Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

Programming Language—Common Lisp

7. Objects

7.1 Object Creation and Initialization

The *generic function* **make-instance** creates and returns a new *instance* of a *class*. The first argument is a *class* or the *name* of a *class*, and the remaining arguments form an ***initialization argument list***.

The initialization of a new *instance* consists of several distinct steps, including the following: combining the explicitly supplied initialization arguments with default values for the unsupplied initialization arguments, checking the validity of the initialization arguments, allocating storage for the *instance*, filling *slots* with values, and executing user-supplied *methods* that perform additional initialization. Each step of **make-instance** is implemented by a *generic function* to provide a mechanism for customizing that step. In addition, **make-instance** is itself a *generic function* and thus also can be customized.

The object system specifies system-supplied primary *methods* for each step and thus specifies a well-defined standard behavior for the entire initialization process. The standard behavior provides four simple mechanisms for controlling initialization:

- Declaring a *symbol* to be an initialization argument for a *slot*. An initialization argument is declared by using the `:initarg` slot option to **defclass**. This provides a mechanism for supplying a value for a *slot* in a call to **make-instance**.

- Supplying a default value form for an initialization argument. Default value forms for initialization arguments are defined by using the `:default-initargs` class option to **defclass**. If an initialization argument is not explicitly provided as an argument to **make-instance**, the default value form is evaluated in the lexical environment of the **defclass** form that defined it, and the resulting value is used as the value of the initialization argument.

- Supplying a default initial value form for a *slot*. A default initial value form for a *slot* is defined by using the `:initform` slot option to **defclass**. If no initialization argument associated with that *slot* is given as an argument to **make-instance** or is defaulted by `:default-initargs`, this default initial value form is evaluated in the lexical environment of the **defclass** form that defined it, and the resulting value is stored in the *slot*. The `:initform` form for a *local slot* may be used when creating an *instance*, when updating an *instance* to conform to a redefined *class*, or when updating an *instance* to conform to the definition of a different *class*. The `:initform` form for a *shared slot* may be used when defining or re-defining the *class*.

- Defining *methods* for **initialize-instance** and **shared-initialize**. The slot-filling behavior described above is implemented by a system-supplied primary *method* for **initialize-instance** which invokes **shared-initialize**. The *generic function* **shared-initialize** implements the parts of initialization shared by these four situations: when making an *instance*, when re-initializing an *instance*, when updating an *instance* to conform to a redefined *class*, and when updating an *instance* to conform to the definition of a different *class*. The system-supplied primary *method* for **shared-initialize** directly implements the slot-filling behavior described above, and **initialize-instance** simply invokes **shared-initialize**.

7.1.1 Initialization Arguments

An initialization argument controls *object* creation and initialization. It is often convenient to use keyword *symbols* to name initialization arguments, but the *name* of an initialization argument can be any *symbol*, including **nil**. An initialization argument can be used in two ways: to fill a *slot* with a value or to provide an argument for an initialization *method*. A single initialization argument can be used for both purposes.

An *initialization argument list* is a *property list* of initialization argument names and values. Its structure is identical to a *property list* and also to the portion of an argument list processed for **&key** parameters. As in those lists, if an initialization argument name appears more than once in an initialization argument list, the leftmost occurrence supplies the value and the remaining occurrences are ignored. The arguments to **make-instance** (after the first argument) form an *initialization argument list*.

An initialization argument can be associated with a *slot*. If the initialization argument has a value in the *initialization argument list*, the value is stored into the *slot* of the newly created *object*, overriding any `:initform` form associated with the *slot*. A single initialization argument can initialize more than one *slot*. An initialization argument that initializes a *shared slot* stores its value into the *shared slot*, replacing any previous value.

An initialization argument can be associated with a *method*. When an *object* is created and a particular initialization argument is supplied, the *generic functions* **initialize-instance**, **shared-initialize**, and **allocate-instance** are called with that initialization argument's name and value as a keyword argument pair. If a value for the initialization argument is not supplied in the *initialization argument list*, the *method*'s *lambda list* supplies a default value.

Initialization arguments are used in four situations: when making an *instance*, when re-initializing an *instance*, when updating an *instance* to conform to a redefined *class*, and when updating an *instance* to conform to the definition of a different *class*.

Because initialization arguments are used to control the creation and initialization of an *instance* of some particular *class*, we say that an initialization argument is "an initialization argument for" that *class*.

7.1.2 Declaring the Validity of Initialization Arguments

Initialization arguments are checked for validity in each of the four situations that use them. An initialization argument may be valid in one situation and not another. For example, the system-supplied primary *method* for **make-instance** defined for the *class* **standard-class** checks the validity of its initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid in that situation.

There are two means for declaring initialization arguments valid.

- Initialization arguments that fill *slots* are declared as valid by the `:initarg` slot option to **defclass**. The `:initarg` slot option is inherited from *superclasses*. Thus the set of valid initialization arguments that fill *slots* for a *class* is the union of the initialization arguments that fill *slots* declared as valid by that *class* and its *superclasses*. Initialization arguments that fill *slots* are valid in all four contexts.

- Initialization arguments that supply arguments to *methods* are declared as valid by defining those *methods*. The keyword name of each keyword parameter specified in the *method*'s *lambda list* becomes an initialization argument for all *classes* for which the *method* is applicable. The presence of `&allow-other-keys` in the *lambda list* of an applicable method disables validity checking of initialization arguments. Thus *method* inheritance controls the set of valid initialization arguments that supply arguments to *methods*. The *generic functions* for which *method* definitions serve to declare initialization arguments valid are as follows:

  - Making an *instance* of a *class*: **allocate-instance**, **initialize-instance**, and **shared-initialize**. Initialization arguments declared as valid by these *methods* are valid when making an *instance* of a *class*.

  - Re-initializing an *instance*: **reinitialize-instance** and **shared-initialize**. Initialization arguments declared as valid by these *methods* are valid when re-initializing an *instance*.

  - Updating an *instance* to conform to a redefined *class*: **update-instance-for-redefined-class** and **shared-initialize**. Initialization arguments declared as valid by these *methods* are valid when updating an *instance* to conform to a redefined *class*.

  - Updating an *instance* to conform to the definition of a different *class*: **update-instance-for-different-class** and **shared-initialize**. Initialization arguments declared as valid by these *methods* are valid when updating an *instance* to conform to the definition of a different *class*.

The set of valid initialization arguments for a *class* is the set of valid initialization arguments that either fill *slots* or supply arguments to *methods*, along with the predefined initialization argument `:allow-other-keys`. The default value for `:allow-other-keys` is **nil**. Validity checking of initialization arguments is disabled if the value of the initialization argument `:allow-other-keys` is *true*.

7.1.3 Defaulting of Initialization Arguments

A default value *form* can be supplied for an initialization argument by using the `:default-initargs` *class* option. If an initialization argument is declared valid by some particular *class*, its default value form might be specified by a different *class*. In this case `:default-initargs` is used to supply a default value for an inherited initialization argument.

The `:default-initargs` option is used only to provide default values for initialization arguments; it does not declare a *symbol* as a valid initialization argument name. Furthermore, the `:default-initargs` option is used only to provide default values for initialization arguments when making an *instance*.

The argument to the `:default-initargs` class option is a list of alternating initialization argument names and *forms*. Each *form* is the default value form for the corresponding initialization argument. The default value *form* of an initialization argument is used and evaluated only if that initialization argument does not appear in the arguments to **make-instance** and is not defaulted by a more specific *class*. The default value *form* is evaluated in the lexical environment of the **defclass** form that supplied it; the resulting value is used as the initialization argument's value.

The initialization arguments supplied to **make-instance** are combined with defaulted initialization arguments to produce a *defaulted initialization argument list*. A *defaulted initialization argument list* is a list of alternating initialization argument names and values in which unsupplied initialization arguments are defaulted and in which the explicitly supplied initialization arguments appear earlier in the list than the defaulted initialization arguments. Defaulted initialization arguments are ordered according to the order in the *class precedence list* of the *classes* that supplied the default values.

There is a distinction between the purposes of the `:default-initargs` and the `:initform` options with respect to the initialization of *slots*. The `:default-initargs` class option provides a mechanism for the user to give a default value *form* for an initialization argument without knowing whether the initialization argument initializes a *slot* or is passed to a *method*. If that initialization argument is not explicitly supplied in a call to **make-instance**, the default value *form* is used, just as if it had been supplied in the call. In contrast, the `:initform` slot option provides a mechanism for the user to give a default initial value form for a *slot*. An `:initform` form is used to initialize a *slot* only if no initialization argument associated with that *slot* is given as an argument to **make-instance** or is defaulted by `:default-initargs`.

The order of evaluation of default value *forms* for initialization arguments and the order of evaluation of `:initform` forms are undefined. If the order of evaluation is important, **initialize-instance** or **shared-initialize** *methods* should be used instead.

7.1.4 Rules for Initialization Arguments

The `:initarg` slot option may be specified more than once for a given *slot*.

The following rules specify when initialization arguments may be multiply defined:

- A given initialization argument can be used to initialize more than one *slot* if the same initialization argument name appears in more than one `:initarg` slot option.

- A given initialization argument name can appear in the *lambda list* of more than one initialization *method*.

- A given initialization argument name can appear both in an `:initarg` slot option and in the *lambda list* of an initialization *method*.

If two or more initialization arguments that initialize the same *slot* are given in the arguments to **make-instance**, the leftmost of these initialization arguments in the *initialization argument list* supplies the value, even if the initialization arguments have different names.

If two or more different initialization arguments that initialize the same *slot* have default values and none is given explicitly in the arguments to **make-instance**, the initialization argument that appears in a `:default-initargs` class option in the most specific of the *classes* supplies the value. If a single `:default-initargs` class option specifies two or more initialization arguments that initialize the same *slot* and none is given explicitly in the arguments to **make-instance**, the leftmost in the `:default-initargs` class option supplies the value, and the values of the remaining default value *forms* are ignored.

Initialization arguments given explicitly in the arguments to **make-instance** appear to the left of defaulted initialization arguments. Suppose that the classes C_1 and C_2 supply the values of defaulted initialization arguments for different *slots*, and suppose that C_1 is more specific than C_2; then the defaulted initialization argument whose value is supplied by C_1 is to the left of the defaulted initialization argument whose value is supplied by C_2 in the *defaulted initialization argument list*. If a single `:default-initargs` class option supplies the values of initialization arguments for two different *slots*, the initialization argument whose value is specified farther to the left in the `:default-initargs` class option appears farther to the left in the *defaulted initialization argument list*.

If a *slot* has both an `:initform` form and an `:initarg` slot option, and the initialization argument is defaulted using `:default-initargs` or is supplied to **make-instance**, the captured `:initform` form is neither used nor evaluated.

The following is an example of the above rules:

```
(defclass q () ((x :initarg a)))
(defclass r (q) ((x :initarg b))
  (:default-initargs a 1 b 2))
```

Form	**Defaulted Initialization Argument List**	**Contents of Slot X**
`(make-instance 'r)`	`(a 1 b 2)`	`1`
`(make-instance 'r 'a 3)`	`(a 3 b 2)`	`3`
`(make-instance 'r 'b 4)`	`(b 4 a 1)`	`4`
`(make-instance 'r 'a 1 'a 2)`	`(a 1 a 2 b 2)`	`1`

7.1.5 Shared-Initialize

The *generic function* **shared-initialize** is used to fill the *slots* of an *instance* using initialization arguments and `:initform` forms when an *instance* is created, when an *instance* is re-initialized, when an *instance* is updated to conform to a redefined *class*, and when an *instance* is updated to conform to a different *class*. It uses standard *method* combination. It takes the following arguments: the *instance* to be initialized, a specification of a set of *names* of *slots accessible* in that *instance*, and any number of initialization arguments. The arguments after the first two must form an *initialization argument list*.

The second argument to **shared-initialize** may be one of the following:

- It can be a (possibly empty) *list* of *slot* names, which specifies the set of those *slot* names.

- It can be the symbol **t**, which specifies the set of all of the *slots*.

There is a system-supplied primary *method* for **shared-initialize** whose first *parameter specializer* is the *class* **standard-object**. This *method* behaves as follows on each *slot*, whether shared or local:

- If an initialization argument in the *initialization argument list* specifies a value for that *slot*, that value is stored into the *slot*, even if a value has already been stored in the *slot* before the *method* is run. The affected *slots* are independent of which *slots* are indicated by the second argument to **shared-initialize**.

- Any *slots* indicated by the second argument that are still unbound at this point are initialized according to their `:initform` forms. For any such *slot* that has an `:initform` form, that *form* is evaluated in the lexical environment of its defining **defclass** form and the result is stored into the *slot*. For example, if a *before method* stores a value in the *slot*, the `:initform` form will not be used to supply a value for the *slot*. If the second argument specifies a *name* that does not correspond to any *slots accessible* in the *instance*, the results are unspecified.

- The rules mentioned in Section 7.1.4 (Rules for Initialization Arguments) are obeyed.

The generic function **shared-initialize** is called by the system-supplied primary *methods* for **reinitialize-instance**, **update-instance-for-different-class**, **update-instance-for-redefined-class**, and **initialize-instance**. Thus, *methods* can be written for **shared-initialize** to specify actions that should be taken in all of these contexts.

7.1.6 Initialize-Instance

The *generic function* **initialize-instance** is called by **make-instance** to initialize a newly created *instance*. It uses *standard method combination*. *Methods* for **initialize-instance** can be defined in order to perform any initialization that cannot be achieved simply by supplying initial values for *slots*.

During initialization, **initialize-instance** is invoked after the following actions have been taken:

- The *defaulted initialization argument list* has been computed by combining the supplied *initialization argument list* with any default initialization arguments for the *class*.
- The validity of the *defaulted initialization argument list* has been checked. If any of the initialization arguments has not been declared as valid, an error is signaled.
- A new *instance* whose *slots* are unbound has been created.

The generic function **initialize-instance** is called with the new *instance* and the defaulted initialization arguments. There is a system-supplied primary *method* for **initialize-instance** whose *parameter specializer* is the *class* **standard-object**. This *method* calls the generic function **shared-initialize** to fill in the *slots* according to the initialization arguments and the `:initform` forms for the *slots*; the generic function **shared-initialize** is called with the following arguments: the *instance*, **t**, and the defaulted initialization arguments.

Note that **initialize-instance** provides the *defaulted initialization argument list* in its call to **shared-initialize**, so the first step performed by the system-supplied primary *method* for **shared-initialize** takes into account both the initialization arguments provided in the call to **make-instance** and the *defaulted initialization argument list*.

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

The object system provides two *functions* that are useful in the bodies of **initialize-instance** methods. The *function* **slot-boundp** returns a *generic boolean* value that indicates whether a specified *slot* has a value; this provides a mechanism for writing *after methods* for **initialize-instance** that initialize *slots* only if they have not already been initialized. The *function* **slot-makunbound** causes the *slot* to have no value.

7.1.7 Definitions of Make-Instance and Initialize-Instance

The generic function **make-instance** behaves as if it were defined as follows, except that certain optimizations are permitted:

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

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

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

The elided code in the definition of **make-instance** augments the `initargs` with any *defaulted initialization arguments* and checks the resulting initialization arguments to determine whether an initialization argument was supplied that neither filled a *slot* nor supplied an argument to an applicable *method*.

The generic function **initialize-instance** behaves as if it were defined as follows, except that certain optimizations are permitted:

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

These procedures can be customized.

Customizing at the Programmer Interface level includes using the `:initform`, `:initarg`, and `:default-initargs` options to **defclass**, as well as defining *methods* for **make-instance**, **allocate-instance**, and **initialize-instance**. It is also possible to define *methods* for **shared-initialize**, which would be invoked by the generic functions **reinitialize-instance**, **update-instance-for-redefined-class**, **update-instance-for-different-class**, and **initialize-instance**. The meta-object level supports additional customization.

Implementations are permitted to make certain optimizations to **initialize-instance** and **shared-initialize**. The description of **shared-initialize** in Chapter 7 mentions the possible optimizations.

7.2 Changing the Class of an Instance

The *function* **change-class** can be used to change the *class* of an *instance* from its current class, C_{from}, to a different class, C_{to}; it changes the structure of the *instance* to conform to the definition of the class C_{to}.

Note that changing the *class* of an *instance* may cause *slots* to be added or deleted. Changing the *class* of an *instance* does not change its identity as defined by the **eq** function.

When **change-class** is invoked on an *instance*, a two-step updating process takes place. The first step modifies the structure of the *instance* by adding new *local slots* and discarding *local slots* that are not specified in the new version of the *instance*. The second step initializes the newly added *local slots* and performs any other user-defined actions. These two steps are further described in the two following sections.

7.2.1 Modifying the Structure of the Instance

In order to make the *instance* conform to the class C_{to}, *local slots* specified by the class C_{to} that are not specified by the class C_{from} are added, and *local slots* not specified by the class C_{to} that are specified by the class C_{from} are discarded.

The values of *local slots* specified by both the class C_{to} and the class C_{from} are retained. If such a *local slot* was unbound, it remains unbound.

The values of *slots* specified as shared in the class C_{from} and as local in the class C_{to} are retained.

This first step of the update does not affect the values of any *shared slots*.

7.2.2 Initializing Newly Added Local Slots

The second step of the update initializes the newly added *slots* and performs any other user-defined actions. This step is implemented by the generic function **update-instance-for-different-class**. The generic function **update-instance-for-different-class** is invoked by **change-class** after the first step of the update has been completed.

The generic function **update-instance-for-different-class** is invoked on arguments computed by **change-class**. The first argument passed is a copy of the *instance* being updated and is an *instance* of the class C_{from}; this copy has *dynamic extent* within the generic function **change-class**. The second argument is the *instance* as updated so far by **change-class** and is an *instance* of the class C_{to}. The remaining arguments are an *initialization argument list*.

There is a system-supplied primary *method* for **update-instance-for-different-class** that has two parameter specializers, each of which is the *class* **standard-object**. First this *method* checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. (For more information, see Section 7.1.2 (Declaring the Validity of Initialization Arguments).) Then it calls the generic function **shared-initialize** with the following arguments: the new *instance*, a list of *names* of the newly added *slots*, and the initialization arguments it received.

7.2.3 Customizing the Change of Class of an Instance

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

Methods for **shared-initialize** may be defined to customize *class* redefinition. For more information, see Section 7.1.5 (Shared-Initialize).

7.3 Reinitializing an Instance

The generic function **reinitialize-instance** may be used to change the values of *slots* according to initialization arguments.

The process of reinitialization changes the values of some *slots* and performs any user-defined actions. It does not modify the structure of an *instance* to add or delete *slots*, and it does not use any `:initform` forms to initialize *slots*.

The generic function **reinitialize-instance** may be called directly. It takes one required argument, the *instance*. It also takes any number of initialization arguments to be used by *methods* for **reinitialize-instance** or for **shared-initialize**. The arguments after the required *instance* must form an *initialization argument list*.

There is a system-supplied primary *method* for **reinitialize-instance** whose *parameter specializer* is the *class* **standard-object**. First this *method* checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. (For more information, see Section 7.1.2 (Declaring the Validity of Initialization Arguments).) Then it calls the generic function **shared-initialize** with the following arguments: the *instance*, **nil**, and the initialization arguments it received.

7.3.1 Customizing Reinitialization

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

Methods for **shared-initialize** may be defined to customize *class* redefinition. For more information, see Section 7.1.5 (Shared-Initialize).

7.4 Meta-Objects

The implementation of the object system manipulates *classes*, *methods*, and *generic functions*. The object system contains a set of *generic functions* defined by *methods* on *classes*; the behavior of those *generic functions* defines the behavior of the object system. The *instances* of the *classes* on which those *methods* are defined are called meta-objects.

7.4.1 Standard Meta-objects

The object system supplies a set of meta-objects, called standard meta-objects. These include the *class* **standard-object** and *instances* of the classes **standard-method**, **standard-generic-function**, and **method-combination**.

- The *class* **standard-method** is the default *class* of *methods* defined by the **defmethod** and **defgeneric** *forms*.

- The *class* **standard-generic-function** is the default *class* of *generic functions* defined by the forms **defmethod**, **defgeneric**, and **defclass**.

- The *class* named **standard-object** is an *instance* of the *class* **standard-class** and is a *superclass* of every *class* that is an *instance* of **standard-class** except itself and **structure-class**.

- Every *method* combination object is an *instance* of a *subclass* of *class* **method-combination**.

7.5 Slots

7.5.1 Introduction to Slots

An *object* of *metaclass* **standard-class** has zero or more named *slots*. The *slots* of an *object* are determined by the *class* of the *object*. Each *slot* can hold one value. The *name* of a *slot* is a *symbol* that is syntactically valid for use as a variable name.

When a *slot* does not have a value, the *slot* is said to be *unbound*. When an unbound *slot* is read, the *generic function* **slot-unbound** is invoked. The system-supplied primary *method* for **slot-unbound** on *class* **t** signals an error. If **slot-unbound** returns, its *primary value* is used that time as the *value* of the *slot*.

The default initial value form for a *slot* is defined by the `:initform` slot option. When the `:initform` form is used to supply a value, it is evaluated in the lexical environment in which the **defclass** form was evaluated. The `:initform` along with the lexical environment in which the **defclass** form was evaluated is called a *captured initialization form*. For more details, see Section 7.1 (Object Creation and Initialization).

A *local slot* is defined to be a *slot* that is *accessible* to exactly one *instance*, namely the one in which the *slot* is allocated. A *shared slot* is defined to be a *slot* that is visible to more than one *instance* of a given *class* and its *subclasses*.

A *class* is said to define a *slot* with a given *name* when the **defclass** form for that *class* contains a *slot specifier* with that *name*. Defining a *local slot* does not immediately create a *slot*; it causes a *slot* to be created each time an *instance* of the *class* is created. Defining a *shared slot* immediately creates a *slot*.

The `:allocation` slot option to **defclass** controls the kind of *slot* that is defined. If the value of the `:allocation` slot option is `:instance`, a *local slot* is created. If the value of `:allocation` is `:class`, a *shared slot* is created.

A *slot* is said to be *accessible* in an *instance* of a *class* if the *slot* is defined by the *class* of the *instance* or is inherited from a *superclass* of that *class*. At most one *slot* of a given *name* can be *accessible* in an *instance*. A *shared slot* defined by a *class* is *accessible* in all *instances* of that *class*. A detailed explanation of the inheritance of *slots* is given in Section 7.5.3 (Inheritance of Slots and Slot Options).

7.5.2 Accessing Slots

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

The *function* **slot-value** can be used with any of the *slot* names specified in the **defclass** form to *access* a specific *slot accessible* in an *instance* of the given *class*.

The macro **defclass** provides syntax for generating *methods* to read and write *slots*. If a reader *method* is requested, a *method* is automatically generated for reading the value of the *slot*, but no *method* for storing a value into it is generated. If a writer *method* is requested, a *method* is automatically generated for storing a value into the *slot*, but no *method* for reading its value is generated. If an accessor *method* is requested, a *method* for reading the value of the *slot* and a *method* for storing a value into the *slot* are automatically generated. Reader and writer *methods* are implemented using **slot-value**.

When a reader or writer *method* is specified for a *slot*, the name of the *generic function* to which the generated *method* belongs is directly specified. If the *name* specified for the writer *method* is the symbol `name`, the *name* of the *generic function* for writing the *slot* is the symbol `name`, and the *generic function* takes two arguments: the new value and the *instance*, in that order. If the *name* specified for the accessor *method* is the symbol `name`, the *name* of the *generic function* for reading the *slot* is the symbol `name`, and the *name* of the *generic function* for writing the *slot* is the list `(setf name)`.

A *generic function* created or modified by supplying `:reader`, `:writer`, or `:accessor` *slot* options can be treated exactly as an ordinary *generic function*.

Note that **slot-value** can be used to read or write the value of a *slot* whether or not reader or writer *methods* exist for that *slot*. When **slot-value** is used, no reader or writer *methods* are invoked.

The macro **with-slots** can be used to establish a *lexical environment* in which specified *slots* are lexically available as if they were variables. The macro **with-slots** invokes the *function* **slot-value** to *access* the specified *slots*.

The macro **with-accessors** can be used to establish a lexical environment in which specified *slots* are lexically available through their accessors as if they were variables. The macro **with-accessors** invokes the appropriate accessors to *access* the specified *slots*.

7.5.3 Inheritance of Slots and Slot Options

The set of the *names* of all *slots accessible* in an *instance* of a *class* C is the union of the sets of *names* of *slots* defined by C and its *superclasses*. The structure of an *instance* is the set of *names* of *local slots* in that *instance*.

In the simplest case, only one *class* among C and its *superclasses* defines a *slot* with a given *slot* name. If a *slot* is defined by a *superclass* of C, the *slot* is said to be inherited. The characteristics of the *slot* are determined by the *slot specifier* of the defining *class*. Consider the defining *class* for a slot S. If the value of the `:allocation` slot option is `:instance`, then S is a *local slot* and each *instance* of C has its own *slot* named S that stores its own value. If the value of the `:allocation` slot option is `:class`, then S is a *shared slot*, the *class* that defined S stores the value, and all *instances* of C can *access* that single *slot*. If the `:allocation` slot option is omitted, `:instance` is used.

In general, more than one *class* among C and its *superclasses* can define a *slot* with a given *name*. In such cases, only one *slot* with the given name is *accessible* in an *instance* of C, and the characteristics of that *slot* are a combination of the several *slot* specifiers, computed as follows:

- All the *slot specifiers* for a given *slot* name are ordered from most specific to least specific, according to the order in C's *class precedence list* of the *classes* that define them. All references to the specificity of *slot specifiers* immediately below refers to this ordering.

- The allocation of a *slot* is controlled by the most specific *slot specifier*. If the most specific *slot specifier* does not contain an `:allocation` slot option, `:instance` is used. Less specific *slot specifiers* do not affect the allocation.

- The default initial value form for a *slot* is the value of the `:initform` slot option in the most specific *slot specifier* that contains one. If no *slot specifier* contains an `:initform` slot option, the *slot* has no default initial value form.

- The contents of a *slot* will always be of type `(and` T_1 `...` T_n`)` where $T_1 \ldots T_n$ are the values of the `:type` slot options contained in all of the *slot specifiers*. If no *slot specifier* contains the `:type` slot option, the contents of the *slot* will always be of *type* `t`. The consequences of attempting to store in a *slot* a value that does not satisfy the *type* of the *slot* are undefined.

- The set of initialization arguments that initialize a given *slot* is the union of the initialization arguments declared in the `:initarg` slot options in all the *slot specifiers*.

- The *documentation string* for a *slot* is the value of the `:documentation` slot option in the most specific *slot* specifier that contains one. If no *slot specifier* contains a `:documentation` slot option, the *slot* has no *documentation string*.

A consequence of the allocation rule is that a *shared slot* can be *shadowed*. For example, if a class C_1 defines a *slot* named S whose value for the `:allocation` slot option is `:class`, that *slot* is *accessible* in *instances* of C_1 and all of its *subclasses*. However, if C_2 is a *subclass* of C_1 and also defines a *slot* named S, C_1's *slot* is not shared by *instances* of C_2 and its *subclasses*. When a class C_1 defines a *shared slot*, any subclass C_2 of C_1 will share this single *slot* unless the **defclass** form for C_2 specifies a *slot* of the same *name* or there is a *superclass* of C_2 that precedes C_1 in the *class precedence list* of C_2 that defines a *slot* of the same name.

A consequence of the type rule is that the value of a *slot* satisfies the type constraint of each *slot specifier* that contributes to that *slot*. Because the result of attempting to store in a *slot* a value that does not satisfy the type constraint for the *slot* is undefined, the value in a *slot* might fail to satisfy its type constraint.

The `:reader`, `:writer`, and `:accessor` slot options create *methods* rather than define the characteristics of a *slot*. Reader and writer *methods* are inherited in the sense described in Section 7.6.7 (Inheritance of Methods).

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

7.6 Generic Functions and Methods

7.6.1 Introduction to Generic Functions

A ***generic function*** is a function whose behavior depends on the *classes* or identities of the *arguments* supplied to it. A *generic function object* is associated with a set of *methods*, a *lambda list*, a *method combination*$_2$, and other information.

Like an *ordinary function*, a *generic function* takes *arguments*, performs a series of operations, and perhaps returns useful *values*. An *ordinary function* has a single body of *code* that is always *executed* when the *function* is called. A *generic function* has a set of bodies of *code* of which a subset is selected for *execution*. The selected bodies of *code* and the manner of their combination are determined by the *classes* or identities of one or more of the *arguments* to the *generic function* and by its *method combination*.

Ordinary functions and *generic functions* are called with identical syntax.

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

A *binding* of a *function name* to a *generic function* can be *established* in one of several ways. It can be *established* in the *global environment* by **ensure-generic-function**, **defmethod** (implicitly, due to **ensure-generic-function**) or **defgeneric** (also implicitly, due to **ensure-generic-function**). No *standardized* mechanism is provided for *establishing* a *binding* of a *function name* to a *generic function* in the *lexical environment*.

When a **defgeneric** form is evaluated, one of three actions is taken (due to **ensure-generic-function**):

- If a generic function of the given name already exists, the existing generic function object is modified. Methods specified by the current **defgeneric** form are added, and any methods in the existing generic function that were defined by a previous **defgeneric** form are removed. Methods added by the current **defgeneric** form might replace methods defined by **defmethod**, **defclass**, **define-condition**, or **defstruct**. No other methods in the generic function are affected or replaced.

- If the given name names an *ordinary function*, a *macro*, or a *special operator*, an error is signaled.

- Otherwise a generic function is created with the methods specified by the method definitions in the **defgeneric** form.

Some *operators* permit specification of the options of a *generic function*, such as the *type* of *method combination* it uses or its *argument precedence order*. These *operators* will be referred to as "operators that specify generic function options." The only *standardized operator* in this category is **defgeneric**.

Some *operators* define *methods* for a *generic function*. These *operators* will be referred to as ***method-defining operators***; their associated *forms* are called *method-defining forms*. The *standardized method-defining operators* are listed in Figure 7–1.

defgeneric	**defmethod**	**defclass**
define-condition	**defstruct**	

Figure 7–1. Standardized Method-Defining Operators

Note that of the *standardized method-defining operators* only **defgeneric** can specify *generic function* options. **defgeneric** and any *implementation-defined operators* that can specify *generic function* options are also referred to as "operators that specify generic function options."

7.6.2 Introduction to Methods

Methods define the class-specific or identity-specific behavior and operations of a *generic function*.

A *method object* is associated with *code* that implements the method's behavior, a sequence of *parameter specializers* that specify when the given *method* is applicable, a *lambda list*, and a sequence of *qualifiers* that are used by the method combination facility to distinguish among *methods*.

A method object is not a function and cannot be invoked as a function. Various mechanisms in the object system take a method object and invoke its method function, as is the case when a generic function is invoked. When this occurs it is said that the method is invoked or called.

A method-defining form contains the *code* that is to be run when the arguments to the generic function cause the method that it defines to be invoked. When a method-defining form is evaluated, a method object is created and one of four actions is taken:

- If a *generic function* of the given name already exists and if a *method object* already exists that agrees with the new one on *parameter specializers* and *qualifiers*, the new *method object* replaces the old one. For a definition of one method agreeing with another on *parameter specializers* and *qualifiers*, see Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers).

- If a *generic function* of the given name already exists and if there is no *method object* that agrees with the new one on *parameter specializers* and *qualifiers*, the existing *generic function object* is modified to contain the new *method object*.

- If the given *name* names an *ordinary function*, a *macro*, or a *special operator*, an error is signaled.

- Otherwise a *generic function* is created with the *method* specified by the *method-defining form*.

If the *lambda list* of a new *method* is not *congruent* with the *lambda list* of the *generic function*, an error is signaled. If a *method-defining operator* that cannot specify *generic function* options creates a new *generic function*, a *lambda list* for that *generic function* is derived from the *lambda list* of the *method* in the *method-defining form* in such a way as to be *congruent* with it. For a discussion of ***congruence***, see Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function).

Each method has a *specialized lambda list*, which determines when that method can be applied. A *specialized lambda list* is like an *ordinary lambda list* except that a specialized parameter may occur instead of the name of a required parameter. A specialized parameter is a list `(`*variable-name parameter-specializer-name*`)`, where *parameter-specializer-name* is one of the following:

a *symbol*

denotes a *parameter specializer* which is the *class* named by that *symbol*.

a *class*

denotes a *parameter specializer* which is the *class* itself.

`(eql` *form*`)`

denotes a *parameter specializer* which satisfies the *type specifier* `(eql` *object*`)`, where *object* is the result of evaluating *form*. The form *form* is evaluated in the lexical environment in which the method-defining form is evaluated. Note that *form* is evaluated only once, at the time the method is defined, not each time the generic function is called.

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

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

Given a generic function and a set of arguments, an applicable method is a method for that generic function whose parameter specializers are satisfied by their corresponding arguments. The following definition specifies what it means for a method to be applicable and for an argument to satisfy a *parameter specializer*.

Let $\langle A_1, \ldots, A_n \rangle$ be the required arguments to a generic function in order. Let $\langle P_1, \ldots, P_n \rangle$ be the *parameter specializers* corresponding to the required parameters of the method M in order. The method M is applicable when each A_i is of the *type* specified by the *type specifier* P_i. Because every valid *parameter specializer* is also a valid *type specifier*, the *function* **typep** can be used during method selection to determine whether an argument satisfies a *parameter specializer*.

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

Methods can have *qualifiers*, which give the method combination procedure a way to distinguish among methods. A method that has one or more *qualifiers* is called a *qualified method*. A method with no *qualifiers* is called an *unqualified method*. A *qualifier* is any *non-list*. The *qualifiers* defined by the *standardized* method combination types are *symbols*.

In this specification, the terms "*primary method*" and "*auxiliary method*" are used to partition *methods* within a method combination type according to their intended use. In standard method combination, *primary methods* are *unqualified methods* and *auxiliary methods* are methods with a single *qualifier* that is one of `:around`, `:before`, or `:after`. *Methods* with these *qualifiers* are called *around methods*, *before methods*, and *after methods*, respectively. When a method combination type is defined using the short form of **define-method-combination**, *primary methods* are methods qualified with the name of the type of method combination, and auxiliary methods have the *qualifier* `:around`. Thus the terms "*primary method*" and "*auxiliary method*" have only a relative definition within a given method combination type.

7.6.3 Agreement on Parameter Specializers and Qualifiers

Two *methods* are said to agree with each other on *parameter specializers* and *qualifiers* if the following conditions hold:

1. Both methods have the same number of required parameters. Suppose the *parameter specializers* of the two methods are $P_{1,1} \ldots P_{1,n}$ and $P_{2,1} \ldots P_{2,n}$.

2. For each $1 \leq i \leq n$, $P_{1,i}$ agrees with $P_{2,i}$. The *parameter specializer* $P_{1,i}$ agrees with $P_{2,i}$ if $P_{1,i}$ and $P_{2,i}$ are the same class or if $P_{1,i}$ = `(`**eql** *object*$_1$`)`, $P_{2,i}$ = `(`**eql** *object*$_2$`)`, and `(`**eql** *object*$_1$ *object*$_2$`)`. Otherwise $P_{1,i}$ and $P_{2,i}$ do not agree.

3. The two *lists* of *qualifiers* are the *same* under **equal**.

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

These rules define the congruence of a set of *lambda lists*, including the *lambda list* of each method for a given generic function and the *lambda list* specified for the generic function itself, if given.

1. Each *lambda list* must have the same number of required parameters.

2. Each *lambda list* must have the same number of optional parameters. Each method can supply its own default for an optional parameter.

3. If any *lambda list* mentions **&rest** or **&key**, each *lambda list* must mention one or both of them.

4. If the *generic function lambda list* mentions **&key**, each method must accept all of the keyword names mentioned after **&key**, either by accepting them explicitly, by specifying **&allow-other-keys**, or by specifying **&rest** but not **&key**. Each method can accept additional keyword arguments of its own. The checking of the validity of keyword names is done in the generic function, not in each method. A method is invoked as if the keyword argument pair whose name is `:allow-other-keys` and whose value is *true* were supplied, though no such argument pair will be passed.

5. The use of **&allow-other-keys** need not be consistent across *lambda lists*. If **&allow-other-keys** is mentioned in the *lambda list* of any applicable *method* or of the *generic function*, any keyword arguments may be mentioned in the call to the *generic function*.

6. The use of **&aux** need not be consistent across methods.

   If a *method-defining operator* that cannot specify *generic function* options creates a *generic function*, and if the *lambda list* for the method mentions keyword arguments, the *lambda list* of the generic function will mention **&key** (but no keyword arguments).

7.6.5 Keyword Arguments in Generic Functions and Methods

When a generic function or any of its methods mentions **&key** in a *lambda list*, the specific set of keyword arguments accepted by the generic function varies according to the applicable methods. The set of keyword arguments accepted by the generic function for a particular call is the union of the keyword arguments accepted by all applicable methods and the keyword arguments mentioned after **&key** in the generic function definition, if any. A method that has **&rest** but not **&key** does not affect the set of acceptable keyword arguments. If the *lambda list* of any applicable method or of the generic function definition contains **&allow-other-keys**, all keyword arguments are accepted by the generic function.

The *lambda list* congruence rules require that each method accept all of the keyword arguments mentioned after **&key** in the generic function definition, by accepting them explicitly, by specifying **&allow-other-keys**, or by specifying **&rest** but not **&key**. Each method can accept additional keyword arguments of its own, in addition to the keyword arguments mentioned in the generic function definition.

If a *generic function* is passed a keyword argument that no applicable method accepts, an error should be signaled; see Section 3.5 (Error Checking in Function Calls).

7.6.5.1 Examples of Keyword Arguments in Generic Functions and Methods

For example, suppose there are two methods defined for `width` as follows:

```
(defmethod width ((c character-class) &key font) ...)

(defmethod width ((p picture-class) &key pixel-size) ...)
```

Assume that there are no other methods and no generic function definition for `width`. The evaluation of the following form should signal an error because the keyword argument `:pixel-size` is not accepted by the applicable method.

```
(width (make-instance ‘character-class :char #\Q)
       :font 'baskerville :pixel-size 10)
```

The evaluation of the following form should signal an error.

```
(width (make-instance ‘picture-class :glyph (glyph #\Q))
       :font 'baskerville :pixel-size 10)
```

The evaluation of the following form will not signal an error if the class named `character-picture-class` is a subclass of both `picture-class` and `character-class`.

```
(width (make-instance ‘character-picture-class :char #\Q)
       :font 'baskerville :pixel-size 10)
```

7.6.6 Method Selection and Combination

When a *generic function* is called with particular arguments, it must determine the code to execute. This code is called the ***effective method*** for those *arguments*. The *effective method* is a combination of the *applicable methods* in the *generic function* that *calls* some or all of the *methods*.

If a *generic function* is called and no *methods* are *applicable*, the *generic function* **no-applicable-method** is invoked, with the *results* from that call being used as the *results* of the call to the original *generic function*. Calling **no-applicable-method** takes precedence over checking for acceptable keyword arguments; see Section 7.6.5 (Keyword Arguments in Generic Functions and Methods).

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

7.6.6.1 Determining the Effective Method

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

1. Select the applicable methods.

2. Sort the applicable methods by precedence order, putting the most specific method first.

3. Apply method combination to the sorted list of applicable methods, producing the effective method.

7.6.6.1.1 Selecting the Applicable Methods

This step is described in Section 7.6.2 (Introduction to Methods).

7.6.6.1.2 Sorting the Applicable Methods by Precedence Order

To compare the precedence of two methods, their *parameter specializers* are examined in order. The default examination order is from left to right, but an alternative order may be specified by the `:argument-precedence-order` option to **defgeneric** or to any of the other operators that specify generic function options.

The corresponding *parameter specializers* from each method are compared. When a pair of *parameter specializers* agree, the next pair are compared for agreement. If all corresponding parameter specializers agree, the two methods must have different *qualifiers*; in this case, either method can be selected to precede the other. For information about agreement, see Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers).

If some corresponding *parameter specializers* do not agree, the first pair of *parameter specializers* that do not agree determines the precedence. If both *parameter specializers* are classes, the more specific of the two methods is the method whose *parameter specializer* appears earlier in the *class precedence list* of the corresponding argument. Because of the way in which the set of applicable methods is chosen, the *parameter specializers* are guaranteed to be present in the class precedence list of the class of the argument.

If just one of a pair of corresponding *parameter specializers* is `(eql` *object*`)`, the *method* with that *parameter specializer* precedes the other *method*. If both *parameter specializers* are **eql** *expressions*, the specializers must agree (otherwise the two *methods* would not both have been applicable to this argument).

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

7.6.6.1.3 Applying method combination to the sorted list of applicable methods

In the simple case—if standard method combination is used and all applicable methods are primary methods—the effective method is the most specific method. That method can call the next most specific method by using the *function* **call-next-method**. The method that **call-next-method** will call is referred to as the ***next method***. The predicate **next-method-p** tests whether a next method exists. If **call-next-method** is called and there is no next most specific method, the generic function **no-next-method** is invoked.

In general, the effective method is some combination of the applicable methods. It is described by a *form* that contains calls to some or all of the applicable methods, returns the value or values that will be returned as the value or values of the generic function, and optionally makes some of the methods accessible by means of **call-next-method**.

The role of each method in the effective method is determined by its *qualifiers* and the specificity of the method. A *qualifier* serves to mark a method, and the meaning of a *qualifier* is determined by the way that these marks are used by this step of the procedure. If an applicable method has an unrecognized *qualifier*, this step signals an error and does not include that method in the effective method.

When standard method combination is used together with qualified methods, the effective method is produced as described in Section 7.6.6.2 (Standard Method Combination).

Another type of method combination can be specified by using the `:method-combination` option of **defgeneric** or of any of the other operators that specify generic function options. In this way this step of the procedure can be customized.

New types of method combination can be defined by using the **define-method-combination** *macro*.

7.6.6.2 Standard Method Combination

Standard method combination is supported by the *class* **standard-generic-function**. It is used if no other type of method combination is specified or if the built-in method combination type **standard** is specified.

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

An auxiliary method is a method whose *qualifier* is `:before`, `:after`, or `:around`. Standard method combination allows no more than one *qualifier* per method; if a method definition specifies more than one *qualifier* per method, an error is signaled.

- A *before method* has the keyword `:before` as its only *qualifier*. A *before method* specifies *code* that is to be run before any *primary methods*.

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

- An *around method* has the keyword `:around` as its only *qualifier*. An *around method* specifies *code* that is to be run instead of other *applicable methods*, but which might contain explicit *code* which calls some of those *shadowed methods* (via **call-next-method**).

The semantics of standard method combination is as follows:

- If there are any *around methods*, the most specific *around method* is called. It supplies the value or values of the generic function.

- Inside the body of an *around method*, **call-next-method** can be used to call the *next method*. When the next method returns, the *around method* can execute more code, perhaps based on the returned value or values. The *generic function* **no-next-method** is invoked if **call-next-method** is used and there is no *applicable method* to call. The *function* **next-method-p** may be used to determine whether a *next method* exists.

- If an *around method* invokes **call-next-method**, the next most specific *around method* is called, if one is applicable. If there are no *around methods* or if **call-next-method** is called by the least specific *around method*, the other methods are called as follows:

  – All the *before methods* are called, in most-specific-first order. Their values are ignored. An error is signaled if **call-next-method** is used in a *before method*.

  – The most specific primary method is called. Inside the body of a primary method, **call-next-method** may be used to call the next most specific primary method. When that method returns, the previous primary method can execute more code, perhaps based on the returned value or values. The generic function **no-next-method** is invoked if **call-next-method** is used and there are no more applicable primary methods. The *function* **next-method-p** may be used to determine whether a *next method* exists. If **call-next-method** is not used, only the most specific *primary method* is called.

  – All the *after methods* are called in most-specific-last order. Their values are ignored. An error is signaled if **call-next-method** is used in an *after method*.

- If no *around methods* were invoked, the most specific primary method supplies the value or values returned by the generic function. The value or values returned by the invocation of **call-next-method** in the least specific *around method* are those returned by the most specific primary method.

In standard method combination, if there is an applicable method but no applicable primary method, an error is signaled.

The *before methods* are run in most-specific-first order while the *after methods* are run in least-specific-first order. The design rationale for this difference can be illustrated with an example. Suppose class C_1 modifies the behavior of its superclass, C_2, by adding *before methods* and *after methods*. Whether the behavior of the class C_2 is defined directly by methods on C_2 or is inherited from its superclasses does not affect the relative order of invocation of methods on instances of the class C_1. Class C_1's *before method* runs before all of class C_2's methods. Class C_1's *after method* runs after all of class C_2's methods.

By contrast, all *around methods* run before any other methods run. Thus a less specific *around method* runs before a more specific primary method.

If only primary methods are used and if **call-next-method** is not used, only the most specific method is invoked; that is, more specific methods shadow more general ones.

7.6.6.3 Declarative Method Combination

The macro **define-method-combination** defines new forms of method combination. It provides a mechanism for customizing the production of the effective method. The default procedure for producing an effective method is described in Section 7.6.6.1 (Determining the Effective Method).

There are two forms of **define-method-combination**. The short form is a simple facility while the long form is more powerful and more verbose. The long form resembles **defmacro** in that the body is an expression that computes a Lisp form; it provides mechanisms for implementing arbitrary control structures within method combination and for arbitrary processing of method *qualifiers*.

7.6.6.4 Built-in Method Combination Types

The object system provides a set of built-in method combination types. To specify that a generic function is to use one of these method combination types, the name of the method combination type is given as the argument to the `:method-combination` option to **defgeneric** or to the `:method-combination` option to any of the other operators that specify generic function options.

The names of the built-in method combination types are listed in Figure 7–2.

+	**append**	**max**	**nconc**	**progn**
and	**list**	**min**	**or**	**standard**

Figure 7–2. Built-in Method Combination Types

The semantics of the **standard** built-in method combination type is described in Section 7.6.6.2 (Standard Method Combination). The other built-in method combination types are called simple built-in method combination types.

The simple built-in method combination types act as though they were defined by the short form of **define-method-combination**. They recognize two roles for *methods*:

- An *around method* has the keyword symbol `:around` as its sole *qualifier*. The meaning of `:around` *methods* is the same as in standard method combination. Use of the functions **call-next-method** and **next-method-p** is supported in *around methods*.

- A primary method has the name of the method combination type as its sole *qualifier*. For example, the built-in method combination type `and` recognizes methods whose sole *qualifier* is `and`; these are primary methods. Use of the functions **call-next-method** and **next-method-p** is not supported in *primary methods*.

The semantics of the simple built-in method combination types is as follows:

- If there are any *around methods*, the most specific *around method* is called. It supplies the value or values of the *generic function*.

- Inside the body of an *around method*, the function **call-next-method** can be used to call the *next method*. The *generic function* **no-next-method** is invoked if **call-next-method** is used and there is no applicable method to call. The *function* **next-method-p** may be used to determine whether a *next method* exists. When the *next method* returns, the *around method* can execute more code, perhaps based on the returned value or values.

- If an *around method* invokes **call-next-method**, the next most specific *around method* is called, if one is applicable. If there are no *around methods* or if **call-next-method** is called by the least specific *around method*, a Lisp form derived from the name of the built-in method combination type and from the list of applicable primary methods is evaluated to produce the value of the generic function. Suppose the name of the method combination type is *operator* and the call to the generic function is of the form

  $$(\textit{generic-function}\ a_1 \ldots a_n)$$

  Let $M_1, \ldots, M_k$ be the applicable primary methods in order; then the derived Lisp form is

  $$(\textit{operator}\ \langle M_1\ a_1 \ldots a_n \rangle \ldots \langle M_k\ a_1 \ldots a_n \rangle)$$

  If the expression $\langle M_i\ a_1 \ldots a_n \rangle$ is evaluated, the method M_i will be applied to the arguments $a_1 \ldots a_n$. For example, if *operator* is `or`, the expression $\langle M_i\ a_1 \ldots a_n \rangle$ is evaluated only if $\langle M_j\ a_1 \ldots a_n \rangle$, $1 \le j < i$, returned `nil`.

  The default order for the primary methods is `:most-specific-first`. However, the order can be reversed by supplying `:most-specific-last` as the second argument to the `:method-combination` option.

The simple built-in method combination types require exactly one *qualifier* per method. An error is signaled if there are applicable methods with no *qualifiers* or with *qualifiers* that are not supported by the method combination type. An error is signaled if there are applicable *around methods* and no applicable primary methods.

7.6.7 Inheritance of Methods

A subclass inherits methods in the sense that any method applicable to all instances of a class is also applicable to all instances of any subclass of that class.

The inheritance of methods acts the same way regardless of which of the *method-defining operators* created the methods.

The inheritance of methods is described in detail in Section 7.6.6 (Method Selection and Combination).

function-keywords *Standard Generic Function*

Syntax:

function-keywords *method* → *keys, allow-other-keys-p*

Method Signatures:

function-keywords (*method* **standard-method**)

Arguments and Values:

method—a *method*.

keys—a *list*.

allow-other-keys-p—a *generalized boolean*.

Description:

Returns the keyword parameter specifiers for a *method*.

Two values are returned: a *list* of the explicitly named keywords and a *generalized boolean* that states whether **&allow-other-keys** had been specified in the *method* definition.

Examples:

```
 (defmethod gf1 ((a integer) &optional (b 2)
                 &key (c 3) ((:dee d) 4) e ((eff f)))
   (list a b c d e f))
→ #<STANDARD-METHOD GF1 (INTEGER) 36324653>
 (find-method #'gf1 '() (list (find-class 'integer)))
→ #<STANDARD-METHOD GF1 (INTEGER) 36324653>
 (function-keywords *)
→ (:C :DEE :E EFF), false
 (defmethod gf2 ((a integer))
   (list a b c d e f))
→ #<STANDARD-METHOD GF2 (INTEGER) 42701775>
 (function-keywords (find-method #'gf1 '() (list (find-class 'integer))))
→ (), false
 (defmethod gf3 ((a integer) &key b c d &allow-other-keys)
   (list a b c d e f))
 (function-keywords *)
→ (:B :C :D), true
```

Affected By:

defmethod

See Also:

defmethod

ensure-generic-function *Function*

Syntax:

ensure-generic-function *function-name* `&key` *argument-precedence-order declare documentation environment generic-function-class lambda-list method-class method-combination*

→ *generic-function*

Arguments and Values:

function-name—a *function name*.

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

`Method-combination` – method combination object.

`Environment` – the same as the **&environment** argument to macro expansion functions and is used to distinguish between compile-time and run-time environments.

generic-function—a *generic function object*.

Description:

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

If *function-name* is not *fbound* in the *global environment*, a new *generic function* is created. If `(fdefinition` *function-name*`)` is an *ordinary function*, a *macro*, or a *special operator*, an error is signaled.

If *function-name* is a *list*, it must be of the form `(setf` *symbol*`)`. If *function-name* specifies a *generic function* that has a different value for any of the following arguments, the *generic function* is modified to have the new value: `:argument-precedence-order`, `:declare`, `:documentation`, `:method-combination`.

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

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

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

Affected By:

Existing function binding of *function-name*.

Exceptional Situations:

If `(fdefinition` *function-name*`)` is an *ordinary function*, a *macro*, or a *special operator*, an error of *type* **error** is signaled.

If *function-name* specifies a *generic function* that has a different value for the `:lambda-list` argument, and the new value is not congruent with the *lambda list* of any existing *method*, an error of *type* **error** is signaled.

If *function-name* specifies a *generic function* that has a different value for the `:generic-function-class` argument and if the new generic function class not is compatible with the old, an error of *type* **error** is signaled.

See Also:

defgeneric

allocate-instance *Standard Generic Function*

Syntax:

allocate-instance *class* `&rest` *initargs* `&key` `&allow-other-keys` → *new-instance*

Method Signatures:

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

allocate-instance (*class* **structure-class**) `&rest` *initargs*

Arguments and Values:

class—a *class*.

initargs—a *list* of *keyword/value pairs* (initialization argument *names* and *values*).

new-instance—an *object* whose *class* is *class*.

Description:

The generic function **allocate-instance** creates and returns a new instance of the *class*, without initializing it. When the *class* is a *standard class*, this means that the *slots* are *unbound*; when the *class* is a *structure class*, this means the *slots' values* are unspecified.

The caller of **allocate-instance** is expected to have already checked the initialization arguments.

The *generic function* **allocate-instance** is called by **make-instance**, as described in Section 7.1 (Object Creation and Initialization).

See Also:

defclass, **make-instance**, **class-of**, Section 7.1 (Object Creation and Initialization)

Notes:

The consequences of adding *methods* to **allocate-instance** is unspecified. This capability might be added by the *Metaobject Protocol*.

reinitialize-instance *Standard Generic Function*

Syntax:

reinitialize-instance *instance* `&rest` *initargs* `&key` `&allow-other-keys` → *instance*

Method Signatures:

reinitialize-instance (*instance* **standard-object**) `&rest` *initargs*

Arguments and Values:

instance—an *object*.

initargs—an *initialization argument list*.

Description:

The *generic function* **reinitialize-instance** can be used to change the values of *local slots* of an *instance* according to *initargs*. This *generic function* can be called by users.

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

Side Effects:

The *generic function* **reinitialize-instance** changes the values of *local slots*.

Exceptional Situations:

The system-supplied primary *method* for **reinitialize-instance** signals an error if an *initarg* is supplied that is not declared as valid.

See Also:

initialize-instance, **shared-initialize**, **update-instance-for-redefined-class**, **update-instance-for-different-class**, **slot-boundp**, **slot-makunbound**, Section 7.3 (Reinitializing an Instance), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

Notes:

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

shared-initialize *Standard Generic Function*

Syntax:

shared-initialize *instance slot-names* `&rest` *initargs* `&key` `&allow-other-keys` → *instance*

Method Signatures:

shared-initialize (*instance* **standard-object**) *slot-names* `&rest` *initargs*

Arguments and Values:

instance—an *object*.

slot-names—a *list* or `t`.

initargs—a *list* of *keyword/value pairs* (of initialization argument *names* and *values*).

Description:

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

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

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

- If an *initarg* in the *initialization argument list* specifies a value for that *slot*, that value is stored into the *slot*, even if a value has already been stored in the *slot* before the *method* is run.

- Any *slots* indicated by *slot-names* that are still unbound at this point are initialized according to their `:initform` forms. For any such *slot* that has an `:initform` form, that *form* is evaluated in the lexical environment of its defining **defclass** *form* and the result is stored into the *slot*. For example, if a *before method* stores a value in the *slot*, the `:initform` form will not be used to supply a value for the *slot*.

- The rules mentioned in Section 7.1.4 (Rules for Initialization Arguments) are obeyed.

The *slots-names* argument specifies the *slots* that are to be initialized according to their `:initform` forms if no initialization arguments apply. It can be a *list* of slot *names*, which specifies the set of those slot *names*; or it can be the *symbol* **t**, which specifies the set of all of the *slots*.

See Also:

initialize-instance, **reinitialize-instance**, **update-instance-for-redefined-class**, **update-instance-for-different-class**, **slot-boundp**, **slot-makunbound**, Section 7.1 (Object Creation and Initialization), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

Notes:

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

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

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

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

Syntax:

update-instance-for-different-class *previous current* `&rest` *initargs* `&key &allow-other-keys`
→ *implementation-dependent*

Method Signatures:

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

Arguments and Values:

previous—a copy of the original *instance*.

current—the original *instance* (altered).

initargs—an *initialization argument list*.

Description:

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

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

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

Methods on **update-instance-for-different-class** can be defined to initialize *slots* differently from **change-class**. The default behavior of **change-class** is described in Section 7.2 (Changing the Class of an Instance).

The arguments to **update-instance-for-different-class** are computed by **change-class**. When **change-class** is invoked on an *instance*, a copy of that *instance* is made; **change-class** then destructively alters the original *instance*. The first argument to **update-instance-for-different-class**, *previous*, is that copy; it holds the old *slot* values temporarily. This argument has dynamic extent within **change-class**; if it is referenced in any way once **update-instance-for-different-class** returns, the results are undefined. The second argument to **update-instance-for-different-class**, *current*, is the altered original *instance*. The intended use of *previous* is to extract old *slot* values by using **slot-value** or **with-slots** or by invoking a reader generic function, or to run other *methods* that were applicable to *instances* of the original *class*.

Examples:

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

Exceptional Situations:

The system-supplied primary *method* on **update-instance-for-different-class** signals an error if an initialization argument is supplied that is not declared as valid.

See Also:

change-class, **shared-initialize**, Section 7.2 (Changing the Class of an Instance), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

Notes:

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

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

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

Syntax:

update-instance-for-redefined-class *instance*
added-slots discarded-slots
property-list
`&rest` *initargs* `&key` `&allow-other-keys`

$\rightarrow$ {*result*}*

update-instance-for-redefined-class

Method Signatures:

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

Arguments and Values:

instance—an *object*.

added-slots—a *list*.

discarded-slots—a *list*.

property-list—a *list*.

initargs—an *initialization argument list*.

result—an *object*.

Description:

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

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

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

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

Examples:

update-instance-for-redefined-class

```
 (defclass position () ())

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

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

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

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

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

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

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

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

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

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

```
          theta (atan new-y x))
      new-y)))
```

Exceptional Situations:

The system-supplied primary *method* on **update-instance-for-redefined-class** signals an error if an *initarg* is supplied that is not declared as valid.

See Also:

make-instances-obsolete, **shared-initialize**, Section 4.3.6 (Redefining Classes), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

Notes:

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

change-class *Standard Generic Function*

Syntax:

change-class *instance new-class* `&key` `&allow-other-keys` → *instance*

Method Signatures:

change-class (*instance* **standard-object**) (*new-class* **standard-class**) `&rest` *initargs*

change-class (*instance* **t**) (*new-class* **symbol**) `&rest` *initargs*

Arguments and Values:

instance—an *object*.

new-class—a *class designator*.

initargs—an *initialization argument list*.

Description:

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

change-class

If in the old *class* there is any *slot* of the same name as a local *slot* in the *new-class*, the value of that *slot* is retained. This means that if the *slot* has a value, the value returned by **slot-value** after **change-class** is invoked is **eql** to the value returned by **slot-value** before **change-class** is invoked. Similarly, if the *slot* was unbound, it remains unbound. The other *slots* are initialized as described in Section 7.2 (Changing the Class of an Instance).

After completing all other actions, **change-class** invokes **update-instance-for-different-class**. The generic function **update-instance-for-different-class** can be used to assign values to slots in the transformed instance. See Section 7.2.2 (Initializing Newly Added Local Slots).

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

Examples:

```
 (defclass position () ())

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

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

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

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

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

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

;;; The result is that the instance bound to p1 is now an instance of
;;; the class rho-theta-position.   The update-instance-for-different-class
;;; method performed the initialization of the rho and theta slots based
```

```
;;; on the value of the x and y slots, which were maintained by
;;; the old instance.
```

See Also:

update-instance-for-different-class, Section 7.2 (Changing the Class of an Instance)

Notes:

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

slot-boundp *Function*

Syntax:

slot-boundp *instance slot-name* → *generalized-boolean*

Arguments and Values:

instance—an *object*.

slot-name—a *symbol* naming a *slot* of *instance*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if the *slot* named *slot-name* in *instance* is bound; otherwise, returns *false*.

Exceptional Situations:

If no *slot* of the *name* *slot-name* exists in the *instance*, **slot-missing** is called as follows:

```
(slot-missing (class-of instance)
              instance
              slot-name
              'slot-boundp)
```

(If **slot-missing** is invoked and returns a value, a *boolean equivalent* to its *primary value* is returned by **slot-boundp**.)

The specific behavior depends on *instance*'s *metaclass*. An error is never signaled if *instance* has *metaclass* **standard-class**. An error is always signaled if *instance* has *metaclass* **built-in-class**. The consequences are undefined if *instance* has any other *metaclass*–an error might or might not be signaled in this situation. Note in particular that the behavior for *conditions* and *structures* is not specified.

See Also:

slot-makunbound, **slot-missing**

Notes:

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

Although no *implementation* is required to do so, implementors are strongly encouraged to implement the *function* **slot-boundp** using the *function* `slot-boundp-using-class` described in the *Metaobject Protocol*.

slot-exists-p *Function*

Syntax:

slot-exists-p *object slot-name* → *generalized-boolean*

Arguments and Values:

object—an *object*.

slot-name—a *symbol*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if the *object* has a *slot* named *slot-name*.

Affected By:

defclass, **defstruct**

See Also:

defclass, **slot-missing**

Notes:

Although no *implementation* is required to do so, implementors are strongly encouraged to implement the *function* **slot-exists-p** using the *function* `slot-exists-p-using-class` described in the *Metaobject Protocol*.

slot-makunbound *Function*

Syntax:

slot-makunbound *instance slot-name* → *instance*

Arguments and Values:

instance – instance.

Slot-name—a *symbol*.

Description:

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

Exceptional Situations:

If no *slot* of the name *slot-name* exists in the *instance*, **slot-missing** is called as follows:

```
(slot-missing (class-of instance)
              instance
              slot-name
              'slot-makunbound)
```

(Any values returned by **slot-missing** in this case are ignored by **slot-makunbound**.)

The specific behavior depends on *instance*'s *metaclass*. An error is never signaled if *instance* has *metaclass* **standard-class**. An error is always signaled if *instance* has *metaclass* **built-in-class**. The consequences are undefined if *instance* has any other *metaclass*–an error might or might not be signaled in this situation. Note in particular that the behavior for *conditions* and *structures* is not specified.

See Also:

slot-boundp, **slot-missing**

Notes:

Although no *implementation* is required to do so, implementors are strongly encouraged to implement the *function* **slot-makunbound** using the *function* `slot-makunbound-using-class` described in the *Metaobject Protocol*.

slot-missing *Standard Generic Function*

Syntax:

slot-missing *class object slot-name operation* `&optional` *new-value* → {*result*}*

Method Signatures:

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

Arguments and Values:

class—the *class* of *object*.

object—an *object*.

slot-name—a *symbol* (the *name* of a would-be *slot*).

operation—one of the *symbols* **setf**, **slot-boundp**, **slot-makunbound**, or **slot-value**.

new-value—an *object*.

result—an *object*.

Description:

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

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

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

The optional *new-value* argument to **slot-missing** is used when the operation is attempting to set the value of the *slot*.

If **slot-missing** returns, its values will be treated as follows:

- If the *operation* is **setf** or **slot-makunbound**, any *values* will be ignored by the caller.
- If the *operation* is **slot-value**, only the *primary value* will be used by the caller, and all other values will be ignored.

- If the *operation* is **slot-boundp**, any *boolean equivalent* of the *primary value* of the *method* might be is used, and all other values will be ignored.

Exceptional Situations:

The default *method* on **slot-missing** signals an error of *type* **error**.

See Also:

defclass, **slot-exists-p**, **slot-value**

Notes:

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

slot-unbound *Standard Generic Function*

Syntax:

slot-unbound *class instance slot-name* → {*result*}*

Method Signatures:

slot-unbound (*class* **t**) *instance slot-name*

Arguments and Values:

class—the *class* of the *instance*.

instance—the *instance* in which an attempt was made to *read* the *unbound slot*.

slot-name—the *name* of the *unbound slot*.

result—an *object*.

Description:

The generic function **slot-unbound** is called when an unbound *slot* is read in an *instance* whose metaclass is **standard-class**. The default *method* signals an error of *type* **unbound-slot**. The name slot of the **unbound-slot** *condition* is initialized to the name of the offending variable, and the instance slot of the **unbound-slot** *condition* is initialized to the offending instance.

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

If **slot-unbound** returns, only the *primary value* will be used by the caller, and all other values will be ignored.

Exceptional Situations:

The default *method* on **slot-unbound** signals an error of *type* **unbound-slot**.

See Also:

slot-makunbound

Notes:

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

slot-value *Function*

Syntax:

slot-value *object slot-name* → *value*

Arguments and Values:

object—an *object*.

name—a *symbol*.

value—an *object*.

Description:

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

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

Examples:

```
 (defclass foo ()
   ((a :accessor foo-a :initarg :a :initform 1)
    (b :accessor foo-b :initarg :b)
    (c :accessor foo-c :initform 3)))
→ #<STANDARD-CLASS FOO 244020371>
 (setq foo1 (make-instance 'foo :a 'one :b 'two))
→ #<FOO 36325624>
 (slot-value foo1 'a) → ONE
 (slot-value foo1 'b) → TWO
 (slot-value foo1 'c) → 3
 (setf (slot-value foo1 'a) 'uno) → UNO
 (slot-value foo1 'a) → UNO
 (defmethod foo-method ((x foo))
```

```
   (slot-value x 'a))
→ #<STANDARD-METHOD FOO-METHOD (FOO) 42720573>
 (foo-method foo1) → UNO
```

Exceptional Situations:

If an attempt is made to read a *slot* and no *slot* of the name *slot-name* exists in the *object*, **slot-missing** is called as follows:

```
(slot-missing (class-of instance)
              instance
              slot-name
              'slot-value)
```

(If **slot-missing** is invoked, its *primary value* is returned by **slot-value**.)

If an attempt is made to write a *slot* and no *slot* of the name *slot-name* exists in the *object*, **slot-missing** is called as follows:

```
(slot-missing (class-of instance)
              instance
              slot-name
              'setf
              new-value)
```

(If **slot-missing** returns in this case, any *values* are ignored.)

The specific behavior depends on *object*'s *metaclass*. An error is never signaled if *object* has *metaclass* **standard-class**. An error is always signaled if *object* has *metaclass* **built-in-class**. The consequences are unspecified if *object* has any other *metaclass*–an error might or might not be signaled in this situation. Note in particular that the behavior for *conditions* and *structures* is not specified.

See Also:

slot-missing, **slot-unbound**, **with-slots**

Notes:

Although no *implementation* is required to do so, implementors are strongly encouraged to implement the *function* **slot-value** using the *function* `slot-value-using-class` described in the *Metaobject Protocol*.

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

method-qualifiers *Standard Generic Function*

Syntax:

method-qualifiers *method* → *qualifiers*

Method Signatures:

method-qualifiers (*method* **standard-method**)

Arguments and Values:

method—a *method*.

qualifiers—a *proper list*.

Description:

Returns a *list* of the *qualifiers* of the *method*.

Examples:

```
 (defmethod some-gf :before ((a integer)) a)
→ #<STANDARD-METHOD SOME-GF (:BEFORE) (INTEGER) 42736540>
 (method-qualifiers *) → (:BEFORE)
```

See Also:

define-method-combination

no-applicable-method *Standard Generic Function*

Syntax:

no-applicable-method *generic-function* &rest *function-arguments* → {*result*}*

Method Signatures:

no-applicable-method (*generic-function* **t**)
&rest *function-arguments*

Arguments and Values:

generic-function—a *generic function* on which no *applicable method* was found.

function-arguments—*arguments* to the *generic-function*.

result—an *object*.

Description:

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

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

Exceptional Situations:

The default *method* signals an error of *type* **error**.

See Also:

no-next-method *Standard Generic Function*

Syntax:

no-next-method *generic-function method* &rest *args* → {*result*}*

Method Signatures:

no-next-method (*generic-function* **standard-generic-function**)
(*method* **standard-method**)
&rest *args*

Arguments and Values:

generic-function – *generic function* to which *method* belongs.

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

args – arguments to **call-next-method**.

result—an *object*.

Description:

The *generic function* **no-next-method** is called by **call-next-method** when there is no *next method*.

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

Exceptional Situations:

The system-supplied *method* on **no-next-method** signals an error of *type* **error**.

See Also:

call-next-method

remove-method *Standard Generic Function*

Syntax:

remove-method *generic-function method* → *generic-function*

Method Signatures:

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

Arguments and Values:

generic-function—a *generic function*.

method—a *method*.

Description:

The *generic function* **remove-method** removes a *method* from *generic-function* by modifying the *generic-function* (if necessary).

remove-method must not signal an error if the *method* is not one of the *methods* on the *generic-function*.

See Also:

find-method

make-instance *Standard Generic Function*

Syntax:

make-instance *class* `&rest` *initargs* `&key` `&allow-other-keys` → *instance*

Method Signatures:

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

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

Arguments and Values:

class—a *class*, or a *symbol* that names a *class*.

initargs—an *initialization argument list*.

instance—a *fresh instance* of *class* *class*.

Description:

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

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

The initialization arguments are checked within **make-instance**.

The *generic function* **make-instance** may be used as described in Section 7.1 (Object Creation and Initialization).

Exceptional Situations:

If any of the initialization arguments has not been declared as valid, an error of *type* **error** is signaled.

See Also:

defclass, **class-of**, **allocate-instance**, **initialize-instance**, Section 7.1 (Object Creation and Initialization)

make-instances-obsolete *Standard Generic Function*

Syntax:

make-instances-obsolete *class* → *class*

Method Signatures:

make-instances-obsolete (*class* **standard-class**)

make-instances-obsolete (*class* **symbol**)

Arguments and Values:

class—a *class designator*.

Description:

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

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

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

Examples:

See Also:

update-instance-for-redefined-class, Section 4.3.6 (Redefining Classes)

make-load-form *Standard Generic Function*

Syntax:

make-load-form *object* `&optional` *environment* → *creation-form*[, *initialization-form*]

Method Signatures:

make-load-form (*object* **standard-object**) `&optional` *environment*

make-load-form (*object* **structure-object**) `&optional` *environment*

make-load-form (*object* **condition**) `&optional` *environment*

make-load-form (*object* **class**) `&optional` *environment*

Arguments and Values:

object—an *object*.

environment—an *environment object*.

creation-form—a *form*.

initialization-form—a *form*.

Description:

The *generic function* **make-load-form** creates and returns one or two *forms*, a *creation-form* and an *initialization-form*, that enable **load** to construct an *object* equivalent to *object*. *Environment* is an *environment object* corresponding to the *lexical environment* in which the *forms* will be processed.

The *file compiler* calls **make-load-form** to process certain *classes* of *literal objects*; see Section 3.2.4.4 (Additional Constraints on Externalizable Objects).

Conforming programs may call **make-load-form** directly, providing *object* is a *generalized instance* of **standard-object**, **structure-object**, or **condition**.

The creation form is a *form* that, when evaluated at **load** time, should return an *object* that is equivalent to *object*. The exact meaning of equivalent depends on the *type* of *object* and is up to

make-load-form

the programmer who defines a *method* for **make-load-form**; see Section 3.2.4 (Literal Objects in Compiled Files).

The initialization form is a *form* that, when evaluated at **load** time, should perform further initialization of the *object*. The value returned by the initialization form is ignored. If **make-load-form** returns only one value, the initialization form is **nil**, which has no effect. If *object* appears as a constant in the initialization form, at **load** time it will be replaced by the equivalent *object* constructed by the creation form; this is how the further initialization gains access to the *object*.

Both the *creation-form* and the *initialization-form* may contain references to any *externalizable object*. However, there must not be any circular dependencies in creation forms. An example of a circular dependency is when the creation form for the object `X` contains a reference to the object `Y`, and the creation form for the object `Y` contains a reference to the object `X`. Initialization forms are not subject to any restriction against circular dependencies, which is the reason that initialization forms exist; see the example of circular data structures below.

The creation form for an *object* is always *evaluated* before the initialization form for that *object*. When either the creation form or the initialization form references other *objects* that have not been referenced earlier in the *file* being *compiled*, the *compiler* ensures that all of the referenced *objects* have been created before *evaluating* the referencing *form*. When the referenced *object* is of a *type* which the *file compiler* processes using **make-load-form**, this involves *evaluating* the creation form returned for it. (This is the reason for the prohibition against circular references among creation forms).

Each initialization form is *evaluated* as soon as possible after its associated creation form, as determined by data flow. If the initialization form for an *object* does not reference any other *objects* not referenced earlier in the *file* and processed by the *file compiler* using **make-load-form**, the initialization form is evaluated immediately after the creation form. If a creation or initialization form F does contain references to such *objects*, the creation forms for those other objects are evaluated before F, and the initialization forms for those other *objects* are also evaluated before F whenever they do not depend on the *object* created or initialized by F. Where these rules do not uniquely determine an order of *evaluation* between two creation/initialization forms, the order of *evaluation* is unspecified.

While these creation and initialization forms are being evaluated, the *objects* are possibly in an uninitialized state, analogous to the state of an *object* between the time it has been created by **allocate-instance** and it has been processed fully by **initialize-instance**. Programmers writing *methods* for **make-load-form** must take care in manipulating *objects* not to depend on *slots* that have not yet been initialized.

It is *implementation-dependent* whether **load** calls **eval** on the *forms* or does some other operation that has an equivalent effect. For example, the *forms* might be translated into different but equivalent *forms* and then evaluated, they might be compiled and the resulting functions called by **load**, or they might be interpreted by a special-purpose function different from **eval**. All that is required is that the effect be equivalent to evaluating the *forms*.

make-load-form

The *method specialized* on **class** returns a creation *form* using the *name* of the *class* if the *class* has a *proper name* in *environment*, signaling an error of *type* **error** if it does not have a *proper name*. *Evaluation* of the creation *form* uses the *name* to find the *class* with that *name*, as if by *calling* **find-class**. If a *class* with that *name* has not been defined, then a *class* may be computed in an *implementation-defined* manner. If a *class* cannot be returned as the result of *evaluating* the creation *form*, then an error of *type* **error** is signaled.

Both *conforming implementations* and *conforming programs* may further *specialize* **make-load-form**.

Examples:

```
(defclass obj ()
   ((x :initarg :x :reader obj-x)
    (y :initarg :y :reader obj-y)
    (dist :accessor obj-dist)))
→ #<STANDARD-CLASS OBJ 250020030>
(defmethod shared-initialize :after ((self obj) slot-names &rest keys)
  (declare (ignore slot-names keys))
  (unless (slot-boundp self 'dist)
    (setf (obj-dist self)
          (sqrt (+ (expt (obj-x self) 2) (expt (obj-y self) 2))))))
→ #<STANDARD-METHOD SHARED-INITIALIZE (:AFTER) (OBJ T) 26266714>
(defmethod make-load-form ((self obj) &optional environment)
  (declare (ignore environment))
  ;; Note that this definition only works because X and Y do not
  ;; contain information which refers back to the object itself.
  ;; For a more general solution to this problem, see revised example below.
  `(make-instance ',(class-of self)
                  :x ',(obj-x self) :y ',(obj-y self)))
→ #<STANDARD-METHOD MAKE-LOAD-FORM (OBJ) 26267532>
(setq obj1 (make-instance 'obj :x 3.0 :y 4.0)) → #<OBJ 26274136>
(obj-dist obj1) → 5.0
(make-load-form obj1) → (MAKE-INSTANCE 'OBJ :X '3.0 :Y '4.0)
```

In the above example, an equivalent *instance* of `obj` is reconstructed by using the values of two of its *slots*. The value of the third *slot* is derived from those two values.

Another way to write the **make-load-form** *method* in that example is to use **make-load-form-saving-slots**. The code it generates might yield a slightly different result from the **make-load-form** *method* shown above, but the operational effect will be the same. For example:

```
;; Redefine method defined above.
(defmethod make-load-form ((self obj) &optional environment)
   (make-load-form-saving-slots self
```

make-load-form

```
                                :slot-names '(x y)
                                :environment environment))
→ #<STANDARD-METHOD MAKE-LOAD-FORM (OBJ) 42755655>
;; Try MAKE-LOAD-FORM on object created above.
(make-load-form obj1)
→ (ALLOCATE-INSTANCE '#<STANDARD-CLASS OBJ 250020030>),
   (PROGN
     (SETF (SLOT-VALUE '#<OBJ 26274136> 'X) '3.0)
     (SETF (SLOT-VALUE '#<OBJ 26274136> 'Y) '4.0)
     (INITIALIZE-INSTANCE '#<OBJ 26274136>))
```

In the following example, *instances* of `my-frob` are "interned" in some way. An equivalent *instance* is reconstructed by using the value of the name slot as a key for searching existing *objects*. In this case the programmer has chosen to create a new *object* if no existing *object* is found; alternatively an error could have been signaled in that case.

```
(defclass my-frob ()
   ((name :initarg :name :reader my-name)))
(defmethod make-load-form ((self my-frob) &optional environment)
  (declare (ignore environment))
  `(find-my-frob ',(my-name self) :if-does-not-exist :create))
```

In the following example, the data structure to be dumped is circular, because each parent has a list of its children and each child has a reference back to its parent. If **make-load-form** is called on one *object* in such a structure, the creation form creates an equivalent *object* and fills in the children slot, which forces creation of equivalent *objects* for all of its children, grandchildren, etc. At this point none of the parent *slots* have been filled in. The initialization form fills in the parent *slot*, which forces creation of an equivalent *object* for the parent if it was not already created. Thus the entire tree is recreated at **load** time. At compile time, **make-load-form** is called once for each *object* in the tree. All of the creation forms are evaluated, in *implementation-dependent* order, and then all of the initialization forms are evaluated, also in *implementation-dependent* order.

```
(defclass tree-with-parent () ((parent :accessor tree-parent)
                               (children :initarg :children)))
(defmethod make-load-form ((x tree-with-parent) &optional environment)
  (declare (ignore environment))
  (values
    ;; creation form
    `(make-instance ',(class-of x) :children ',(slot-value x 'children))
    ;; initialization form
    `(setf (tree-parent ',x) ',(slot-value x 'parent))))
```

In the following example, the data structure to be dumped has no special properties and an equivalent structure can be reconstructed simply by reconstructing the *slots*' contents.

```
(defstruct my-struct a b c)
```

```
(defmethod make-load-form ((s my-struct) &optional environment)
  (make-load-form-saving-slots s :environment environment))
```

Exceptional Situations:

The *methods specialized* on **standard-object**, **structure-object**, and **condition** all signal an error of *type* **error**.

It is *implementation-dependent* whether *calling* **make-load-form** on a *generalized instance* of a *system class* signals an error or returns creation and initialization *forms*.

See Also:

compile-file, **make-load-form-saving-slots**, Section 3.2.4.4 (Additional Constraints on Externalizable Objects) Section 3.1 (Evaluation), Section 3.2 (Compilation)

Notes:

The *file compiler* calls **make-load-form** in specific circumstances detailed in Section 3.2.4.4 (Additional Constraints on Externalizable Objects).

Some *implementations* may provide facilities for defining new *subclasses* of *classes* which are specified as *system classes*. (Some likely candidates include **generic-function**, **method**, and **stream**). Such *implementations* should document how the *file compiler* processes *instances* of such *classes* when encountered as *literal objects*, and should document any relevant *methods* for **make-load-form**.

make-load-form-saving-slots *Function*

Syntax:

make-load-form-saving-slots *object* `&key` *slot-names environment*
→ *creation-form, initialization-form*

Arguments and Values:

object—an *object*.

slot-names—a *list*.

environment—an *environment object*.

creation-form—a *form*.

initialization-form—a *form*.

Description:

Returns *forms* that, when *evaluated*, will construct an *object* equivalent to *object*, without *executing initialization forms*. The *slots* in the new *object* that correspond to initialized *slots* in *object* are initialized using the values from *object*. Uninitialized *slots* in *object* are not initialized in the new *object*. **make-load-form-saving-slots** works for any *instance* of **standard-object** or **structure-object**.

Slot-names is a *list* of the names of the *slots* to preserve. If *slot-names* is not supplied, its value is all of the *local slots*.

make-load-form-saving-slots returns two values, thus it can deal with circular structures. Whether the result is useful in an application depends on whether the *object*'s *type* and slot contents fully capture the application's idea of the *object*'s state.

Environment is the environment in which the forms will be processed.

See Also:

make-load-form, **make-instance**, **setf**, **slot-value**, **slot-makunbound**

Notes:

make-load-form-saving-slots can be useful in user-written **make-load-form** methods.

When the *object* is an *instance* of **standard-object**, **make-load-form-saving-slots** could return a creation form that *calls* **allocate-instance** and an initialization form that contains *calls* to **setf** of **slot-value** and **slot-makunbound**, though other *functions* of similar effect might actually be used.

with-accessors *Macro*

Syntax:

with-accessors ({*slot-entry*}*) *instance-form* {*declaration*}* {*form*}*
→ {*result*}*

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

Arguments and Values:

variable-name—a *variable name*; not evaluated.

accessor-name—a *function name*; not evaluated.

instance-form—a *form*; evaluated.

declaration—a **declare** *expression*; not evaluated.

forms—an *implicit progn*.

results—the *values* returned by the *forms*.

with-accessors

Description:

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

Examples:

```
 (defclass thing ()
           ((x :initarg :x :accessor thing-x)
            (y :initarg :y :accessor thing-y)))
→ #<STANDARD-CLASS THING 250020173>
 (defmethod (setf thing-x) :before (new-x (thing thing))
   (format t "~&Changing X from ~D to ~D in ~S.~%"
           (thing-x thing) new-x thing))
 (setq thing1 (make-instance 'thing :x 1 :y 2)) → #<THING 43135676>
 (setq thing2 (make-instance 'thing :x 7 :y 8)) → #<THING 43147374>
 (with-accessors ((x1 thing-x) (y1 thing-y))
                 thing1
   (with-accessors ((x2 thing-x) (y2 thing-y))
                   thing2
     (list (list x1 (thing-x thing1) y1 (thing-y thing1)
                 x2 (thing-x thing2) y2 (thing-y thing2))
           (setq x1 (+ y1 x2))
           (list x1 (thing-x thing1) y1 (thing-y thing1)
                 x2 (thing-x thing2) y2 (thing-y thing2))
           (setf (thing-x thing2) (list x1))
           (list x1 (thing-x thing1) y1 (thing-y thing1)
                 x2 (thing-x thing2) y2 (thing-y thing2)))))
▷ Changing X from 1 to 9 in #<THING 43135676>.
▷ Changing X from 7 to (9) in #<THING 43147374>.
→ ((1 1 2 2 7 7 8 8)
     9
     (9 9 2 2 7 7 8 8)
     (9)
     (9 9 2 2 (9) (9) 8 8))
```

Affected By:

defclass

Exceptional Situations:

The consequences are undefined if any *accessor-name* is not the name of an accessor for the *instance*.

See Also:

with-slots, **symbol-macrolet**

Notes:

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

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

expands into the equivalent of

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

where Q_i is

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

with-slots *Macro*

Syntax:

with-slots ({*slot-entry*}*) *instance-form* {*declaration*}* {*form*}*
→ {*result*}*

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

Arguments and Values:

slot-name—a *slot name*; not evaluated.

variable-name—a *variable name*; not evaluated.

instance-form—a *form*; evaluted to produce *instance*.

instance—an *object*.

declaration—a **declare** *expression*; not evaluated.

forms—an *implicit progn*.

results—the *values* returned by the *forms*.

with-slots

Description:

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

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

Examples:

```
 (defclass thing ()
           ((x :initarg :x :accessor thing-x)
            (y :initarg :y :accessor thing-y)))
→ #<STANDARD-CLASS THING 250020173>
 (defmethod (setf thing-x) :before (new-x (thing thing))
   (format t "~&Changing X from ~D to ~D in ~S.~%"
           (thing-x thing) new-x thing))
 (setq thing (make-instance 'thing :x 0 :y 1)) → #<THING 62310540>
 (with-slots (x y) thing (incf x) (incf y)) → 2
 (values (thing-x thing) (thing-y thing)) → 1, 2
 (setq thing1 (make-instance 'thing :x 1 :y 2)) → #<THING 43135676>
 (setq thing2 (make-instance 'thing :x 7 :y 8)) → #<THING 43147374>
 (with-slots ((x1 x) (y1 y))
             thing1
   (with-slots ((x2 x) (y2 y))
               thing2
     (list (list x1 (thing-x thing1) y1 (thing-y thing1)
                 x2 (thing-x thing2) y2 (thing-y thing2))
           (setq x1 (+ y1 x2))
           (list x1 (thing-x thing1) y1 (thing-y thing1)
                 x2 (thing-x thing2) y2 (thing-y thing2))
           (setf (thing-x thing2) (list x1))
           (list x1 (thing-x thing1) y1 (thing-y thing1)
                 x2 (thing-x thing2) y2 (thing-y thing2)))))
▷ Changing X from 7 to (9) in #<THING 43147374>.
→ ((1 1 2 2 7 7 8 8)
    9
    (9 9 2 2 7 7 8 8)
    (9)
    (9 9 2 2 (9) (9) 8 8))
```

Affected By:

defclass

Exceptional Situations:

The consequences are undefined if any *slot-name* is not the name of a *slot* in the *instance*.

See Also:

with-accessors, **slot-value**, **symbol-macrolet**

Notes:

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

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

expands into the equivalent of

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

where Q_i is

`(`*slot-entry*$_i$ () `(slot-value` *in* '*slot-entry*$_i$`))`

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

`(`*variable-name*$_i$ () `(slot-value` *in* '*slot-name*$_i$`))`

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

`(`*variable-name*$_i$ *slot-name*$_i$`)`

defclass *Macro*

Syntax:

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

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

slot-name::= *symbol*

slot-option::= {`:reader` *reader-function-name*}* |
{`:writer` *writer-function-name*}* |

defclass

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

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

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

Arguments and Values:

Class-name—a *non-nil symbol*.

Superclass-name–a *non-nil symbol*.

Slot-name–a *symbol*. The *slot-name* argument is a *symbol* that is syntactically valid for use as a variable name.

Reader-function-name—a *non-nil symbol*. `:reader` can be supplied more than once for a given *slot*.

Writer-function-name—a *generic function* name. `:writer` can be supplied more than once for a given *slot*.

Reader-function-name—a *non-nil symbol*. `:accessor` can be supplied more than once for a given *slot*.

Allocation-type—(member `:instance :class`). `:allocation` can be supplied once at most for a given *slot*.

Initarg-name—a *symbol*. `:initarg` can be supplied more than once for a given *slot*.

Form—a *form*. `:init-form` can be supplied once at most for a given *slot*.

Type-specifier—a *type specifier*. `:type` can be supplied once at most for a given *slot*.

Class-option— refers to the *class* as a whole or to all class *slots*.

Initarg-list—a *list* of alternating initialization argument *names* and default initial value *forms*. `:default-initargs` can be supplied at most once.

Class-name—a *non-nil symbol*. `:metaclass` can be supplied once at most.

new-class—the new *class object*.

Description:

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

defclass

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

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

The *class-name* argument specifies the *proper name* of the new *class*. If a *class* with the same *proper name* already exists and that *class* is an *instance* of **standard-class**, and if the **defclass** form for the definition of the new *class* specifies a *class* of *class* **standard-class**, the existing *class* is redefined, and instances of it (and its *subclasses*) are updated to the new definition at the time that they are next *accessed*. For details, see Section 4.3.6 (Redefining Classes).

Each *superclass-name* argument specifies a direct *superclass* of the new *class*. If the *superclass* list is empty, then the *superclass* defaults depending on the *metaclass*, with **standard-object** being the default for **standard-class**.

The new *class* will inherit *slots* and *methods* from each of its direct *superclasses*, from their direct *superclasses*, and so on. For a discussion of how *slots* and *methods* are inherited, see Section 4.3.4 (Inheritance).

The following slot options are available:

- The `:reader` slot option specifies that an *unqualified method* is to be defined on the *generic function* named *reader-function-name* to read the value of the given *slot*.

- The `:writer` slot option specifies that an *unqualified method* is to be defined on the *generic function* named *writer-function-name* to write the value of the *slot*.

- The `:accessor` slot option specifies that an *unqualified method* is to be defined on the generic function named *reader-function-name* to read the value of the given *slot* and that an *unqualified method* is to be defined on the *generic function* named `(setf` *reader-function-name*`)` to be used with **setf** to modify the value of the *slot*.

- The `:allocation` slot option is used to specify where storage is to be allocated for the given *slot*. Storage for a *slot* can be located in each instance or in the *class object* itself. The value of the *allocation-type* argument can be either the keyword `:instance` or the keyword `:class`. If the `:allocation` slot option is not specified, the effect is the same as specifying `:allocation :instance`.

  - If *allocation-type* is `:instance`, a *local slot* of the name *slot-name* is allocated in each instance of the *class*.

- If *allocation-type* is `:class`, a shared *slot* of the given name is allocated in the *class object* created by this **defclass** form. The value of the *slot* is shared by all *instances* of the *class*. If a class C_1 defines such a *shared slot*, any subclass C_2 of C_1 will share this single *slot* unless the **defclass** form for C_2 specifies a *slot* of the same *name* or there is a superclass of C_2 that precedes C_1 in the class precedence list of C_2 and that defines a *slot* of the same *name*.

- The `:initform` slot option is used to provide a default initial value form to be used in the initialization of the *slot*. This *form* is evaluated every time it is used to initialize the *slot*. The lexical environment in which this *form* is evaluated is the lexical environment in which the **defclass** form was evaluated. Note that the lexical environment refers both to variables and to functions. For *local slots*, the dynamic environment is the dynamic environment in which **make-instance** is called; for shared *slots*, the dynamic environment is the dynamic environment in which the **defclass** form was evaluated. See Section 7.1 (Object Creation and Initialization).

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

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

- The `:type` slot option specifies that the contents of the *slot* will always be of the specified data type. It effectively declares the result type of the reader generic function when applied to an *object* of this *class*. The consequences of attempting to store in a *slot* a value that does not satisfy the type of the *slot* are undefined. The `:type` slot option is further discussed in Section 7.5.3 (Inheritance of Slots and Slot Options).

- The `:documentation` slot option provides a *documentation string* for the *slot*. `:documentation` can be supplied once at most for a given *slot*.

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

- The `:default-initargs` class option is followed by a list of alternating initialization argument *names* and default initial value forms. If any of these initialization arguments does not appear in the initialization argument list supplied to **make-instance**, the corresponding default initial value form is evaluated, and the initialization argument *name* and the *form*'s value are added to the end of the initialization argument list before the instance is created; see Section 7.1 (Object Creation and Initialization). The default initial value form is evaluated each time it is used. The lexical environment in which this *form* is evaluated is the lexical environment in which the **defclass** form was evaluated. The

defclass

dynamic environment is the dynamic environment in which **make-instance** was called. If an initialization argument *name* appears more than once in a `:default-initargs` class option, an error is signaled.

- The `:documentation` class option causes a *documentation string* to be attached with the *class object*, and attached with kind **type** to the *class-name*. `:documentation` can be supplied once at most.

- The `:metaclass` class option is used to specify that instances of the *class* being defined are to have a different metaclass than the default provided by the system (the *class* **standard-class**).

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

- It is not required that the *superclasses* of a *class* be defined before the **defclass** form for that *class* is evaluated.

- All the *superclasses* of a *class* must be defined before an *instance* of the *class* can be made.

- A *class* must be defined before it can be used as a parameter specializer in a **defmethod** form.

The object system can be extended to cover situations where these rules are not obeyed.

Some slot options are inherited by a *class* from its *superclasses*, and some can be shadowed or altered by providing a local slot description. No class options except `:default-initargs` are inherited. For a detailed description of how *slots* and slot options are inherited, see Section 7.5.3 (Inheritance of Slots and Slot Options).

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

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

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

If a **defclass** *form* appears as a *top level form*, the *compiler* must make the *class name* be recognized as a valid *type name* in subsequent declarations (as for **deftype**) and be recognized as a valid *class name* for **defmethod** *parameter specializers* and for use as the `:metaclass` option of a subsequent **defclass**. The *compiler* must make the *class* definition available to be returned by **find-class** when its *environment argument* is a value received as the *environment parameter* of a *macro*.

Exceptional Situations:

If there are any duplicate slot names, an error of *type* **program-error** is signaled.

If an initialization argument *name* appears more than once in `:default-initargs` class option, an error of *type* **program-error** is signaled.

If any of the following slot options appears more than once in a single slot description, an error of *type* **program-error** is signaled: `:allocation`, `:initform`, `:type`, `:documentation`.

It is required that all implementations signal an error of *type* **program-error** if they observe a class option or a slot option that is not implemented locally.

See Also:

documentation, **initialize-instance**, **make-instance**, **slot-value**, Section 4.3 (Classes), Section 4.3.4 (Inheritance), Section 4.3.6 (Redefining Classes), Section 4.3.5 (Determining the Class Precedence List), Section 7.1 (Object Creation and Initialization)

defgeneric *Macro*

Syntax:

defgeneric *function-name gf-lambda-list* ⟦↓*option* | {↓*method-description*}*⟧
→ *new-generic*

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

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

Arguments and Values:

function-name—a *function name*.

generic-function-class—a *non-nil symbol* naming a *class*.

gf-declaration—an **optimize** *declaration specifier*; other *declaration specifiers* are not permitted.

gf-documentation—a *string*; not evaluated.

gf-lambda-list—a *generic function lambda list*.

defgeneric

method-class—a *non-nil symbol* naming a *class*.

method-combination-argument—an *object.*

method-combination-name—a *symbol* naming a *method combination type*.

method-qualifiers, *specialized-lambda-list*, *declarations*, *documentation*, *forms*—as per **defmethod**.

new-generic—the *generic function object*.

parameter-name—a *symbol* that names a *required parameter* in the *lambda-list*. (If the `:argument-precedence-order` option is specified, each *required parameter* in the *lambda-list* must be used exactly once as a *parameter-name*.)

Description:

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

If *function-name* is a *list* it must be of the form `(setf` *symbol*`)`. If `(fboundp` *function-name*`)` is *false*, a new *generic function* is created. If `(fdefinition` *function-name*`)` is a *generic function*, that *generic function* is modified. If *function-name* names an *ordinary function*, a *macro*, or a *special operator*, an error is signaled.

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

Each *method-description* defines a *method* on the *generic function*. The *lambda list* of each *method* must be congruent with the *lambda list* specified by the *gf-lambda-list* option. If no *method* descriptions are specified and a *generic function* of the same name does not already exist, a *generic function* with no *methods* is created.

The *gf-lambda-list* argument of **defgeneric** specifies the shape of *lambda lists* for the *methods* on this *generic function*. All *methods* on the resulting *generic function* must have *lambda lists* that are congruent with this shape. If a **defgeneric** form is evaluated and some *methods* for that *generic function* have *lambda lists* that are not congruent with that given in the **defgeneric** form, an error is signaled. For further details on method congruence, see Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function).

The *generic function* passes to the *method* all the argument values passed to it, and only those; default values are not supported. Note that optional and keyword arguments in method definitions, however, can have default initial value forms and can use supplied-p parameters.

The following options are provided. Except as otherwise noted, a given option may occur only once.

- The `:argument-precedence-order` option is used to specify the order in which the required arguments in a call to the *generic function* are tested for specificity when selecting a particular *method*. Each required argument, as specified in the *gf-lambda-list* argument, must be included exactly once as a *parameter-name* so that the full and unambiguous precedence order is supplied. If this condition is not met, an error is signaled.

- The **declare** option is used to specify declarations that pertain to the *generic function*.

  An **optimize** *declaration specifier* is allowed. It specifies whether method selection should be optimized for speed or space, but it has no effect on *methods*. To control how a *method* is optimized, an **optimize** declaration must be placed directly in the **defmethod** *form* or method description. The optimization qualities **speed** and **space** are the only qualities this standard requires, but an implementation can extend the object system to recognize other qualities. A simple implementation that has only one method selection technique and ignores **optimize** *declaration specifiers* is valid.

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

  The **declare** option may be specified more than once. The effect is the same as if the lists of *declaration specifiers* had been appended together into a single list and specified as a single **declare** option.

- The `:documentation` argument is a *documentation string* to be attached to the *generic function object*, and to be attached with kind **function** to the *function-name*.

- The `:generic-function-class` option may be used to specify that the *generic function* is to have a different *class* than the default provided by the system (the *class* **standard-generic-function**). The *class-name* argument is the name of a *class* that can be the *class* of a *generic function*. If *function-name* specifies an existing *generic function* that has a different value for the `:generic-function-class` argument and the new generic function *class* is compatible with the old, **change-class** is called to change the *class* of the *generic function*; otherwise an error is signaled.

- The `:method-class` option is used to specify that all *methods* on this *generic function* are to have a different *class* from the default provided by the system (the *class* **standard-method**). The *class-name* argument is the name of a *class* that is capable of being the *class* of a *method*.

- The `:method-combination` option is followed by a symbol that names a type of method combination. The arguments (if any) that follow that symbol depend on the type of method combination. Note that the standard method combination type does not support any arguments. However, all types of method combination defined by the short form of **define-method-combination** accept an optional argument named *order*, defaulting to `:most-specific-first`, where a value of `:most-specific-last` reverses the order of the primary *methods* without affecting the order of the auxiliary *methods*.

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

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

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

defgeneric is not required to perform any compile-time side effects. In particular, the *methods* are not installed for invocation during compilation. An *implementation* may choose to store information about the *generic function* for the purposes of compile-time error-checking (such as checking the number of arguments on calls, or noting that a definition for the function name has been seen).

Examples:

Exceptional Situations:

If *function-name* names an *ordinary function*, a *macro*, or a *special operator*, an error of *type* **program-error** is signaled.

Each required argument, as specified in the *gf-lambda-list* argument, must be included exactly once as a *parameter-name*, or an error of *type* **program-error** is signaled.

The *lambda list* of each *method* specified by a *method-description* must be congruent with the *lambda list* specified by the *gf-lambda-list* option, or an error of *type* **error** is signaled.

If a **defgeneric** form is evaluated and some *methods* for that *generic function* have *lambda lists* that are not congruent with that given in the **defgeneric** form, an error of *type* **error** is signaled.

A given *option* may occur only once, or an error of *type* **program-error** is signaled.

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

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

See Also:

defmethod, **documentation**, **ensure-generic-function**, **generic-function**, Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function)

defmethod *Macro*

Syntax:

defmethod *function-name* {*method-qualifier*}* *specialized-lambda-list*
⟦{*declaration*}* | *documentation*⟧ {*form*}*

→ *new-method*

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

method-qualifier::= *non-list*

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

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

Arguments and Values:

declaration—a **declare** *expression*; not evaluated.

documentation—a *string*; not evaluated.

var—a *variable name*.

eql-specializer-form—a *form*.

Form—a *form*.

Initform—a *form*.

Supplied-p-parameter—variable name.

new-method—the new *method object*.

Description:

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

If (fboundp *function-name*) is **nil**, a *generic function* is created with default values for the argument precedence order (each argument is more specific than the arguments to its right in the argument list), for the generic function class (the *class* **standard-generic-function**), for the method class (the *class* **standard-method**), and for the method combination type (the standard method combination type). The *lambda list* of the *generic function* is congruent with the *lambda*

defmethod

list of the *method* being defined; if the **defmethod** form mentions keyword arguments, the *lambda list* of the *generic function* will mention `&key` (but no keyword arguments). If *function-name* names an *ordinary function*, a *macro*, or a *special operator*, an error is signaled.

If a *generic function* is currently named by *function-name*, the *lambda list* of the *method* must be congruent with the *lambda list* of the *generic function*. If this condition does not hold, an error is signaled. For a definition of congruence in this context, see Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function).

Each *method-qualifier* argument is an *object* that is used by method combination to identify the given *method*. The method combination type might further restrict what a method *qualifier* can be. The standard method combination type allows for *unqualified methods* and *methods* whose sole *qualifier* is one of the keywords `:before`, `:after`, or `:around`.

The *specialized-lambda-list* argument is like an ordinary *lambda list* except that the *names* of required parameters can be replaced by specialized parameters. A specialized parameter is a list of the form `(`*var parameter-specializer-name*`)`. Only required parameters can be specialized. If *parameter-specializer-name* is a *symbol* it names a *class*; if it is a *list*, it is of the form `(eql` *eql-specializer-form*`)`. The parameter specializer name `(eql` *eql-specializer-form*`)` indicates that the corresponding argument must be **eql** to the *object* that is the value of *eql-specializer-form* for the *method* to be applicable. The *eql-specializer-form* is evaluated at the time that the expansion of the **defmethod** macro is evaluated. If no *parameter specializer name* is specified for a given required parameter, the *parameter specializer* defaults to the *class* **t**. For further discussion, see Section 7.6.2 (Introduction to Methods).

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

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

If the *generic function* already has a *method* that agrees with the *method* being defined on *parameter specializers* and *qualifiers*, **defmethod** replaces the existing *method* with the one now being defined. For a definition of agreement in this context. see Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers).

The *parameter specializers* are derived from the *parameter specializer names* as described in Section 7.6.2 (Introduction to Methods).

The expansion of the **defmethod** macro "refers to" each specialized parameter (see the description of **ignore** within the description of **declare**). This includes parameters that have an explicit *parameter specializer name* of **t**. This means that a compiler warning does not occur if the body of the *method* does not refer to a specialized parameter, while a warning might occur if the body of the *method* does not refer to an unspecialized parameter. For this reason, a parameter that specializes on **t** is not quite synonymous with an unspecialized parameter in this context.

Declarations at the head of the method body that apply to the method's *lambda variables* are treated as *bound declarations* whose *scope* is the same as the corresponding *bindings*.

Declarations at the head of the method body that apply to the functional bindings of **call-next-method** or **next-method-p** apply to references to those functions within the method body *forms*. Any outer *bindings* of the *function names* **call-next-method** and **next-method-p**, and declarations associated with such *bindings* are *shadowed*$_2$ within the method body *forms*.

The *scope* of *free declarations* at the head of the method body is the entire method body, which includes any implicit local function definitions but excludes *initialization forms* for the *lambda variables*.

defmethod is not required to perform any compile-time side effects. In particular, the *methods* are not installed for invocation during compilation. An *implementation* may choose to store information about the *generic function* for the purposes of compile-time error-checking (such as checking the number of arguments on calls, or noting that a definition for the function name has been seen).

Documentation is attached as a *documentation string* to the *method object*.

Affected By:

The definition of the referenced *generic function*.

Exceptional Situations:

If *function-name* names an *ordinary function*, a *macro*, or a *special operator*, an error of *type* **error** is signaled.

If a *generic function* is currently named by *function-name*, the *lambda list* of the *method* must be congruent with the *lambda list* of the *generic function*, or an error of *type* **error** is signaled.

See Also:

defgeneric, **documentation**, Section 7.6.2 (Introduction to Methods), Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function), Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

find-class *Accessor*

Syntax:

find-class *symbol* `&optional` *errorp environment* → *class*

(**setf** (**find-class** *symbol* `&optional` *errorp environment*) *new-class*)

Arguments and Values:

symbol—a *symbol*.

errorp—a *generalized boolean*. The default is *true*.

environment – same as the **&environment** argument to macro expansion functions and is used to distinguish between compile-time and run-time environments. The **&environment** argument has *dynamic extent*; the consequences are undefined if the **&environment** argument is referred to outside the *dynamic extent* of the macro expansion function.

class—a *class object*, or **nil**.

Description:

Returns the *class object* named by the *symbol* in the *environment*. If there is no such *class*, **nil** is returned if *errorp* is *false*; otherwise, if *errorp* is *true*, an error is signaled.

The *class* associated with a particular *symbol* can be changed by using **setf** with **find-class**; or, if the new *class* given to **setf** is **nil**, the *class* association is removed (but the *class object* itself is not affected). The results are undefined if the user attempts to change or remove the *class* associated with a *symbol* that is defined as a *type specifier* in this standard. See Section 4.3.7 (Integrating Types and Classes).

When using **setf** of **find-class**, any *errorp* argument is *evaluated* for effect, but any *values* it returns are ignored; the *errorp parameter* is permitted primarily so that the *environment parameter* can be used.

The *environment* might be used to distinguish between a compile-time and a run-time environment.

Exceptional Situations:

If there is no such *class* and *errorp* is *true*, **find-class** signals an error of *type* **error**.

See Also:

defmacro, Section 4.3.7 (Integrating Types and Classes)

next-method-p *Local Function*

Syntax:

next-method-p ⟨*no arguments*⟩ → *generalized-boolean*

Arguments and Values:

generalized-boolean—a *generalized boolean*.

Description:

The locally defined function **next-method-p** can be used within the body *forms* (but not the *lambda list*) defined by a *method-defining form* to determine whether a next *method* exists.

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

Whether or not **next-method-p** is *fbound* in the *global environment* is *implementation-dependent*; however, the restrictions on redefinition and *shadowing* of **next-method-p** are the same as for *symbols* in the `COMMON-LISP` *package* which are *fbound* in the *global environment*. The consequences of attempting to use **next-method-p** outside of a *method-defining form* are undefined.

See Also:

call-next-method, **defmethod**, **call-method**

call-method, make-method *Local Macro*

Syntax:

call-method *method* `&optional` *next-method-list* → {*result*}*

make-method *form* → *method-object*

Arguments and Values:

method—a *method object*, or a *list* (see below); not evaluated.

method-object—a *method object*.

next-method-list—a *list* of *method objects*; not evaluated.

results—the *values* returned by the *method* invocation.

Description:

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

Whether or not **call-method** is *fbound* in the *global environment* is *implementation-dependent*; however, the restrictions on redefinition and *shadowing* of **call-method** are the same as for *symbols* in the `COMMON-LISP` *package* which are *fbound* in the *global environment*. The consequences of attempting to use **call-method** outside of an *effective method form* are undefined.

The macro **call-method** invokes the specified *method*, supplying it with arguments and with definitions for **call-next-method** and for **next-method-p**. If the invocation of **call-method** is lexically inside of a **make-method**, the arguments are those that were supplied to that method.

Otherwise the arguments are those that were supplied to the generic function. The definitions of **call-next-method** and **next-method-p** rely on the specified *next-method-list*.

If *method* is a *list*, the first element of the *list* must be the symbol **make-method** and the second element must be a *form*. Such a *list* specifies a *method object* whose *method* function has a body that is the given *form*.

Next-method-list can contain *method objects* or *lists*, the first element of which must be the symbol **make-method** and the second element of which must be a *form*.

Those are the only two places where **make-method** can be used. The *form* used with **make-method** is evaluated in the *null lexical environment* augmented with a local macro definition for **call-method** and with bindings named by symbols not *accessible* from the `COMMON-LISP-USER` *package*.

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

If *next-method-list* is not supplied, the **call-next-method** function available to *method* signals an error of *type* **control-error** and the **next-method-p** function available to *method* returns **nil**.

Examples:

See Also:

call-next-method, **define-method-combination**, **next-method-p**

call-next-method *Local Function*

Syntax:

call-next-method `&rest` *args* → {*result*}*

Arguments and Values:

arg—an *object*.

results—the *values* returned by the *method* it calls.

Description:

The *function* **call-next-method** can be used within the body *forms* (but not the *lambda list*) of a *method* defined by a *method-defining form* to call the *next method*.

If there is no next *method*, the generic function **no-next-method** is called.

The type of method combination used determines which *methods* can invoke **call-next-method**. The standard *method combination* type allows **call-next-method** to be used within primary *methods* and *around methods*. For generic functions using a type of method combination defined by the short form of **define-method-combination**, **call-next-method** can be used in *around methods* only.

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

When **call-next-method** is called with arguments, the *next method* is called with those arguments.

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

The *function* **call-next-method** returns any *values* that are returned by the *next method*.

The *function* **call-next-method** has *lexical scope* and *indefinite extent* and can only be used within the body of a *method* defined by a *method-defining form*.

Whether or not **call-next-method** is *fbound* in the *global environment* is *implementation-dependent*; however, the restrictions on redefinition and *shadowing* of **call-next-method** are the same as for *symbols* in the `COMMON-LISP` *package* which are *fbound* in the *global environment*. The consequences of attempting to use **call-next-method** outside of a *method-defining form* are undefined.

Affected By:

defmethod, **call-method**, **define-method-combination**.

Exceptional Situations:

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

See Also:

define-method-combination, **defmethod**, **next-method-p**, **no-next-method**, **call-method**, Section 7.6.6 (Method Selection and Combination), Section 7.6.6.2 (Standard Method Combination), Section 7.6.6.4 (Built-in Method Combination Types)

compute-applicable-methods *Standard Generic Function*

Syntax:

compute-applicable-methods *generic-function function-arguments* → *methods*

Method Signatures:

compute-applicable-methods (*generic-function* **standard-generic-function**)

Arguments and Values:

generic-function—a *generic function*.

function-arguments—a *list* of arguments for the *generic-function*.

methods—a *list* of *method objects*.

Description:

Given a *generic-function* and a set of *function-arguments*, the function **compute-applicable-methods** returns the set of *methods* that are applicable for those arguments sorted according to precedence order. See Section 7.6.6 (Method Selection and Combination).

Affected By:

defmethod

See Also:

Section 7.6.6 (Method Selection and Combination)

define-method-combination *Macro*

Syntax:

define-method-combination *name* ⟦↓*short-form-option*⟧
→ *name*

define-method-combination *name lambda-list*
({*method-group-specifier*}*)
[(`:arguments` . *args-lambda-list*)]
[(`:generic-function` *generic-function-symbol*)]
⟦{*declaration*}* | *documentation*⟧
{*form*}*

→ *name*

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

method-group-specifier::=(*name* {{*qualifier-pattern*}$^+$ | *predicate*} ⟦↓*long-form-option*⟧)

long-form-option::=`:description` *description* |
`:order` *order* |
`:required` *required-p*

Arguments and Values:

args-lambda-list—a *define-method-combination arguments lambda list*.

declaration—a **declare** *expression*; not evaluated.

description—a *format control*.

documentation—a *string*; not evaluated.

forms—an *implicit progn* that must compute and return the *form* that specifies how the *methods* are combined, that is, the *effective method*.

generic-function-symbol—a *symbol*.

identity-with-one-argument—a *generalized boolean*.

lambda-list—*ordinary lambda list*.

name—a *symbol*. Non-*keyword*, *non-nil symbols* are usually used.

operator—an *operator*. *Name* and *operator* are often the *same symbol*. This is the default, but it is not required.

order—`:most-specific-first` or `:most-specific-last`; evaluated.

predicate—a *symbol* that names a *function* of one argument that returns a *generalized boolean*.

qualifier-pattern—a *list*, or the *symbol* `*`.

required-p—a *generalized boolean*.

Description:

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

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

define-method-combination

allows arbitrary processing of method *qualifiers*.

Short Form

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

Keyword options for the short form are the following:

- The `:documentation` option is used to document the method-combination type; see description of long form below.

- The `:identity-with-one-argument` option enables an optimization when its value is *true* (the default is *false*). If there is exactly one applicable method and it is a primary method, that method serves as the effective method and *operator* is not called. This optimization avoids the need to create a new effective method and avoids the overhead of a *function* call. This option is designed to be used with operators such as **progn**, **and**, +, and **max**.

- The `:operator` option specifies the *name* of the operator. The *operator* argument is a *symbol* that can be the *name* of a *function*, *macro*, or *special form*.

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

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

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

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

For a discussion of built-in method combination types, see Section 7.6.6.4 (Built-in Method Combination Types).

define-method-combination

Long Form

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

The *lambda-list* receives any arguments provided after the *name* of the method combination type in the `:method-combination` option to **defgeneric**.

A list of method group specifiers follows. Each specifier selects a subset of the applicable methods to play a particular role, either by matching their *qualifiers* against some patterns or by testing their *qualifiers* with a *predicate*. These method group specifiers define all method *qualifiers* that can be used with this type of method combination.

The *car* of each *method-group-specifier* is a *symbol* which *names* a *variable*. During the execution of the *forms* in the body of **define-method-combination**, this *variable* is bound to a list of the *methods* in the method group. The *methods* in this list occur in the order specified by the `:order` option.

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

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

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

If there is an applicable method that does not fall into any method group, the *function* **invalid-method-error** is called.

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

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

define-method-combination

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

- The `:order` option specifies the order of methods. The *order* argument is a *form* that evaluates to `:most-specific-first` or `:most-specific-last`. If it evaluates to any other value, an error is signaled. If `:order` is not supplied, it defaults to `:most-specific-first`.

- The `:required` option specifies whether at least one method in this method group is required. If its value is *true* and the method group is empty (that is, no applicable methods match the *qualifier* patterns or satisfy the predicate), an error is signaled. If `:required` is not supplied, it defaults to **nil**.

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

The body *forms* compute and return the *form* that specifies how the methods are combined, that is, the effective method. The effective method is evaluated in the *null lexical environment* augmented with a local macro definition for **call-method** and with bindings named by symbols not *accessible* from the `COMMON-LISP-USER` *package*. Given a method object in one of the *lists* produced by the method group specifiers and a *list* of next methods, **call-method** will invoke the method such that **call-next-method** has available the next methods.

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

The list `(:arguments .` *lambda-list*`)` can appear before any declarations or *documentation* *string*. This form is useful when the method combination type performs some specific behavior as part of the combined method and that behavior needs access to the arguments to the *generic function*. Each parameter variable defined by *lambda-list* is bound to a *form* that can be inserted into the effective method. When this *form* is evaluated during execution of the effective method, its value is the corresponding argument to the *generic function*; the consequences of using such a *form* as the *place* in a **setf** *form* are undefined. Argument correspondence is computed by dividing the `:arguments` *lambda-list* and the *generic function* *lambda-list* into three sections: the *required parameters*, the *optional parameters*, and the *keyword* and *rest parameters*. The *arguments* supplied to the *generic function* for a particular *call* are also divided into three sections; the required *arguments* section contains as many *arguments* as the *generic function* has *required parameters*, the optional *arguments* section contains as many arguments as the *generic function* has *optional parameters*, and the keyword/rest *arguments* section contains the remaining arguments. Each *parameter* in the required and optional sections of the `:arguments` *lambda-list* accesses the argument at the same position in the corresponding section of the *arguments*. If the section of the `:arguments` *lambda-list* is shorter, extra *arguments* are ignored. If the section of the `:arguments` *lambda-list* is longer, excess *required parameters* are bound to forms that evaluate to **nil** and excess *optional parameters* are *bound* to their initforms. The *keyword parameters* and *rest parameters* in the `:arguments` *lambda-list* access the keyword/rest section of the *arguments*. If the `:arguments` *lambda-list* contains **&key**, it behaves as if it also contained **&allow-other-keys**.

In addition, **&whole** *var* can be placed first in the `:arguments` *lambda-list*. It causes *var* to be *bound* to a *form* that *evaluates* to a *list* of all of the *arguments* supplied to the *generic function*. This is different from **&rest** because it accesses all of the arguments, not just the keyword/rest *arguments*.

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

The body *forms* are evaluated inside of the *bindings* created by the *lambda list* and method group specifiers. Declarations at the head of the body are positioned directly inside of *bindings* created by the *lambda list* and outside of the *bindings* of the method group variables. Thus method group variables cannot be declared in this way. **locally** may be used around the body, however.

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

Documentation is attached as a *documentation string* to *name* (as kind **method-combination**) and to the *method combination object*.

Note that two methods with identical specializers, but with different *qualifiers*, are not ordered by the algorithm described in Step 2 of the method selection and combination process described in Section 7.6.6 (Method Selection and Combination). Normally the two methods play different roles in the effective method because they have different *qualifiers*, and no matter how they are ordered in the result of Step 2, the

define-method-combination

effective method is the same. If the two methods play the same role and their order matters, an error is signaled. This happens as part of the *qualifier* pattern matching in **define-method-combination**.

If a **define-method-combination** *form* appears as a *top level form*, the *compiler* must make the *method combination name* be recognized as a valid *method combination name* in subsequent **defgeneric** *forms*. However, the *method combination* is executed no earlier than when the **define-method-combination** *form* is executed, and possibly as late as the time that *generic functions* that use the *method combination* are executed.

Examples:

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

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

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

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

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

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

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

;The default method-combination technique
 (define-method-combination standard ()
         ((around (:around))
          (before (:before))
          (primary () :required t)
          (after (:after)))
   (flet ((call-methods (methods)
```

```
                    (mapcar #'(lambda (method)
                                `(call-method ,method))
                            methods)))
              (let ((form (if (or before after (rest primary))
                              `(multiple-value-prog1
                                 (progn ,@(call-methods before)
                                        (call-method ,(first primary)
                                                     ,(rest primary)))
                                 ,@(call-methods (reverse after)))
                              `(call-method ,(first primary)))))
                (if around
                    `(call-method ,(first around)
                                  (,@(rest around)
                                   (make-method ,form)))
                    form))))

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

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

define-method-combination

```
      (if around
          `(call-method ,(first around)
                        (,@(rest around)
                         (make-method ,form)))
          form)))

;The same thing, using the :order and :required keyword options
 (define-method-combination or
         (&optional (order ':most-specific-first))
         ((around (:around))
          (primary (or) :order order :required t))
   (let ((form (if (rest primary)
                   `(or ,@(mapcar #'(lambda (method)
                                      `(call-method ,method))
                                  primary))
                   `(call-method ,(first primary)))))
      (if around
          `(call-method ,(first around)
                        (,@(rest around)
                         (make-method ,form)))
          form)))

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

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

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

;;; Example of the use of :arguments
 (define-method-combination progn-with-lock ()
         ((methods ()))
   (:arguments object)
   `(unwind-protect
        (progn (lock (object-lock ,object))
```

```
               ,@(mapcar #'(lambda (method)
                             `(call-method ,method))
                         methods))
          (unlock (object-lock ,object))))
```

Side Effects:

The *compiler* is not required to perform any compile-time side-effects.

Exceptional Situations:

Method combination types defined with the short form require exactly one *qualifier* per method. An error of *type* **error** is signaled if there are applicable methods with no *qualifiers* or with *qualifiers* that are not supported by the method combination type. At least one primary method must be applicable or an error of *type* **error** is signaled.

If an applicable method does not fall into any method group, the system signals an error of *type* **error** indicating that the method is invalid for the kind of method combination in use.

If the value of the `:required` option is *true* and the method group is empty (that is, no applicable methods match the *qualifier* patterns or satisfy the predicate), an error of *type* **error** is signaled.

If the `:order` option evaluates to a value other than `:most-specific-first` or `:most-specific-last`, an error of *type* **error** is signaled.

See Also:

call-method, **call-next-method**, **documentation**, **method-qualifiers**, **method-combination-error**, **invalid-method-error**, **defgeneric**, Section 7.6.6 (Method Selection and Combination), Section 7.6.6.4 (Built-in Method Combination Types), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

Notes:

The `:method-combination` option of **defgeneric** is used to specify that a *generic function* should use a particular method combination type. The first argument to the `:method-combination` option is the *name* of a method combination type and the remaining arguments are options for that type.

find-method *Standard Generic Function*

Syntax:

find-method *generic-function method-qualifiers specializers* `&optional` *errorp*
→ *method*

find-method

Method Signatures:

find-method (*generic-function* **standard-generic-function**)
method-qualifiers specializers `&optional` *errorp*

Arguments and Values:

generic-function—a *generic function*.

method-qualifiers—a *list*.

specializers—a *list*.

errorp—a *generalized boolean*. The default is *true*.

method—a *method object*, or **nil**.

Description:

The *generic function* **find-method** takes a *generic function* and returns the *method object* that agrees on *qualifiers* and *parameter specializers* with the *method-qualifiers* and *specializers* arguments of **find-method**. *Method-qualifiers* contains the method *qualifiers* for the *method*. The order of the method *qualifiers* is significant. For a definition of agreement in this context, see Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers).

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

If there is no such *method* and *errorp* is *true*, **find-method** signals an error. If there is no such *method* and *errorp* is *false*, **find-method** returns **nil**.

Examples:

```
 (defmethod some-operation ((a integer) (b float)) (list a b))
→ #<STANDARD-METHOD SOME-OPERATION (INTEGER FLOAT) 26723357>
 (find-method #'some-operation '() (mapcar #'find-class '(integer float)))
→ #<STANDARD-METHOD SOME-OPERATION (INTEGER FLOAT) 26723357>
 (find-method #'some-operation '() (mapcar #'find-class '(integer integer)))
▷ Error: No matching method
 (find-method #'some-operation '() (mapcar #'find-class '(integer integer)) nil)
→ NIL
```

Affected By:

add-method, **defclass**, **defgeneric**, **defmethod**

Exceptional Situations:

If the *specializers* argument does not correspond in length to the number of required arguments of the *generic-function*, an an error of *type* **error** is signaled.

If there is no such *method* and *errorp* is *true*, **find-method** signals an error of *type* **error**.

See Also:

Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers)

add-method *Standard Generic Function*

Syntax:

add-method *generic-function method* → *generic-function*

Method Signatures:

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

Arguments and Values:

generic-function—a *generic function object*.

method—a *method object*.

Description:

The generic function **add-method** adds a *method* to a *generic function*.

If *method* agrees with an existing *method* of *generic-function* on *parameter specializers* and *qualifiers*, the existing *method* is replaced.

Exceptional Situations:

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

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

See Also:

defmethod, **defgeneric**, **find-method**, **remove-method**, Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers)

initialize-instance *Standard Generic Function*

Syntax:

initialize-instance *instance* `&rest` *initargs* `&key` `&allow-other-keys` → *instance*

Method Signatures:

initialize-instance (*instance* **standard-object**) `&rest` *initargs*

Arguments and Values:

instance—an *object*.

initargs—a *defaulted initialization argument list*.

Description:

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

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

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

See Also:

shared-initialize, **make-instance**, **slot-boundp**, **slot-makunbound**, Section 7.1 (Object Creation and Initialization), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

class-name *Standard Generic Function*

Syntax:

class-name *class* → *name*

Method Signatures:

class-name (*class* **class**)

Arguments and Values:

class—a *class object*.

name—a *symbol*.

Description:

Returns the *name* of the given *class*.

See Also:

find-class, Section 4.3 (Classes)

Notes:

If S is a *symbol* such that S =`(class-name` C`)` and C =`(find-class` S`)`, then S is the proper name of C. For further discussion, see Section 4.3 (Classes).

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

(setf class-name) *Standard Generic Function*

Syntax:

(setf class-name) *new-value class* → *new-value*

Method Signatures:

(setf class-name) *new-value* (*class* **class**)

Arguments and Values:

new-value—a *symbol*.

class—a *class*.

Description:

The generic function `(setf class-name)` sets the name of a *class* object.

See Also:

find-class, *proper name*, Section 4.3 (Classes)

class-of *Function*

Syntax:

class-of *object* → *class*

Arguments and Values:

object—an *object*.

class—a *class object*.

Description:

Returns the *class* of which the *object* is a *direct instance*.

Examples:

```
(class-of 'fred) → #<BUILT-IN-CLASS SYMBOL 610327300>
(class-of 2/3) → #<BUILT-IN-CLASS RATIO 610326642>

(defclass book () ()) → #<STANDARD-CLASS BOOK 33424745>
(class-of (make-instance 'book)) → #<STANDARD-CLASS BOOK 33424745>

(defclass novel (book) ()) → #<STANDARD-CLASS NOVEL 33424764>
(class-of (make-instance 'novel)) → #<STANDARD-CLASS NOVEL 33424764>

(defstruct kons kar kdr) → KONS
(class-of (make-kons :kar 3 :kdr 4)) → #<STRUCTURE-CLASS KONS 250020317>
```

See Also:

make-instance, **type-of**

unbound-slot *Condition Type*

Class Precedence List:

unbound-slot, **cell-error**, **error**, **serious-condition**, **condition**, **t**

Description:

The *object* having the unbound slot is initialized by the `:instance` initialization argument to **make-condition**, and is *accessed* by the *function* **unbound-slot-instance**.

The name of the cell (see **cell-error**) is the name of the slot.

See Also:

cell-error-name, **unbound-slot-object**, Section 9.1 (Condition System Concepts)

unbound-slot-instance *Function*

Syntax:

unbound-slot-instance *condition* → *instance*

Arguments and Values:

condition—a *condition* of *type* **unbound-slot**.

instance—an *object*.

Description:

Returns the instance which had the unbound slot in the *situation* represented by the *condition*.

See Also:

cell-error-name, **unbound-slot**, Section 9.1 (Condition System Concepts)