11. ERRORS AND BREAKS

LOOPS provides an interface to the Medley error system. This allows appropriate detection and recovery from errors that are LOOPS errors rather than Lisp errors. The full power of the Medley error system is available to help you determine and repair the causes of errors. In addition, under certain circumstances, LOOPS will attempt to repair an error and continue if you agree.

This chapter describes the functions and methods LOOPS uses to handle error conditions. It also describes the error messages generated by LOOPS.

11.1 Error Handling Functions and Methods

LOOPS provides several ways to trap and process many common errors. A default processing is available for most errors, and this processing can be specialized for actions you may require.

The following table shows the items in this section.

Name	Type	Description
HELPCHECK	Function	Provides an interface to the Common Lisp error system.
LoopsHelp	NoSpread Function	Generates an error if **LoopsDebugFlg**=NIL, else calls **HELP**.
LoopsDebugFlg	Variable	Controls the behavior of **LoopsHelp**.
ErrorOnNameConflict	Variable	Calls **HELPCHECK** when you attempt to give an object the same name as an existing object.
CVMissing	Method	Sent by access functions when you attempt to access a class variable that does not exist.
CVValueMissing	Method	Sent by access functions when you attempt to access a class variable that has no value.
IVMissing	Method	Sent by access functions when you attempt to access an instance variable that does not exist.
IVValueMissing	Method	Sent by access functions when you attempt to access an instance variable that has no value.
MessageNotUnderstood	Method	Sent when a message has no corresponding selector.

(**HELPCHECK** *mess1 ... messN*) [Function]

Purpose/Behavior: **HELPCHECK** is the LOOPS interface with the Common Lisp error system. When LOOPS detects an error, it generally calls this function with up to four argument messages describing what is wrong and possibly what to do about

it. **HELPCHECK** calls **BREAK1** to put you into a break window and returns whatever the call to **BREAK1** returns. For example, if you type `OK`, it returns T. If you type "RETURN 'someValue", it returns that value. In some instances, LOOPS uses such returned values to repair errors and continue execution.

Arguments: *mess1 ... messN*
Messages to print at the break.

Returns: Value depends on what you type in the break window; see Behavior**.**

Example: The following code causes a break window with the message "Are you certain?". If you type "OK" in the break window, the message "He said OK" will print.

```
(IF (HELPCHECK "Are you certain?")
    THEN (PRINT "He said OK"))
```

(**LoopsHelp** *mess1 ... messN*) [NoSpread Function]

Purpose/Behavior: Generates an error. Calls **HELP** if **LoopsDebugFlg** is T, otherwise calls **ERROR**. Use **LoopsHelp** whenever you want to give the user a way to recover from errors when **LoopsDebugFlg** is T. For example, have **LoopsHelp** print messages like "FOO is not the name of a class. Type RETURN '<classname> to continue using <classname>."

Arguments: *mess1 ... messN*
Messages to print at the break.

Returns: Value depends on what you type in the break window; see **HELPCHECK**, above.

LoopsDebugFlg [Variable]

Purpose/Behavior: Controls the behavior of **LoopsHelp**. If it is T, all calls to **LoopsHelp** generate a break. If it is NIL, such calls that occur near the top of the stack or after a short computation cause a message to be printed and a return to the next level. The default value is T. See **BREAKCHK** in the *Interlisp-D Reference Manual* for more information.

ErrorOnNameConflict [Variable]

Purpose/Behavior: If T, an attempt to give an object the same name as an existing object causes a call to **HELPCHECK**. If you type "OK" in the resulting break window, the process continues and the original object is unnamed. The default value is NIL.

(← *self* **CVMissing** *object varName propName typeFlg newValue*) [Method of Class]

Purpose: Sent by access functions when there is an attempt to access a class variable that does not exist.

Behavior: Calls **LoopsHelp** with the message

varName not a CV of *self*

This method can be specialized to take more sophisticated action by using the other arguments which are provided.

When, in an instance, an attempt is made to access a class variable that does not exist, the message **CVMissing** is sent to the instance's class with the instance in question as *object*.

Note: This method can be invoked if an instance variable is missing.

Arguments:

object The object upon which the access was attempted.

typeFlg The name of the access function (**GetValue**, **GetValueOnly**, **PutValue**, **PutValueOnly**) which caused this message to be sent. The function name allows the type of access to be determined.

varName The name of the variable on which access was attempted.

propName The name of the property on which access was attempted. If NIL, the value of the class variable *varName* was accessed.

newValue The value to which the class variable was to be set.

Categories: Class

Example: Specialize this method to automatically add the class variable which is missing to the class described by *self*. Assuming the class of *self* is **SomeClass**, the method definition is

```
(Method ((SomeClass CVMissing)
         self object varName propName typeFlg
         newValue)
(← self AddCV varName newValue))
```

(← *self* **CVValueMissing** *object varName propName typeFlg*) [Method of Class]

Purpose: Sent by access functions when there is an attempt to access a class variable that has no value. This method can also be invoked if an instance variable is missing and you attempt to access it.

Behavior: If *propName* is NIL it returns the value of **NotSetValue**, otherwise it returns the value of **NoValueFound**.

The default setting for **NoValueFound** is NIL. The default setting **NotSetValue** is an annotatedValue. See Chapter 8, Active Values, for an explanation of **NotSetValue**.

This method can be specialized to take more sophisticated action by using the other arguments which are provided. See the example for **CVMissing**, above.

Arguments:

object The object on which the access was attempted.

typeFlg The name of the access function (**GetValue**, **GetValueOnly**, **PutValue**, **PutValueOnly**) which caused this message to be sent. The function name allows the type of access to be determined.

varName The name of the variable on which access was attempted.

propName The name of the property on which access was attempted. If NIL, the value of the class variable *varName* was accessed.

Returns: Value depends on the arguments; see Behavior.

Categories: Class

(← *self* **IVMissing** *varName propName typeFlg newValue*) [Method of Object]

Purpose: Sent by access functions when there is an attempt to access an instance variable that cannot be found in *self*.

Behavior: Tries to remedy the situation, but if it fails, it calls **LoopsHelp** with the message

varName not an IV of *self*

If the instance variable is present in the object's class, the instance variable will be copied to *self*. This can happen when a class is changed after an instance has been created.

If the instance variable is not present in the class, it attempts to find a class variable of the same name in the class. If one is found, it is used according to its **:allocation** property.

- If the property is **dynamicCached**, the instance variable is added by copying the class variable regardless of the type of access.
- If the property is **dynamic**, the type of access is determined from *typeFlg*, which is the name of the access function. The value of the class variable is returned for a get and the instance variable is created only on a put.
- If the property is **class**, the class variable's value is returned or set and no instance variable is created.

If all else fails, an attempt is made to fix the spelling of *varName* and, if a possible fixed spelling is found, the process starts over.

If an instance variable is not found, the arguments are not used, but could be in a specialization of this method. See the example in **CVMissing** above.

Arguments:

typeFlg The name of the access function (**GetValue**, **GetValueOnly**, **PutValue**, **PutValueOnly**) which caused this message to be sent. The function name allows the type of access to be determined.

varName The name of the variable on which access was attempted.

propName The name of the property on which access was attempted. If NIL, the value of the instance variable *varName* was accessed.

newValue The value to which the instance variable was to be set.

Returns: Value depends on the arguments; see Behavior.

Categories: Object

(← *self* **IVValueMissing** *varName propName typeFlg newValue*) [Method of Object]

Purpose: Sent by access functions when there is an attempt to access an instance variable which has no value in *self*.

Behavior: Looks up the class hierarchy to find a value. If none is found, **SHOULDNT** (see the *Interlisp-D Reference Manual*) is called with the message

Error in Put or GetValue.

The arguments are not used, but could be in a specialization of this method. See the example in **CVMissing**, above.

This method is used internally to handle inheritance of instance variable values. If this error occurs, the LOOPS system has probably been corrupted.

Arguments:

typeFlg The name of the access function (**GetValue**, **GetValueOnly**, **PutValue**, **PutValueOnly**) and allows the type of access to be determined.

The other arguments are passed from the access function.

Categories: Object

(← *self* **MessageNotUnderstood** *selector messageArguments superFlg*) [Method of Object]

Purpose: Sent when a message has no corresponding selector in *self*.

Behavior: Attempts to fix the spelling of *selector*. If this fails, it generates an error.

Arguments: *selector* The name of the message that was not understood.

messageArguments
The arguments of the message *selector*.

superFlg If T, an attempt was made to locate the method *selector* in the supers of *self*.

Categories: Object

Example: Define a class that acts as the class of Lisp numbers, and use the **MessageNotUnderstood** message to translate messages into function calls.

```
37←(DefineClass 'Number)
#.($ Number)

38←(← ($ Number) SpecializeMethod 'MessageNotUnderstood)
```

The **MessageNotUnderstood** method is defined in the editor, making the body of the method as follows:

```
(if (GETD selector)
  then (APPLY selector messageArguments))
  else (←Super))
Number.MessageNotUnderstood
```

Use the class **Number** as the LOOPS class for Lisp numbers.

```
39←(PUTHASH 'SMALLP ($ Number) LispClassTable)
#.($ Number)

40←(PUTHASH 'FIXP ($ Number) LispClassTable)
#.($ Number)

41←(PUTHASH 'FLOATP ($ Number) LispClassTable)
#.($ Number)
```

Test it out.

```
42←(← 4 PLUS 5)
9
```


11.2 ERROR MESSAGES

11.2 Error Messages

This section contains the LOOPS error messages along with their explanations. Atoms which are in *italics* are replaced with specific values when the messages are generated. Messages generated by calls to **SHOULDNT** indicate problems in LOOPS system code. Messages generated by direct calls to **ERROR**, that is not via calls to the LOOPS function **LoopsHelp**, may indicate problems with the system or with user code.

Errors appear in their respective categories:

- Errors that occur when accessing classes and instances in LOOPS.

- Errors that occur when sending messages to LOOPS objects.
- Errors dealing with naming objects.
- Errors encountered when using annotated values and active values.
- Other error messages that may be encountered when using LOOPS.

11.2.1 Classes and Instances

This section describes errors that occur when accessing classes and instances.

type not recognized part of class

Explanation: The *type* argument to the method **ListAttribute** does not correspond to one of the parts of a class.

name not a CV of *self*

Explanation: A reference has been made to a class value that does not exist.

Error in Put or GetValue

Explanation: An attempt has been made to access an instance variable that has no value in an object or in any of its supers.

varName not an IV of *self*

Explanation: An attempt has been made to access an instance variable and it does not exist in the object or its supers, and a class variable of the same name does not exist either.

varName is not a local instance variable of class *name*. Type OK to ignore error and go on.

Explanation: An attempt has been made to delete an instance variable which is not in the class.

newValue is not a class. Type OK to replace metaclass of *classRec* with $Class

Explanation: A call has been made to **PutClass** or **PutClassOnly** with either *propName* erroneously set to NIL or left out, or the new metaclass set to something that is not a valid class.

varName is not a CV of *Class* so cannot be moved from there

Explanation: An attempt has been made to move a class variable from a class where it does not exist. Possible causes include wrong source class or misspelled class variable name.

class has subclasses. You cannot **Destroy** classes that have subclasses.
Type OK to use **Destroy!** if that is what you want.

Explanation: Sending the message **Destroy** to a class with subclasses will leave the subclasses referring to nonexistent superclasses. **Destroy!** destroys all of the subclasses as well. Be sure this is what you want before you type "OK**"** **.**

11.2.2 Methods and Messages

This section describes errors that occur when sending messages to LOOPS objects.

GetValue, PutValue, GetValueOnly, PutValueOnly or **GetIVHere** *self args* not possible

Explanation: An attempt has been made to access a value in an abstract class, which cannot have any values.

← or ←**Super** *self selector* -- not understood

Explanation: Neither the object to which the message was sent nor any of its ancestors has such a method selector.

(← NIL selector --) not understood

Explanation: An attempt has been made to send a message to NIL. One way to do this is to execute (_($ foo) ...), where foo does not name a LOOPS object.

class does not contain the selector *selector*. Type RETURN 'selectorName to try again

Explanation: An attempt has been made to delete a nonexistent method. If the problem is that the wrong method selector was typed or the selector was misspelled, typing "RETURN 'correctName" will fix the problem.

selector is not local for *self* To copy anyway, type OK

Explanation: The object to which **CopyMethod** was sent does not contain *selector*, but one of its supers does. This is not necessarily an error.

selector is not a selector for *self*

Explanation: Neither the object to which **CopyMethod** was sent nor any or its supers contains *selector*.

newClass is not a class. Type OK to use oldClass

Explanation: Something may be missing from the argument to **HELPCHECK,** since nothing is printed after oldClass. Alternatively, the destination class specified in **CopyMethod** is neither a class nor a valid class name.

Typing "OK" causes the method to be copied to the class to which the message was sent. The net result can be to copy a method down from one of the class's supers or to make a copy within the class with a new selector.

name is not a defined function

Explanation: The selector named in **CopyMethod** exists but it does not have a function defined for it. It is possible the class has been loaded but the method has not or that the function definition for the method was somehow erroneously destroyed.

name not a currently defined class. Cannot add method to class.
Type OK to create class and go on.

Explanation: An attempt has been made to add a method to a nonexistent class.

If the class should exist, but has not been created yet, type "OK" to let LOOPS create it automatically. If the class has yet to be loaded, abort and load it first.

Can't find source for *fn*

Explanation: The source file containing a method of a class that is being moved via **MoveToFile** cannot be found. **WHEREIS** is used to try to find it. Either add the necessary file to **FILELST** or use **LOADFNS** to load the function(s).

11.2.3 Naming Objects

This section describes errors that occur when naming objects.

name is already used as a name for an object

Explanation: **ErrorOnNameConflict** has been set to T and an object with the given name already exists. Typing "OK" will cause the new object to be created anyway.

Can't name object NIL

Explanation: The *name* argument to the method **SetName** has been left out.

name should be a symbol to be a name

Explanation: The method **SetName** has been given a non-symbolic name.

name cannot be a class name. Type OK to ignore

Explanation: A non-symbolic class name has somehow gotten into the CLASSES of a file.

Typing "OK" will continue writing the file, but will not remove the offending name.

Can't rename a class without specifying name.
Type RETURN <newName> to continue and rename class: *self*

Explanation: The *newName* argument has been left out of **Rename**. Classes can not be named NIL.

Typing "RETURN 'aNewName" renames the class.

name not defined as a class or an instance. Type OK to ignore and go on.

Explanation: A name which refers to a nonexistent class or instance is in the **CLASSES** or **INSTANCES** file command of a file.

Typing "OK" continues writing out the file, but does not remove the offending name.

name not the name of an instance! Type OK to proceed.

Explanation: A name that refers to a nonexistent instance is in the **THESE-INSTANCES** file command of a file.

Typing "OK" continues writing out the file, but does not do anything to correct the source of the problem; that is, it does not remove the name from the filecoms or find out why it does not exist.

name is a defined object, but is not a class.

Explanation: The name of some LOOPS object that is not a class has been used as an argument where a class name should have been used.

11.2.4 Annotated and Active Values

This section describes errors that occur when using annotated values and active values.

Active value not found, so can't replace it.

Explanation: The old active value specified in **ReplaceActiveValue** does not exist or has been specifed incorrectly.

Unknown access type *type*

Explanation: An improper *type* has been given to the message **AddActiveValue** or **DeleteActiveValue**.

Invalid type *type*

Explanation: An active value has an incorrect type specifier.

Conflicting active value wrapping precedence *self activeValue otherPrecedence*

Explanation: An attempt has been made to add an annotated value with wrapping precedence T or NIL to an existing annotated value with the same wrapping precedence.

Unknown access type *type*

Explanation: **GetWrappedValue** or **PutWrappedValue** has been given an incorrect type.

Can't set the local state of #.NotSetValue

Explanation: **PutWrappedValueOnly** has been erroneously sent to a #.NotSetValue.

11.2.5 Miscellaneous

This section describes other errors that can occur when using LOOPS.

Use one of METHODS IVS CVS for type. RETURN one of these symbols to go on.

Explanation: An incorrect type has been specified to the method **Wherels**.

To continue, enter the type into the break window. For example, enter "RETURN 'METHODS".

Name not installed because of error in source

Explanation: The source specification of a class has been corrupted in some way. It may be necessary to manually redefine the class or edit the file.

Time is not set! Call (**SETTIME** dd-mmm-yy hh:mm:ss) and then type in OK

Explanation: LOOPS uses the date and time to create unique internal names for objects; thus, the time must be set before any objects are created. Call **SETTIME** and then type "OK". For example, (**SETTIME** "15-APR-87 12:00:00") sets time at noon on April 15, 1987.

self varName propName not broken. Type OK to go on

Explanation: Either an attempt has been made to unbreak a value which was not broken or the value was specified incorrectly.

[This page intentionally left blank]