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 C1 and C2 supply the values of defaulted initialization arguments for different *slots*, and suppose that C1 is more specific than C2; then the defaulted initialization argument whose value is supplied by C1 is to the left of the defaulted initialization argument whose value is supplied by C2 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))

                              Defaulted
Form                          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, Cfrom, to a different class, Cto; it changes the structure of the *instance* to conform to the definition of the class Cto.

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 Cto, *local slots* specified by the class Cto that are not specified by the class Cfrom are added, and *local slots* not specified by the class Cto that are specified by the class Cfrom are discarded.

The values of *local slots* specified by both the class Cto and the class Cfrom are retained. If such a *local slot* was unbound, it remains unbound.

The values of *slots* specified as shared in the class Cfrom and as local in the class Cto 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 Cfrom; 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 Cto. 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 T1 ... Tn)` where T1 ...Tn 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 C1 defines a *slot* named S whose value for the `:allocation` slot option is `:class`, that *slot* is *accessible* in *instances* of C1 and all of its *subclasses*. However, if C2 is a *subclass* of C1 and also defines a *slot* named S, C1's *slot* is not shared by *instances* of C2 and its *subclasses*. When a class C1 defines a *shared slot*, any subclass C2 of C1 will share this single *slot* unless the **defclass** form for C2 specifies a *slot* of the same *name* or there is a *superclass* of C2 that precedes C1 in the *class precedence list* of C2 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 the next figure.

```
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 <A1, ..., An> be the required arguments to a generic function in order. Let <P1, ..., Pn> be the *parameter specializers* corresponding to the required parameters of the method M in order. The method M is applicable when each Ai is of the *type* specified by the *type specifier* Pi. 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 P1,1...P1,n and P2,1...P2,n.
2. For each 1<=i<=n, P1,i agrees with P2,i. The *parameter specializer* P1,i agrees with P2,i if P1,i and P2,i are the same class or if P1,i=`(`**eql** *object*1`)`, P2,i=`(`**eql** *object*2`)`, and `(`**eql** *object*1 *object*2`)`. Otherwise P1,i and P2,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 C1 modifies the behavior of its superclass, C2, by adding *before methods* and *after methods*. Whether the behavior of the class C2 is defined directly by methods on C2 or is inherited from its superclasses does not affect the relative order of invocation of methods on instances of the class C1. Class C1's *before method* runs before all of class C2's methods. Class C1's *after method* runs after all of class C2'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 the next figure.

```
+    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

    (*generic-function* a1...an)

Let M1,...,Mk be the applicable primary methods in order; then the derived Lisp form is

    (*operator* <M1 a1...an>...<Mk a1...an>)

If the expression <Mi a1...an> is evaluated, the method Mi will be applied to the arguments a1...an. For example, if *operator* is `or`, the expression <Mi a1...an> is evaluated only if <Mj a1...an>, 1<=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).