-effective-slot-definition**.

The final step of class finalization is computing the full set of initialization arguments for the class. This is done by calling the generic function **compute-default-initargs**. The value returned by this generic function is associated with the class metaobject and can be accessed by calling **class-default-initargs**.

If the class was previously finalized, **finalize-inheritance** may call **make-instances-obsolete**. The circumstances under which this happens are describe in the section of the CLOS specification called "Redefining Classes."

Forward-referenced classes, which provide a temporary definition for a class which has been referenced but not yet defined, can never be finalized. An error is signalled if **finalize-inheritance** is called on a forward-referenced class.

Instance Structure Protocol

The instance structure protocol is responsible for implementing the behavior of the slot access functions like **slot-value** and **(setf slot-value)**.

For each CLOS slot access function other than **slot-exists-p**, there is a corresponding generic function which actually provides the behavior of the function. When called, the slot access function finds the pertinent effective slot definition metaobject, calls the corresponding generic function and returns its result. The arguments passed on to the generic function include one additional value, the class of the *object* argument, which always immediately precedes the *object* argument

The correspondences between slot access function and underlying slot access generic function are as follows:

Slot Access Function	**Corresponding Slot Access Generic Function**
slot-boundp	**slot-boundp-using-class**
slot-makunbound	**slot-makunbound-using-class**
slot-value	**slot-value-using-class**
(setf slot-value)	**(setf slot-value-using-class)**

At the lowest level, the instance structure protocol provides only limited mechanisms for portable programs to control the implementation of instances and to directly access the storage associated with instances without going through the indirection of slot access. This is done to allow portable programs to perform certain commonly requested slot access optimizations.

In particular, portable programs can control the implementation of, and obtain direct access to, slots with allocation **:instance** and type **t**. These are called *directly accessible slots*.

The relevant specified around-method on **compute-slots** determines the implementation of instances by deciding how each slot in the instance will be stored. For each directly accessible slot, this method allocates a *location* and associates it with the effective slot definition metaobject. The location can be accessed by calling the **slot-definition-location** generic function. Locations are non-negative integers. For a given class, the locations increase consecutively, in the order that the directly accessible slots appear in the list of effective slots. (Note that here, the next paragraph, and the specification of this around-method are the only places where the value returned by **compute-slots** is described as a list rather than a set.)

Given the location of a directly accessible slot, the value of that slot in an instance can be accessed with the appropriate accessor. For **standard-class**, this accessor is the function **standard-instance-access**. For **funcallable-standard-class**, this accessor is the function **funcallable-standard-instance-access**. In each case, the arguments to the accessor are the instance and the slot location, in that order. See the definition of each accessor in Chapter 6 for additional restrictions on the use of these function.

Portable programs are permitted to affect and rely on the allocation of locations only in the following limited way: By first defining a portable primary method on **compute-slots** which orders the returned value in a predictable way, and then relying on the defined behavior of the specified around-method to assign locations to all directly accessible slots. Portable programs may compile-in calls to low-level accessors which take advantage of the resulting predictable allocation of slot locations.

Example:

The following example shows the use of this mechanism to implement a new class metaobject class, **ordered-class** and class option **:slot-order**. This option provides control over the allocation of slot locations. In this simple example implementation, the **:slot-order** option is not inherited by subclasses; it controls only instances of the class itself.

```
(defclass ordered-class (standard-class)
     ((slot-order :initform ()
                  :initarg :slot-order
                  :reader class-slot-order)))

(defmethod compute-slots ((class ordered-class))
  (let ((order (class-slot-order class)))
    (sort (copy-list (call-next-method))
          #'(lambda (a b)
              (< (position (slot-definition-name a) order)
                 (position (slot-definition-name a) order))))))
```

Following is the source code the user of this extension would write. Note that because the code above doesn't implement inheritance of the **:slot-order** option, the function **distance** must not be called on instances of subclasses of **point**; it can only be called on instances of **point** itself.

```
(defclass point ()
     ((x :initform 0)
      (y :initform 0))
  (:metaclass ordered-class)
  (:slot-order x y))

(defun distance (point)
  (sqrt (/ (+ (expt (standard-instance-access point 0) 2)
              (expt (standard-instance-access point 1) 2))
           2.0)))
```

In more realistic uses of this mechanism, the calls to the low-level instance structure accessors would not actually appear textually in the source program, but rather would be generated by a meta-level analysis program run during the process of compiling the source program.

Funcallable Instances

Instances of classes which are themselves instances of **funcallable-standard-class** or one of its subclasses are called *funcallable instances*. Funcallable instances can only be created by **allocate-instance (funcallable-standard-class)**.

Like standard instances, funcallable instances have slots with the normal behavior. They differ from standard instances in that they can be used as functions as well; that is, they can be passed to **funcall** and **apply**, and they can be stored as the definition of a function name. Associated with each funcallable instance is the function which it runs when it is called. This function can be changed with **set-funcallable-instance-function**.

Example:

The following simple example shows the use of funcallable instances to create a simple, **defstruct**-like facility. (Funcallable instances are useful when a program needs to construct and maintain a set of functions and information about those functions. They make it possible to maintain both as the same object rather than two separate objects linked, for example, by hash tables.)

```
(defclass constructor ()
     ((name :initarg :name :accessor constructor-name)
      (fields :initarg :fields :accessor constructor-fields))
  (:metaclass funcallable-standard-class))

(defmethod initialize-instance :after ((c constructor) &key)
  (with-slots (name fields) c
    (set-funcallable-instance-function
      c
      #'(lambda ()
          (let ((new (make-array (1+ (length fields)))))
            (setf (aref new 0) name)
            new)))))

(setq c1 (make-instance 'constructor
                        :name 'position :fields '(x y)))
#<CONSTRUCTOR 262437>

(setq p1 (funcall c1))
#<ARRAY 3 263674>
```

Generic Function Invocation Protocol

Associated with each generic function is its discriminating function. Each time the generic function is called, the discriminating function is called to provide the behavior of the generic function. The discriminating function receives the full set of arguments received by the generic function. It must lookup and execute the appropriate methods, and return the appropriate values.

The discriminating function is computed by the highest layer of the generic function invocation protocol, **compute-discriminating-function**. Whenever a generic function metaobject is initialized, reinitialized, or a method is added or removed, the discriminating function is recomputed. The new discriminating function is then stored with **set-funcallable-instance-function**.

Discriminating functions call **compute-applicable-methods** and **compute-applicable-methods-using-classes** to compute the methods applicable to the generic functions arguments. Applicable methods are combined by **compute-effective-method** to produce an *effective method*. Provisions are made to allow memoization of the method applicability and effective methods computations. (See the description of **compute-discriminating-function** for details.)

The body of method definitions are processed by **make-method-lambda**. The result of this generic function is a lambda expression which is processed by either **compile** or the file compiler to produce a *method function*. The arguments received by the method function are controlled by the **call-method** forms appearing in the effective methods. By default, method functions accept two arguments: a list of arguments to the generic function, and a list of next methods. The list of next methods corresponds to the next methods argument to **call-method**. If **call-method** appears with additional arguments, these will be passed to the method functions as well; in these cases, **make-method-lambda** must have created the method lambdas to expect additional arguments.

Dependent Maintenance Protocol

It is convenient for portable metaobjects to be able to memoize information about other metaobjects, portable or otherwise. Because class and generic function metaobjects can be reinitialized, and generic function metaobjects can be modified by adding and removing methods, a means must be provided to update this memoized information.

The dependent maintenance protocol supports this by providing a way to register an object which should be notified whenever a class or generic function is modified. An object which has been registered this way is called a *dependent* of the class or generic function metaobject. The dependents of class and generic function metaobjects are maintained with **add-dependent** and **remove-dependent**. The dependents of a class or generic function metaobject can be accessed with **map-dependents**. Dependents are notified about a modification by calling **update-dependent**. (See the specification of **update-dependent** for detailed description of the circumstances under which it is called.)

To prevent conflicts between two portable programs, or between portable programs and the implementation, portable code must not register metaobjects themselves as dependents. Instead, portable programs which need to record a metaobject as a dependent, should encapsulate that metaobject in some other kind of object, and record that object as the dependent. The results are undefined if this restriction is violated.

Example:

This example shows a general facility for encapsulating metaobjects before recording them as dependents. The facility defines a basic kind of encapsulating object: an updater. Specializations of the basic class can be defined with appropriate special updating behavior. In this way, information about the updating required is associated with each updater rather than with the metaobject being updated.

Updaters are used to encapsulate any metaobject which requires updating when a given class or generic function is modified. The function **record-updater** is called to both create an updater and add it to the dependents of the class or generic function. Methods on the generic function **update-dependent**, specialized to the specific class of updater do the appropriate update work.

```
(defclass updater ()
     ((dependent :initarg :dependent :reader dependent)))

(defun record-updater (class dependee dependent &rest initargs)
  (let ((updater (apply #'make-instance class :dependent dependent
                                              initargs)))
    (add-dependent dependee updater)
    updater))
```

A **flush-cache-updater** simply flushes the cache of the dependent when it is updated.

```
(defclass flush-cache-updater (updater) ())

(defmethod update-dependent (dependee (updater flush-cache-updater)
                             &rest args)
```

```
  (declare (ignore args))
  (flush-cache (dependent updater)))
```

Chapter 6

Generic Functions and Methods

This chapter describes each of the functions and generic functions that make up the CLOS Metaobject Protocol. The descriptions appear in alphabetical order with the exception that all the reader generic functions for each kind of metaobject are grouped together. So, for example, **method-function** would be found with **method-qualifiers** and other method metaobject readers under "Readers for Method Metaobjects."

The description of functions follows the same formas used in the CLOS specification. The description of generic functions is similar to that in the CLOS specification, but some minor changes have been made in the way methods are presented.

The following is an example of the format for the syntax description of a generic function:

gf1

x y `&optional` *z* `&key` *k*

This description indicates that **gf1** is a generic function with two required parameters, *x* and *y*, an optional parameter *z* and a keyword parameter *k*.

The description of a generic function includes a description of its behavior. This provides the general behavior, or protocol of the generic function. All methods defined on the generic function, both portable and specified, must have behavior consistent with this description.

Every generic function described in this section is an instance of the class **standard-generic-function** and uses standard method combination.

The description of a generic function also includes descriptions of the specified methods for that generic function. In the description of these methods, a *method signature* is used to describe the parameters and parameter specializers of each method. The following is an example of the format for a method signature:

gf1 *Primary Method*

 (*x* *class*) *y* `&optional` *z* `&key` *k*

This signature indicates that this primary method on the generic function **gf1** has two required parameters, named *x* and *y*. In addition, there is an optional parameter *z* and a keyword parameter *k*. This signature also indicates that the method's parameter specializers are the classes named **class** and **t**.

The description of each method includes a description of the behavior particular to that method.

An abbreviated syntax is used when referring to a method defined elsewhere in the document. This abbreviated syntax includes the name of the generic function, the qualifiers, and the parameter specializers. A reference to the method with the signature shown above is written as: **gf1 (class t)**.

add-dependent *Generic Function*

Syntax

add-dependent

 metaobject dependent

Arguments

The *metaobject* argument is a class or generic function metaobject.

The *dependent* argument is an object.

Values

The value returned by this generic function is unspecified.

Purpose

This generic function adds *dependent* to the dependents of *metaobject*. If *dependent* is already in the set of dependents it is not added again (no error is signaled).

The generic function **map-dependents** can be called to access the set of dependents of a class or generic function. The generic function **remove-dependent** can be called to remove an object from the set of dependents of a class or generic function. The effect of calling **add-dependent** or **remove-dependent** while a call to **map-dependents** on the same class or generic function is in progress is unspecified.

The situations in which **add-dependent** is called are not specified.

Methods

add-dependent *Primary Method*

 (*class* `standard-class`) *dependent*

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

remove-dependent (standard-class t)
map-dependents (standard-class t)

add-dependent *Primary Method*
(*class* `funcallable-standard-class`) *dependent*

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

remove-dependent (funcallable-standard-class t)
map-dependents (funcallable-standard-class t)

add-dependent *Primary Method*
(*generic-function* `standard-generic-function`) *dependent*

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

remove-dependent (standard-generic-function t)
map-dependents (standard-generic-function t)

Remarks
See the "Dependent Maintenance Protocol" section for remarks about the use of this facility.

add-direct-method *Generic Function*

Syntax
add-direct-method
specializer method

Arguments
The *specializer* argument is a specializer metaobject.

The *method* argument is a method metaobject.

Values
The value returned by this generic function is unspecified.

Purpose

This generic function is called to maintain a set of backpointers from a specializer to the set of methods specialized to it. If *method* is already in the set, it is not added again (no error is signaled).

This set can be accessed as a list by calling the generic function **specializer-direct-methods**. Methods are removed from the set by **remove-direct-method**.

The generic function **add-direct-method** is called by **add-method** whenever a method is added to a generic function. It is called once for each of the specializers of the method. Note that in cases where a specializer appears more than once in the specializers of a method, this generic function will be called more than once with the same specializer as argument.

The results are undefined if the *specializer* argument is not one of the specializers of the *method* argument.

Methods

add-direct-method *Primary Method*

(*specializer* `class`)
(*method* `method`)

This method implements the behavior of the generic function for class specializers. No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

remove-direct-method (class method)
specializer-direct-generic-functions (class)
specializer-direct-methods (class)

add-direct-method *Primary Method*

(*specializer* `eql-specializer`)
(*method* `method`)

This method implements the behavior of the generic function for **eql** specializers. No behavior is specified for this method beyond that which is specified for the generic function.

add-direct-subclass *Generic Function*

Syntax

add-direct-subclass
superclass subclass

Arguments

The *superclass* argument is a class metaobject.

The *subclass* argument is a class metaobject.

Values

The value returned by this generic function is unspecified.

Purpose

This generic function is called to maintain a set of backpointers from a class to its direct subclasses. This generic function adds *subclass* to the set of direct subclasses of *superclass*.

When a class is initialized, this generic function is called once for each direct superclass of the class.

When a class is reinitialized, this generic function is called once for each added direct superclass of the class. The generic function **remove-direct-subclass** is called once for each deleted direct superclass of the class.

Methods

add-direct-subclass *Primary Method*

(*superclass* `class`)
(*subclass* `class`)

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

remove-direct-subclass (class class)
class-direct-subclasses (class)

add-method *Generic Function*

Syntax

add-method
generic-function method

Arguments

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

The *method* argument is a method metaobject.

Values

The *generic-function* argument is returned.

Purpose

This generic function associates an unattached method with a generic function.

An error is signaled if the lambda list of the method is not congruent with the lambda list of the generic function. An error is also signaled if the method is already associated with some other generic function.

If the given method agrees with an existing method of the generic function on parameter specializers and qualifiers, the existing method is removed by calling **remove-method** before the new method is added. See the section of the CLOS Specification called "Agreement on Parameter Specializers and Qualifiers" for a definition of agreement in this context.

Associating the method with the generic function then proceeds in four steps: (i) add *method* to the set returned by **generic-function-methods** and arrange for **method-generic-function** to return *generic-function*; (ii) call **add-direct-method** for each of the method's specializers; (iii) call **compute-discriminating-function** and install its result with **set-funcallable-instance-function**; and (iv) update the dependents of the generic function.

The generic function **add-method** can be called by the user or the implementation.

Methods

add-method *Primary Method*

(*generic-function* `standard-generic-function`)
(*method* `standard-method`)

No behavior is specified for this method beyond that which is specified for the generic function.

allocate-instance *Generic Function*

Syntax

allocate-instance
class `&rest` *initargs*

Arguments

The *class* argument is a class metaobject.

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

Values

The value returned is a newly allocated instance of *class*.

Purpose

This generic function is called to create a new, uninitialized instance of a class. The interpretation of the concept of an "uninitialized" instance depends on the class metaobject class.

Before allocating the new instance, **class-finalized-p** is called to see if *class* has been finalized. If it has not been finalized, **finalize-inheritance** is called before the new instance is allocated.

Methods

allocate-instance *Primary Method*
  (*class* `standard-class`) `&rest` *initargs*

This method allocates storage in the instance for each slot with allocation **:instance**. These slots are unbound. Slots with any other allocation are ignored by this method (no error is signaled).

allocate-instance *Primary Method*
  (*class* `funcallable-standard-class`) `&rest` *initargs*

This method allocates storage in the instance for each slot with allocation **:instance**. These slots are unbound. Slots with any other allocation are ignored by this method (no error is signaled).

The funcallable instance function of the instance is undefined—the results are undefined if the instance is applied to arguments before **set-funcallable-instance-function** has been used to set the funcallable instance function.

allocate-instance *Primary Method*
  (*class* `built-in-class`) `&rest` *initargs*

This method signals an error.

---

class-... *Generic Function*

The following generic functions are described together under "Readers for Class Metaobjects" (page 75): **class-default-initargs**, **class-direct-default-initargs**, **class-direct-slots**, **class-direct-subclasses**, **class-direct-superclasses**, **class-finalized-p**, **class-name**, **class-precedence-list**, **class-prototype** and **class-slots**.

---

compute-applicable-methods *Generic Function*

Syntax

compute-applicable-methods
  generic-function arguments

Arguments

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

The *arguments* argument is a list of objects.

Values

This generic function returns a possibly empty list of method metaobjects.

Purpose

This generic function determines the method applicability of a generic function given a list of required arguments. The returned list of method metaobjects is sorted by precedence order with the most specific method appearing first. If no methods are applicable to the supplied arguments the empty list is returned.

When a generic function is invoked, the discriminating function must determine the ordered list of methods applicable to the arguments. Depending on the generic function and the arguments, this is done in one of three ways: using a memoized value; calling **compute-applicable-methods-using-classes**; or calling **compute-applicable-methods**. (Refer to the description of **compute-discriminating-function** for the details of this process.)

The *arguments* argument is permitted to contain more elements than the generic function accepts required arguments; in these cases the extra arguments will be ignored. An error is signaled if *arguments* contains fewer elements than the generic function accepts required arguments.

The list returned by this generic function will not be mutated by the implementation. The results are undefined if a portable program mutates the list returned by this generic function.

Methods

compute-applicable-methods *Primary Method*
  (*generic-function* `standard-generic-function`)
  arguments

This method signals an error if any method of the generic function has a specializer which is neither a class metaobject nor an **eql** specializer metaobject.

Otherwise, this method computes the sorted list of applicable methods according to the rules described in the section of the CLOS Specification called "Method Selection and Combination."

This method can be overridden. Because of the consistency requirements between this generic function and **compute-applicable-methods-using-classes**, doing so may require also overriding **compute-applicable-methods-using-classes (standard-generic-function t)**.

compute-applicable-methods-using-classes *Generic Function*

Syntax

compute-applicable-methods-using-classes
  generic-function classes

Arguments

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

The *classes* argument is a list of class metaobjects.

Values

This generic function returns two values. The first is a possibly empty list of method metaobjects. The second is either true or false.

Purpose

This generic function is called to attempt to determine the method applicability of a generic function given only the classes of the required arguments.

If it is possible to completely determine the ordered list of applicable methods based only on the supplied classes, this generic function returns that list as its first value and true as its second value. The returned list of method metaobjects is sorted by precedence order, the most specific method coming first. If no methods are applicable to arguments with the specified classes, the empty list and true are returned.

If it is not possible to completely determine the ordered list of applicable methods based only on the supplied classes, this generic function returns an unspecified first value and false as its second value.

When a generic function is invoked, the discriminating function must determine the ordered list of methods applicable to the arguments. Depending on the generic function and the arguments, this is done in one of three ways: using a memoized value; calling **compute-applicable-methods-using-classes**; or calling **compute-applicable-methods**. (Refer to the description of **compute-discriminating-function** for the details of this process.)

The following consistency relationship between **compute-applicable-methods-using-classes** and **compute-applicable-methods** must be maintained: for any given generic function and set of arguments, if **compute-applicable-methods-using-classes** returns a second value of true, the first value must be equal to the value that would be returned by a corresponding call to **compute-applicable-methods**. The results are undefined if a portable method on either of these generic functions causes this consistency to be violated.

The list returned by this generic function will not be mutated by the implementation. The results are undefined if a portable program mutates the list returned by this generic function.

Methods

compute-applicable-methods-using-classes *Primary Method*
(*generic-function* `standard-generic-function`)
classes

If any method of the generic function has a specializer which is neither a class metaobject nor an **eql** specializer metaobject, this method signals an error.

In cases where the generic function has no methods with **eql** specializers, or has no methods with **eql** specializers that could be applicable to arguments of the supplied classes, this method returns the ordered list of applicable methods as its first value and true as its second value.

Otherwise this method returns an unspecified first value and false as its second value.

This method can be overridden. Because of the consistency requirements between this generic function and **compute-applicable-methods**, doing so may require also overriding **compute-applicable-methods (standard-generic-function t)**.

REMARKS

This generic function exists to allow user extensions which alter method lookup rules, but which base the new rules only on the classes of the required arguments, to take advantage of the class-based method lookup memoization found in many implementations. (There is of course no requirement for an implementation to provide this optimization.)

Such an extension can be implemented by two methods, one on this generic function and one on **compute-applicable-methods**. Whenever the user extension is in effect, the first method will return a second value of true. This should allow the implementation to absorb these cases into its own memoization scheme.

To get appropriate performance, other kinds of extensions may require methods on **compute-discriminating-function** which implement their own memoization scheme.

---

compute-class-precedence-list *Generic Function*

SYNTAX

compute-class-precedence-list
class

ARGUMENTS

The *class* argument is a class metaobject.

VALUES

The value returned by this generic function is a list of class metaobjects.

PURPOSE

This generic-function is called to determine the class precedence list of a class.

The result is a list which contains each of *class* and its superclasses once and only once. The first element of the list is *class* and the last element is the class named **t**.

All methods on this generic function must compute the class precedence list as a function of the ordered direct superclasses of the superclasses of *class*. The results are undefined if the rules used to compute the class precedence list depend on any other factors.

When a class is finalized, **finalize-inheritance** calls this generic function and associates the returned value with the class metaobject. The value can then be accessed by calling **class-precedence-list**.

The list returned by this generic function will not be mutated by the implementation. The results are undefined if a portable program mutates the list returned by this generic function.

METHODS

compute-class-precedence-list *Primary Method*
(*class* `class`)

This method computes the class precedence list according to the rules described in the section of the CLOS Specification called "Determining the Class Precedence List."

This method signals an error if *class* or any of its superclasses is a forward referenced class.

This method can be overridden.

---

compute-default-initargs *Generic Function*

SYNTAX

compute-default-initargs
class

ARGUMENTS

The *class* argument is a class metaobject.

VALUES

The value returned by this generic function is a list of canonicalized default initialization arguments.

PURPOSE

This generic-function is called to determine the default initialization arguments for a class.

The result is a list of canonicalized default initialization arguments, with no duplication among initialization argument names.

All methods on this generic function must compute the default initialization arguments as a function of only: (i) the class precedence list of *class*, and (ii) the direct default initialization arguments of each class in that list. The results are undefined if the rules used to compute the default initialization arguments depend on any other factors.

When a class is finalized, **finalize-inheritance** calls this generic function and associates the returned value with the class metaobject. The value can then be accessed by calling **class-default-initargs**.

The list returned by this generic function will not be mutated by the implementation. The results are undefined if a portable program mutates the list returned by this generic function.

Methods

compute-default-initargs *Primary Method*
    (*class* `standard-class`)

compute-default-initargs *Primary Method*
    (*class* `funcallable-standard-class`)

These methods compute the default initialization arguments according to the rules described in the section of the CLOS Specification called "Defaulting of Initialization Arguments."

These methods signal an error if *class* or any of its superclasses is a forward referenced class.

These methods can be overridden.

compute-discriminating-function *Generic Function*

Syntax

compute-discriminating-function
    generic-function

Arguments

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

Values

The value returned by this generic function is a function.

Purpose

This generic function is called to determine the discriminating function for a generic function. When a generic function is called, the *installed* discriminating function is called with the full set of arguments received by the generic function, and must implement the behavior of calling the generic function: determining the ordered set of applicable methods, determining the effective method, and running the effective method.

To determine the ordered set of applicable methods, the discriminating function first calls **compute-applicable-methods-using-classes**. If **compute-applicable-methods-using-classes** returns a second value of false, the discriminating function then calls **compute-applicable-methods**.

When **compute-applicable-methods-using-classes** returns a second value of true, the discriminating function is permitted to memoize the first returned value as follows. The discriminating function may reuse the list of applicable methods without calling **compute-applicable-methods-using-classes** again provided that:

(i) the generic function is being called again with required arguments which are instances of the same classes,

(ii) the generic function has not been reinitialized,

(iii) no method has been added to or removed from the generic function,

(iv) for all the specializers of all the generic function's methods which are classes, their class precedence lists have not changed and

(v) for any such memoized value, the class precedence list of the class of each of the required arguments has not changed.

Determination of the effective method is done by calling **compute-effective-method**. When the effective method is run, each method's function is called, and receives as arguments: (i) a list of the arguments to the generic function, and (ii) whatever other arguments are specified in the **call-method** form indicating that the method should be called. (See **make-method-lambda** for more information about how method functions are called.)

The generic function **compute-discriminating-function** is called, and its result installed, by **add-method**, **remove-method**, **initialize-instance** and **reinitialize-instance**.

Methods

compute-discriminating-function *Primary Method*

(*generic-function* `standard-generic-function`)

No behavior is specified for this method beyond that specified for the generic function.

This method can be overridden.

compute-effective-method *Generic Function*

Syntax

compute-effective-method

generic-function method-combination methods

Arguments

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

The *method-combination* argument is a method combination metaobject.

The *methods* argument is a list of method metaobjects.

Values

This generic function returns two values. The first is an effective method, the second is a list of effective method options.

Purpose

This generic function is called to determine the effective method from a sorted list of method metaobjects.

An effective method is a form that describes how the applicable methods are to be combined. Inside of effective method forms are **call-method** forms which indicate that a particular method is to be called. The arguments to the **call-method** form indicate exactly how the method function of the method should be called. (See **make-method-lambda** for more details about method functions.)

An effective method option has the same interpretation and syntax as either the **:arguments** or the **:generic-function** option in the long form of **define-method-combination**.

More information about the form and interpretation of effective methods and effective method options can be found under the description of the **define-method-combination** macro in the CLOS specification.

This generic function can be called by the user or the implementation. It is called by discriminating functions whenever a sorted list of applicable methods must be converted to an effective method.

METHODS

compute-effective-method *Primary Method*
    (*generic-function* `standard-generic-function`)
    method-combination
    methods

This method computes the effective method according to the rules of the method combination type implemented by *method-combination*.

This method can be overridden.

compute-effective-slot-definition *Generic Function*

SYNTAX

compute-effective-slot-definition
    class name direct-slot-definitions

ARGUMENTS

The *class* argument is a class metaobject.

The *name* argument is a slot name.

The *direct-slot-definitions* argument is an ordered list of direct slot definition metaobjects. The most specific direct slot definition metaobject appears first in the list.

VALUES

The value returned by this generic function is an effective slot definition metaobject.

PURPOSE

This generic function determines the effective slot definition for a slot in a class. It is called by **compute-slots** once for each slot accessible in instances of *class*.

This generic function uses the supplied list of direct slot definition metaobjects to compute the inheritance of slot properties for a single slot. The returned effective slot definition represents the result of computing the inheritance. The name of the new effective slot definition is the same as the name of the direct slot definitions supplied.

The class of the effective slot definition metaobject is determined by calling **effective-slot-definition-class**. The effective slot definition is then created by calling **make-instance**. The initialization arguments passed in this call to **make-instance** are used to initialize the new effective slot definition metaobject. See "Initialization of Slot Definition Metaobjects" for details.

Methods

compute-effective-slot-definition *Primary Method*

(*class* `standard-class`)
name
direct-slot-definitions

This method implements the inheritance and defaulting of slot options following the rules described in the "Inheritance of Slots and Options" section of the CLOS Specification.

This method can be extended, but the value returned by the extending method must be the value returned by this method.

compute-effective-slot-definition *Primary Method*

(*class* `funcallable-standard-class`)
name
direct-slot-definitions

This method implements the inheritance and defaulting of slot options following the rules described in the "Inheritance of Slots and Options" section of the CLOS Specification.

This method can be extended, but the value returned by the extending method must be the value returned by this method.

compute-slots *Generic Function*

Syntax

compute-slots
class

Arguments

The *class* argument is a class metaobject.

Values

The value returned is a set of effective slot definition metaobjects.

Purpose

This generic function computes a set of effective slot definition metaobjects for the class *class*. The result is a list of effective slot definition metaobjects: one for each slot that will be accessible in instances of *class*.

This generic function proceeds in 3 steps:

The first step collects the full set of direct slot definitions from the superclasses of *class*.

The direct slot definitions are then collected into individual lists, one list for each slot name associated with any of the direct slot definitions. The slot names are compared with **eql**. Each such list is then sorted into class precedence list order. Direct slot definitions coming from classes earlier in the class precedence list of *class* appear before those coming from classes later in the class precedence list. For each slot name, the generic function **compute-effective-slot-definition** is called to compute an effective slot definition. The result of **compute-slots** is a list of these effective slot definitions, in unspecified order.

In the final step, the location for each effective slot definition is set. This is done by specified around-methods; portable methods cannot take over this behavior. For more information on the slot definition locations, see the section "Instance Structure Protocol."

The list returned by this generic function will not be mutated by the implementation. The results are undefined if a portable program mutates the list returned by this generic function.

Methods

compute-slots *Primary Method*
(*class* `standard-class`)

This method implements the specified behavior of the generic function.

This method can be overridden.

compute-slots *Primary Method*
(*class* `funcallable-standard-class`)

This method implements the specified behavior of the generic function.

This method can be overridden.

compute-slots *Around-Method*
(*class* `standard-class`)

This method implements the specified behavior of computing and storing slot locations. This method cannot be overridden.

compute-slots *Around-Method*
(*class* `funcallable-standard-class`)

This method implements the specified behavior of computing and storing slot locations. This method cannot be overridden.

direct-slot-definition-class *Generic Function*

SYNTAX

direct-slot-definition-class
class `&rest` *initargs*

ARGUMENTS

The *class* argument is a class metaobject.

The *initargs* argument is a set of initialization arguments and values.

VALUES

The value returned is a subclass of the class **direct-slot-definition**.

PURPOSE

When a class is initialized, each of the canonicalized slot specifications must be converted to a direct slot definition metaobject. This generic function is called to determine the class of that direct slot definition metaobject.

The *initargs* argument is simply the canonicalized slot specification for the slot.

METHODS

direct-slot-definition-class *Primary Method*
(*class* `standard-class`)
`&rest` *initargs*

This method returns the class **standard-direct-slot-definition**.

This method can be overridden.

direct-slot-definition-class *Primary Method*
(*class* `funcallable-standard-class`)
`&rest` *initargs*

This method returns the class **standard-direct-slot-definition**.

This method can be overridden.

effective-slot-definition-class *Generic Function*

SYNTAX

effective-slot-definition-class
class `&rest` *initargs*

ARGUMENTS

The *class* argument is a class metaobject.

The *initargs* argument is a set of initialization arguments and values.

VALUES
The value returned is a subclass of the class **effective-slot-definition-class**.

PURPOSE
This generic function is called by **compute-effective-slot-definition** to determine the class of the resulting effective slot definition metaobject. The *initargs* argument is the set of initialization arguments and values that will be passed to **make-instance** when the effective slot definition metaobject is created.

METHODS
effective-slot-definition-class *Primary Method*
  (*class* `standard-class`)
  `&rest initargs`

This method returns the class **standard-effective-slot-definition**.
  This method can be overridden.

effective-slot-definition-class *Primary Method*
  (*class* `funcallable-standard-class`)
  `&rest initargs`

This method returns the class **standard-effective-slot-definition**.
  This method can be overridden.

---

ensure-class *Function*

SYNTAX
ensure-class
  name `&key &allow-other-keys`

ARGUMENTS
The *name* argument is a symbol.

Some of the keyword arguments accepted by this function are actually processed by **ensure-class-using-class**, others are processed during initialization of the class metaobject (as described in the section called “Initialization of Class Metaobjects”).

VALUES
The result is a class metaobject.

PURPOSE
This function is called to define or redefine a class with the specified name, and can be called by the user or the implementation. It is the functional equivalent of **defclass**, and is called by the expansion of the **defclass** macro.

The behavior of this function is actually implemented by the generic function **ensure-class-using-class**. When **ensure-class** is called, it immediately calls **ensure-class-using-class** and returns that result as its own.

The first argument to **ensure-class-using-class** is computed as follows:

- If *name* names a class (**find-class** returns a class when called with *name*) use that class.
- Otherwise use **nil**.

The second argument is *name*. The remaining arguments are the complete set of keyword arguments received by the **ensure-class** function.

---

ensure-class-using-class *Generic Function*

Syntax

ensure-class-using-class

```
      class name &key :direct-default-initargs :direct-slots
                      :direct-superclasses :name
                      :metaclass

                 &allow-other-keys
```

Arguments

The *class* argument is a class metaobject or **nil**.

The *name* argument is a class name.

The **:metaclass** argument is a class metaobject class or a class metaobject class name. If this argument is not supplied, it defaults to the class named **standard-class**. If a class name is supplied, it is interpreted as the class with that name. If a class name is supplied, but there is no such class, an error is signaled.

The **:direct-superclasses** argument is a list of which each element is a class metaobject or a class name. An error is signaled if this argument is not a proper list.

For the interpretation of additional keyword arguments, see "Initialization of Class Metaobjects" (page 57).

Values

The result is a class metaobject.

Purpose

This generic function is called to define or modify the definition of a named class. It is called by the **ensure-class** function. It can also be called directly.

The first step performed by this generic function is to compute the set of initialization arguments which will be used to create or reinitialize the named class. The initialization arguments are computed from the full set of keyword arguments received by this generic function as follows:

- The **:metaclass** argument is not included in the initialization arguments.

- If the **:direct-superclasses** argument was received by this generic function, it is converted into a list of class metaobjects. This conversion does not affect the structure of the supplied **:direct-superclasses** argument. For each element in the **:direct-superclasses** argument:

  - If the element is a class metaobject, that class metaobject is used.
  - If the element names a class, that class metaobject is used.
  - Otherwise an instance of the class **forward-referenced-class** is created and used. The proper name of the newly created forward referenced class metaobject is set to *name*.

- All other keyword arguments are included directly in the initialization arguments.

If the *class* argument is **nil**, a new class metaobject is created by calling the **make-instance** generic function with the value of the **:metaclass** argument as its first argument, and the previously computed initialization arguments. The proper name of the newly created class metaobject is set to *name*. The newly created class metaobject is returned.

If the *class* argument is a forward referenced class, **change-class** is called to change its class to the value specified by the **:metaclass** argument. The class metaobject is then reinitialized with the previously initialization arguments. (This is a documented violation of the general constraint that **change-class** not be used with class metaobjects.)

If the class of the *class* argument is not the same as the class specified by the **:metaclass** argument, an error is signaled.

Otherwise, the class metaobject *class* is redefined by calling the **reinitialize-instance** generic function with *class* and the initialization arguments. The *class* argument is then returned.

Methods

ensure-class-using-class *Primary Method*

```
(class class)
name
&key :metaclass
     :direct-superclasses

&allow-other-keys
```

This method implements the behavior of the generic function in the case where the *class* argument is a class.

This method can be overridden.

ensure-class-using-class *Primary Method*

```
(class forward-referenced-class)
name
&key :metaclass
     :direct-superclasses

&allow-other-keys
```

This method implements the behavior of the generic function in the case where the *class* argument is a forward referenced class.

ensure-class-using-class *Primary Method*

```
(class null)
name
&key :metaclass
     :direct-superclasses

&allow-other-keys
```

This method implements the behavior of the generic function in the case where the *class* argument is **nil**.

---

ensure-generic-function *Function*

Syntax

ensure-generic-function
function-name `&key &allow-other-keys`

Arguments

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

Some of the keyword arguments accepted by this function are actually processed by **ensure-generic-function-using-class**, others are processed during initialization of the generic function metaobject (as described in the section called "Initialization of Generic Function Metaobjects").

Values

The result is a generic function metaobject.

Purpose

This function is called to define a globally named generic function or to specify or modify options and declarations that pertain to a globally named generic function as a whole. It can be called by the user or the implementation.

It is the functional equivalent of **defgeneric**, and is called by the expansion of the **defgeneric** and **defmethod** macros.

The behavior of this function is actually implemented by the generic function **ensure-generic-function-using-class**. When **ensure-generic-function** is called, it immediately calls **ensure-generic-function-using-class** and returns that result as its own.

The first argument to **ensure-generic-function-using-class** is computed as follows:

- If *function-name* names a non-generic function, a macro, or a special form, an error is signaled.
- If *function-name* names a generic function, that generic function metaobject is used.
- Otherwise, **nil** is used.

The second argument is *function-name*. The remaining arguments are the complete set of keyword arguments received by **ensure-generic-function**.

ensure-generic-function-using-class *Generic Function*

Syntax

ensure-generic-function-using-class
generic-function
function-name
```
&key :argument-precedence-order :declarations
     :documentation :generic-function-class
     :lambda-list :method-class
     :method-combination :name

&allow-other-keys
```

Arguments

The *generic-function* argument is a generic function metaobject or **nil**.

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

The **:generic-function-class** argument is a class metaobject or a class name. If it is not supplied, it defaults to the class named **standard-generic-function**. If a class name is supplied, it is interpreted as the class with that name. If a class name is supplied, but there is no such class, an error is signaled.

For the interpretation of additional keyword arguments, see "Initialization of Generic Function Metaobjects" (page 61).

Values

The result is a generic function metaobject.

Purpose

The generic function **ensure-generic-function-using-class** is called to define or modify the definition of a globally named generic function. It is called by the **ensure-generic-function** function. It can also be called directly.

The first step performed by this generic function is to compute the set of initialization arguments which will be used to create or reinitialize the globally named generic function. These initialization arguments are computed from the full set of keyword arguments received by this generic function as follows:

- The **:generic-function-class** argument is not included in the initialization arguments.
- If the **:method-class** argument was received by this generic function, it is converted into a class metaobject. This is done by looking up the class name with **find-class**. If there is no such class, an error is signalled.
- All other keyword arguments are included directly in the initialization arguments.

If the *generic-function* argument is **nil**, an instance of the class specified by the **:generic-function-class** argument is created by calling **make-instance** with the previously computed initialization arguments. The function name *function-name* is set to name the generic function. The newly created generic function metaobject is returned.

If the class of the *generic-function* argument is not the same as the class specified by the **:generic-function-class** argument, an error is signaled.

Otherwise the generic function *generic-function* is redefined by calling the **reinitialize-instance** generic function with *generic-function* and the initialization arguments. The *generic-function* argument is then returned.

Methods

ensure-generic-function-using-class *Primary Method*

```
(generic-function generic-function)
function-name
&key :generic-function-class

&allow-other-keys
```

This method implements the behavior of the generic function in the case where *function-name* names an existing generic function.

This method can be overridden.

ensure-generic-function-using-class *Primary Method*
 (*generic-function* `null`)
 function-name
 `&key :generic-function-class`

 `&allow-other-keys`

This method implements the behavior of the generic function in the case where *function-name* names no function, generic function, macro or special form.

eql-specializer-object *Function*

SYNTAX
eql-specializer-object
 eql-specializer

ARGUMENTS
The *eql-specializer* argument is an **eql** specializer metaobject.

VALUES
The value returned by this function is an object.

PURPOSE
This function returns the object associated with *eql-specializer* during initialization. The value is guaranteed to be **eql** to the value originally passed to **intern-eql-specializer**, but it is not necessarily **eq** to that value.

This function signals an error if *eql-specializer* is not an **eql** specializer.

extract-lambda-list *Function*

SYNTAX
extract-lambda-list
 specialized-lambda-list

ARGUMENTS
The *specialized-lambda-list* argument is a specialized lambda list as accepted by **defmethod**.

VALUES
The result is an unspecialized lambda list.

Purpose
This function takes a specialized lambda list and returns the lambda list with the specializers removed. This is a non-destructive operation. Whether the result shares any structure with the argument is unspecified.

If the *specialized-lambda-list* argument does not have legal syntax, an error is signaled. This syntax checking does not check the syntax of the actual specializer names, only the syntax of the lambda list and where the specializers appear.

Examples
```
(extract-lambda-list '((p position)))             ==> (P)

(extract-lambda-list '((p position) x y))         ==> (P X Y)

(extract-lambda-list '(a (b (eql x)) c &rest i))  ==> (A B C &OPTIONAL I)
```

---

extract-specializer-names *Function*

Syntax
extract-specializer-names
specialized-lambda-list

Arguments
The *specialized-lambda-list* argument is a specialized lambda list as accepted by **defmethod**.

Values
The result is a list of specializer names.

Purpose
This function takes a specialized lambda list and returns its specializer names. This is a non-destructive operation. Whether the result shares structure with the argument is unspecified. The results are undefined if the result of this function is modified.

The result of this function will be a list with a number of elements equal to the number of required arguments in *specialized-lambda-list*. Specializers are defaulted to the symbol **t**.

If the *specialized-lambda-list* argument does not have legal syntax, an error is signaled. This syntax checking does not check the syntax of the actual specializer names, only the syntax of the lambda list and where the specializers appear.

Examples
```
(extract-specializer-names '((p position)))            ==> (POSITION)

(extract-specializer-names '((p position) x y))        ==> (POSITION T T)

(extract-specializer-names '(a (b (eql x)) c &rest i)) ==> (T (EQL X) T)
```

finalize-inheritance *Generic Function*

SYNTAX

finalize-inheritance
    class

ARGUMENTS

The *class* argument is a class metaobject.

VALUES

The value returned by this generic function is unspecified.

PURPOSE

This generic function is called to finalize a class metaobject. This is described in the Section named "Class Finalization Protocol."

After **finalize-inheritance** returns, the class metaobject is finalized and the result of calling **class-finalized-p** on the class metaobject will be true.

METHODS

finalize-inheritance *Primary Method*
    (*class* `standard-class`)

finalize-inheritance *Primary Method*
    (*class* `funcallable-standard-class`)

No behavior is specified for these methods beyond that which is specified for the generic function.

finalize-inheritance *Primary Method*
    (*class* `forward-referenced-class`)

This method signals an error.

find-method-combination *Generic Function*

SYNTAX

find-method-combination
    generic-function
    method-combination-type-name
    method-combination-options

ARGUMENTS

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

The *method-combination-type-name* argument is a symbol which names a type of method combination.

The *method-combination-options* argument is a list of arguments to the method combination type.

VALUES
The value returned by this generic function is a method combination metaobject.

PURPOSE
This generic function is called to determine the method combination object used by a generic function.

REMARKS
Further details of method combination metaobjects are not specified.

funcallable-standard-instance-access *Function*

SYNTAX
funcallable-standard-instance-access
instance location

ARGUMENTS
The *instance* argument is an object.

The *location* argument is a slot location.

VALUES
The result of this function is an object.

PURPOSE
This function is called to provide direct access to a slot in an instance. By usurping the normal slot lookup protocol, this function is intended to provide highly optimized access to the slots associated with an instance.

The following restrictions apply to the use of this function:

- The *instance* argument must be a funcallable instance (it must have been returned by **allocate-instance (funcallable-standard-class)**).
- The *instance* argument cannot be an non-updated obsolete instance.
- The *location* argument must be a location of one of the directly accessible slots of the instance's class.
- The slot must be bound.

The results are undefined if any of these restrictions are not met.

generic-function-... *Generic Function*

The following generic functions are described together under "Readers for Generic Function Metaobjects" (page 79): **generic-function-argument-precedence-order**, **generic-function-declarations**, **generic-function-lambda-list**, **generic-function-method-class**, **generic-function-method-combination**, **generic-function-methods** and **generic-function-name**.

Initialization of Class Metaobjects

A class metaobject can be created by calling **make-instance**. The initialization arguments establish the definition of the class. A class metaobject can be redefined by calling **reinitialize-instance**. Some classes of class metaobject do not support redefinition; in these cases, **reinitialize-instance** signals an error.

Initialization of a class metaobject must be done by calling **make-instance** and allowing it to call **initialize-instance**. Reinitialization of a class metaobject must be done by calling **reinitialize-instance**. Portable programs must not call **initialize-instance** directly to initialize a class metaobject. Portable programs must not call **shared-initialize** directly to initialize or reinitialize a class metaobject. Portable programs must not call **change-class** to change the class of any class metaobject or to turn a non-class object into a class metaobject.

Since metaobject classes may not be redefined, no behavior is specified for the result of calls to **update-instance-for-redefined-class** on class metaobjects. Since the class of class metaobjects may not be changed, no behavior is specified for the result of calls to **update-instance-for-different-class** on class metaobjects.

During initialization or reinitialization, each initialization argument is checked for errors and then associated with the class metaobject. The value can then be accessed by calling the appropriate accessor as shown in Table 6.1.

This section begins with a description of the error checking and processing of each initialization argument. This is followed by a table showing the generic functions that can be used to access the stored initialization arguments. Initialization behavior specific to the different specified class metaobject classes comes next. The section ends with a set of restrictions on portable methods affecting class metaobject initialization and reinitialization.

In these descriptions, the phrase "this argument defaults to *value*" means that when that initialization argument is not supplied, initialization or reinitialization is performed as if *value* had been supplied. For some initialization arguments this could be done by the use of default initialization arguments, but whether it is done this way is not specified. Implementations are free to define default initialization arguments for specified class metaobject classes. Portable programs are free to define default initialization arguments for portable subclasses of the class **class**.

Unless there is a specific note to the contrary, then during reinitialization, if an initialization argument is not supplied, the previously stored value is left unchanged.

- The **:direct-default-initargs** argument is a list of canonicalized default initialization arguments.

  An error is signaled if this value is not a proper list, or if any element of the list is not a canonicalized default initialization argument.

  If the class metaobject is being initialized, this argument defaults to the empty list.

- The **:direct-slots** argument is a list of canonicalized slot specifications.

  An error is signaled if this value is not a proper list or if any element of the list is not a canonicalized slot specification.

  After error checking, this value is converted to a list of direct slot definition metaobjects before it is associated with the class metaobject. Conversion of each canonicalized slot specification to a direct slot definition metaobject is a two-step process. First, the generic function **direct-slot-definition-class** is called with the class metaobject and the canonicalized slot specification to determine the class of the new direct slot definition metaobject; this permits both the class metaobject and the canonicalized slot specification to control the resulting direct slot definition metaobject class. Second, **make-instance** is applied to the direct slot definition metaobject class and the canonicalized slot specification. This conversion could be implemented as shown in the following code:

  ```
  (defun convert-to-direct-slot-definition (class canonicalized-slot)
    (apply #'make-instance
           (apply #'direct-slot-definition-class
                  class canonicalized-slot)
           canonicalized-slot))
  ```

  If the class metaobject is being initialized, this argument defaults to the empty list.

  Once the direct slot definition metaobjects have been created, the specified reader and writer methods are created. The generic functions **reader-method-class** and **writer-method-class** are called to determine the classes of the method metaobjects created.

- The **:direct-superclasses** argument is a list of class metaobjects. Classes which do not support multiple inheritance signal an error if the list contains more than one element.

  An error is signaled if this value is not a proper list or if **validate-superclass** applied to *class* and any element of this list returns false.

  When the class metaobject is being initialized, and this argument is either not supplied or is the empty list, this argument defaults as follows: if the class is an instance of **standard-class** or one of its subclasses the default value is a list of the class **standard-object**; if the class is an instance of **funcallable-standard-class** or one of its subclasses the default value is list of the class **funcallable-standard-object**.

  After any defaulting of the value, the generic function **add-direct-subclass** is called once for each element of the list.

  When the class metaobject is being reinitialized and this argument is supplied, the generic function **remove-direct-subclass** is called once for each class metaobject in the previously stored value but not in the new value; the generic function **add-direct-subclass** is called once for each class metaobject in the new value but not in the previously stored value.

- The **:documentation** argument is a string or **nil**.

  An error is signaled if this value is not a string or **nil**.

  If the class metaobject is being initialized, this argument defaults to **nil**.

- The **:name** argument is an object.

  If the class is being initialized, this argument defaults to **nil**.

After the processing and defaulting of initialization arguments described above, the value of each initialization argument is associated with the class metaobject. These values can then be accessed by calling the corresponding generic function. The correspondences are as follows:

Initialization Argument	**Generic Function**
:direct-default-initargs	**class-direct-default-initargs**
:direct-slots	**class-direct-slots**
:direct-superclasses	**class-direct-superclasses**
:documentation	**documentation**
:name	**class-name**

Table 6.1 Initialization arguments and accessors for class metaobjects.

Instances of the class **standard-class** support multiple inheritance and reinitialization. Instances of the class **funcallable-standard-class** support multiple inheritance and reinitialization. For forward referenced classes, all of the initialization arguments default to **nil**.

Since built-in classes cannot be created or reinitialized by the user, an error is signaled if **initialize-instance** or **reinitialize-instance** are called to initialize or reinitialize a derived instance of the class **built-in-class**.

Methods

It is not specified which methods provide the initialization and reinitialization behavior described above. Instead, the information needed to allow portable programs to specialize this behavior is presented as a set of restrictions on the methods a portable program can define. The model is that portable initialization methods have access to the class metaobject when either all or none of the specified initialization has taken effect.

These restrictions govern the methods that a portable program can define on the generic functions **initialize-instance**, **reinitialize-instance**, and **shared-initialize**. These restrictions apply only to methods on these generic functions for which the first specializer is a subclass of the class **class**. Other portable methods on these generic functions are not affected by these restrictions.

- Portable programs must not define methods on **shared-initialize**.
- For **initialize-instance** and **reinitialize-instance**:
  - Portable programs must not define primary methods.
  - Portable programs may define around-methods, but these must be extending, not overriding methods.
  - Portable before-methods must assume that when they are run, none of the initialization behavior described above has been completed.
  - Portable after-methods must assume that when they are run, all of the initialization behavior described above has been completed.

  The results are undefined if any of these restrictions are violated.

Initialization of Generic Function Metaobjects

A generic function metaobject can be created by calling **make-instance**. The initialization arguments establish the definition of the generic function. A generic function metaobject can be redefined by calling **reinitialize-instance**. Some classes of generic function metaobject do not support redefinition; in these cases, **reinitialize-instance** signals an error.

Initialization of a generic function metaobject must be done by calling **make-instance** and allowing it to call **initialize-instance**. Reinitialization of a generic-function metaobject must be done by calling **reinitialize-instance**. Portable programs must not call **initialize-instance** directly to initialize a generic function metaobject. Portable programs must not call **shared-initialize** directly to initialize or reinitialize a generic function metaobject. Portable programs must not call **change-class** to change the class of any generic function metaobject or to turn a non-generic-function object into a generic function metaobject.

Since metaobject classes may not be redefined, no behavior is specified for the result of calls to **update-instance-for-redefined-class** on generic function metaobjects. Since the class of a generic function metaobject may not be changed, no behavior is specified for the results of calls to **update-instance-for-different-class** on generic function metaobjects.

During initialization or reinitialization, each initialization argument is checked for errors and then associated with the generic function metaobject. The value can then be accessed by calling the appropriate accessor as shown in Table 6.2.

This section begins with a description of the error checking and processing of each initialization argument. This is followed by a table showing the generic functions that can be used to access the stored initialization arguments. The section ends with a set of restrictions on portable methods affecting generic function metaobject initialization and reinitialization.

In these descriptions, the phrase "this argument defaults to *value*" means that when that initialization argument is not supplied, initialization or reinitialization is performed as if *value* had been supplied. For some initialization arguments this could be done by the use of default initialization arguments, but whether it is done this way is not specified. Implementations are free to define default initialization arguments for specified generic function metaobject classes. Portable programs are free to define default initialization arguments for portable subclasses of the class **generic-function**.

Unless there is a specific note to the contrary, then during reinitialization, if an initialization argument is not supplied, the previously stored value is left unchanged.

- The **:argument-precedence-order** argument is a list of symbols.

  An error is signaled if this argument appears but the **:lambda-list** argument does not appear. An error is signaled if this value is not a proper list or if this value is not a permutation of the symbols from the required arguments part of the **:lambda-list** initialization argument.

When the generic function is being initialized or reinitialized, and this argument is not supplied, but the **:lambda-list** argument is supplied, this value defaults to the symbols from the required arguments part of the **:lambda-list** argument, in the order they appear in that argument. If neither argument is supplied, neither are initialized (see the description of **:lambda-list**.)

- The **:declarations** argument is a list of declarations.

  An error is signaled if this value is not a proper list or if each of its elements is not a legal declaration.

  When the generic function is being initialized, and this argument is not supplied, it defaults to the empty list.

- The **:documentation** argument is a string or **nil**.

  An error is signaled if this value is not a string or **nil**.

  If the generic function is being initialized, this argument defaults to **nil**.

- The **:lambda-list** argument is a lambda list.

  An error is signaled if this value is not a proper generic function lambda list.

  When the generic function is being initialized, and this argument is not supplied, the generic function's lambda list is not initialized. The lambda list will be initialized later, either when the first method is added to the generic function, or a later reinitialization of the generic function.

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

- The **:method-class** argument is a class metaobject.

  An error is signaled if this value is not a subclass of the class **method**.

  When the generic function is being initialized, and this argument is not supplied, it defaults to the class **standard-method**.

- The **:name** argument is an object.

  If the generic function is being initialized, this argument defaults to **nil**.

After the processing and defaulting of initialization arguments described above, the value of each initialization argument is associated with the generic function metaobject. These values can then be accessed by calling the corresponding generic function. The correspondences are as follows:

Initialization Argument	**Generic Function**
:argument-precedence-order	**generic-function-argument-precedence-order**
:declarations	**generic-function-declarations**
:documentation	**documentation**
:lambda-list	**generic-function-lambda-list**
:method-combination	**generic-function-method-combination**
:method-class	**generic-function-method-class**
:name	**generic-function-name**

Table 6.2 Initialization arguments and accessors for generic function metaobjects.

METHODS
It is not specified which methods provide the initialization and reinitialization behavior described above. Instead, the information needed to allow portable programs to specialize this behavior is presented as a set of restrictions on the methods a portable program can define. The model is that portable initialization methods have access to the generic function metaobject when either all or none of the specified initialization has taken effect.

These restrictions govern the methods that a portable program can define on the generic functions **initialize-instance**, **reinitialize-instance**, and **shared-initialize**. These restrictions apply only to methods on these generic functions for which the first specializer is a subclass of the class **generic-function**. Other portable methods on these generic functions are not affected by these restrictions.

- Portable programs must not define methods on **shared-initialize**.
- For **initialize-instance** and **reinitialize-instance**:
  - Portable programs must not define primary methods.
  - Portable programs may define around-methods, but these must be extending, not overriding methods.
  - Portable before-methods must assume that when they are run, none of the initialization behavior described above has been completed.
  - Portable after-methods must assume that when they are run, all of the initialization behavior described above has been completed.

The results are undefined if any of these restrictions are violated.

Initialization of Method Metaobjects

A method metaobject can be created by calling **make-instance**. The initialization arguments establish the definition of the method. A method metaobject cannot be redefined; calling **reinitialize-instance** signals an error.

Initialization of a method metaobject must be done by calling **make-instance** and allowing it to call **initialize-instance**. Portable programs must not call **initialize-instance** directly to initialize a method metaoject. Portable programs must not call **shared-initialize** directly to initialize a method metaobject. Portable programs must not call **change-class** to change the class of any method metaobject or to turn a non-method object into a method metaobject.

Since metaobject classes may not be redefined, no behavior is specified for the result of calls to **update-instance-for-redefined-class** on method metaobjects. Since the class of a method metaobject cannot be changed, no behavior is specified for the result of calls to **update-instance-for-different-class** on method metaobjects.

During initialization, each initialization argument is checked for errors and then associated with the method metaobject. The value can then be accessed by calling the appropriate accessor as shown in Table 6.3.

This section begins with a description of the error checking and processing of each initialization argument. This is followed by a table showing the generic functions that can be used to access the stored initialization arguments. The section ends with a set of restrictions on portable methods affecting method metaobject initialization.

In these descriptions, the phrase "this argument defaults to *value*" means that when that initialization argument is not supplied, initialization is performed as if *value* had been supplied. For some initialization arguments this could be done by the use of default initialization arguments, but whether it is done this way is not specified. Implementations are free to define default initialization arguments for specified method metaobject classes. Portable programs are free to define default initialization arguments for portable subclasses of the class **method**.

- The **:qualifiers** argument is a list of method qualifiers. An error is signaled if this value is not a proper list, or if any element of the list is not a non-null atom. This argument defaults to the empty list.

- The **:lambda-list** argument is the unspecialized lambda list of the method. An error is signaled if this value is not a proper lambda list. If this value is not supplied, an error is signaled.

- The **:specializers** argument is a list of the specializer metaobjects for the method. An error is signaled if this value is not a proper list, or if the length of the list differs from the number of required arguments in the **:lambda-list** argument, or if any element of the list is not a specializer metaobject. If this value is not supplied, an error is signaled.

- The **:function** argument is a method function. It must be compatible with the methods on **compute-effective-method** defined for this class of method and generic function with which it will be used. That is, it must accept the same number of arguments as all uses of **call-method** that will call it supply. (See **compute-effective-method** for more information.) An error is signaled if this argument is not supplied.
- When the method being initialized is an instance of a subclass of **standard-accessor-method**, the **:slot-definition** initialization argument must be provided. Its value is the direct slot definition metaobject which defines this accessor method. An error is signaled if the value is not an instance of a subclass of **direct-slot-definition**.
- The **:documentation** argument is a string or **nil**. An error is signaled if this value is not a string or **nil**. This argument defaults to **nil**.

After the processing and defaulting of initialization arguments described above, the value of each initialization argument is associated with the method metaobject. These values can then be accessed by calling the corresponding generic function. The correspondences are as follows:

Initialization Argument	**Generic Function**
:qualifiers	**method-qualifiers**
:lambda-list	**method-lambda-list**
:specializers	**method-specializers**
:function	**method-function**
:slot-definition	**accessor-method-slot-definition**
:documentation	**documentation**

Table 6.3 Initialization arguments and accessors for method metaobjects.

Methods

It is not specified which methods provide the initialization behavior described above. Instead, the information needed to allow portable programs to specialize this behavior is presented in as a set of restrictions on the methods a portable program can define. The model is that portable initialization methods have access to the method metaobject when either all or none of the specified initialization has taken effect.

These restrictions govern the methods that a portable program can define on the generic functions **initialize-instance**, **reinitialize-instance**, and **shared-initialize**. These restrictions apply only to methods on these generic functions for which the first specializer is a subclass of the class **method**. Other portable methods on these generic functions are not affected by these restrictions.

- Portable programs must not define methods on **shared-initialize** or **reinitialize-instance**.
- For **initialize-instance**:
  - Portable programs must not define primary methods.
  - Portable programs may define around-methods, but these must be extending, not overriding methods.
  - Portable before-methods must assume that when they are run, none of the initialization behavior described above has been completed.
  - Portable after-methods must assume that when they are run, all of the initialization behavior described above has been completed.

The results are undefined if any of these restrictions are violated.

Initialization of Slot Definition Metaobjects

A slot definition metaobject can be created by calling **make-instance**. The initialization arguments establish the definition of the slot definition. A slot definition metaobject cannot be redefined; calling **reinitialize-instance** signals an error.

Initialization of a slot definition metaobject must be done by calling **make-instance** and allowing it to call **initialize-instance**. Portable programs must not call **initialize-instance** directly to initialize a slot definition metaobject. Portable programs must not call **shared-initialize** directly to initialize a slot definition metaobject. Portable programs must not call **change-class** to change the class of any slot definition metaobject or to turn a non-slot-definition object into a slot definition metaobject.

Since metaobject classes may not be redefined, no behavior is specified for the result of calls to **update-instance-for-redefined-class** on slot definition metaobjects. Since the class of a slot definition metaobject cannot be changed, no behavior is specified for the result of calls to **update-instance-for-different-class** on slot definition metaobjects.

During initialization, each initialization argument is checked for errors and then associated with the slot definition metaobject. The value can then be accessed by calling the appropriate accessor as shown in Table 6.4.

This section begins with a description of the error checking and processing of each initialization argument. This is followed by a table showing the generic functions that can be used to access the stored initialization arguments.

In these descriptions, the phrase "this argument defaults to *value*" means that when that initialization argument is not supplied, initialization is performed as if *value* had been supplied. For some initialization arguments this could be done by the use of default initialization arguments, but whether it is done this way is not specified. Implementations are free to define default initialization arguments for specified slot definition metaobject classes. Portable programs are free to define default initialization arguments for portable subclasses of the class **slot-definition**.

- The **:name** argument is a slot name. An error is signaled if this argument is not a symbol which can be used as a variable name. An error is signaled if this argument is not supplied.

- The **:initform** argument is a form. The **:initform** argument defaults to **nil**. An error is signaled if the **:initform** argument is supplied, but the **:initfunction** argument is not supplied.

- The **:initfunction** argument is a function of zero arguments which, when called, evaluates the **:initform** in the appropriate lexical environment. The **:initfunction** argument defaults to false. An error is signaled if the **:initfunction** argument is supplied, but the **:initform** argument is not supplied.

- The **:type** argument is a type specifier name. An error is signaled otherwise. The **:type** argument defaults to the symbol **t**.
- The **:allocation** argument is a symbol. An error is signaled otherwise. The **:allocation** argument defaults to the symbol **:instance**.
- The **:initargs** argument is a list of symbols. An error is signaled if this argument is not a proper list, or if any element of this list is not a symbol. The **:initargs** argument defaults to the empty list.
- The **:readers** argument is a list of function names. An error is signaled if it is not a proper list, or if any element is not a valid function name. It defaults to the empty list. An error is signaled if this argument is supplied and the metaobject is not a direct slot definition.
- The **:writers** argument is a list of function names. An error is signaled if it is not a proper list, or if any element is not a valid function name. It defaults to the empty list. An error is signaled if this argument is supplied and the metaobject is not a direct slot definition.
- The **:documentation** argument is a string or **nil**. An error is signaled otherwise. The **:documentation** argument defaults to **nil**.

After the processing and defaulting of initialization arguments described above, the value of each initialization argument is associated with the slot definition metaobject. These values can then be accessed by calling the corresponding generic function. The correspondences are as follows:

Initialization Argument	Generic Function
:name	**slot-definition-name**
:initform	**slot-definition-initform**
:initfunction	**slot-definition-initfunction**
:type	**slot-definition-type**
:allocation	**slot-definition-allocation**
:initargs	**slot-definition-initargs**
:readers	**slot-definition-readers**
:writers	**slot-definition-writers**
:documentation	**documentation**

Table 6.4 Initialization arguments and accessors for slot definition metaobjects.

Methods

It is not specified which methods provide the initialization and reinitialization behavior described above. Instead, the information needed to allow portable programs to specialize this behavior is presented as a set of restrictions on the methods a portable program can define. The model is that portable initialization methods have access to the slot definition metaobject when either all or none of the specified initialization has taken effect.

These restrictions govern the methods that a portable program can define on the generic functions **initialize-instance**, **reinitialize-instance**, and **shared-initialize**. These restrictions apply only to methods on these generic functions for which the first specializer is a subclass of the class **slot-definition**. Other portable methods on these generic functions are not affected by these restrictions.

- Portable programs must not define methods on **shared-initialize** or **reinitialize-instance**.
- For **initialize-instance**:
  - Portable programs must not define primary methods.
  - Portable programs may define around-methods, but these must be extending, not overriding methods.
  - Portable before-methods must assume that when they are run, none of the initialization behavior described above has been completed.
  - Portable after-methods must assume that when they are run, all of the initialization behavior described above has been completed.

The results are undefined if any of these restrictions are violated.

intern-eql-specializer *Function*

Syntax

intern-eql-specializer
    object

Arguments

The *object* argument is any Lisp object.

Values

The result is the **eql** specializer metaobject for *object*.

Purpose

This function returns the unique **eql** specializer metaobject for *object*, creating one if necessary. Two calls to **intern-eql-specializer** with **eql** arguments will return the same (i.e., **eq**) value.

Remarks

The result of calling **eql-specializer-object** on the result of a call to **intern-eql-specializer** is only guaranteed to be **eql** to the original *object* argument, not necessarily **eq**.

make-instance *Generic Function*

Syntax

make-instance
    class `&rest` *initargs*

Arguments

The *class* argument is a class metaobject or a class name.

The *initargs* argument is a list of alternating initialization argument names and values.

Values

The result is a newly allocated and initialized instance of *class*.

Purpose

The generic function **make-instance** creates and returns a new instance of the given class. Its behavior and use is described in the CLOS specification.

Methods

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

This method simply invokes **make-instance** recursively on the arguments `(find-class` *class*`)` and *initargs*.

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

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

These methods implement the behavior of **make-instance** described in the CLOS specification section named "Object Creation and Initialization."

make-method-lambda *Generic Function*

Syntax

make-method-lambda
generic-function method lambda-expression environment

Arguments

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

The *method* argument is a (possibly uninitialized) method metaobject.

The *lambda-expression* argument is a lambda expression.

The *environment* argument is the same as the **&environment** argument to macro expansion functions.

Values

This generic function returns two values. The first is a lambda expression, the second is a list of initialization arguments and values.

Purpose

This generic function is called to produce a lambda expression which can itself be used to produce a method function for a method and generic function with the specified classes. The generic function and method the method function will be used with are not required to be the given ones. Moreover, the method metaobject may be uninitialized.

Either the function **compile**, the special form **function** or the function **coerce** must be used to convert the lambda expression a method function. The method function itself can be applied to arguments with **apply** or **funcall**.

When a method is actually called by an effective method, its first argument will be a list of the arguments to the generic function. Its remaining arguments will be all but the first argument passed to **call-method**. By default, all method functions must accept two arguments: the list of arguments to the generic function and the list of next methods.

For a given generic function and method class, the applicable methods on **make-method-lambda** and **compute-effective-method** must be consistent in the following way: each use of **call-method** returned by the method on **compute-effective-method** must have the same number of arguments, and the method lambda returned by the method on **make-method-lambda** must accept a corresponding number of arguments.

Note that the system-supplied implementation of **call-next-method** is not required to handle extra arguments to the method function. Users who define additional arguments to the method function must either redefine or forego **call-next-method**. (See the example below.)

When the method metaobject is created with **make-instance**, the method function must be the value of the **:function** initialization argument. The additional initialization arguments, returned as the second value of this generic function, must also be passed in this call to **make-instance**.

METHODS

make-method-lambda *Primary Method*

(*generic-function* `standard-generic-function`)
(*method* `standard-method`)
lambda-expression
environment

This method returns a method lambda which accepts two arguments, the list of arguments to the generic function, and the list of next methods. What initialization arguments may be returned in the second value are unspecified.

This method can be overridden.

Example:

This example shows how to define a kind of method which, from within the body of the method, has access to the actual method metaobject for the method. This simplified code overrides whatever method combination is specified for the generic function, implementing a simple method combination supporting only primary methods, **call-next-method** and **next-method-p**. (In addition, its a simplified version of **call-next-method** which does no error checking.)

Notice that the extra lexical function bindings get wrapped around the body before **call-next-method** is called. In this way, the user's definition of **call-next-method** and **next-method-p** are sure to override the system's definitions.

```
(defclass my-generic-function (standard-generic-function)
     ()
  (:default-initargs :method-class (find-class 'my-method)))

(defclass my-method (standard-method) ())

(defmethod make-method-lambda ((gf my-generic-function)
                               (method my-method)
                               lambda-expression
                               environment)
  (declare (ignore environment))
  `(lambda (args next-methods this-method)
     (,(call-next-method gf method
         `(lambda ,(cadr lambda-expression)
            (flet ((this-method () this-method)
                   (call-next-method (&rest cnm-args)
                     (funcall (method-function (car next-methods))
                              (or cnm-args args)
                              (cdr next-methods)
                              (car next-methods)))
                   (next-method-p ()
                     (not (null next-methods))))
              ,@(cddr lambda-expression)))
          environment)
       args next-methods)))

(defmethod compute-effective-method ((gf my-generic-function)
                                     method-combination
                                     methods)
  `(call-method ,(car methods) ,(cdr methods) ,(car methods)))
```

map-dependents *Generic Function*

SYNTAX

map-dependents
    metaobject function

ARGUMENTS

The *metaobject* argument is a class or generic function metaobject.

The *function* argument is a function which accepts one argument.

VALUES

The value returned is unspecified.

PURPOSE

This generic function applies *function* to each of the dependents of *metaobject*. The order in which the dependents are processed is not specified, but *function* is applied to each dependent once and only once. If, during the mapping, **add-dependent** or **remove-dependent** is called to alter the dependents of *metaobject*, it is not specified whether the newly added or removed dependent will have *function* applied to it.

METHODS

map-dependents *Primary Method*
(*metaobject* `standard-class`) *function*

This method has no specified behavior beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

add-dependent (standard-class t)
remove-dependent (standard-class t)

map-dependents *Primary Method*
(*metaobject* `funcallable-standard-class`) *function*

This method has no specified behavior beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

add-dependent (funcallable-standard-class t)
remove-dependent (funcallable-standard-class t)

map-dependents *Primary Method*
(*metaobject* `standard-generic-function`) *function*

This method has no specified behavior beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

add-dependent (standard-generic-function t)
remove-dependent (standard-generic-function t)

REMARKS

See the "Dependent Maintenance Protocol" section for remarks about the use of this facility.

method-... *Generic Function*

The following generic functions are described together under "Readers for Method Metaobjects" (page 81): **method-function**, **method-generic-function**, **method-lambda-list**, **method-specializers**, **method-qualifiers** and **accessor-method-slot-definition**.

Readers for Class Metaobjects

In this and the immediately following sections, the "reader" generic functions which simply return information associated with a particular kind of metaobject are presented together. General information is presented first, followed by a description of the purpose of each, and ending with the specified methods for these generic functions.

The reader generic functions which simply return information associated with class metaobjects are presented together in this section.

Each of the reader generic functions for class metaobjects has the same syntax, accepting one required argument called *class*, which must be an class metaobject; otherwise, an error is signaled. An error is also signaled if the class metaobject has not been initialized.

These generic functions can be called by the user or the implementation.

For any of these generic functions which returns a list, such lists will not be mutated by the implementation. The results are undefined if a portable program allows such a list to be mutated.

class-default-initargs *Generic Function*
class

Returns a list of the default initialization arguments for *class*. Each element of this list is a canonicalized default initialization argument. The empty list is returned if *class* has no default initialization arguments.

During finalization **finalize-inheritance** calls **compute-default-initargs** to compute the default initialization arguments for the class. That value is associated with the class metaobject and is returned by **class-default-initargs**.

This generic function signals an error if *class* has not been finalized.

class-direct-default-initargs *Generic Function*
class

Returns a list of the direct default initialization arguments for *class*. Each element of this list is a canonicalized default initialization argument. The empty list is returned if *class* has no direct default initialization arguments. This is the defaulted value of the **:direct-default-initargs** initialization argument that was associated with the class during initialization or reinitialization.

class-direct-slots *Generic Function*
class

Returns a set of the direct slots of *class*. The elements of this set are direct slot definition metaobjects. If the class has no direct slots, the empty set is returned. This is the defaulted value of the **:direct-slots** initialization argument that was associated with the class during initialization and reinitialization.

class-direct-subclasses *Generic Function*
class

Returns a set of the direct subclasses of *class*. The elements of this set are class metaobjects that all mention this class among their direct superclasses. The empty set is returned if *class* has no direct subclasses. This value is maintained by the generic functions **add-direct-subclass** and **remove-direct-subclass**.

class-direct-superclasses *Generic Function*
class

Returns a list of the direct superclasses of *class*. The elements of this list are class metaobjects. The empty list is returned if *class* has no direct superclasses. This is the defaulted value of the **:direct-superclasses** initialization argument that was associated with the class during initialization or reinitialization.

class-finalized-p *Generic Function*
class

Returns true if *class* has been finalized. Returns false otherwise. Also returns false if the class has not been initialized.

class-name *Generic Function*
class

Returns the name of *class*. This value can be any Lisp object, but is usually a symbol, or **nil** if the class has no name. This is the defaulted value of the **:name** initialization argument that was associated with the class during initialization or reinitialization. (Also see **(setf class-name)**.)

class-precedence-list *Generic Function*
class

Returns the class precedence list of *class*. The elements of this list are class metaobjects.

During class finalization **finalize-inheritance** calls **compute-class-precedence-list** to compute the class precedence list of the class. That value is associated with the class metaobject and is returned by **class-precedence-list**.

This generic function signals an error if *class* has not been finalized.

class-prototype *Generic Function*
class

Returns a prototype instance of *class*. Whether the instance is initialized is not specified. The results are undefined if a portable program modifies the binding of any slot of prototype instance.

This generic function signals an error if *class* has not been finalized.

class-slots *Generic Function*
class

Returns a possibly empty set of the slots accessible in instances of *class*. The elements of this set are effective slot definition metaobjects.

During class finalization **finalize-inheritance** calls **compute-slots** to compute the slots of the class. That value is associated with the class metaobject and is returned by **class-slots**.

This generic function signals an error if *class* has not been finalized.

Methods

The specified methods for the class metaobject reader generic functions are presented below.

Each entry in the table indicates a method on one of the reader generic functions, specialized to a specified class. The number in each entry is a reference to the full description of the method. The full descriptions appear after the table.

	standard-class and **funcallable-standard-class**	**forward-referenced-class**	**built-in-class**
class-default-initargs	2	3	4
class-direct-default-initargs	1	4	4
class-direct-slots	1	4	4
class-direct-subclasses	9	9	7
class-direct-superclasses	1	4	7
class-finalized-p	2	6	5
class-name	1	1	8
class-precedence-list	2	3	7
class-prototype	10	10	10
class-slots	2	3	4

1. This method returns the value which was associated with the class metaobject during initialization or reinitialization.
2. This method returns the value associated with the class metaobject by **finalize-inheritance (standard-class)** or **finalize-inheritance (funcallable-standard-class)**.
3. This method signals an error.
4. This method returns the empty list.
5. This method returns true.
6. This method returns false.
7. This method returns a value derived from the information in Table 5.1, except that implementation-specific modifications are permitted as described in section "Implementation and User Specialization."
8. This method returns the name of the built-in class.
9. This methods returns a value which is maintained by **add-direct-subclass (class class)** and **remove-direct-subclass (class class)**. This method can be overridden only if those methods are overridden as well.
10. No behavior is specified for this method beyond that specified for the generic function.

Readers for Generic Function Metaobjects

The reader generic functions which simply return information associated with generic function metaobjects are presented together in this section.

Each of the reader generic functions for generic function metaobjects has the same syntax, accepting one required argument called *generic-function*, which must be a generic function metaobject; otherwise, an error is signaled. An error is also signaled if the generic function metaobject has not been initialized.

These generic functions can be called by the user or the implementation.

The list returned by this generic function will not be mutated by the implementation. The results are undefined if a portable program mutates the list returned by this generic function.

generic-function-argument-precedence-order *Generic Function*
generic-function

Returns the argument precedence order of the generic function. This value is a list of symbols, a permutation of the required parameters in the lambda list of the generic function. This is the defaulted value of the **:argument-precedence-order** initialization argument that was associated with the generic function metaobject during initialization or reinitialization.

generic-function-declarations *Generic Function*
generic-function

Returns a possibly empty list of the declarations of the generic function. The elements of this list are declarations. This list is the defaulted value of the **:declarations** initialization argument that was associated with the generic function metaobject during initialization or reinitialization.

generic-function-lambda-list *Generic Function*
generic-function

Returns the lambda list of the generic function. This is the defaulted value of the **:lambda-list** initialization argument that was associated with the generic function metaobject during initialization or reinitialization. An error is signaled if the lambda list has yet to be supplied.

generic-function-method-class *Generic Function*
generic-function

Returns the default method class of the generic function. This class must be a subclass of the class **method**. This is the defaulted value of the **:method-class** initialization argument that was associated with the generic function metaobject during initialization or reinitialization.

generic-function-method-combination *Generic Function*
generic-function

Returns the method combination of the generic function. This is a method combination metaobject. This is the defaulted value of the **:method-combination** initialization argument that was associated with the generic function metaobject during initialization or reinitialization.

generic-function-methods *Generic Function*
generic-function

Returns the set of methods currently connected to the generic function. This is a set of method metaobjects. This value is maintained by the generic functions **add-method** and **remove-method**.

generic-function-name *Generic Function*
generic-function

Returns the name of the generic function, or **nil** if the generic function has no name. This is the defaulted value of the **:name** initialization argument that was associated with the generic function metaobject during initialization or reinitialization. (Also see **(setf generic-function-name)**.)

Methods

The specified methods for the generic function metaobject reader generic functions are presented below.

generic-function-argument-precedence-order *Primary Method*
(*generic-function* `standard-generic-function`)

generic-function-declarations *Primary Method*
(*generic-function* `standard-generic-function`)

generic-function-lambda-list *Primary Method*
(*generic-function* `standard-generic-function`)

generic-function-method-class *Primary Method*
(*generic-function* `standard-generic-function`)

generic-function-method-combination *Primary Method*
(*generic-function* `standard-generic-function`)

generic-function-name *Primary Method*
(*generic-function* `standard-generic-function`)

No behavior is specified for these methods beyond that which is specified for their respective generic functions.

generic-function-methods *Primary Method*
(*generic-function* `standard-generic-function`)

No behavior is specified for this method beyond that which is specified for their respective generic functions.

The value returned by this method is maintained by **add-method (standard-generic-function standard-method)** and **remove-method (standard-generic-function standard-method)**.

Readers for Method Metaobjects

The reader generic functions which simply return information associated with method metaobjects are presented together here in the format described under "Readers for Class Metaobjects."

Each of these reader generic functions have the same syntax, accepting one required argument called *method*, which must be a method metaobject; otherwise, an error is signaled. An error is also signaled if the method metaobject has not been initialized.

These generic functions can be called by the user or the implementation.

For any of these generic functions which returns a list, such lists will not be mutated by the implementation. The results are undefined if a portable program allows such a list to be mutated.

method-function *Generic Function*
method

Returns the method function of *method*. This is the defaulted value of the **:function** initialization argument that was associated with the method during initialization.

method-generic-function *Generic Function*
method

Returns the generic function that *method* is currently connected to, or **nil** if it is not currently connected to any generic function. This value is either a generic function metaobject or **nil**. When a method is first created it is not connected to any generic function. This connection is maintained by the generic functions **add-method** and **remove-method**.

method-lambda-list *Generic Function*
method

Returns the (unspecialized) lambda list of *method*. This value is a Common Lisp lambda list. This is the defaulted value of the **:lambda-list** initialization argument that was associated with the method during initialization.

method-specializers *Generic Function*
method

Returns a list of the specializers of *method*. This value is a list of specializer metaobjects. This is the defaulted value of the **:specializers** initialization argument that was associated with the method during initialization.

method-qualifiers *Generic Function*
method

Returns a (possibly empty) list of the qualifiers of *method*. This value is a list of non-**nil** atoms. This is the defaulted value of the **:qualifiers** initialization argument that was associated with the method during initialization.

accessor-method-slot-definition *Generic Function*
method

This accessor can only be called on accessor methods. It returns the direct slot definition metaobject that defined this method. This is the value of the **:slot-definition** initialization argument associated with the method during initialization.

METHODS

The specified methods for the method metaobject readers are presented below.

method-function *Primary Method*
(*method* `standard-method`)

method-lambda-list *Primary Method*
(*method* `standard-method`)

method-specializers *Primary Method*
(*method* `standard-method`)

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

No behavior is specified for these methods beyond that which is specified for their respective generic functions.

method-generic-function *Primary Method*
(*method* `standard-method`)

No behavior is specified for this method beyond that which is specified for its generic function.

The value returned by this method is maintained by **add-method (standard-generic-function standard-method)** and **remove-method (standard-generic-function standard-method)**.

accessor-method-slot-definition *Primary Method*
(*method* `standard-accessor-method`)

No behavior is specified for this method beyond that which is specified for its generic function.

Readers for Slot Definition Metaobjects

The reader generic functions which simply return information associated with slot definition metaobjects are presented together here in the format described under "Readers for Class Metaobjects."

Each of the reader generic functions for slot definition metaobjects has the same syntax, accepting one required argument called *slot*, which must be a slot definition metaobject; otherwise, an error is signaled. An error is also signaled if the slot definition metaobject has not been initialized.

These generic functions can be called by the user or the implementation.

For any of these generic functions which returns a list, such lists will not be mutated by the implementation. The results are undefined if a portable program allows such a list to be mutated.

Generic Functions

slot-definition-allocation *Generic Function*
slot

Returns the allocation of *slot*. This is a symbol. This is the defaulted value of the **:allocation** initialization argument that was associated with the slot definition metaobject during initialization.

slot-definition-initargs *Generic Function*
slot

Returns the set of initialization argument keywords for *slot*. This is the defaulted value of the **:initargs** initialization argument that was associated with the slot definition metaobject during initialization.

slot-definition-initform *Generic Function*
slot

Returns the initialization form of *slot*. This can be any form. This is the defaulted value of the **:initform** initialization argument that was associated with the slot definition metaobject during initialization. When *slot* has no initialization form, the value returned is unspecified (however, **slot-definition-initfunction** is guaranteed to return **nil**).

slot-definition-initfunction *Generic Function*
slot

Returns the initialization function of *slot*. This value is either a function of no arguments, or **nil**, indicating that the slot has no initialization function. This is the defaulted value of the **:initfunction** initialization argument that was associated with the slot definition metaobject during initialization.

slot-definition-name *Generic Function*
slot

Returns the name of *slot*. This value is a symbol that can be used as a variable name. This is the value of the **:name** initialization argument that was associated with the slot definition metaobject during initialization.

slot-definition-type *Generic Function*
slot

Returns the allocation of *slot*. This is a type specifier name. This is the defaulted value of the **:name** initialization argument that was associated with the slot definition metaobject during initialization.

Methods

The specified methods for the slot definition metaobject readers are presented below.

slot-definition-allocation *Primary Method*
(*slot-definition* `standard-slot-definition`)

slot-definition-initargs *Primary Method*
(*slot-definition* `standard-slot-definition`)

slot-definition-initform *Primary Method*
(*slot-definition* `standard-slot-definition`)

slot-definition-initfunction *Primary Method*
(*slot-definition* `standard-slot-definition`)

slot-definition-name *Primary Method*
(*slot-definition* `standard-slot-definition`)

slot-definition-type *Primary Method*
(*slot-definition* `standard-slot-definition`)

No behavior is specified for these methods beyond that which is specified for their respective generic functions.

Direct Slot Definition Metaobjects

The following additional reader generic functions are defined for direct slot definition metaobjects.

slot-definition-readers *Generic Function*
direct-slot

Returns a (possibly empty) set of readers of the *direct slot*. This value is a list of function names. This is the defaulted value of the **:readers** initialization argument that was associated with the direct slot definition metaobject during initialization.

slot-definition-writers *Generic Function*
direct-slot

Returns a (possibly empty) set of writers of the *direct slot*. This value is a list of function names. This is the defaulted value of the **:writers** initialization argument that was associated with the direct slot definition metaobject during initialization.

slot-definition-readers *Primary Method*
(*direct-slot-definition* `standard-direct-slot-definition`)

slot-definition-writers *Primary Method*
(*direct-slot-definition* `standard-direct-slot-definition`)

No behavior is specified for these methods beyond what is specified for their generic functions.

Effective Slot Definition Metaobjects

The following reader generic function is defined for effective slot definition metaobjects.

slot-definition-location *Generic Function*
effective-slot-definition

Returns the location of *effective-slot-definition*. The meaning and interpretation of this value is described in the section called "Instance Structure Protocol."

slot-definition-location *Primary Method*
(*effective-slot-definition* `standard-effective-slot-definition`)

This method returns the value stored by **compute-slots :around (standard-class)** and **compute-slots :around (funcallable-standard-class)**.

reader-method-class *Generic Function*

Syntax

reader-method-class
class direct-slot `&rest` *initargs*

Arguments

The *class* argument is a class metaobject.

The *direct-slot* argument is a direct slot definition metaobject.

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

Values
The value returned is a class metaobject.

Purpose
This generic function is called to determine the class of reader methods created during class initialization and reinitialization. The result must be a subclass of **standard-reader-method**.

The *initargs* argument must be the same as will be passed to **make-instance** to create the reader method. The *initargs* must include **:slot-definition** with *slot-definition* as its value.

Methods
reader-method-class *Primary Method*
    (*class* `standard-class`)
    (*direct-slot* `standard-direct-slot-definition`)
    `&rest` *initargs*

reader-method-class *Primary Method*
    (*class* `funcallable-standard-class`)
    (*direct-slot* `standard-direct-slot-definition`)
    `&rest` *initargs*

These methods return the class **standard-reader-method**. These methods can be overridden.

---

remove-dependent *Generic Function*

Syntax
remove-dependent
    metaobject dependent

Arguments
The *metaobject* argument is a class or generic function metaobject.

The *dependent* argument is an object.

Values
The value returned by this generic function is unspecified.

Purpose
This generic function removes *dependent* from the dependents of *metaobject*. If *dependent* is not one of the dependents of *metaobject*, no error is signaled.

The generic function **map-dependents** can be called to access the set of dependents of a class or generic function. The generic function **add-dependent** can be called to add an object from the set of dependents of a class or generic function. The effect of calling

add-dependent or **remove-dependent** while a call to **map-dependents** on the same class or generic function is in progress is unspecified.

The situations in which **remove-dependent** is called are not specified.

Methods

remove-dependent *Primary Method*
(*class* `standard-class`) *dependent*

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

add-dependent (standard-class t)
map-dependents (standard-class t)

remove-dependent *Primary Method*
(*class* `funcallable-standard-class`) *dependent*

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

add-dependent (funcallable-standard-class t)
map-dependents (funcallable-standard-class t)

remove-dependent *Primary Method*
(*generic-function* `standard-generic-function`) *dependent*

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

add-dependent (standard-generic-function t)
map-dependents (standard-generic-function t)

Remarks

See the "Dependent Maintenance Protocol" section for remarks about the use of this facility.

remove-direct-method *Generic Function*

Syntax

remove-direct-method
specializer method

Arguments

The *specializer* argument is a specializer metaobject.

The *method* argument is a method metaobject.

Values

The value returned by **remove-direct-method** is unspecified.

Purpose

This generic function is called to maintain a set of backpointers from a specializer to the set of methods specialized to it. If *method* is in the set it is removed. If it is not, no error is signaled.

This set can be accessed as a list by calling the generic function **specializer-direct-methods**. Methods are added to the set by **add-direct-method**.

The generic function **remove-direct-method** is called by **remove-method** whenever a method is removed from a generic function. It is called once for each of the specializers of the method. Note that in cases where a specializer appears more than once in the specializers of a method, this generic function will be called more than once with the same specializer as argument.

The results are undefined if the *specializer* argument is not one of the specializers of the *method* argument.

Methods

remove-direct-method *Primary Method*
(*specializer* `class`)
(*method* `method`)

This method implements the behavior of the generic function for class specializers. No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

add-direct-method (class method)
specializer-direct-generic-functions (class)
specializer-direct-methods (class)

remove-direct-method *Primary Method*
(*specializer* `eql-specializer`)
(*method* `method`)

This method implements the behavior of the generic function for **eql** specializers. No behavior is specified for this method beyond that which is specified for the generic function.

---

remove-direct-subclass *Generic Function*

Syntax

remove-direct-subclass
superclass subclass

Arguments

The *superclass* argument is a class metaobject.

The *subclass* argument is a class metaobject.

Values

The value returned by this generic function is unspecified.

Purpose

This generic function is called to maintain a set of backpointers from a class to its direct subclasses. It removes *subclass* from the set of direct subclasses of *superclass*. No error is signaled if *subclass* is not in this set.

Whenever a class is reinitialized, this generic function is called once with each deleted direct superclass of the class.

Methods

remove-direct-subclass *Primary Method*
(*superclass* `class`)
(*subclass* `class`)

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

add-direct-subclass (class class)
class-direct-subclasses (class)

remove-method *Generic Function*

Syntax

remove-method
generic-function method

Arguments

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

The *method* argument is a method metaobject.

Values

The *generic-function* argument is returned.

Purpose

This generic function breaks the association between a generic function and one of its methods.

No error is signaled if the method is not among the methods of the generic function.

Breaking the association between the method and the generic function proceeds in four steps: (i) remove *method* from the set returned by **generic-function-methods** and arrange for **method-generic-function** to return **nil**; (ii) call **remove-direct-method** for each of the method's specializers; (iii) call **compute-discriminating-function** and install its result with **set-funcallable-instance-function**; and (iv) update the dependents of the generic function.

The generic function **remove-method** can be called by the user or the implementation.

Methods

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

No behavior is specified for this method beyond that which is specified for the generic function.

set-funcallable-instance-function *Function*

Syntax

set-funcallable-instance-function
funcallable-instance function

Arguments

The *funcallable-instance* argument is a funcallable instance (it must have been returned by **allocate-instance (funcallable-standard-class)**).

The *function* argument is a function.

Values

The value returned by this function is unspecified.

Purpose

This function is called to set or to change the function of a funcallable instance. After **set-funcallable-instance-function** is called, any subsequent calls to *funcallable-instance* will run the new function.

(setf class-name) *Function*

Syntax

(setf class-name) *Generic Function*
new-name class

Arguments

The *class* argument is a class metaobject.

The *new-name* argument is any Lisp object.

Results

This function returns its *new-name* argument.

Purpose

This function changes the name of *class* to *new-name*. This value is usually a symbol, or **nil** if the class has no name.

This function works by calling **reinitialize-instance** with *class* as its first argument, the symbol **:name** as its second argument and *new-name* as its third argument.

(setf generic-function-name) *Function*

Syntax

(setf generic-function-name) *Generic Function*
new-name generic-function

Arguments

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

The *new-name* argument is a function name or **nil**.

Results

This function returns its *new-name* argument.

Purpose

This function changes the name of *generic-function* to *new-name*. This value is usually a function name (i.e., a symbol or a list of the form **(setf** *symbol***)**) or **nil**, if the generic function is to have no name.

This function works by calling **reinitialize-instance** with *generic-function* as its first argument, the symbol **:name** as its second argument and *new-name* as its third argument.

(setf slot-value-using-class) *Generic Function*

Syntax

(setf slot-value-using-class)
new-value class object slot

Arguments

The *new-value* argument is an object.

The *class* argument is a class metaobject. It is the class of the *object* argument.

The *object* argument is an object.

The *slot* argument is an effective slot definition metaobject.

Values

This generic function returns the *new-value* argument.

Purpose

The generic function **(setf slot-value-using-class)** implements the behavior of the **(setf slot-value)** function. It is called by **(setf slot-value)** with the class of *object* as its second argument and the pertinent effective slot definition metaobject as its fourth argument.

The generic function **(setf slot-value-using-class)** sets the value contained in the given slot of the given object to the given new value; any previous value is lost.

The results are undefined if the *class* argument is not the class of the *object* argument, or if the *slot* argument does not appear among the set of effective slots associated with the *class* argument.

Methods

(setf slot-value-using-class) *Primary Method*
  new-value
  (*class* `standard-class`)
  object
  (*slot* `standard-effective-slot-definition`)

(setf slot-value-using-class) *Primary Method*
  new-value
  (*class* `funcallable-standard-class`)
  object
  (*slot* `standard-effective-slot-definition`)

These methods implement the full behavior of this generic function for slots with allocation **:instance** and **:class**. If the supplied slot has an allocation other than **:instance** or **:class** an error is signaled.

Overriding these methods is permitted, but may require overriding other methods in the standard implementation of the slot access protocol.

(setf slot-value-using-class) *Primary Method*
  new-value
  (*class* `built-in-class`)
  object
  slot

This method signals an error.

slot-boundp-using-class *Generic Function*

Syntax

slot-boundp-using-class
  class object slot

Arguments

The *class* argument is a class metaobject. It is the class of the *object* argument.

The *object* argument is an object.

The *slot* argument is an effective slot definition metaobject.

Values

This generic function returns true or false.

Purpose

This generic function implements the behavior of the **slot-boundp** function. It is called by **slot-boundp** with the class of *object* as its first argument and the pertinent effective slot definition metaobject as its third argument.

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

The results are undefined if the *class* argument is not the class of the *object* argument, or if the *slot* argument does not appear among the set of effective slots associated with the *class* argument.

Methods

slot-boundp-using-class *Primary Method*
(*class* `standard-class`)
object
(*slot* `standard-effective-slot-definition`)

slot-boundp-using-class *Primary Method*
(*class* `funcallable-standard-class`)
object
(*slot* `standard-effective-slot-definition`)

These methods implement the full behavior of this generic function for slots with allocation **:instance** and **:class**. If the supplied slot has an allocation other than **:instance** or **:class** an error is signaled.

Overriding these methods is permitted, but may require overriding other methods in the standard implementation of the slot access protocol.

slot-boundp-using-class *Primary Method*
(*class* `built-in-class`)
object
slot

This method signals an error.

Remarks

In cases where the class metaobject class does not distinguish unbound slots, true should be returned.

---

slot-definition-... *Generic Function*

The following generic functions are described together under "Readers for Slot Definition Metaobjects" (page 83): **slot-definition-allocation**, **slot-definition-initargs**,

slot-definition-initform, **slot-definition-initfunction**, **slot-definition-location**, **slot-definition-name**, **slot-definition-readers**, **slot-definition-writers** and **slot-definition-type**.

slot-makunbound-using-class *Generic Function*

Syntax

slot-makunbound-using-class
class object slot

Arguments

The *class* argument is a class metaobject. It is the class of the *object* argument.

The *object* argument is an object.

The *slot* argument is an effective slot definition metaobject.

Values

This generic function returns its *object* argument.

Purpose

This generic function implements the behavior of the **slot-makunbound** function. It is called by **slot-makunbound** with the class of *object* as its first argument and the pertinent effective slot definition metaobject as its third argument.

The generic function **slot-makunbound-using-class** restores a slot in an object to its unbound state. The interpretation of "restoring a slot to its unbound state" depends on the class metaobject class.

The results are undefined if the *class* argument is not the class of the *object* argument, or if the *slot* argument does not appear among the set of effective slots associated with the *class* argument.

Methods

slot-makunbound-using-class *Primary Method*
(*class* `standard-class`)
object
(*slot* `standard-effective-slot-definition`)

slot-makunbound-using-class *Primary Method*
(*class* `funcallable-standard-class`)
object
(*slot* `standard-effective-slot-definition`)

These methods implement the full behavior of this generic function for slots with allocation **:instance** and **:class**. If the supplied slot has an allocation other than **:instance** or **:class** an error is signaled.

Overriding these methods is permitted, but may require overriding other methods in the standard implementation of the slot access protocol.

slot-makunbound-using-class *Primary Method*
(*class* `built-in-class`)
object
slot

This method signals an error.

slot-value-using-class *Generic Function*

SYNTAX
slot-value-using-class
class object slot

ARGUMENTS

The *class* argument is a class metaobject. It is the class of the *object* argument.

The *object* argument is an object.

The *slot* argument is an effective slot definition metaobject.

VALUES

The value returned by this generic function is an object.

PURPOSE

This generic function implements the behavior of the **slot-value** function. It is called by **slot-value** with the class of *object* as its first argument and the pertinent effective slot definition metaobject as its third argument.

The generic function **slot-value-using-class** returns the value contained in the given slot of the given object. If the slot is unbound **slot-unbound** is called.

The results are undefined if the *class* argument is not the class of the *object* argument, or if the *slot* argument does not appear among the set of effective slots associated with the *class* argument.

METHODS

slot-value-using-class *Primary Method*
(*class* `standard-class`)
object
(*slot* `standard-effective-slot-definition`)

slot-value-using-class *Primary Method*
(*class* `funcallable-standard-class`)
object
(*slot* `standard-effective-slot-definition`)

These methods implement the full behavior of this generic function for slots with allocation **:instance** and **:class**. If the supplied slot has an allocation other than **:instance** or **:class** an error is signaled.

Overriding these methods is permitted, but may require overriding other methods in the standard implementation of the slot access protocol.

slot-value-using-class *Primary Method*
(*class* `built-in-class`)
object
slot

This method signals an error.

specializer-direct-generic-functions *Generic Function*

SYNTAX

specializer-direct-generic-functions
specializer

ARGUMENTS

The *specializer* argument is a specializer metaobject.

VALUES

The result of this generic function is a possibly empty list of generic function metaobjects.

PURPOSE

This generic function returns the possibly empty set of those generic functions which have a method with *specializer* as a specializer. The elements of this set are generic function metaobjects. This value is maintained by the generic functions **add-direct-method** and **remove-direct-method**.

METHODS

specializer-direct-generic-functions *Primary Method*
  (*specializer* `class`)

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

add-direct-method (class method)
remove-direct-method (class method)
specializer-direct-methods (class)

specializer-direct-generic-functions *Primary Method*
  (*specializer* `eql-specializer`)

No behavior is specified for this method beyond that which is specified for the generic function.

specializer-direct-methods *Generic Function*

SYNTAX

specializer-direct-methods
  specializer

ARGUMENTS

The *specializer* argument is a specializer metaobject.

VALUES

The result of this generic function is a possibly empty list of method metaobjects.

PURPOSE

This generic function returns the possibly empty set of those methods, connected to generic functions, which have *specializer* as a specializer. The elements of this set are method metaobjects. This value is maintained by the generic functions **add-direct-method** and **remove-direct-method**.

METHODS

specializer-direct-methods *Primary Method*
  (*specializer* `class`)

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

add-direct-method (class method)
remove-direct-method (class method)
specializer-direct-generic-functions (class)

specializer-direct-methods *Primary Method*
(*specializer* `eql-specializer`)

No behavior is specified for this method beyond that which is specified for the generic function.

---

standard-instance-access *Function*

SYNTAX
standard-instance-access
instance location

ARGUMENTS
The *instance* argument is an object.
The *location* argument is a slot location.

VALUES
The result of this function is an object.

PURPOSE
This function is called to provide direct access to a slot in an instance. By usurping the normal slot lookup protocol, this function is intended to provide highly optimized access to the slots associated with an instance.

The following restrictions apply to the use of this function:

- The *instance* argument must be a standard instance (it must have been returned by **allocate-instance (standard-class)**).
- The *instance* argument cannot be an non-updated obsolete instance.
- The *location* argument must be a location of one of the directly accessible slots of the instance's class.
- The slot must be bound.

The results are undefined if any of these restrictions are not met.

update-dependent *Generic Function*

Syntax

update-dependent
metaobject dependent `&rest` *initargs*

Arguments

The *metaobject* argument is a class or generic function metaobject. It is the metaobject being reinitialized or otherwise modified.

The *dependent* argument is an object. It is the dependent being updated.

The *initargs* argument is a list of the initialization arguments for the metaobject redefinition.

Values

The value returned by **update-dependent** is unspecified.

Purpose

This generic function is called to update a dependent of *metaobject*.

When a class or a generic function is reinitialized each of its dependents is updated. The *initargs* argument to **update-dependent** is the set of initialization arguments received by **reinitialize-instance**.

When a method is added to a generic function, each of the generic function's dependents is updated. The *initargs* argument is a list of two elements: the symbol **add-method**, and the method that was added.

When a method is removed from a generic function, each of the generic function's dependents is updated. The *initargs* argument is a list of two elements: the symbol **remove-method**, and the method that was removed.

In each case, **map-dependents** is used to call **update-dependent** on each of the dependents. So, for example, the update of a generic function's dependents when a method is added could be performed by the following code:

```
(map-dependents generic-function
                #'(lambda (dep)
                    (update-dependent generic-function
                                      dep
                                      'add-method
                                      new-method)))
```

Methods

There are no specified methods on this generic function.

Remarks

See the "Dependent Maintenance Protocol" section for remarks about the use of this facility.

validate-superclass *Generic Function*

SYNTAX

validate-superclass
  class superclass

ARGUMENTS

The *class* argument is a class metaobject.

The *superclass* argument is a class metaobject.

VALUES

This generic function returns true or false.

PURPOSE

This generic function is called to determine whether the class *superclass* is suitable for use as a superclass of *class*.

This generic function can be be called by the implementation or user code. It is called during class metaobject initialization and reinitialization, before the direct superclasses are stored. If this generic function returns false, the initialization or reinitialization will signal an error.

METHODS

validate-superclass *Primary Method*
  (*class* `class`)
  (*superclass* `class`)

This method returns true in three situations:

(i) If the *superclass* argument is the class named **t**,

(ii) if the class of the *class* argument is the same as the class of the *superclass* argument or

(iii) if the classes one of the arguments is **standard-class** and the class of the other is **funcallable-standard-class**.

In all other cases, this method returns false.

This method can be overridden.

REMARKS

Defining a method on **validate-superclass** requires detailed knowledge of of the internal protocol followed by each of the two class metaobject classes. A method on **validate-superclass** which returns true for two different class metaobject classes declares that they are compatible.

writer-method-class *Generic Function*

Syntax

writer-method-class

class direct-slot `&rest` *initargs*

Arguments

The *class* argument is a class metaobject.

The *direct-slot* argument is a direct slot definition metaobject.

The *initargs* argument is a list of initialization arguments and values.

Values

The value returned is a class metaobject.

Purpose

This generic function is called to determine the class of writer methods created during class initialization and reinitialization. The result must be a subclass of **standard-writer-method**.

The *initargs* argument must be the same as will be passed to **make-instance** to create the reader method. The *initargs* must include **:slot-definition** with *slot-definition* as its value.

Methods

writer-method-class *Primary Method*

(*class* `standard-class`)
(*direct-slot* `standard-direct-slot-definition`)
`&rest` *initargs*

writer-method-class *Primary Method*

(*class* `funcallable-standard-class`)
(*direct-slot* `standard-direct-slot-definition`)
`&rest` *initargs*

These methods returns the class **standard-writer-method**. These methods can be overridden.

Bibliography

[CLtLII] Steele, Guy *Common Lisp: The Language,* Second Edition, Digital Press, 1990.

[X3J13] Bobrow, Daniel G. Linda G. Demichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon *Common Lisp Object System Specification,* X3J13 Document 88-002R, June 1988; appears in *Lisp and Symbolic Computation* **1**, 3/4, January 1989, 245–394, and as Chapter 28 of [CLtLII], 770–864.