8. ACTIVE VALUES

Active values are used in LOOPS to transpose the access of data within an object to a message being sent to a different object. Typical uses include:

- Causing side effects to occur when data is accessed
- Debugging
- Initializing variables
- Maintaining constraints between variables

An **ActiveValue** is an instance of a subclass of the LOOPS class **ActiveValue**. When an **ActiveValue** instance is installed in the value of a variable, further references to that variable cause messages to be sent to the instance.

LOOPS provides several kinds of active values which are described in this chapter. You can obtain new behavior by specializing one of the existing LOOPS **ActiveValue** subclasses.

When **GetValue** notices that an **ActiveValue** is installed on the variable, it sends the **GetWrappedValue** message to the **ActiveValue**. Similarly, when **PutValue** notices that an **ActiveValue** is installed on the variable, it sends the **PutWrappedValue** message to the **ActiveValue**. The value returned from the get or put is the value returned from the message send. Each specialization of **ActiveValue** either inherits these methods from its superclasses or specializes them to call user code. The messages are received and processed by the **ActiveValue** instances.

For example, assume that you are modeling a simulation that requires the value of an instance variable called windSpeed to be a random value. You can make the value of windSpeed into an active value called ($ RandomWindSpeedAV1). Now, if you try to determine the value of windSpeed by entering

```
(@ ($ SomeAirport) windSpeed)
```

the value returned from this expression is the value returned from

```
(← ($ RandomWindSpeedAV1) GetWrappedValue . otherArgs)
```

This returns the required random value.

The variable containing the **ActiveValue** may still have a current value. Most system **ActiveValue** subclasses are specializations of **LocalStateActiveValue**, which uses an instance variable **localState** in the **ActiveValue** to hold the value.

For efficient implementation, LOOPS uses a special Interlisp data type, the annotatedValue data type, to "wrap" each **ActiveValue** instance when it is installed as a value within an object; the **annotatedValue** contains the **ActiveValue** instance. That is, if the value of an instance variable is said to be an active value, in actuality, the value of the instance variable is an annotatedValue which contains the active value. This allows every **GetValue** or **PutValue** to use Interlisp's microcoded type checking mechanism to see if it should be processed normally or via the **ActiveValue** mechanism. This extra layer is largely invisible in application programs. LOOPS also contains a class

AnnotatedValue to handle the occasional accident when a user forgets about the distinction between annotatedValue and **ActiveValue**, and attempts to treat an annotatedValue as a LOOPS object.

The class **ActiveValue** defines the general protocol followed by all active value objects. Methods setting up the basic functionality of **ActiveValues** are defined in this class and inherited by all its specializations. Methods defined in this class include **AVPrintSource** to specify how annotatedValues print, **AddActiveValue** to install an **ActiveValues**, and **DeleteActiveValue** to delete an installed **ActiveValue**.

The class **ActiveValue** itself is an abstract class; that is, it is a placeholder in the class hierarchy that is not intended to be instantiated. When this documentation refers to an active value object, it is referring to an instantiation of a specialization of the class **ActiveValue**.

Note: The current **ActiveValue** is different from the **activeValue** implementation in the Buttress version of LOOPS. See Appendix A, Active Values in Buttress LOOPS, for more information.

8.1 Using Active Values

A general template is available when using active values. As with all templates, you should not blindly follow it. A good understanding of the active value mechanism is necessary to avoid errors in more complicated situations.

- Determine the functionality that you want the active value to provide. For example, will it cause a side effect to occur on access of data? Will it maintain constraints between two pieces of data? The required functionality will give an indication of which active value class you should use.
- Specialize one of the active value classes to satisfy your specific requirements, if necessary.
- Create an instance of the active value class that you have chosen or created.
- Initialize the contents of that instance, if necessary.
- Install that active value instance on the data that you want to become ative. This is accomplished by using the **AddActiveValue** message.

In a number of situations, you may want to install an active value on an instance variable for every instance of a class. One technique for doing this is discussed in Section 8.5, "Active Values in Class Structures."

8.2 Specializations of the Class ActiveValue

The class **ActiveValue** is an abstract class, and therefore cannot be instantiated. This class contains a number of methods, described in Section 8.3, "ActiveValue Methods," that are necessary for the active value mechanism to function. As a user, you will be making active values which are instances of some subclass of **ActiveValue**, either one of those already provided or one that you created. Figure 8-1 shows the class **ActiveValue** and its specializations. This section describes the subclasses of **ActiveValue** in order of their appearance in this figure. Also included is information on specializing active values.

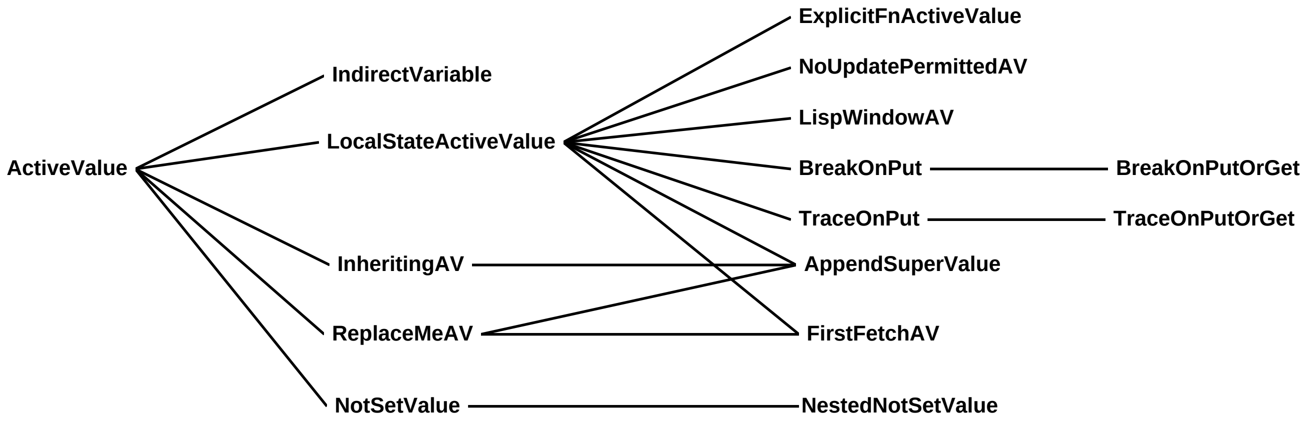


*Figure 8-1. The Class **ActiveValue** and its Specializations*

8.2.1 IndirectVariable

This specialization sets up the functionality of an **ActiveValue** to return the value of another variable as its value. It is analogous to the concept of indirect addressing in other computer languages.

Note: Indirect variables must be the innermost of nested active values. Wrapping precedence (see Section 8.3, "Active Value Methods") insures this.

IndirectVariable [Class]

Purpose: Enables variable values to be accessed indirectly from other variables. This simulates two variables sharing the same memory location. This is a useful technique for implementing simulations and enforcing constraints.

Behavior: When a Fetch is made on the variable containing the **IndirectVariable** instance, this active value retrieves and returns the value of the tracked variable. If a Store is made with the variable containing the **IndirectVariable** instance, this active value stores the new value in the tracked variable. Essentially, this forces the two variables to share the same data.

Instance Variables:

object	Object instance containing the tracked value.
varName	The name of the variable being tracked.
propName	If non-NIL, the name of the variable property being tracked.
type	Type of variable being tracked. Value can be CV, IV or NIL, which defaults to IV.

Examples: Several examples are included to show the use of **IndirectVariable**.

Example 1: Consider a chemical reactor simulation where you have a tank draining into a pipe. The output pressure of the tank needs to equal the input pressure of the pipe. The following demonstrates this.

First, build the appropriate pipe and tank classes and make instances of them.

```
78← (DefineClass 'Tank)
#,($C Tank)

79← (DefineClass 'Pipe)
#,($C Pipe)
```

```
80← (← ($ Tank) AddIV 'outputPressure)
outputPressure

81← (← ($ Pipe)  AddIV  'inputPressure)
inputPressure

82← (← ($ Tank) New 'tank1)
#,($& Tank (NYW0.0X%:.aF4.6R8 . 5))

83← (← ($ Pipe) New 'pipe1)
#,($& Pipe (NYW0.0X%:.aF4.6R8 . 6))
```

Create an instance of **IndirectVariable** and initialize its contents to point to the tank's output pressure.

```
84← (← ($ IndirectVariable) New 'indVar1)
#,($& IndirectVariable (NYW0.0X%:.aF4.6R8 . 7))

85← (←@ ($ indVar1) object ($ tank1))
#,($& Tank (NYW0.0X%:.aF4.6R8 . 5))

86← (←@ ($ indVar1)   varName 'outputPressure)
outputPressure
```

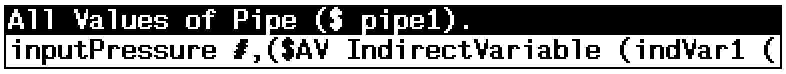
Install the active value instance as the value of the pipe's input pressure.

```
87← (← ($ indVar1) AddActiveValue ($ pipe1) 'inputPressure)
#,($AV IndirectVariable (indVar1 (NYW0.0X%:.aF4.6R8 . 7)) (object
#,($& Tank (NYW0.0X%:.aF4.6R8 . 5))) (varName outputPressure))
```

Accesses to either the pipe's input pressure or the tank's output pressure produce the same result.

```
90← (@ ($ pipe1) inputPressure)
NIL

92← (←@ ($ pipe1)  inputPressure 100)
100

94← (@ ($ tank1)  outputPressure)
100

95← (←@ ($ tank1)  outputPressure 200)
200

96← (@ ($ pipe1) inputPressure)
200
```

An inspector window of ($ pipe1) appears as follows:



```
All Values of Pipe ($ pipe1).
inputPressure #,($AV IndirectVariable (indVar1 (
```

Example 2: Consider a conveyor that must be three feet above a bin. Assume both have an instance variable named height.

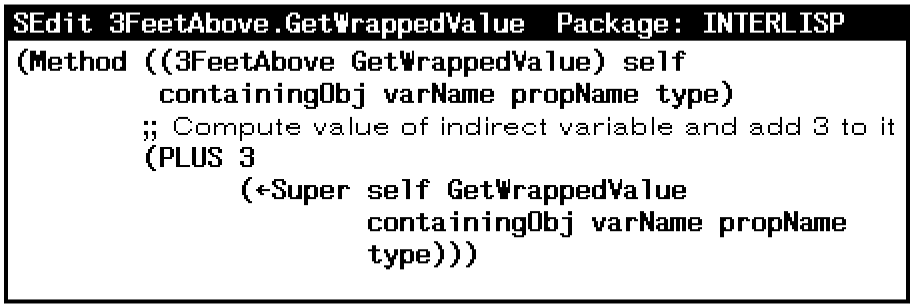
First, create the classes and instances.

```
53← (DefineClass 'Bin)
#,($C Bin)

54← (DefineClass 'Conveyor)
#,($C Conveyor)

55← (← ($ Bin) AddIV 'height 0)
height
```

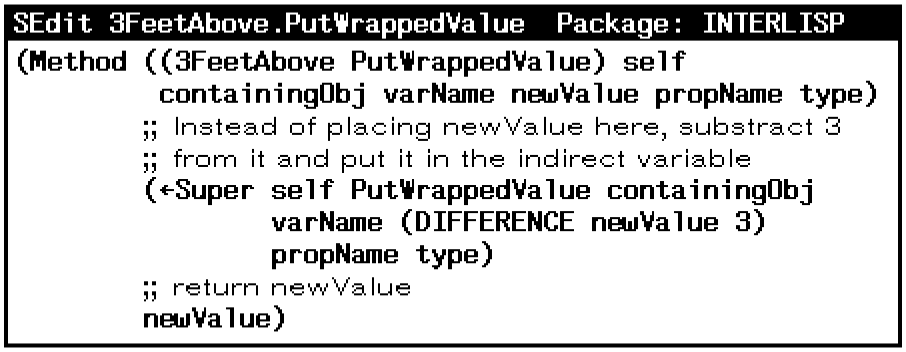
```
56← (← ($ Conveyor) AddIV 'height 0)
height

57← (← ($ Bin) New 'bin1)
#,($& Bin (|DAW0.1Y:.H53.]99| . 506))

58← (← ($ Conveyor) New 'conveyor1)
#,($& Conveyor (|DAW0.1Y:.H53.]99| . 507))
```

Create a specialization of **IndirectVariable** and specialize the methods **GetWrappedValue** and **PutWrappedValue**. You need to specialize **IndirectVariable** because you do not want to maintain equality between the two variables, but instead want to maintain a different mathematical relationship. The **_Super**s are used to use the default behavior of **IndirectVariable** which takes care of retrieving or storing the data into the tracked variable.

```
59← (DefineClass '3FeetAbove
'(IndirectVariable))
#,($C 3FeetAbove)
```



```
SEdit 3FeetAbove.GetWrappedValue  Package: INTERLISP
(Method ((3FeetAbove GetWrappedValue) self
          containingObj varName propName type)
        ;; Compute value of indirect variable and add 3 to it
        (PLUS 3
              (←Super self GetWrappedValue
                      containingObj varName propName
                      type)))
```



```
SEdit 3FeetAbove.PutWrappedValue  Package: INTERLISP
(Method ((3FeetAbove PutWrappedValue) self
          containingObj varName newValue propName type)
        ;; Instead of placing newValue here, substract 3
        ;; from it and put it in the indirect variable
        (←Super self PutWrappedValue containingObj
                varName (DIFFERENCE newValue 3)
                propName type)
        ;; return newValue
        newValue)
```

Create an instance of **3FeetAbove** and initialize its contents to point to the bin's height.

```
65← (← ($ 3FeetAbove) New '3fa1)
#,($& 3FeetAbove (|DAW0.1Y:.H53.]99| . 505))

66← (←@ ($ 3fa1) object ($ bin1))
#,($& Bin (|DAW0.1Y:.H53.]99| . 506))

67← (←@ ($ 3fa1) varName 'height)
height
```

Install this instance of **3FeetAbove** as the value of the conveyor's height.

```
68← (← ($ 3fa1) AddActiveValue ($ conveyor1) 'height)
#,($AV 3FeetAbove (3fa1 (|DAW0.1Y:.H53.]99| . 505))
    (object #,($& Bin (|DAW0.1Y:.H53.]99| . 506)))
          (varName height))
```

The height of **bin1** defaults to 0, what is the height of **conveyor1**?

```
69← (@ ($ bin1) height)
0
```

```
70← (@ ($ conveyor1) height)
3
```

Set either **bin1's** height or **conveyor1's** height and notice how they track each other.

```
71← (←@ ($ bin1) height 15)
15

72← (@ ($ conveyor1) height)
18

73← (←@ ($ conveyor1) height 21)
21

74← (@ ($ bin1) height)
18
```

8.2.2 LocalStateActiveValue

This specialization sets up a class of **ActiveValue** that contains the instance variable **localState**, which is used primarily for storing the value of the referenced variable.

If you need an active value that will produce your own specific side-effect, you will most likely use your own specialization of **LocalStateActiveValue.** The data that would have been accessed, had an active value not been installed, is stored in the **localState** instance variable.

LocalStateActiveValue [Class]

Purpose: Creates a subclass of **ActiveValue** with an instance variable to hold the current value of the referenced variable.

Behavior: Holds the data that normally is stored in the variable where it is installed. At installation time, the current variable value is placed in the **localState** instance variable of the **ActiveValue**. Subclasses of **LocalStateActiveValues** are the most common **ActiveValue** instances.

The class **LocalStateActiveValue** is commonly specialized. In particular, it is usually desirable to specialize the methods **GetWrappedValue** and **PutWrappedValue** associated with new subclasses of **LocalStateActiveValue**. These methods implement the active value messages sent when the variable is accessed.

Instance Variable: **localState** Stores the value of the referenced variable.

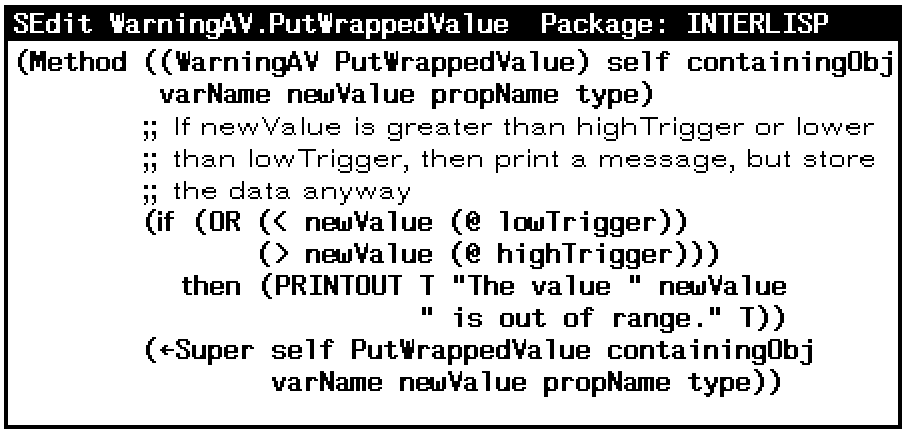
Examples: Several examples are included to show the use of **LocalStateActiveValue**.

Example 1: In this example, an active value will print a message if an attempt is made to store an out-of-range value in an instance variable.

Define a subclass of **LocalStateActiveValue** and give it two instance variables that will store the values of the limits.

```
99← (DefineClass 'WarningAV '(LocalStateActiveValue))
#,($C WarningAV)

100← (← ($ WarningAV) AddIV 'lowTrigger 0)
lowTrigger

101← (← ($ WarningAV) AddIV 'highTrigger 100)
highTrigger
```

Specialize **LocalStateActiveValue**'s **PutWrappedValue** method to create one for **WarningAV**.



```
SEdit WarningAV.PutWrappedValue  Package: INTERLISP
(Method ((WarningAV PutWrappedValue) self containingObj
         varName newValue propName type)
        ;; If newValue is greater than highTrigger or lower
        ;; than lowTrigger, then print a message, but store
        ;; the data anyway
        (if (OR (< newValue (@ lowTrigger))
                (> newValue (@ highTrigger)))
           then (PRINTOUT T "The value " newValue
                         " is out of range." T))
        (←Super self PutWrappedValue containingObj
                varName newValue propName type))
```

Create an instance of a Bin and attach an instance of a **WarningAV** to its height.

```
4← (← ($ Bin) New 'bin3)
#,($& Bin (|DAW0.1Y:.H53.]99| . 513))

5← (←New ($ WarningAV) AddActiveValue ($ bin3) 'height)
#,($& WarningAV (|DAW0.1Y:.H53.]99| . 514))
```

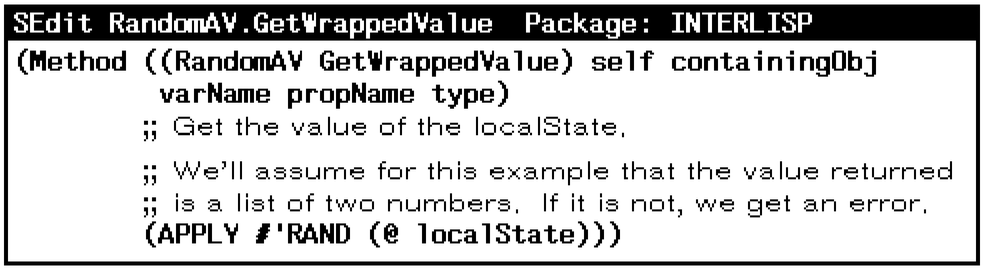
Attempt to store various values into the bin's height.

```
7← (←@ ($ bin3) height 10)
10

8← (←@ ($ bin3) height -10)
The value -10 is out of range.
-10

9← (←@ ($ bin3) height 110)
The value 110 is out of range.
110

10← (@ ($ bin3) height)
110
```

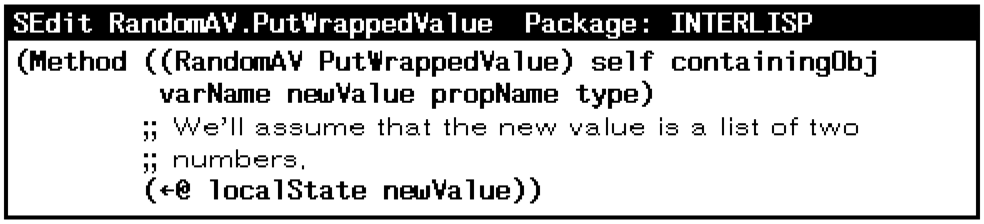
Example 2: In this example, an active value will return a random number when it is read from. Puts to it will change the range of the random value returned on gets. This will use **localState** for something other than storing the data for active values that provide only pure side-effect behavior.

```
99← (DefineClass 'RandomAV '(LocalStateActiveValue))
#,($C RandomAV)

100← (← ($ RandomAV) AddIV 'localState '(0 100))
localState
```

Specialize **LocalStateActiveValue**'s **PutWrappedValue** and **GetWrappedValue** methods to create them for **RandomAV**.



```
SEdit RandomAV.GetWrappedValue  Package: INTERLISP
(Method ((RandomAV GetWrappedValue) self containingObj
          varName propName type)
         ;; Get the value of the localState.

         ;; We'll assume for this example that the value returned
         ;; is a list of two numbers.  If it is not, we get an error.
         (APPLY #'RAND (@ localState)))
```



```
SEdit RandomAV.PutWrappedValue  Package: INTERLISP
(Method ((RandomAV PutWrappedValue) self containingObj
          varName newValue propName type)
         ;; We'll assume that the new value is a list of two
         ;; numbers.
         (←@ localState newValue))
```

Now, try to test this.

```
36← (DefineClass 'RandomTest)
#,($C RandomTest)

39← (← IT AddIV 'randomIV)
randomIV

40← (← ($ RandomTest) New 'rt1)
#,($& RandomTest (DCW0.0X%:.aF4.5S; . 518))

41← (←New ($ RandomAV) AddActiveValue ($ rt1) 'randomIV)
#,($& RandomAV (DCW0.0X%:.aF4.5S; . 519))

42← (@ ($ rt1) randomIV)
24

43← redo
32

44← redo
9

45← redo
49

46← (←@ ($ rt1) randomIV '(4.0 5.0))
(4.0 5.0)

47← (@ ($ rt1) randomIV)
4.190201

48← REDO
4.1129

49← REDO
4.380234

50← REDO
4.397278
```

8.2.2.1 ExplicitFnActiveValue

ExplicitFnActiveValue emulates the activeValue implementation from the Butttress version of LOOPS. Users are discouraged from using this particular form of active values within new projects.

See the *LOOPS Users' Modules* for details on LOOPSBACKWARDS, which describes **ExplicitFnActiveValue**. See Appendix A, Active Values in Buttress LOOPS, for details on the compatibility of **ActiveValue** with activeValue.

8.2.2.2 NoUpdatePermittedAV

This specialization sets up a class of **ActiveValue** that prevents the value of a variable from being replaced.

NoUpdatePermittedAV [Class]

Purpose: Prevents the value of a variable from being replaced using the **PutValue** method.

Behavior: Stores the current value of the variable in **localState**, then prevents it from being updated. **GetWrappedValue** requests return the value found in **localState**, but **PutWrappedValue** requests cause a break with the break message **NoUpdatePermitted!**, or a message if sent from the Exec.

Example: Suppose an identification number for a piece of data should never be changed. Installing a **NoUpdatePermittedAV** in the data's ID number will cause a break if a replacement attempt is made.

Start with a user-defined class named **Datum**. Make a **Datum** instance named **Datum1**. Set the instance variable named **idNumber** to the value 999. Look at the instance. Make a new instance of **NoUpdatePermittedAV**, and name it **NumberGuard**. Install the **ActiveValue** in the instance variable **idNumber** of the instance **Datum1**. Look at the **ActiveValue** instance; the **localState** instance variable contains the previous value of **idNumber**. To test this **ActiveValue**, attempt to replace the **idNumber** of **Datum1** with a new value.

```
67← (DefineClass 'Datum)
#,($C Datum)

68← (← ($ Datum) AddIV 'idNumber 0)
idNumber

69← (← ($ Datum) New 'Datum1)
#,($& Datum (|DAW0.1Y:.H53.]99| . 524))

70← (←@ ($ Datum1) idNumber 999)
999

71← (← ($ Datum1) PP)
(DEFINST Datum (Datum1 (|DAW0.1Y:.H53.]99| . 524)) (idNumber 999))
#,($& Datum (|DAW0.1Y:.H53.]99| . 524))

74← (← ($ NoUpdatePermittedAV) New 'NumberGuard)
#,($& NoUpdatePermittedAV (|DAW0.1Y:.H53.]99| . 525))

75← (← ($ NumberGuard) AddActiveValue ($ Datum1) 'idNumber)
#,($AV NoUpdatePermittedAV (NumberGuard (|DAW0.1Y:.H53.]99| . 525))
       (localState 999))

76← (← ($ Datum1) PP)
(DEFINST Datum (Datum1 (|DAW0.1Y:.H53.]99| . 524))
         (idNumber #,($AV NoUpdatePermittedAV (NumberGuard
                (|DAW0.1Y:.H53.]99| . 525)) (localState 999))))
#,($& Datum (|DAW0.1Y:.H53.]99| . 524))

77← (←@ ($ Datum1) idNumber 888)
No update permitted!
NIL
```

8.2.2.3 LispWindowAV

This specialization sets up a class of **ActiveValue** used by the system to guarantee that the **window** instance variable within a LOOPS **Window** instance contains an Interlisp window. This class provides functionality required by the LOOPS system, and should not generally used by LOOPS users.

LispWindowAV [Class]

Purpose: Guarantees that a variable contains a window which has been made into a LOOPS window.

Behavior: Meant to be installed only in the instance variable **window** of instances of class **Window.** A specialization of **LocalStateActiveValue**. Checks to see if its **localState** is a window, and assures that other instance variables of the window instance are set correctly. See Chapter 19, Windows, for further details.

8.2.2.4 Breaking and Tracing Active Values

The following active values are all specializations of **LocalStateActiveValue** and are used for debugging, as described in Chapter 12, Breaking and Tracing. This chapter also describes **UnbreakIt**, which unbreaks or untraces a method of a class. These classes provide functionality required by the LOOPS system, and are not generally used by LOOPS users.

Note: All breaks and traces occur before the variable is read or modified.

BreakOnPut [Class]

Purpose: Breaks when a replacement attempt is made.

Behavior: Breaks when a replacement attempt is made. Local variables bound at the time of the break are **containingObj**, **varName**, and **propName**.

BreakOnPutOrGet [Class]

Purpose: Breaks when a retrieval or replacement of a variable is made. This is a specialization of **BreakOnPut**.

Behavior: Break occurs before any access to the variable where it is installed. Local variables bound at the time of the break are **containingObj**, **varName**, and **propName**.

TraceOnPut [Class]

Purpose: Traces replacements of a variable.

Behavior: Has a specialized **PutWrappedValue** method that causes the values of the arguments **containingObj**, **varName**, and **propName** to print in the trace window when the variable is about to be modified.

TraceOnPutOrGet [Class]

Purpose: Traces retrievals and replacements of a variable. This is a specialization of **TraceOnPut**.

Behavior: The **GetWrappedValue** method is also specialized so that the variable is traced before any access of the variable where it is installed.

8.2.2.5 AppendSuperValue

This specialization allows the value of a variable to reside only partially in the local instance or class. This is a specialization of the **ReplaceMeAV**, **InheritingAV**, and **LocalStateActiveValue** classes.

AppendSuperValue [Class]

Purpose: Allows the value of a variable to be defined by both a local value and an inherited value.

Behavior: When an instance of **AppendSuperValue** is installed in a variable, Get-references return its **localState** appended to the end of the inherited value the variable would have if it had no local value. Any **PutValue** to the variable replaces the active value, not just the **localState**; **InheritingAV** and its specializations are designed for use more in class variables where replacement is infrequent.

Examples: Several examples are included to show the use of **AppendSuperValue**.

Example 1: Append the **localState** of the instance variable **idNumber** to the default value specified in the class description.

```
23←(DefineClass ’Datum)
#,($C Datum)

24←(← ($ Datum) AddIV ’idNumber ’(5))
idNumber

25←(← ($ Datum) New ’Datum1)
#,($ Datum1)

26←(@ ($ Datum1) idNumber)
(5)

27←(←@ ($ Datum1) idNumber ’(9))
(9)

28←(@ ($ Datum1) idNumber)
(9)

29←(←New ($ AppendSuperValue) AddActiveValue ($ Datum1) ’idNumber)
#,($& AppendSuperValue (45 . 54648))

30←(@ ($ Datum1) idNumber)
(5 9)
```

Example 2: In this example, there are two classes of cars; the **Two-tone-Car** class is a subclass of the class **Car**. Each **Car** class has the instance variable color. The default value for color in the class **Car** is (white).

```
89← (DefineClass ’Car)
#,($C Car)

90← (DefineClass ’Two-tone-Car ’(Car))
#,($C Two-tone-Car)

91← (← ($ Car) AddIV ’color ’(white))
color
```

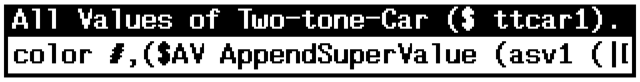
```
92← (← ($ Two-tone-Car) AddIV 'color)
color
```

The default value for color in the class **Two-tone-Car** is an instance of **AppendSuperValue** with its localState set to (blue). The technique for adding active values as default values in a class is discussed in Section 8.3.1, "Adding and Deleting Active Values.".

```
100← (← ($ AppendSuperValue) New 'asv1)

1← (←@ ($ asv1) localState '(blue))

2← (PutClassIV ($ Two-tone-Car) 'color
                          (create annotatedValue
                             annotatedValue ← ($ asv1)))
#,($AV AppendSuperValue (asv1 (|DAW0.1Y:.H53.]99| . 528))
                        (localState (blue)))

9← (← ($ Car) New 'car1)
#,($& Car (|DAW0.1Y:.H53.]99| . 531))

10← (← ($ Two-tone-Car) New 'ttcar1)
#,($& Two-tone-Car (|DAW0.1Y:.H53.]99| . 532))

11← (@ ($ car1) color)
(white)
```

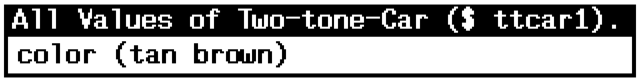
When an instance of a **Two-tone-Car** is created the default value for its instance variable color is the combination of the values in both the classes **Car** and **Two-tone-Car**. The first inspector shows the existence of the active value that provides this behavior. As soon as one puts a value for color in this instance, the **AppendSuperValue** active value is replaced by the new value as shown in the second inspector.

```
12← (@ ($ ttcar1) color)
(white blue)

13← (INSPECT ($ ttcar1))
{WINDOW}#50,5000
```



```
All Values of Two-tone-Car ($ ttcar1).
color #,($AV AppendSuperValue (asv1 (|[
```

```
14← (←@ ($ ttcar1) color '(tan brown))
(tan brown)
```



```
All Values of Two-tone-Car ($ ttcar1).
color (tan brown)
```

For another example, see the **TitleItems** class variable of the class **ClassBrowser**, where **AppendSuperValue** is used to add menu items to an inherited menu.

8.2.2.6 FirstFetchAV

This specialization has instances that have an expression as the value of the instance variable **localState**. These active values allow a form to be evaluated the first time that they are read.

FirstFetchAV [Class]

Purpose: This is a specialization of the **ReplaceMeAV** mixin and **LocalStateActiveValue**. Instances of this class have an expression as the value of the instance variable **localState**.

Behavior: On the first get access, the expression in **localState** is evaluated. The resulting value replaces the **FirstFetchAV** so the variable is no longer an active value. On the first put access, the put value replaces the **FirstFetchAV** so the variable is no longer an active value. A **FirstFetchAV** is often used as the default value for a variable. This class also specializes the method **AVPrintSource** so that instances print as follows when wrapped in an annotatedValue:

```
#,(Defer contentsOfLocalState)
```

CAUTION

FirstFetchAVs cannot be shared. Unlike lists, SEdit does not make copies of active values. Hence, if active values are copied in SEdit, they will share structure, and if one is modified, all will be changed.

Workaround: Use **CopyActiveValue** to copy the active value instance and the local state into each instance which uses the **FirstFetchAV**. See Section 8.3.4, "Shared ActiveValues in Variable Inheritance," for information on **CopyActiveValue**.

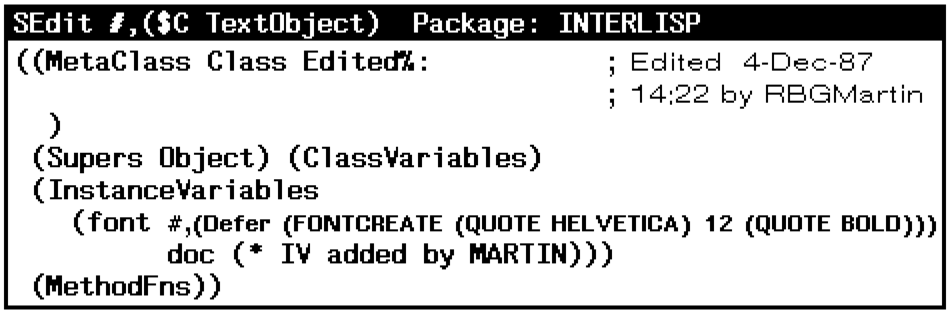
Example: An example application of **FirstFetchAV** is an instance variable that stores a font descriptor. A font descriptor in a class definition does not save correctly;only the pointer to the descriptor is saved. A **FirstFetchAV** stores the expression used to create the descriptor. So, for example the expression held in the **localState** of the **FirstFetchAV** is

```
(FONTCREATE 'HELVETICA 12 'BOLD)
```

On the first access of the instance variable, the font descriptor produced by calling **FONTCREATE** replaces the **FirstFetchAV**.

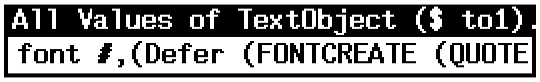
The complete example follows.

```
29← (DefineClass 'TextObject)
#,($C TextObject)

30← (← ($ TextObject) AddIV 'font)
font

31← (← ($ FirstFetchAV) New 'ffav1)
#,($& FirstFetchAV (|DAW0.1Y:.H53.]99| . 535))

32← (←@ ($ ffav1) localState '(FONTCREATE 'HELVETICA 12 'BOLD)]
(FONTCREATE (QUOTE HELVETICA) 12 (QUOTE BOLD))

33← (PutClassIV ($ TextObject) 'font
                    (create annotatedValue
                      annotatedValue ← ($ ffav1)))
#,(Defer (FONTCREATE (QUOTE HELVETICA) 12 (QUOTE BOLD)))

34← (← ($ TextObject) Edit)
TextObject
```



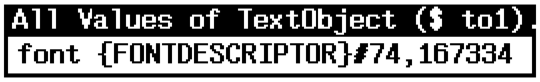
```
SEdit #,($C TextObject)  Package: INTERLISP
((MetaClass Class Edited%:           ; Edited 4-Dec-87
                                     ; 14:22 by RBGMartin
  )
 (Supers Object) (ClassVariables)
 (InstanceVariables
   (font #,(Defer (FONTCREATE (QUOTE HELVETICA) 12 (QUOTE BOLD)))
         doc (* IV added by MARTIN)))
 (MethodFns))
```

```
35← (← ($ TextObject) New 'to1)
#,($& TextObject (|DAW0.1Y:.H53.]99| . 536))

36← (INSPECT ($ to1))
{WINDOW}#47,125470
```



```
All Values of TextObject ($ to1).
font #,(Defer (FONTCREATE (QUOTE
```

```
37← (@ ($ to1) font)
{FONTDESCRIPTOR}#74,167334
```



```
All Values of TextObject ($ to1).
font {FONTDESCRIPTOR}#74,167334
```

8.2.3 InheritingAV

This specialization of **ActiveValue** is used as a mixin to add the **InheritedValue** method.

InheritingAV [Class]

Purpose: Used as a mixin to add the **InheritedValue** method.

Behavior: An abstract class, adds a method **InheritedValue** which allows looking at the value a variable would have if it had no local value, as **NotSetValue** would work. Used as a mixin to add this capability to other specializations of **ActiveValue**.

Example: Used as super class of **AppendSuperValue** to provide incremental menus in various parts of LOOPS.

(← *self* **InheritedValue** *containingObj varName propName type*) [Method of InheritingAV]

Purpose/Behavior: Allows viewing the value a variable would have inherited if it had no local value yet assigned. Similar to the way **NotSetValue** works, it is removed by an assignment to the variable.

Arguments:

self	**InheritingAV** instance.
containingObj	The instance or class that contains the variable to be viewed.
varName	In the *containingObj* the variable to be viewed.
propName	Name of an instance variable or class variable property to be looked at. If *propName* is NIL, the variable itself is viewed.

type	*O*ne of IV, CV, or NIL: a *type* of IV or NIL indicates that the variable is an instance variable or an instance variable property of *containingObj*; a *type* of CV indicates a class variable or class variable property of *containingObj*.

Returns: The value which would have been inherited if the variable had no local value.

8.2.4 ReplaceMeAV

This specialization of the class **ActiveValue** sets up the functionality to replace itself on the first Put- access.

ReplaceMeAV [Class]

Purpose: Specializes the method **PutWrappedValue** to simply replace itself on the first Put- request.

Behavior: No variables are defined in this class. It is an abstract class not intended for instantiation. It is a mixin (see Chapter 3, Classes) to be combined in specialization with another class to add its functionality to the subclass.

Example: **FirstFetchAV** combines **LocalStateActiveValue** and **ReplaceMeAV** to get an **ActiveValue** that replaces itself with the value of an expression stored in the instance variable **localState**.

8.2.5 NotSetValue

This section describes where and when instances of this class appear in user-defined objects.

CAUTION

Do not specialize the classes **NotSetValue** and **NestedNotSetValue**. The documentation is provided here only to explain the functionality that these classes provide to the LOOPS system.

NotSetValue [Class]

Purpose: This specialization of the class **ActiveValue** is unique in that it was created primarily for implementing instance variable inheritance. It has no instance variable to hold a local value and is replaced after the first Put- variable access.

Behavior When an instance of any LOOPS object is created, its instance variables are initialized to contain the value of the variable **NotSetValue**. **NotSetValue** is an annotatedValue whose **ActiveValue** is the only instance of the class **NotSetValue**. The value of **NotSetValue** stored in an instance variable may be replaced within other initialization procedures of new instances that are invoked by the methods **NewWithValues** and **NewInstance** and the instance variable property **:initForm**.

The class **NotSetValue** specializes the default **ActiveValue** protocol to trigger instance variable inheritance. An annotatedValue check is always done by **GetValue** and **PutValue**. LOOPS speeds up instance generation by always initializing instance variables to the value **NotSetValue**. If a retrieval attempt is made on the variable, **NotSetValue** finds the inherited value and returns that value. If no requests are made for the value of the variable, there is no overhead for the instance variable.

The term local value refers to the values LOOPS has actually written into that instance’s instance variables. The local value is always equal to **NotSetValue** before the first Put- access, and to a new value after the first Put- access.

The annotatedValue **#,NotSetValue** is bound to the Lisp variable **NotSetValue**. It must always be on the inside of any sequence of nested **ActiveValues**. Its **WrappingPrecedence** method returns NIL, ensuring this functionality. **NotSetValue** has no **localState** instance variable to hold any nested **ActiveValues**.

See Section 8.3.4, "Shared ActiveValues in Variable Inheritance," for information on **ActiveValues** as default values.

Example: Consider the class **Datum** with the instance variable **idNumber**. Create a new instance named **Datum2**. A standard **GetValue** or @ call returns the default value of **idNumber**, since nothing else has yet been assigned. The call **GetIVHere** shows that the value is not stored in the instance, but is actually returned by **NotSetValue**.

```
91←(← ($ Datum) New 'Datum2)
#,($ Datum2)

92←(@ ($ Datum2) idNumber)
NIL

93←(GetIVHere ($ Datum2) 'idNumber)
#,NotSetValue
```

8.2.5.1 NestedNotSetValue

This subclass of the class **NotSetValue** is used by the internal of LOOPS to solve the problem of using active values as default values.

8.2.6 User Specializations of Active Values

If new specializations of the class **ActiveValue** are defined, the methods **GetWrappedValueOnly** and **PutWrappedValueOnly** might need to be specialized (LOOPS-defined specializations of **ActiveValue**, such as **LocalStateActiveValue**, have already done this). You may also want to specialize the following methods:

AVPrintSource	Prints an **ActiveValue** instance.
GetWrappedValue	Method associated with getting an **ActiveValue**.
PutWrappedValue	Method associated with putting an **ActiveValue**.
WrappingPrecedence	Returns T, NIL, or a number to specify order of **ActiveValue** nesting.
CopyActiveValue	Copies an annotatedValue and its wrapped **ActiveValue**.

8.3 Active Value Methods

Methods defined in the class **ActiveValue** describe how active values work.

8.3.1 Adding and Deleting Active Values

This section describes the methods to install, delete, and replace active values.

Name	Type	Description
AddActiveValue	Method	Makes a variable or property an active value.
WrappingPrecedence	Method	Returns a value which determines how to nest the active value.
DeleteActiveValue	Method	Deletes an active value.
ReplaceActiveValue	Method	Replaces an active value.

(← *self* **AddActiveValue** *containingObj varName propName type annotatedValue*) [Method of ActiveValue]

Purpose: Accomplishes two tasks fundamental in making a variable or property an active value. First, the **ActiveValue** is wrapped inside an annotatedValue. Second, the *annotatedValue* is placed as the value of the variable.

Behavior: **AddActiveValue** associates the annotatedValue with the variable specified by the arguments. If the argument *annotatedValue* is not specified or is NIL, a new annotatedValue is created containing the **ActiveValue** *self*. When the current value of the variable is already an annotatedValue, the **WrappingPrecedence** message determines if it should be nested in the current one or wrapped around it.

Arguments:

self **ActiveValue** instance.

containingObj
The instance or class that contains the variable where the **ActiveValue** is to be added.

varName In the *containingObj* the variable to be made into an **ActiveValue**.

propName Name of an instance variable or class variable property to be made into an **ActiveValue**. If *propName* is NIL, the **ActiveValue** is associated with the variable itself.

type One of IV, CV, CLASS, METHOD, or NIL.

- A *type* of IV or NIL indicates that *varName* is an instance variable or an instance variable property of *containingObj*.
- A *type* of CV indicates a class variable or class variable property of *containingObj*.
- A *type* of CLASS indicates access to a class object’s instance variables and properties.
- A *type* of METHOD indicates access to a method object ’s instance variables and properties.

annotatedValue
AnnotatedValue object used to contain this **ActiveValue**. If NIL, a new **annotatedValue** is created.

Returns: *annotatedValue*

Example: Adds the **ActiveValue** instance named **($ ActiveValueInstance)** to the object **($ ExampleLoopsWindowInstance)** in the instance variable **width**.

```
71←(← ($ ActiveValueInstance) AddActiveValue ($
ExampleLoopsWindowInstance) ’width)

#,($AV IndirectVariable (ActiveValueInstance (NCV0.0X:.SD7.KR . 8))
(object #,($ ExampleLoopsWIndowInstance))(varName width))
```

(← *self* **WrappingPrecedence**) [Method of ActiveValue]

Purpose: Returns a value which determines how to nest the **ActiveValue** associated with *self*.

Behavior: Varies according to the value returned.

- T

  The **ActiveValue** associated with *self* goes on the outside of any other active values.

- NIL

  This **ActiveValue** goes on the inside.

  If two **ActiveValues** return either T or NIL, a break occurs.

- Number

  Specifies precedence: **ActiveValues** with larger **WrappingPrecedence** values go outside ones with smaller **WrappingPrecedence** values.

CAUTION

It is potentially dangerous to have more than one class with a T or NIL precedence.

ActiveValues that have the instance variable **localState** nest in the following way. When a new **ActiveValue** is added to an existing one with equal **WrappingPrecedence**, the original **ActiveValue** is held in the **localState** of the new one. **ActiveValues** not having an instance variable **localState** must nest inside of ones that do.

To set the **WrappingPrecedence** for a user specialization of **ActiveValue**, specialize this method to return the proper value.

Arguments: *self* **ActiveValue** instance.

Returns: The default method defined in the class **ActiveValue** returns 100. **WrappingPrecedence** for the class **NotSetValue** returns NIL. **WrappingPrecedence** for **IndirectVariable** returns 50.

(← *self* **DeleteActiveValue** *containingObj varName propName type*) [Method of ActiveValue]

Purpose: Deletes an **ActiveValue** from *containingObj*.

Behavior: If the variable defined by the arguments is an **ActiveValue**, it is deleted. If it contains a nested **ActiveValue**, the one matching *self* is deleted; otherwise, nothing happens. No **ActiveValue** messages are triggered. If the deleted **ActiveValue** had a **localState**, it becomes the current value.

Arguments: *self* **ActiveValue** instance.

containingObj	The instance or class that contains the variable where the **ActiveValue** is found.
varName	In the *containingObj* the variable that contains the **ActiveValue**.
propName	Name of an instance variable or class variable property where the **ActiveValue** resides. If *propName* is NIL, the **ActiveValue** is associated with the variable itself.
type	One of IV, CV, CLASS, METHOD, or NIL.

- A *type* of IV or NIL indicates that *varName* is an instance variable or an instance variable property of *containingObj*.
- A *type* of CV indicates a class variable or class variable property of *containingObj*.
- A *type* of CLASS indicates access to a class object's instance variables and properties.
- A *type* of METHOD indicates access to a method object's instance variables and properties.

Returns: The deleted annotatedValue if a match was found, NIL otherwise.

(← *self* **ReplaceActiveValue** *newVal containingObj varName propName type*) [Method of ActiveValue]

Purpose: Replaces an **ActiveValue**.

Behavior: Replaces the **ActiveValue** *self* with *newVal*. The location of the old **ActiveValue** is described by the arguments. No **ActiveValue** messages are triggered.

Arguments:

self	**ActiveValue** instance.
newVal	The new value used to replace *self*.
containingObj	The instance or class that contains the variable where the **ActiveValue** is found.
varName	In the *containingObj* the variable that holds the **ActiveValue**.
propName	Name of an instance variable or class variable property where the **ActiveValue** resides. If *propName* is NIL, the **ActiveValue** is associated with the variable itself.
type	*type* is one of IV, CV, or NIL: a *type* of IV or NIL indicates that the variable is an instance variable or an instance variable property of *containingObj*; a *type* of CV indicates a class variable or class variable property of *containingObj*.

Returns: The value of *newVal*.

8.3.2 Fetching and Replacing Wrapped Values

The value of a variable is wrapped in an **ActiveValue**, usually by keeping it in the instance variable **localState**. Specify the behavior of new **ActiveValue** specializations by specializing the methods **GetWrappedValue** and **PutWrappedValue**. For example, **IndirectVariable.GetWrappedValue** just does a **GetValue** on the slot specified by its **object**, **varName**, **propName**, and **type** instance variables. These methods may perform arbitrary work before returning a value, usually that of **localState**. The methods **GetWrappedValueOnly** and **PutWrappedValueOnly** are available for accessing **localState** and bypassing the **ActiveValue** mechanism.

The following table shows the items in this section.

Name	Type	Description
GetWrappedValue	Method	Contains the code to be triggered by a Get- reference to the variable which has been made an **ActiveValue**.
GetWrappedValueOnly	Method	Provides a mechanism to assist in handling nested **ActiveValues**.
PutWrappedValue	Method	Contains the code to be triggered by a Put- reference to the variable which has been made an **ActiveValue**.
PutWrappedValueOnly	Method	Provides a mechanism to assist in handling nested **ActiveValues**.

(← *self* **GetWrappedValue** *containingObj varName propName type*) [Method of ActiveValue]

Purpose: Contains the code to be triggered by a Get- reference to the variable which has been made an **ActiveValue**.

Behavior: Performs arbitrary actions, but when finished, it must return a value which will be returned as the value of the **Get** to the original variable.

This method is fundamental for **ActiveValues**. When **GetValue** or **GetClassValue** finds an annotatedValue in an instance, it does not return that as the value. Instead, it sends the contained **ActiveValue** the **GetWrappedValue** message. This method is not usually called explicitly by users, but is triggered when the **GetValue** function retrieves the value of a variable that contains an **ActiveValue**. It should be specialized when a new subclass of **ActiveValue** is defined.

Arguments:

self	**ActiveValue** instance.
containingObj	The instance or class that contains the variable where the **ActiveValue** is found.
varName	In the *containingObj* the variable that holds the **ActiveValue**.
propName	Name of an instance variable or class variable property where the **ActiveValue** resides. If *propName* is NIL, the **ActiveValue** is associated with the variable itself.
type	One of IV, CV, or NIL: a *type* of IV or NIL indicates that the variable is an instance variable or an instance variable property of *containingObj*; a *type* of CV indicates a class variable or class variable property of *containingObj*.

Returns: The value returned from the actions performed by **GetWrappedValue** message.

(← *self* **GetWrappedValueOnly**) [Method of ActiveValue]

Purpose: Enables the **ActiveValue** mechanism to deal with different problems of nested **ActiveValues**. You will generally not need to specialize this method, as most uses of **ActiveValues** will specialize a subclass of **ActiveValue**.

Behavior: Specializations of the class **ActiveValue** may need to specialize this method. (**LocalStateActiveValue**, **IndirectVariable**, and **NotSetValue** all have specialized versions of this method.)

The class **LocalStateActiveValue** specialization simply returns the value of *self*'s **localState** without triggering the active value mechanism.

The class **IndirectVariable** specialization simply returns the value of tracked variable without triggering the active value mechanism.

The class **NotSetValue** specialization simply returns the value of **NotSetValue.**

Arguments: *self* **ActiveValue** instance.

Returns: See Behavior.

(← *self* **PutWrappedValue** *containingObj varName newValue propName type*) [Method of ActiveValue]

Purpose: Contains the code to be triggered by a Put- reference to the variable which has been made an **ActiveValue**.

Behavior: The **PutWrappedValue** message is similar to **GetWrappedValue** except that it is triggered when the local state of the value is to be replaced. When **PutValue** or **PutClassValue** attempts to replace an **ActiveValue**, it instead sends the contained **ActiveValue** the **PutWrappedValue** message.

The default method found in the class **ActiveValue** checks for nested **ActiveValues** by sending the **GetWrappedValueOnly** message to *self*. If the result is an **AnnotatedValue**, **PutWrappedValue** forwards the message on the nested **ActiveValue**; otherwise it sends the message **PutWrappedValueOnly** to *self* and returns the result.

Arguments:

self	**ActiveValue** instance.
containingObj	The instance or class that contains the variable where the **ActiveValue** is found.
varName	In the *containingObj* the variable that holds the **ActiveValue**.
newValue	The value used to replace the **ActiveValue** containing *self*.
propName	Name of an instance variable or class variable property where the **ActiveValue** resides. If *propName* is NIL, the **ActiveValue** is associated with the variable itself.
type	One of IV, CV, or NIL: a *type* of IV or NIL indicates that the variable is an instance variable or an instance variable property of *containingObj*; a *type* of CV indicates a class variable or class variable property of *containingObj*.

Returns: The value of *newValue*.

(← *self* **PutWrappedValueOnly** *newValue*) [Method of ActiveValue]

Purpose: Enables the **ActiveValue** mechanism to deal with different problems of nested **ActiveValues**. You will generally not need to specialize this method, as most uses of **ActiveValues** will specialize a subclass of **ActiveValue**.

Behavior: Specializations of the class **ActiveValue** may need to specialize this method. (**LocalStateActiveValue**, **IndirectVariable**, and **NotSetValue** all have specialized versions of this method.)

The class **LocalStateActiveValue** specialization simply stores *newValue* into *self*'s **localState** without triggering the active value mechanism.

The class **IndirectVariable** specialization simply stores *newValue* into tracked variable without triggering the active value mechanism.

The class **NotSetValue** specialization causes a break**.**

Arguments:	*self*	**ActiveValue** instance.
	newValue	The new value for **localState**.
Returns:	See Behavior.	

8.3.3 Get and Put Functions Bypassing the ActiveValue Mechanism

ActiveValues normally convert **GetValue**, **GetClassValue**, **PutValue**, and **PutClassValue** accesses into messages which invoke methods to return a value, usually from the **localState** instance variable of the **ActiveValue**. The following functions allow access to class variables and instance variables without triggering any installed **ActiveValue**. See Chapter 2, Instances, for details.

Name	Type	Description
GetValueOnly	Function	Finds the value of an instance variable without triggering active values.
PutValueOnly	Function	Writes the value of an instance variable without triggering active values.
GetClassValueOnly	Function	Returns the value of a class variable; does not trigger active values.
PutClassValueOnly	Function	Changes the value of a class variable and changes the value of a class variable. The change occurs within the class and therefore causes all instances to access the new value of the variable. Does not trigger active values.

8.3.4 Shared ActiveValues in Variable Inheritance

When a **LocalStateActiveValue** is used as the default value for an instance variable in a class, it must be copied into each instance or else all of the instances try to share a single **localState**. This copying is done automatically by LOOPS when the instance variable is first accessed, which means that all instances will share the same ActiveValue until that first access. Copying an **ActiveValue** implies creating a new annotatedValue, so it must be done with the specialized method **CopyActiveValue**.

ActiveValues with no local state may be shared by several variables. In the most extreme case, one instance of **NotSetValue** is the default for the instance variables of all new instances of all classes.

(← *self* **CopyActiveValue** *annotatedValue*) [Method of ActiveValue]

Purpose: Makes a copy of an **ActiveValue** instance.

Behavior: Copies the *AnnotatedValue* and the wrapped **ActiveValue** handling instance variables as follows:

- Instance variables that contain **AnnotatedValues** are copied using the **CopyActiveValue** method.
- The instance variable **localState** is copied so that each copy has its own unique local state.
- All other instance variables are considered shared, and are not copied.

Arguments: *self* **ActiveValue** instance.

annotatedValue
The annotatedValue that surrounds *self*.

Returns: A new annotatedValue wrapped around a copy of the **ActiveValue** *self*.

8.3.5 Creating Your Own Active Value

This example defines a new kind of active value, a **BlippingActiveValue**, which prints out a "blip" of some kind whenever the variable it wraps is read or written.

First, define the new class as a specialization of **LocalStateActiveValue**, then specialize the **PutWrappedValue** and **GetWrappedValue** methods. This is done with the display editor, so in the example they are printed out via the **PPMethod** method. In each case, a **PRINTOUT** function was added before the call to ←**Super**.

Create an instance of **Window** for a location to install a **BlippingActiveValue** for the example. Line 38 is required to set the value of **height** locally to instance **Window1**; if this is not done, its initial value is the active value **#,NotSetValue**, which would remove any active value as soon as it was accessed.

The last few statements in the example show how read and write accesses to height cause a blip character to be printed before **height** is either read or written, with a "!" character representing a write access triggering **PutWrappedValue**, and a "." character representing a read access triggering **GetWrappedValue**.

```
32←(DefineClass 'BlippingActiveValue '(LocalStateActiveValue]
#,($ BlippingActiveValue)

33←(← ($ BlippingActiveValue) SpecializeMethod 'PutWrappedValue]
BlippingActiveValue.PutWrappedValue

34←(← ($ BlippingActiveValue) SpecializeMethod 'GetWrappedValue]
BlippingActiveValue.GetWrappedValue

35←(← ($ BlippingActiveValue) PPMethod 'PutWrappedValue]
```

(**BlippingActiveValue.PutWrappedValue**

```
  (Method ((BlippingActiveValue PutWrappedValue)
           self containingObj varName newValue propName type)
                **COMMENT** **COMMENT**
          (PRINTOUT PPDefault "!")
          (←Super self PutWrappedValue containingObj varName newValue
                       propName type)))
(BlippingActiveValue.PutWrappedValue)

36←(← ($ BlippingActiveValue) PPMethod 'GetWrappedValue]
```

(**BlippingActiveValue.GetWrappedValue**

```
  (Method ((BlippingActiveValue GetWrappedValue)
           self containingObj varName propName type)
                **COMMENT**  **COMMENT**
          (PRINTOUT PPDefault ".")
          (←Super self GetWrappedValue containingObj varName propName
type)))
(BlippingActiveValue.GetWrappedValue)

37←(← ($ Window) New 'Window1]
#,($ Window1)

38←(←@ ($ Window1) height 9876]
9876

39←(←New ($ BlippingActiveValue) AddActiveValue ($ Window1) 'height]
#,($& BlippingActiveValue (46 . 45056))

40←(←@ ($ Window1) height 300)
!300

41←(@ ($ Window1) height]
.300

42←(FOR I TO 20 SUM (@ ($ Window1) height]
...................6000

43←(←@ ($ Window1) height 123]
!123

44←(FOR I TO 20 DO SUM (←@ ($ Window1) height I]
!!!!!!!!!!!!!!!!!!!!210

45←
```

8.4 ANNOTATED VALUES

8.4 Annotated Values

AnnotatedValue is a LOOPS pseudoclass, and instances of it, called pseudoinstances, are Interlisp data type instances.

The structure of the data type is simple. Each annotatedValue contains one field named annotatedValue. This field contains an **ActiveValue** object. The Interlisp record package macros discussed below let you create and work with instances of the data type annotateValue.

There is also a LOOPS class named **AnnotatedValue**. It is an abstract class so it cannot be instantiated, but paradoxically there are objects which consider it their class. (Actually, it is not paradoxical, but this behavior is implemented at a low level within the LOOPS system.) These are the Lisp data type annotatedValue. In normal use this class can be ignored.

AnnotatedValue [Class]

Purpose: LOOPS class equivalent of Lisp data type annotatedValue.

Behavior: This is a LOOPS class, but not a subclass of **Object**. Its super is the LOOPS class **Tofu**. (See Chapter 4, Metaclasses, for a description of **Tofu**.) **AnnotatedValue** is a LOOPS abstract class, and instances are Interlisp data type instances. LOOPS fields messages sent to the annotatedValue data type instances by using the class definition **AnnotatedValue**.

8.4.1 Explicit Control over Annotated Values

This section describes the macros and methods that allow explicit control over annotated values.

Name	Type	Description
type?	Macro	Performs a type check for an instance of the Lisp data type annotatedValue.

create	Macro	Creates a new instance of the data type annotatedValue.
fetch	Macro	Retrieves the contents of the annotatedValue field of an annotatedValue instance.
replace	Macro	Replaces contents of the annotatedValue field of the annotatedValue instance.
_AV	Macro	Sends a message to the **ActiveValue** object wrapped in an annotatedValue.
MessageNotUnderstood	Method	Forwards messages intended for the wrapped **ActiveValue** to that object.

(**type? annotatedValue** *value*) [Macro]

Purpose: Performs a type check for an instance of the Lisp data type annotatedValue.

Arguments: *value* The value to type check.

Returns: T if value is an instance of the data type annotatedValue, NIL otherwise.

(**create annotatedValue annotatedValue** ← *object*) [Macro]

Purpose: Creates a new instance of the data type annotatedValue**.**

Arguments: *object* An **ActiveValue** object to initialize the field annotatedValue of the new annotatedValue instance. This must be an object that has a method **AVPrintSource** (a method of **ActiveValue**) or this form breaks on evaluation. No type checking of object will be performed by the macro.

Returns: An instance of annotatedValue.

(**fetch annotatedValue of** *value*) [Macro]

Purpose: Retrieves the contents of the annotatedValue field of an annotatedValue instance.

Arguments: *value* An annotatedValue instance.

Returns: Contents of field annotatedValue.

(**replace annotatedValue of** *value* **with** *object*) [Macro]

Purpose: Replaces contents of the annotatedValue field of annotatedValue instance with *object*.

Arguments: *value* An annotatedValue instance.

object **ActiveValue** object to be stored in the field. No type checking is done on *object*.

Returns: If value is not an annotatedValue, generates an error; otherwise the previous contents of the field is returned.

(**_AV** *av selector* **.** *args*) [Macro]

Purpose: Sends a message to the **ActiveValue** object wrapped in an annotatedValue.

Behavior: Equivalent to

(_ (**fetch** annotatedValue **of** *av*) *selector* **.***args*)

Arguments:	*av*	Instance of an annotatedValue.
	selector	Selector for message to send to the **ActiveValue** object.
	args	Arguments to be passed when the message is sent.

Returns: Result of message.

(← *self* **MessageNotUnderstood**) [Method of AnnotatedValue]

Purpose: Forwards messages intended for the wrapped **ActiveValue** to that object.

Behavior: Messages sent to an annotatedValue are forwarded to its wrapped **ActiveValue**. Users should not explicitly send this message.

8.4.2 Saving and Restoring Annotated Values

The following are methods of the class **ActiveValue** that handle annotated values.

(← *self* **AVPrintSource**) [Method of ActiveValue]

Purpose: Prints **ActiveValues**.

Behavior: An annotatedValue determines how it prints out by sending the **AVPrintSource** message to its wrapped **ActiveValue**.

The default method in **ActiveValue** returns a list of the form:

```
("#," $AV className avNames(ivName value propName value ...)(ivName ...) ...)
```

which causes the annotatedValue to print out as

```
#,($AV className avNames(ivName value propName value ...)(ivName ...) ...)
```

Arguments:	*self*	**ActiveValue** instance.
	className	Name of the class of the **ActiveValue**.
	avNames	List of names of *self*; the last element being the unique identifier (UID) of *self*

The list *(ivName value propName value ...)* describes the state of the instance variables of the **ActiveValue**. Including the UID of the **ActiveValue** in the print form enables recovery of the identity of the **ActiveValue**. This enables different annotatedValues to share the same **ActiveValue**, and maintain this sharing across saving to a file and reloading into Lisp.

Returns: A form suitable for use by the Interlisp function **DEFPRINT**. Result should be a pair of the form (item1 . item2); item1 will be printed using **PRIN1**, and item2 will be printed using **PRIN2** (see *Lisp Release Notes* and the *Interlisp-D Reference Manual* description of **DEFPRINT**).

Example:

```
#,($AV IndirectVariable (HeightFromWidth (NCV0.0X:.SD7.KR
. 8))
(object #.($ SquareWindow)) (varName width) (propName NIL)
(type IV))
```

(**$AV** *className avNames . ivForms*) [NLambda, NoSpread Function]

Purpose: Reconstructs an annotatedValue that has been saved to a file.

Arguments:

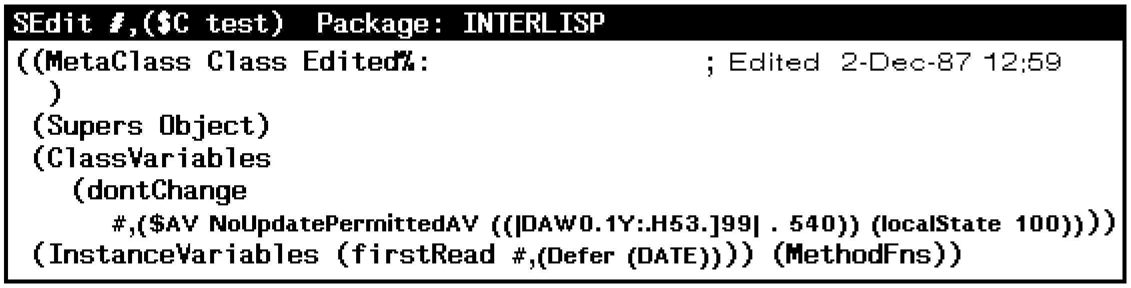
className	Name of the class of **ActiveValue***.*
avNames	A list of the LOOPS names of **ActiveValue** instances.
ivForms	A list describing the state of the instance variables of the **ActiveValue**.

Returns: A new annotatedValue whose **ActiveValue** is reconstructed from the *avNames* and *ivForms*.

8.5 ACTIVE VALUES IN CLASS STRUCTURES

8.5 Active Values in Class Structures

It is possible to have an active value as the default value of an instance variable or the value of a class variable in a class. For example, the following class has an active value installed in the class variable **dontChange** and one installed in the instance variable **firstRead**.



```
SEdit #,($C test)  Package: INTERLISP
((MetaClass Class Edited%:                    ; Edited  2-Dec-87 12:59
  )
 (Supers Object)
 (ClassVariables
    (dontChange
       #,($AV NoUpdatePermittedAV ((|DAW0.1Y:.H53.]99| . 540)) (localState 100))))
 (InstanceVariables (firstRead #,(Defer (DATE)))) (MethodFns))
```

LocalStateActiveValue active values as default IV values are copied down into the instance when their **localState** is smashed, instead of being shared by all instances; this is different from normal default behavior. It is also possible to create a **LocalStateActiveValue** which inherits its **localState** value by giving it a **localState** value of the value of **NotSetValu**e). These copy the inherited value down from the superclass when the **LocalStateActiveValue** is created; if the value in the superclass is changed after the **LocalStateActiveValue** is created, that change will not be reflected in the **LocalStateActiveValue**. Normally inherited values are always tracked by instances that inherit them.

There are two ways to enter active values into the structure of a class: with the editor or programmatically.

It is possible to create active values by typing in a form such as:

($AV activeValueClassName NIL (ivname value propName value ...)(ivname value propName value ...) ...). None of the arguments are evaluated.

To add an active value through the editor, you can type in the above form, select it, and mutate it with the function **EVAL**.

Programmatically, you can use the functions **PutClassIV**, **PutClassValue**, **PutClassValueOnly**, **AddCIV**, **AddCV**, etc. or different methods, such as **Add**, to modify or add class variables and instance variables.

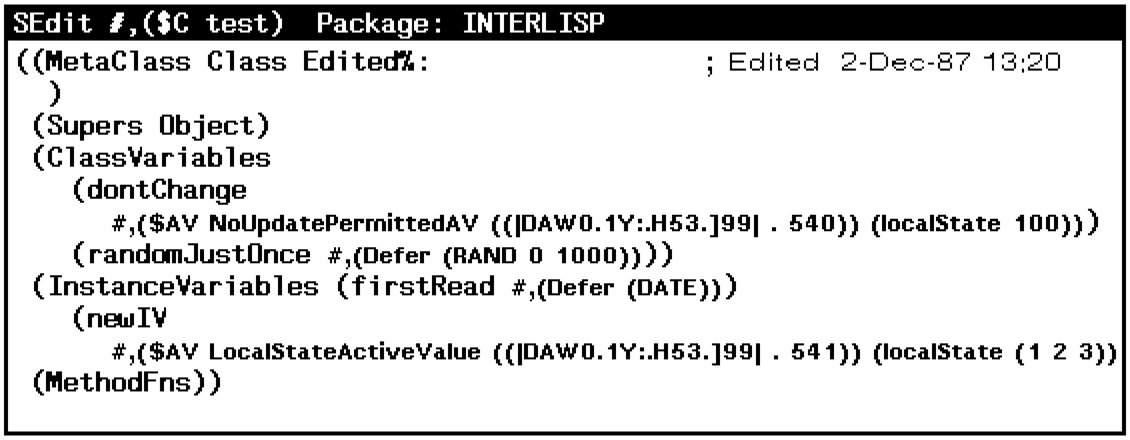
For example, given the above class, **test**:

```
(← ($ test) Add 'CV 'randomJustOnce ($AV FirstFetchAV NIL
(localState (RAND 0 1000))))
```

and

```
(AddCIV  ($ test) 'newIV ($AV LocalStateActiveValue NIL
(localState (1 2 3))))
```

will result in the following:



```
SEdit #,($C test)  Package: INTERLISP
((MetaClass Class Edited%:                    ; Edited  2-Dec-87 13:20
  )
 (Supers Object)
 (ClassVariables
   (dontChange
      #,($AV NoUpdatePermittedAV ((|DAW0.1Y:.H53.]99| . 540)) (localState 100)))
   (randomJustOnce #,(Defer (RAND 0 1000))))
 (InstanceVariables (firstRead #,(Defer (DATE)))
   (newIV
      #,($AV LocalStateActiveValue ((|DAW0.1Y:.H53.]99| . 541)) (localState (1 2 3))
 (MethodFns))
```

An even more general programmatic method that more easily allows customization of an active value uses the annotatedValue data type explicitly. First, you must create an instance of an **ActiveValue** class.

```
(← ($ MyActiveValue) New 'MyAV1)
```

Then the contents of the instance **MyAV1** are initialized. Finally, it is added as the value of a variable in a class structure.

```
(AddCIV  ($ test) 'myNewIV (create annotatedValue
annotatedValue ← ($ MyAV1)))
```

[This page intentionally left blank]