6. METHODS

Methods are the expressions that evaluate when a message is sent to an instance or a class. Methods are analogous to Interlisp-D functions, except that they are defined by a LOOPS class and invoked by sending a message to an instance of that class.

This chapter presents the basic constructs used to create and implement methods. Also included are important methods and functions relevant to the definition and maintenance of methods.

6.1 Categories

LOOPS methods can be divided into categories. This section contains a brief description of each method category. These categories serve as additional documentation only; they do not imply differences in implementation.

Any symbol can be used as a category. Categories can be used as a tool for the organization of methods. Methods may belong to more than one category.

Class [Category]

Messages associated with a class method can only be sent to an object of type class. Methods associated with the class **Class** have this category. See Chapter 3, Classes, for more information on classes.

Object [Category]

The message associated with an object method can only be sent to an object of type object. Methods associated with the class **Object** have this category.

Internal [Category]

Internal methods are low-level system methods, and should not be specialized by users.

Public [Category]

Public methods are defined by the user or the system. These methods can be specialized by users.

Any [Category]

Methods that have not been categorized belong to this category by default.

Masterscope [Category]

Masterscope is an interactive program analysis tool. Methods that are predefined for Masterscope are local only to Masterscope and can be used only when Masterscope has been invoked. Refer to the *Lisp Library Modules Manual* for more information on Masterscope.

(← *self* **AllMethodCategories**) [Method of Class]

Purpose/Behavior: Extracts and lists the categories of all methods defined by the class *self*.

Arguments: *self* Pointer to a class.

Returns: The categories of the methods defined by the class of *self*.

Categories: Class

Example: Line 98 shows the categories of all methods defined in the class *self*.

```
98←(← ($ Class) AllMethodCategories)
(Class Object Masterscope)
```

(← *self* **CategorizeMethods** *categorization*) [Method of Class]

Purpose: Allows you to change how methods are categorized.

Behavior: Varies according to the arguments.

- If *categorization* is NIL, this opens a display editor window with a form that represents the current categorizations. After you have exited from the editor, these new categorizations are installed.
- If *categorization* is non-NIL, it must be of the following form:

  (category1 (selector1 ... selectorN)) (category2 (selector ...)).

  A categorization specified by **CategorizeMethods** deletes any previous categorization; i.e., if method Print for class Thing was in categories Internal and I/O, after doing

  ```
  (← ($ Thing) CategorizeMethods '((Output
  (Print))(Printing (Print))))
  ```

  Print will be only in categories Output and Printing.

Arguments: *self* Pointer to a class.

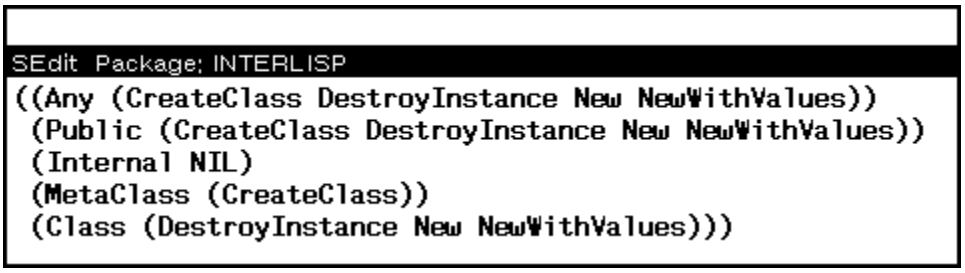
categorization
A list in the form as described in Behavior, or NIL.

Categories: Class

Example: This example shows how to use **CategorizeMethods** with categorization NIL.

```
1←(← ($ MetaClass) CategorizeMethods)
```

The following display editor window appears:



(← *self* **ChangeMethodCategory** *selector newCategory*) [Method of Class]

Purpose: Changes the category of a selected method.

Behavior: Varies according to the arguments.

- If *selector* is NIL, a menu appears showing the selectors for the class of *self*. This is done using the message **PickSelector** to determine the *selector* that is to have its category changed.
- If *selector* is supplied, but not associated with *self*, this message returns NIL.
- If *newCategory* is an atom, adds *selector* to the category. If *newCategory* is a list of atoms, removes *selector* from all its current categories, then adds it to the categories in the list. If *newCategory* is NIL, pops up a menu showing all of the known categories and an additional item, ***other***. If ***other*** is selected, you are prompted to enter a new category name.

Arguments:

self Pointer to a class.

selector Method selector for class of *self* or NIL.

newCategory An atom, a list of atoms, or NIL.

Returns: The new category if there was a change made; else NIL.

Categories: Class

Example: The following command changes the categories of the method associated with **Shape1**.

```
2←(← ($ Window) ChangeMethodCategory
'Shape1 '(Window Internal))
(Window Internal)
```

6.2 STRUCTURE OF METHOD FUNCTIONS

6.2 STRUCTURE OF METHOD FUNCTIONS

6.2 Structure of Method Functions

This section discusses the structure of a LOOPS method.

(**Method** :FUNCTION-TYPE *type ((class selector) self args ...) body...*) [Definer]

Purpose: Similar to **DefineMethod**, but gives more control over the argument list and body syntax. Allows use of Common Lisp lambda argument lists, and Common Lisp syntax in the body of the method. This is the form you will see when editing methods.

Behavior: Defines a method whose argument list is either Interlisp (default) or Common Lisp style. The body of the method may likewise contain either Common Lisp or Interlisp syntax. Common Lisp syntax is distinguished by lexical scoping, etc. (see the *Common Lisp Implementation Notes* for more information).

Arguments:

type The :FUNCTION-TYPE type clause is optional and defaults to :IL.

:IL - The body of the method uses Interlisp syntax, allows CLISP expressions, etc.

:CL - The body of the method uses Common Lisp syntax (is lexically scoped).

class The class to which the method will be attached.

selector The new method's selector.

self This argument must be present and first.

args If type was given as :CL this argument list may contain Common Lisp keys like &OPTIONAL, &KEY and &REST.

body The body of the method. If the type was given as :CL it will be treated as the body of a Common Lisp lambda is, e.g. scoping will be lexical.

Returns: The name of the method function.

Example:

```
12← (Method :FUNCTION-TYPE :CL ((Window Foo) self bar
&OPTIONAL baz &REST glorp)
          (CL:FORMAT T "Bar ~s baz ~s glorp ~s~%" bar baz
glorp))
13← (← ($ Window) New 'Flarb)
14← (← ($ Flarb) Foo 1 2 3 4)
Bar 1 baz 2 glorp (3 4)
```


6.3 CREATING, EDITING, AND DESTROYING METHODS

6.3 Creating, Editing, and Destroying Methods

This section describes the methods and functions which are used to create, rename, delete, and edit LOOPS methods.

Name	Type	Description
DefineMethod	Function	Defines a new method on a class.
DeleteMethod	Function	Deletes a method from a class.
EditMethod	Method	Invokes the editor on a method of a class.
SubclassResponsibility		Macro Appears in the template when you create a new method.

(**DefineMethod** *class selector args expr file* -) [Function]

Purpose: Defines a new method on a class.

Behavior: Varies according to the arguments.

- If *args* is a non-NIL symbol and *expr* is NIL, its function definition is installed as the method for (class selector). This definition must accept an appropriate number of arguments and otherwise work as a LOOPS method. Also, *args* must be a symbol of the form **Name1.Name2** for many of the LOOPS internal routines to handle it properly.
- If *args* is a list of arguments and *expr* is a function, its body will be installed as the definition of **class.selector**.

Arguments:

class	Class in which method is defined.
selector	Method selector (message).
args	List of arguments.
expr	Function definition or NIL.
file	Place where method is stored.

Example: The following expression shows how to add a method called **Increment** to a class called **Documentation**.

```
(DefineMethod ($ Documentation) 'Increment '(Number) '(PLUS number 1]
```

(**DeleteMethod** *class selector prop*) [Function]

Purpose: Deletes a method from a class.

Behavior: Varies according to the arguments.

- If *prop* is NIL or T, the method is deleted from the class.
- If *prop* is T, the function definition is also deleted.

Note: You may also delete methods by using the **ClassInheritance Browser**. Position the mouse on the appropriate class, press the middle mouse button, and select **DeleteMethod** from the resulting menu.

Arguments:

class	Class in which method is defined.
selector	Method selector (message).
prop	T or NIL; determines whether the function definition is deleted.

Example: The following command deletes the method associated with '**MyOpen** from **LatticeBrowser**.

```
(DeleteMethod ($ LatticeBrowser) 'MyOpen)
```

(← *self* **EditMethod** *selector commands okCategories*) [Method of Class]

Purpose: Invokes the display editor on a method of a class.

Behavior: Varies according to the arguments.

- If *selector* is NIL, a menu of selectors is presented using the message **PickSelector** in *okCategories* . This can be a list or a symbol.

- If *selector* is non-NIL, and if it corresponds to a method that is in not *self's* class, you are asked whether the method should be created.
- If *selector* cannot be found, the spelling corrector is invoked to find a correct local selector. If it can be corrected, the local method is used, or an inherited method that is made local is used. When the method is finally determined, **EDITF** (refer to the *Lisp Release Notes* and the *Interlisp-D Reference Manual*) is invoked with *commands* passed as the second argument.

Note: You may also edit methods by using the **ClassInheritance Browser**. Position the mouse on the appropriate class, press the middle mouse button, and select **EditMethod** from the resulting menu.

Arguments:
self Class name.
selector Refers to the method.
commands List of **EDITF** commands.
okCategories Atom or list specifying valid categories.

Categories: Class

(**SubclassResponsibility**) [Macro]

Purpose/Behavior: Appears in the template when you create a new method. It is used to make sure you specialize a method.

6.4 ESCAPING FROM MESSAGE SYNTAX

6.4 Escaping from Message Syntax

The methods described in the previous section manipulate methods in a specific order. Sometimes it may be necessary to invoke multiple inherited methods in some other order. The more general functions in this section have been provided to do this.

CAUTION

These functions do not conform to the conventions of method inheritance and should be used as a last resort and with extreme caution.

The following table shows the items in this section.

Name	Type	Description
DoMethod	Function	Computes the action which should be a method associated with a class and applies it to an object and arguments.
ApplyMethod	Function	Computes the action which should be a method associated with a class and applies it to an object and argument list.

DoFringeMethod	Function	Invokes a method in the class of an object or in each of the super classes for that class.

(**DoMethod** *object selector class arg1 ... argn*) [Function]

Purpose: Computes the action which should be a method associated with *class* and applies it to *object*.

Behavior: All of the arguments are evaluated. If *class* is NIL, **DoMethod** uses the class of *object*. If no method from *class* can be computed from *selector,* an error is generated.

Arguments:

object	Instance to which action is applied.
selector	Evaluates to a method selector.
class	NIL or class in which method name resides.
arg1...argn	The arguments for the method.

(**ApplyMethod** *object selector argList class*) [Function]

Purpose: Same as **DoMethod**.

Behavior: Applies the selected method to the already evaluated arguments in *argList*; otherwise, this is the same as **DoMethod.**

Arguments:

object	Instance to which action is applied.
selector	Evaluates to a method name.
arglist	The arguments for the method.
class	Class in which method name resides.

Example: This example illustrates the MessageNotUnderstood protocol, the function **ApplyMethod**, and the macro **_Super**. This is a specialization of the default **MessageNotUnderstood** message that tries to correct the spelling of the selector. (See Chapter 11, Errors and Breaks, for more information on **MessageNotUnderstood** .)

```
(Method ((DwimObject MessageNotUnderstood)
            self selector mesageArguments superFlg)
  (LET ((correctSelector (FixSelectorSpelling selector)))
    (COND ((correctSelector (ApplyMethod correctSelector mesageArguments))
          (T (_Super))))))
```

Note: *self* is included in the list of **messageArguments**.

(**DoFringeMethods** *object selector arg1 ... argn*) [Function]

Purpose: Invokes method for *selector* in the class of *object* or in each of the super classes for that class.

Behavior: Evaluates all of the arguments. If the method for *selector* in the class of *object* is defined in that class (not through inheritance), **DoFringeMethods** invokes the local method. If there is no local method, **DoFringeMethods** goes down the class of *object,* and for each super invokes its method for selector if one exists. If the supers share supers this can result in the same method being called more than once.

Arguments:

object	Class instance.
selector	Method selector.

arg1...argn Arguments to *selector*.

Returns: NIL

6.5 MOVEMENT BETWEEN CLASSES

6.5 Movement between Classes

This section describes functions and methods that are used in moving methods between classes, as well as stack method macros.

6.5.1 Movement of Methods

The following functions and methods are used to move methods, instance variables, and class variables between classes.

Name	Type	Description
RenameMethod	Function	Renames a function used as a method.
MoveMethod	Function	Moves a method from one class to another.
MoveMethod	Method	Moves a method from one class to another.
MoveMethodToFile	Function	Moves a method to this file if it has the same name as a function on a specified file.
CalledFns	Function	Finds names of all functions called from a set of classes.

(**RenameMethod** *classOrName oldSelector newSelector*) [Function]

Purpose: Renames a function used as a method in *classOrName*.

Behavior: This changes the selector for a method. If no method is associated with *oldSelector* or *newSelector*, this generates an error. Explicit references to *oldSelector* such as

```
(←Super self oldSelector))
```

will not be fixed by **RenameMethod**.

Arguments:

classOrName
Class in which function is defined.

oldSelector Old name of method; invokes method before this function is called.

newSelector
New name of method; invokes method after this function is called.

Returns: If successful, returns *newSelector* in the form **ClassName.Selector**.

Example: The following command renames a method named **Foo** to **Fie** in the class **MyClass**.

```
24←(RenameMethod ($ MyClass) 'Foo 'Fie)
```

(**MoveMethod** *oldClassName newClassName selector newSelector files*) [Function]

Purpose: Moves a method from *oldClassName* to *newClassName*. The method is deleted from *oldClassName.*

Behavior: If *newSelector* is a different name than *selector*, **MoveMethod** renames the method. Explicit references to *oldSelector* such as

```
(←Super self oldSelector))
```

will not be fixed by **RenameMethod**.

Note: You may also move methods by using the **ClassInheritance Browser**. Position the mouse on the appropriate class, press the middle mouse button, and select **MoveMethod** from the resulting menu.

Arguments: *oldClassName*
Source class.

newClassName
Destination class.

selector Method selector to be moved.

newSelector
New name; if NIL, the existing *selector* is preserved.

files Files in which the change is to occur.

Example: The following command moves the method **Buy** from class **Car** to class **Boat** and renames the method to **Purchase**.

```
25←(MoveMethod ($ Car) ($ Boat) 'Buy 'Purchase)
Boat.Purchase
```

(← *self* **MoveMethod** *newClassName selector*) [Method of Class]

Purpose: Moves a method from the class associated with *self* to *newClassName*.

Behavior: Same as the function **MoveMethod**, except that you cannot rename *selector*.

Arguments: *self* Pointer to a class from which the method is taken.

newClassName
Destination class; must be a class, not a class name.

selector Method selector to be moved.

Returns: **NewsClass.Selector**

(**MoveMethodsToFile** *file*) [Function]

Purpose/Behavior: Moves a method to this file if it has the same name as a function on *file*.

Arguments: *file* Name of a file to which methods are moved.

Returns: Normally T; NIL if a method does not have the same name as a function on *file.*

(**CalledFns** *classes definedFlg*) [Function]

Purpose: Finds names of all functions called from a set of *classes*.

Behavior: Varies according to the arguments.

- If *definedFlg* is NIL, all the functions associated with *classes* are returned.
- If *definedFlg* is T, the defined functions are returned.
- If *definedFlg* is 1, the undefined functions are returned.

Arguments: *classes* List of classes to search.

definedFlg NIL, 1, or T.

Returns: NIL or the list of functions.

Example: The following command finds all functions called from the class **Method**.

```
(CalledFns '(Method))
```

6.5.2 Stack Method Macros

This section describes macros that access methods on the stack.

(**ClassNameOfMethodOwner**) [Macro]

Purpose: Uses the stack to perform a help check. Returns the name of the class to which the method on top of the stack belongs.

(**SelectorOfMethodBeingCompiled**) [Macro]

Purpose: Uses the stack to perform a help check. Returns the name of the method being compiled.

(**ArgsOfMethodBeingCompiled**) [Macro]

Purpose: Uses the stack to perform a help check. Returns all arguments associated with the method being compiled.

[This page intentionally left blank]