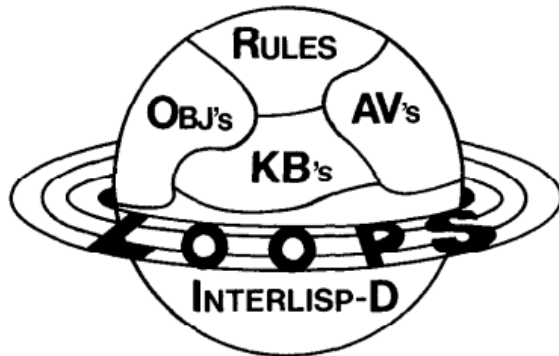


Medley LOOPS: The Basic System

Lisp Object-Oriented Programming  
System  
(LOOPS)  
Volume I: The Basic System

By  
Stephen H. Kaisler

Version 1.2  
July 2024



## Acknowledgement

Many people have contributed to resurrecting, restoring, and modernizing Medley Interlisp. Among them are Nick Briggs, Ron Kaplan, Frank Halasz, Matt Heffron, Bill Stumbo, Paolo Amoroso, and many others.

A special thanks to Larry Masinter, one of original developers, for his assistance in debugging some of test code for the LOOPS active value feature. And, to Paolo Amoroso for his careful review of the text.

## Table of Contents

List of Figures .....	12
List of Tables .....	14
Introduction.....	15
I.1 Some History .....	17
I.2 LOOPS Paradigms.....	17
I.3 Structure of the Documentation.....	19
Chapter One .....	20
Introduction to LOOPS Paradigms .....	20
1.1 Introduction to Object-Oriented Programming.....	21
1.2 Classes and Instances .....	23
1.2.1 Variables and Property Lists .....	23
1.2.2 Properties .....	27
1.2.3 Instances.....	27
1.2.4 Methods.....	29
1.2.5 Metaclasses .....	30
1.2.6 Abstract Classes .....	31
1.3 Generic Class Description.....	32
1.4 Class Hierarchy .....	33
1.4.1 The Concept of Inheritance.....	34
1.4.2 Simple Hierarchy .....	35
1.4.3 A Complex Hierarchy .....	39

## Medley LOOPS: The Basic System

1.5 Interlisp Objects .....	39
1.5.1 Testing for Lisp Data Types.....	40
1.5.2 Assigning Names to LOOPS Objects .....	46
1.5.3 Class Objects and LOOPS Names .....	47
1.5.4 NamedObject .....	48
1.5.5 DatedObject .....	48
1.6 System Variables and Functions .....	49
Chapter Two.....	53
Object-Oriented Programming in LOOPS .....	53
2.1 Creating a New Class.....	53
2.1.1 Creating a New Class via New .....	53
2.1.2 Creating a New Class with NewClass.....	55
2.1.3 Creating a New Class with DefineClass .....	55
2.1.4 LispClassTable.....	58
2.2 Creating an Instance of a Class .....	58
2.2.1 Simple Form.....	59
2.2.2 Creating a New Class as a Subclass.....	60
2.2.3 Initializing a New Class .....	61
2.3 Creating an Instance of a Class Using SEND .....	62
2.4 Instance Variables and Properties .....	63
2.4.1 Instance Variable Operations .....	63
2.5 LOOPS Names.....	64
2.6 Instance Names .....	64
2.6.1 Working with LOOPS Names.....	65

## Medley LOOPS: The Basic System

2.7 Editing a Class .....	66
2.8 The Class Record .....	68
2.8.1 Object Functions .....	69
Chapter Three.....	71
Class Messages and Functions.....	71
3.1 Sending a Message to an Object .....	71
3.2 Checking Objectivity .....	73
3.3 Class Operations .....	73
3.3.1 Creating a New Class.....	73
3.3.2 Editing a Class .....	76
3.3.3 Editing a Method.....	77
3.3.4 Naming an Object .....	78
3.4 Accessing Supers .....	79
3.5 Accessing Variables.....	79
3.5.1 Getting Variable and Property Values .....	82
3.5.2 Putting Variable and Property Values.....	86
3.5.3 Non-triggering Get and Put.....	98
3.5.4 Local Get Functions.....	99
3.5.5 Accessing Class and Method Properties .....	100
3.5.6 Accessing Class Properties .....	102
3.5.7 Adding Variables to a Class.....	105
3.5.8 Generalized Get and Put Functions.....	109
3.5.9 Putting IV Value and Property.....	113
3.5.10 Dual Use of Get and Put Functions.....	114

## Medley LOOPS: The Basic System

3.6 Accessing Methods .....	114
3.6.1 Accessing Method Properties.....	115
3.7 Delete Functions .....	117
3.8 Destroying Classes.....	118
3.8.1 Removing a Class.....	118
3.8.2 Destroying a Class .....	119
3.8.3 Ensuring Removal of Subclasses .....	121
3.9 Inheritance.....	121
3.10 Compact Forms for Accessing Data .....	123
3.10.2 IV Delimiters .....	128
3.11 Class Method Operations .....	129
3.11.1 Defining a Method .....	129
3.11.2 Defining a Method by a Definer .....	137
3.11.3 Defining A Method by Message .....	140
3.11.4 Deleting a Method.....	142
3.11.5 Editing a Method.....	142
3.11.6 SubclassResponsibility.....	145
3.11.7 Alternatives to Executing Methods .....	145
3.12 Manipulating Methods Across Classes .....	149
3.12.1 Renaming a Method .....	149
3.12.2 Moving a Method between Classes .....	150
3.12.3 Alternate Moving a Method .....	151
3.12.4 Moving Methods to a File.....	152
3.12.5 Getting Functions Called from a Class Set .....	152

## Medley LOOPS: The Basic System

3.13 Methods Concerning the Class of an Object.....	153
3.13.1 Finding the Class of an Object.....	153
3.13.2 Getting the Class Name.....	154
3.13.3 Determining an Instance of a Class.....	156
3.13.4 Copying Instances.....	156
Chapter Four .....	158
Instance Functions and Methods.....	158
4.1 Defining a New Instance.....	158
4.1.1 Sending the Class the Message NEW .....	158
4.1.2 Using NewInstance Message .....	162
4.1.3 Creating an Instance with Initial Values .....	164
4.1.4 Creating an Instance with Immediate Messaging .....	165
4.2 Data Storage for New Instance .....	167
4.2.1 IVValueMissing.....	168
4.2.2 NotSetValue.....	169
4.2.3:initWithForm .....	170
4.2.4 Changing the Number of IVs in an Instance.....	171
Chapter Five.....	174
Metaclass Functions and Methods .....	174
5.1 Base Metaclasses .....	174
5.1.1 Abstract Classes .....	175
5.2 Pseudoclasses.....	176
5.2.1 Pseudoclass Functions.....	176
5.3 Metaclass Functions.....	177

## Medley LOOPS: The Basic System

5.3.1 Defining a New Metaclass .....	178
Chapter Six.....	179
Sending Messages Alternatives .....	179
6.1 Sending A Message to a LOOPS Object .....	179
6.2 _! .....	181
6.3 _IV .....	181
6.4 _Try.....	182
6.5 _Super .....	183
6.5.1 _Super? .....	184
6.5.2 _SuperFringe.....	184
6.6 _New .....	184
6.7 FetchMethod .....	186
Chapter Seven .....	188
Introduction to.....	188
Data-Oriented Programming.....	188
7.1 Specifying an Active Value .....	190
7.1.1 getFn and putFn .....	191
7.1.2 Defining an Active Value .....	191
7.1.3 Nested Active Values.....	194
7.1.4 Using Active Values .....	194
7.2 Active Value Functions.....	196
7.2.1 FirstFetch .....	196
7.2.2 Indirect Access.....	198
7.2.3 ReplaceMe .....	199



## Medley LOOPS: The Basic System

7.2.4 User-Defined Function.....	200
7.2.5 Local State Functions.....	201
7.2.6 Annotated Values.....	202
7.2.7 Managing Annotated Values.....	206
7.3 The ActiveValue Class .....	209
7.3.1 Using Active Values .....	209
7.3.2 Specializing an Active Value.....	210
7.3.3 Breaking and Tracing Active Values .....	221
7.3.4 Appending to a Super Value .....	222
7.3.5 InheritedValue.....	223
7.3.6 ReplaceMeAV.....	223
7.3.7 NotSetValue.....	224
7.3.8 User Specializations of Active Values .....	224
7.4 Active Value Methods .....	225
7.4.1 Adding and Deleting Active Values .....	225
7.4.2 Wrapped Value Methods .....	228
7.5 Annotated Properties.....	230
7.6 Defensive Programming .....	231
7.7 ActiveValue Uses.....	232
Chapter 8.....	233
Introduction to.....	233
Rule-Oriented Programming.....	233
8.1 RuleSets and Rules .....	234
8.2 Organizing a Rule-based System .....	235

## Medley LOOPS: The Basic System

8.3 RuleSet.....	235
8.3.1 RuleSet Class Definition.....	236
8.3.2 RuleSetSource.....	237
8.3.3 RuleSet Structure .....	239
8.3.4 RuleSet Methods.....	239
8.3.5 Invoking RuleSets.....	239
8.4 RuleSetMeta.....	240
8.5 RuleSetNode .....	241
8.6 RuleSetSource.....	242
8.7 Rule.....	242
8.7.1 Rule Class Definition.....	243
8.7.2 Variables Used in Rules.....	244
8.7.3 Infix Operators Used in Rules.....	246
8.7.4 Use of Interlisp Functions in Rules.....	247
8.7.5 Use of LOOPS Objects and Message Selectors.....	248
8.8 Running RuleSets.....	250
8.9 Using RuleSets as Methods.....	250
8.9.1 Defining A RuleSet as a Method .....	251
8.10 Control Structures for Selecting Rules.....	251
8.10.1 Singleton Rule Execution.....	253
References.....	254
Appendix A: Running MEDLEY .....	257
A.1 Running Medley.....	257
Appendix B .....	262

## Medley LOOPS: The Basic System

Installing and Running LOOPS .....	262
B.2 Loading LOOPSRULES and GAUGES .....	267
B.3 Setting System Variables.....	267
B.3.1 Connect to the LOOPS System Directory .....	268
Appendix C .....	269
Testing LOOPS Installation.....	269
C.1 LOOPS 1.1 Tests.....	269
Appendix C: Test Applications.....	275
C.1 Source Code for TestAV.txt.....	275
C.2 Source Code for NewTestAv.txt .....	277
Index .....	280

## List of Figures

- 1-1. LOOPS Lattice
- 1-2. Object Representation
- 1-3. Truckin Player Class
- 1-4. A Class and Its Instances
- 1-5. Method Structure
- 1-6. A MetaClass Example
- 1-7. Generic Class Description
- 1-8. The Subclass Point3D
  
- 2-1. LOOPS Forms for Name Manipulation
- 2-2. LOOPS Object Functions
  
- 3-1. Tofu Specializations
- 3-2. Editing a Class Definition
- 3-3. Variable Locations
- 3-4. Method Edit Menu
- 3-5. Method Display
- 3-6. Functions Called From Window
  
- 4-1. Shaping a Window After Creating it.
  
- 5-1. Base Metaclasses
  
- 7-1. ActiveValue and its specializations

## Medley LOOPS: The Basic System

A-1. Medley Directory Contents

B-1. B-1. Loading LOOPS

B-2. Starting Medley with the LOOPSRULES Sysout

B-3. LOOPSRULES Sysout Running

B-x. Directory Variables

## List of Tables

- 1-1. Basic Inheritance Principles
- 1-2. LOOPS System Variables
  
- 2-1. Object Functions
  
- 3-1. Tofu Message Descriptions
- 3-2. Compact Access Forms
- 3-3. Instance Variable Delimiters in Compact Forms
  
- 4-1. IVValueMissing Behavior
  
- 5-1. Base Metaclass Descriptions
  
- 7-1. IndirectVariable Instance Variables
- 7-2. LocalStateActiveValue Instance Variables
- 7-3. NoUpdatePermittedAV Instance Variables
  
- 8-1. Types of LOOPS Variables
- 8-2. Reserved Word Usage
- 8-3. Infix Operators in LOOPS Rules
- 8-4. RuleSet Control Structures

## Introduction

The Medley Interlisp Project has ported the Medley release of Interlisp to modern operating system environments, including Windows, Linux, and MacOS. This port includes the Interlisp Core, selected LispUsers packages, and selected applications developed for Interlisp-D.

This document describes the Installation and Use of the Lisp Object-Oriented Programming System (LOOPS) running on the Medley Interlisp system. LOOPS is unique in that it integrates different paradigms for program development that allow the programmer to utilize the best approach to specifying data structures and manipulating them in efficient and effective ways.

LOOPS was developed at Xerox Palo Alto Research Center (PARC) to support the development of expert systems as part of their research program. LOOPS extends MEDLEY Interlisp with additional programming paradigms, which provide a powerful programming environment for multi-paradigm applications. The primary paradigms provided by LOOPS are (Stefik 2003):

- *Procedure-Oriented Programming*: Interlisp is an imperative, procedure-oriented programming language. Historically, Lisp was one of the first major programming languages based on work by John McCarthy (circa 1956). Programs consist of procedures and data, where procedures operate upon data to transform it and generate new data. Procedures consist of a set of instructions, typically executed sequentially, and formed by the syntactic rules of the language.;

## Medley LOOPS: The Basic System

- *Object-Oriented Programming* (OOP): LOOPS provides “classes and objects, class variables, instance variables, methods, multiple-inheritance mechanisms, and interactive class browsers”;
- *Access-Oriented Programming* (AOP) based on active values attached to variables which are triggered whenever certain operations are applied to the variable; and
- *Rule-Oriented Programming* (ROP) based on a simple forward-chaining(?)/backward-chaining rule language.

Programming experience has shown that it is easier to build a program that can successfully solve a problem when the programming paradigm matches the structure of the problem space of the domain of interest. Complex problems often require multiple approaches to representing data and computations where the problem can be decomposed into subproblems which can be addressed by different programming approaches, but working together can represent a more effective solution to the problem.

LOOPS was originally implemented in Interlisp-D for the D-machines developed by Xerox PARC for direct execution of both Interlisp and Smalltalk. After Common Lisp became an accepted standard for the Lisp community at large, it was recoded and extended to incorporate some Common Lisp features and was renamed Common Loops (Bobrow, Kahn, Kiczales, Masinter, Stefik, and Zdybel. 1986).

**NOTE: Capitalization Style: LOOPS used CamelCase rather than ALL CAPS as the capitalization style in its documentation. We have preserved the use of CamelCase in this volume.**



## I.1 Some History

LOOPS was based on research performed at Stanford University, Massachusetts Institute of Technology (MIT), other universities, and Xerox PARC. Some of these previous efforts include:

- Knowledge Representation Language (KRL), which developed concepts on frame-based knowledge representation concurrently with OOP concepts (Bobrow 77).
- Units, which provided a testbed for experiments in problem solving using OOP concepts (Stefik 79).
- EMYCIN, an early rule-based system for diagnostic system, which demonstrated the power of reasoning systems given incomplete data (VanMelle 80).
- Smalltalk, which pioneered many concepts in object-oriented programming and was developed at the same time as Interlisp at Xerox PARC (Ingalls 78, Goldberg 81, Goldberg 82).
- Flavors, developed at MIT for the Lisp Machine(s), and which also pioneered object-oriented programming, but also focused on non-hierarchical inheritance (Cannon 82).

Numerous papers that influenced LOOPS are mentioned in the References.

## I.2 LOOPS Paradigms

LOOPS incorporates four programming paradigms. As Stefik, Bobrow, Mittal, and Conway 1983 note, LOOPS was developed to support knowledge programming and the building of knowledge-based systems. An important principle was that different paradigms were appropriate for different purposes, e.g., different representation and problem solving purposes. This approach is substantially different

## Medley LOOPS: The Basic System

from the commonly accepted idea that a single programming paradigm is suitable for every type of problem.

The primary paradigms provided by LOOPS are (Stefik 2003):

- *Procedure-Oriented Programming*: Medley Interlisp is an imperative, procedure-oriented programming language. Historically, Lisp was one of the first major programming languages based on work by John McCarthy (circa 1956). Programs consist of procedures and data, where procedures operate upon data to transform it and generate new data. Procedures consist of a set of instructions, typically executed sequentially, and formed by the syntactic rules of the language. Interlisp-D is shown at the base of the LOOPS logo as it provides the foundation on which the rest of LOOPS is built.
- *Object-Oriented Programming (OOP)*: Information is organized into objects which are classes or instance of classes. More complex (“larger”) objects are built from simpler objects. Objects are arranged in a hierarchy or tree with the most general objects at the top of the tree. LOOPS provides “classes and objects, class variables, instance variables, methods, multiple-inheritance, and interactive class browsers)” among other elements of its ecosystem.
- *Access-Oriented Programming (AOP)* based on active values attached to variables. We can think of active values as entities associated with variables which monitor their value. An active value is an entity that can be triggered (e.g. activated) when the value of the variable is either read or changed. An active value can have an expression which performs additional computations when reading or putting a value to the variable. For example, the value of a variable can be displayed as it changes using a gauge implemented as an active value.

## Medley LOOPS: The Basic System

Gauges are described in Volume II: *Medley LOOPS: Tools and Utilities*.

- *Rule-Oriented Programming* (ROP) is a paradigm for building decision-making processes in a knowledge-based program. It is based on a simple forward-chaining(?) / backward-chaining rule language. Rule-based programming is described in Volume III: *Medley Loops: Rule-based Systems*.

### I.3 Structure of the Documentation

This volume, *Medley LOOPS: The Basic System*, the first of three describing elements of LOOPS and its application, will introduce the basic elements of LOOPS – classes, active values, and rules, and then focus on object-oriented programming in LOOPS with active values.

A second volume, *Medley LOOPS Tools and Utilities*, discusses LOOPS extensions to Medley utilities, such as windows, browsers, and the file manager.

A third volume, *Medley LOOPS: Rule-Based Systems*, will describe the LOOPSRULES features and how to write rule-based systems. *The Truckin' Game, A LOOPS Application*, describes the structure and use of LOOPS to develop an interactive game.

## Chapter One

### Introduction to LOOPS Paradigms

LOOPS supports four programming paradigms:

1. Procedure-oriented programming;
2. Object-oriented programming;
3. Data/Access programming;
4. Rule-based programming.

These are reflected in the LOOPS lattice, Figure 1-1, which depicts the major components of the LOOPS ecosystem.

# Medley LOOPS: The Basic System

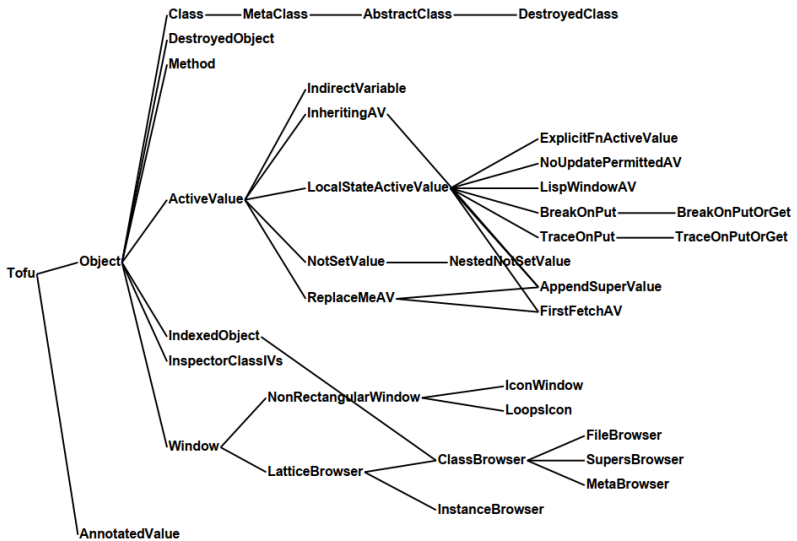


Figure 1-1. LOOPS Lattice

Source: LOOPS Reference Manual, Medley Release, November 1991

It is assumed that the reader is familiar with Interlisp procedure-oriented programming, so it will not be discussed further.

## 1.1 Introduction to Object-Oriented Programming

LOOPS provides a rich infrastructure for describing data to be represented and manipulated. The object-oriented paradigm represents programs as objects consisting of both procedures, called *methods*, and data, called *variables*. Objects have local data and local procedures to manipulate that data as depicted in Figure 1-2.

## Medley LOOPS: The Basic System

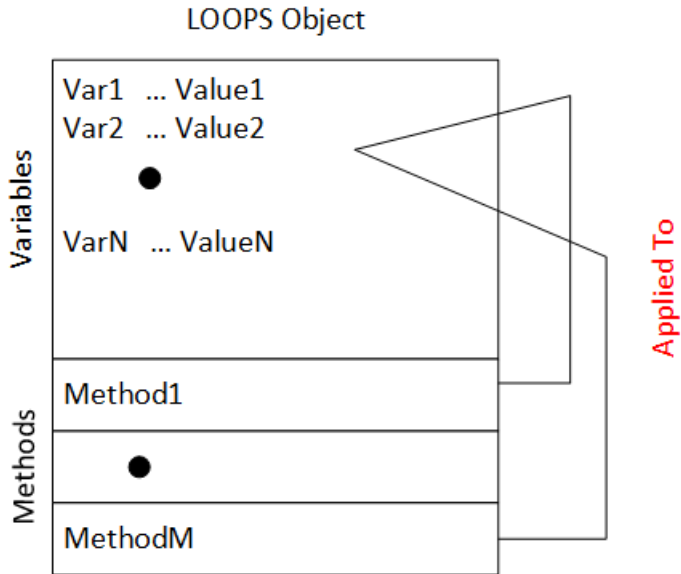


Figure 1-2. Object Representation

Source: Adapted from LOOPS Reference Manual

A LOOPS class is a (partial) description of one or more objects. Every object is an instance of exactly one class. All instances have the same structure, but are differentiated by the values of their variables.

As Stefik and Bobrow (1986) noted, actions come from sending messages between objects. A class instance responds to messages sent to it which activate methods belonging to the class. Instances of classes respond to message by invoking the methods defined in a class (or its superclass(es)). The methods are Interlisp functions (Stefik 1979). But, rather than calling the function directly, LOOPS sends a message to an object which causes it to select a method and execute it.

## Medley LOOPS: The Basic System

As Stefik and Bobrow (1986) further note, this uses the principle of data abstraction to isolate the method's implementation from its invocation and execution. The calling program does not know of the method's implementation and should not make assumptions about its implementation.

An object's methods are invoked to manipulate its variables – get and put values, sometimes transform them, and sometimes to compute another value. One object may invoke a procedure in another object by sending a message to that object and receiving a message with a result back. This section describes the object-oriented paradigm as it is implemented within LOOPS.

### 1.2 Classes and Instances

The basic structure in LOOPS is a class. A *class* serves as a description, a template if you will, for one or more similar objects. An *instance* is an object described by a class. Every object in LOOPS is an instance of exactly one class.

All LOOPS and user classes are subclasses of a class named *Class*. As we will see, one can create a new class using the function *DEFCLASS* or sending the message *NEW* to *Class*. The new class is a subclass of the class specified as its *MetaClass* (or parent class).

#### 1.2.1 Variables and Property Lists

*Variables* are containers that hold values and are used to describe a class or an instance. A *class variable* is defined in the class description. It is *shared* by all subclasses and instances of a class. It is generally used to store information about a class as a whole. An

## Medley LOOPS: The Basic System

*instance variable* contains information about a specific instance of a class. A definition for a class Point might look like:

```
(Point
  (x 0)
  (y 0)
)
```

where x and y represent the coordinates of the point. X and Y are referred to as the *descriptors* of the object Point. The value 0 is the default value that is assigned when any instance of a point is created. Both types of variables have names and values, and may have other properties.

[Note: Some O-O books use the term “properties” to describe the variables x and y. Some O-O books also use the term *attributes*. In this volume, we use descriptors because property is reserved for another aspect of a variable or method.]

Figure 1-3 represents the Player class from the Truckin game, which is described in LOOPS Volume III: Rule-Based Systems. This example is reformatted for readability.

```
(DEFCLASS Player
  (MetaClass PlayerMeta
    doc "Participant in the Truckin Simulation."
    Edited%: (* sm%: "16-SEP-83 15:42"))
  (Supers SystemPlayer)
  (ClassVariables
    (Handicap 0)
```



## Medley LOOPS: The Basic System

```

                                doc "Free time allowed to
                                compensate for slowness"))
(InstanceVariables
  (timeUsed 0
    DefaultGauge LCD
    doc "total time used so far")
  (movesMade 0
    DefaultGauge LCD
    Doc "actual number of moves made.
    Used by TimeGameMaster")
  (pendingRequest NIL
    inProcess NIL
    whenSent 0
    doc "pending request. inProcess –
    is the request already sent
    to Master for processing.
    whenSent - time when process
    sent in IDATE form)
  (maxMove 0
    doc "maxMove that can be made
    in current attempt")
  (processHandle NIL
    doc "process handle for the
    player's UserProcess")
  (startedAt 0
    doc "CLOCK time when player
    process was last started")
  (unchargedTime 0
    Doc "time not charged for in
    a given move")
  (wakeReason NIL
```

## Medley LOOPS: The Basic System

```

                Doc "value to be returned when
                    player process is resumed")
(staySuspend NIL
                Doc "set to T when player suspended
                    pending request completion")
(schCount      0
                Doc "number of times player
                    was scheduled")
(remoteMachine NIL
                doc "name of mc on which running")
))
```

Figure 1-3. Truckin Player Class

This looks complex. We will not describe it here, but just point out some of the features of the definition so you can begin to recognize them as we proceed.

The name of the class is Player. A player is a participant in the Truckin game. Its MetaClass is PlayerMeta (not shown here), but which provides a template for all types of players in Truckin. In particular, the superclass of Player is SystemPlayer (not shown here).

Player has one class variable, Handicap, which is associated with every player. Handicap has a value and a property, doc, which describes what a handicap is.

Player has instance variables which describe data the player needs to participate in the Truckin game. Each instance of Player will have these variables with values specific to that instance.

## Medley LOOPS: The Basic System

### *1.2.2 Properties*

Many of the variables and methods have properties that further describe characteristics of the variable, including constraints, documentation, etc.

A *property* is an attribute of either a variable or a method, which has a value that can provide additional information about the variable or method. Properties are stored on property lists associated with the variable or method. For example, one property you will see quite often in the LOOPS code is ‘doc’, which has a value that is a string and provides documentation about the object it is attached to.

Unlike some other object-oriented programming languages (OOPs), property lists are extendible and dynamically modifiable. This allows a programmer to add new properties to a class or instance description by adding them to the property list.

It is strongly recommended that you add properties to your variables and methods, especially, the *doc* property, which describes the role and use of the variable.

### *1.2.3 Instances*

An *instance* of a class is a description of a particular entity in a LOOPS program. Each instance inherits copies of the parent classes’ instance variables. An instance may have a local value of some or all of the parent classes’ class variables. When an instance is sent a message, LOOPS uses the selector, e.g., the name of the method in the message to find the appropriate method to use in the parent class or

## Medley LOOPS: The Basic System

one of its superclasses. Every object in LOOPS is an instance of one class. Figure 1-4 depicts a class with instances.

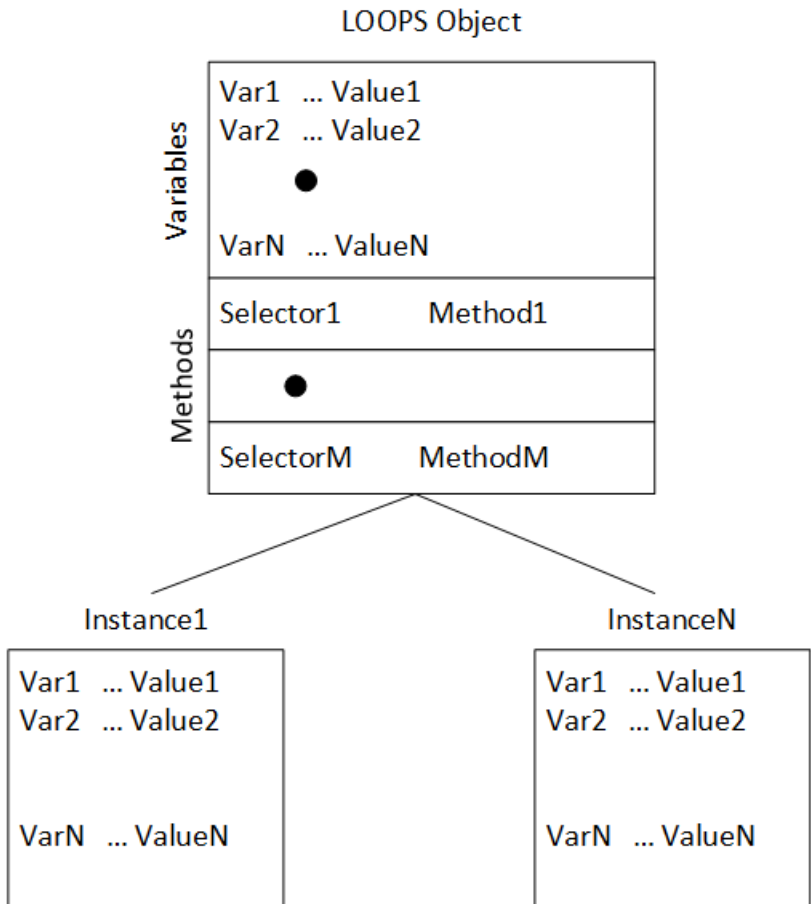


Figure 1-4. A Class and Its Instances

## Medley LOOPS: The Basic System

### 1.2.4 Methods

A class has *behavior* that is represented by a set of methods. A method is a procedural construction like a subroutine or function in other programming languages. Associated with a message is a *selector* represented by a Lisp atom. A selector typically has the same name as a method in the object, which allows LOOPS to invoke the appropriate method when an object receives a message. Figure 1-5 depicts the general structure of a method.

```
(Method  <name> (<parameters>)  
  <statement-1>  
  [<more statements>]  
)
```

A method receives its arguments, if any, by a *message*. A method may have zero or more parameters defined for it. All methods - implicitly - receive an argument, *self*, which represents the handle of the receiving object. Users do not have to encode *self* in a method definition.

All class instances have the same set of methods. Two instances of a class may respond differently to a message based on the values of their instance variables. The different behaviors are encoded in the method body.

This approach differs from procedure-oriented programming (POP) in that the object determines what method to use to respond to a message, whereas in POP, the calling procedure determines what procedure to use.

## Medley LOOPS: The Basic System

If a method is not defined within a class when invoked through one of its instances, LOOPS searches upward through the *class hierarchy* to find the method definition. If the method is not found in any of the superclasses of the class receiving the message, an error occurs.

A special type of method invocation can be used to determine the class or method name dynamically at run-time, which provides extensible flexibility in program construction.

### *1.2.5 Metaclasses*

Classes may have *subclasses*. The class that has subclasses is referred to as a *metaClass* or a superclass. When a class is sent a message, the method handling that message may be defined in the class itself or in its metaClass. If its definition resides in the class and it is not a local method, we say the class has *inherited* the method from its metaClass.

If the method does not reside in the class, but in its metaClass, its metaClass determines the response. Subclasses may have additional methods defined for them that are not specified within the metaClass. These are termed *local methods*. This allows the user to elaborate the functionality of a class. A *class hierarchy* is a class with its subclasses (and their subclasses, if they have them, recursively) arranged as a tree to show the dependency relationship.

The user should think of a metaClass as a template for possibly several classes each of which somehow distinguishes a particular set of objects with properties. In section 1.2.1, PlayerMeta is a metaClass for Player from the Truckin application.

Leverage comes from allowing different subclasses to respond to the same message, but in possibly different ways. A program can treat uniformly objects from different classes. This uses the principle of modularity by reusing pieces of code in more than one class.

## 1.2.6 Abstract Classes

Another class available in the LOOPS ecosystem is **AbstractClass**. Abstract classes are useful when creating classes that implement general functionality, which are then specialized into instantiable classes. Instances of this class are classes that cannot be instantiated. An example of an AbstractClass is `ActiveValue`, which is described in Chapter 7.

Figure 1-6 is an example of a MetaClass and its instances.

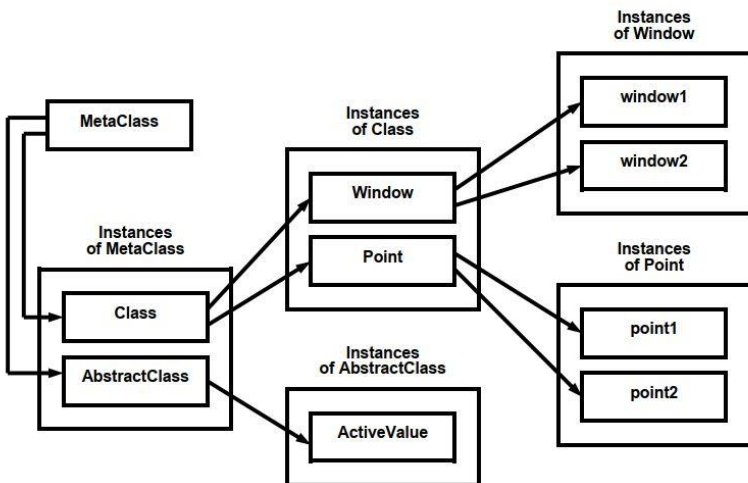


Figure 1-6. A MetaClass Example

Source: PARC LRM91

**NOTE: It seems SuperClass was used in earlier documentation, but MetaClass was introduced to explicitly capture the idea of a class of classes. Thus, in LOOPS documentation you will find references to both superclasses and metaclasses. Our understanding is that these names refer to the same type of object in LOOPS.**

## 1.3 Generic Class Description

A *generic class description* has the form presented in Figure 1-7. Figure 1-8 depicts a graphic picture using a notational mechanism borrowed from UML.

```
(DEFCLASS <class-name>
  (Supers)           ; a list of immediate superclasses (metaclasses).
  (ClassVariables   ; a list of variables that describe this class and
                    ; differentiate it from its superclasses.
    (<CV-1> <CValue-1> [doc <description-1>])
    ...
    (<CV-n> <CValue-n> [doc <description-n>]))
  (InstanceVariables ; a list of instance variables which are inherited
                    ; (defined in) each instance of the class.
    (<IV-1> <IValue-1> [doc <description-1>])
    ...
    (<IV-n> <IValue-n> [doc <description-n>]))
  (Methods          ; a list of methods implementing the behavior ;
                    ; of this class.
    (<method-name-1> <argument-list-1> [doc <description>])
```



## Medley LOOPS: The Basic System

```
...  
(<method-name-n> <arguments-list-n> [doc <description>])  
)
```

Figure 1-7. Generic Class Description



Figure 1-8. Graphic Depiction of a Class

### 1.4 Class Hierarchy

Classes are arranged in a *class hierarchy* beginning with the most general class, Object, and organized as an *inheritance network* descending from it. We use the term ‘network’ here because LOOPS allows a class to descend (inherit from) multiple superclasses.

## Medley LOOPS: The Basic System

We said that every instance is defined by a single class. If we want to define a combination of classes – each contributing some attributes to the concept being described, we must define a new class that inherits from each of the superclasses whose attributes are to be combined to describe the entity.

### *1.4.1 The Concept of Inheritance*

*Inheritance* is a major organizing principle in general-purpose OOPLs. It allows programmers to specify many objects that are “almost like” other objects, but differ only in a few incremental changes. Thus, the programmer does not have to specify redundant information that is common to every subclass of a class. It also minimizes errors because information only needs to be updated in one place. The structure of classes and subclasses is referred to as an *inheritance network*.

Inheritance adheres to the following principles described in Table 1-1.

**Table 1-1. Basic Inheritance Principles**

An instance object inherits the instance variables and message responses from its superclass.
All descriptions in a class are inherited by a subclass unless overridden in a subclass.
Methods do not have to be defined in a subclass unless their behavior is being overridden.
Class variables do not have to be assigned a value in a subclass unless their value is being overridden.

## Medley LOOPS: The Basic System

When a class variable or a method is referenced in an instance of a subclass, a search is made up the class hierarchy for the class in which the variable or method is defined to retrieve the value.

### *1.4.2 Simple Hierarchy*

A simple inheritance hierarchy consists of one superclass for each class. A superclass may have multiple subclasses. In this case, the hierarchy is organized as Directed Acyclic Graph (DAG). Here is an example using a general notation:

```
(DEFINECLASS Point
  (x 0)
  (y 0)
  ... methods for manipulating the instances
)

(DEFINECLASS ColoredPoint
  (Supers Point)
  (color "blue")
  (methods
    (setColor (newcolor) ... docs set color to newcolor)
  )
)
```

As we see, Point has two instance variables (IVs): x and y. In the class definition of Point, the default values are 0 for x and 0 for y. When a new instance of Point is created, unless otherwise specified, the values of x and y are initially set to 0.

## Medley LOOPS: The Basic System

In `ColoredPoint`, a new attribute, `Color`, which further characterizes a `Point`, is defined. It is used to define a `ColoredPoint` as a subclass of `Point`. Thus, points can be colored or not depending on the class the new point belongs to.

Now, let us create a point in 3-dimensional space, which we will call `Point3D`. This subclass `Point3D` of `Point` will have a new IV, `z`, which represents its location along the `z`-axis. We depict this arrangement in Figure 1-9.

We note several things in this figure:

1. Instance `pt1` is an instance of class `Point`;
2. Class `Point` has a class variable (CV) `lastPoint`;
3. Class `Point` has two instance variables: `x` and `y`;
4. `Pt1` has instance variables `x` and `y`;
5. Class `Point3D` is a subclass of class `Point`;
6. Class `Point3D` does not have CV `lastPoint`;
7. Class `Point3D` declares a new IV `z`;
8. Implicitly, class `Point3D` inherits IVs `x` and `y` from class `Point`;
9. Instance `pt2` is a subclass of class `Point3D`;
10. `Pt2` has three IVs: `x`, `y`, and `z` inherited from class `Point3D`;
11. Class `Point3D` has the same selector `A1`, which is modified from class `Point` and new selector `C`.

# Medley LOOPS: The Basic System

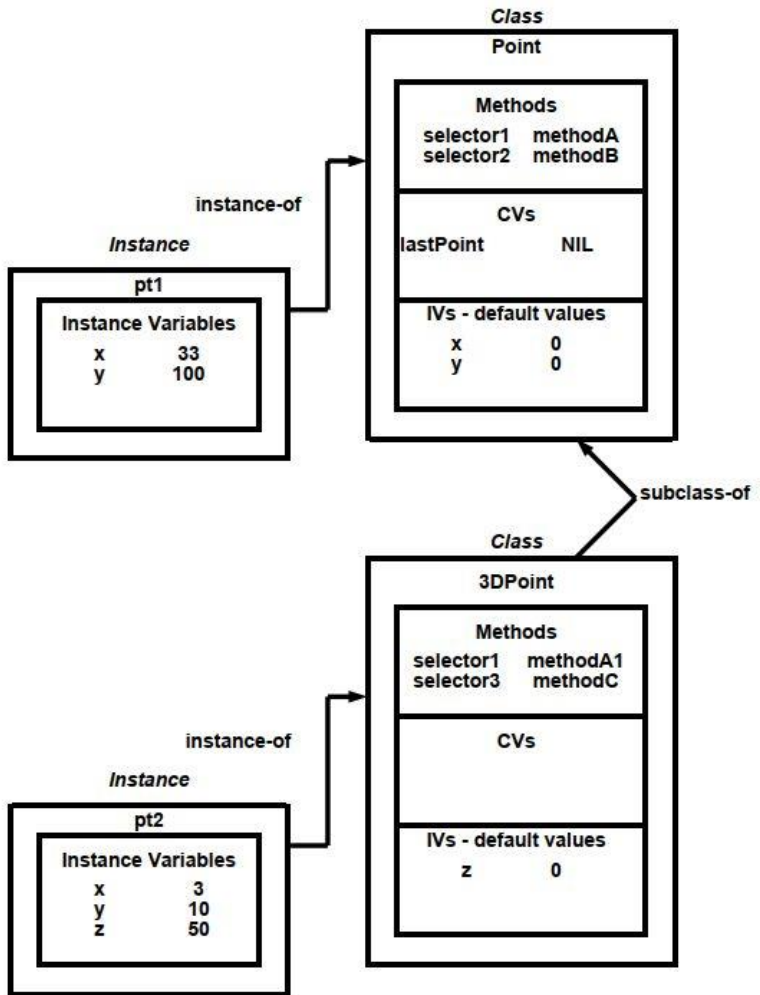


Figure 1-9. The Subclass Point3D

Source: PARC LRM91

As Figure 1-9 demonstrates, a simple hierarchy consists of a single superclass for a class. All instance variables specified in the superclass are inherited and present in the subclass. Of course, the superclass may have its own superclasses and, thus, has all of their instance variables as well.

### **1.4.2.1 Multiple Superclasses**

A LOOPS class may have more than one superclass. Multiple superclasses allow separation of functionality across superclasses, but support the concept of *object composition*, which allows a new class to combine the features and methods from many classes. This powerful inheritance mechanism allows substantial flexibility in defining applications using a variety of combinations of classes.

The concept of multiple inheritance is both powerful and fraught with danger.

### **1.4.2.2 Name Conflict Resolution**

When more than one Superclass is specified for a new class, it is possible that some of the names of variables in the superclasses may be the same. The superclasses are specified as a list of classes when defining the new class.

When referencing a variable or method that has been specified in one or more superclasses, which superclass should be used to provide the variable or method to any operations in the class? LOOPS uses the order of the names in the superclass list to establish the precedence for search when looking for the value of an variable or method in a

## Medley LOOPS: The Basic System

superclass - from left to right in the superclass list. The first occurrence of the name in a superclass is the one that is used to resolve the *name conflict resolution* to determine the value.

### 1.4.3 A Complex Hierarchy

LOOPS allows a subclass to inherit from multiple superclasses. Consider the following example:

```
(DEFINECLASS Point3D
  (Supers Point2D)
  (z 0)
)

(DEFINECLASS ColoredPoint3d
  (Supers Point3d ColoredPoint2D)
)
```

Class ColoredPoint3D has the IVs inherited from Point3D and from ColoredPoint. It responds to setColor by traversing up the hierarchy to ColoredPoint. It responds to the methods in class Point3D by traversing up the hierarchy because Point3D is named before ColoredPoint.

## 1.5 Interlisp Objects

To access an Interlisp object, one needs to have a *handle* (sometimes, called a “pointer”) to it. The handle should be assigned to an Interlisp variable or a LOOPS instance variable, usually via a SETQ statement. Lisp objects can be passed as arguments to functions to be

## Medley LOOPS: The Basic System

examined and operated upon by Lisp functions by passing their handles.

<b>Handle</b>
<p>A <i>handle</i> is a reference to an object in memory. A handle is NOT a pointer, although the term has sometimes been used interchangeably with ‘pointer’. Rather, a handle is a value that allows the Medley Interlisp run-time system to locate an object – either Medley Interlisp or Medley LOOPS - in virtual memory. A handle has a value that can serve as an index into a table of objects, some of which are located in physical memory and some of which are located in virtual memory. The object table entry has fields describing attributes of the object.</p>

A handle can reference a Lisp object or a LOOPS object. Which type of object is determined by the values of an attribute in the object table.

### *1.5.1 Testing for Lisp Data Types*

LOOPS defines three Lisp data objects: annotatedValue, class, and instance. LOOPS provides macros to test the data type of a Lisp object. We can load RichardI from the Plantagenet data set:

```
(* ; "This data set describes the Plantagenet Family")
```

```
(* ; "Prepared by Steve Kaisler")
```

```
(DEFINE-FILE-INFO -PACKAGE "INTERLISP" -READTABLE  
"INTERLISP" -BASE 10)
```

```
(* ; "Richard I")
```



## Medley LOOPS: The Basic System

```
(SETQ RichardI (SEND Person New))
```

```
(SEND RichardI SetName 'RichardI)
```

```
(PutValue RichardI 'Gender 'Male)
```

```
(PutValue RichardI 'Birthdate (LIST 09 08 1157))
```

```
(PutValue RichardI 'Deathdate (LIST 04 06 1199))
```

```
(putFather RichardI 'HenryII)
```

```
(putMother RichardI 'EleanorOfAquitaine)
```

```
(PRINT "Loaded RichardI")
```

```
STOP
```

```
2/11+ (LOAD 'RichardI.txt T T)
{DSK}<home>steve>LOOPS-MAIN>RichardI.txt;1
;
;
;
;
#,($& Person (92 . 65280))
#,$ RichardI)
Male
(9 8 1157)
(4 6 1199)
HenryII
self is an unbound variable.
```

**(NOTE: Remember files to be loaded by Interlisp must be terminated by a STOP atom.)**

The SETQ statements assigns the handle to a variable named RichardI. The SEND line assigns the name to the variable which allows it to be used in LOOPS expressions without using the (\$ ...) notation (see below).

## Medley LOOPS: The Basic System

The last two arguments, “T T”, direct Interlisp to print the results of each statement as it is executed.

In this example, we see that RichardI is an instance of Person, which is a class. The first line, #,(\$& Person (92 . 65200)), after the “;’s”, specifies the handle, which allows the run-time system to locate the object named RichardI in virtual memory. The second line, #, (\$ RichardI), is the result of the SetName message. It specifies the name of this instance is RichardI, so we can use RichardI to reference this instance in other Lisp expressions. The other lines present the values of other attributes describing RichardI

Note: To send a message to an object, we can use the following form in the Interlisp EXEC window:

```
2/57← (SETQ RichardI (← Person SetName 'RichardI))
#,$C RichardI
```

On the keyboard, we would type “... (\_ Person ...), but this is translated by the Interlisp readtable into “←”. Alternatively, the function to send a message to an object is SEND as indicated in the listing above. SEND is described in Section 3.1.

Note: The Plantagenet data set is presented in Appendix ??.

### 1.5.1.1 Testing for Lisp

To test if the value of a variable is a Lisp object, you can use the **Object?** macro as follows:

Macro:        Object?

## Medley LOOPS: The Basic System

Arguments: X, an arbitrary lisp variable.  
Value: Returns T, if a Lisp Object;  
otherwise, NIL.

For example, testing RichardI:

```
2/12+ (Object? RichardI)  
T
```

So, yes, RichardI is a Lisp object.

Note: Since we assigned the name RichardI to the object RichardI, we can now reference it by its name.

### 1.5.1.2 Testing for a Class

To test if a variable is a class, you can use the macro **Class?**:

Macro: Class?  
Arguments: X, a possible class.  
Return: Returns T, if X is a class;  
otherwise, NIL.

For example, to test RichardI as a class:

```
2/13+ (Class? RichardI)  
NIL  
.....
```

So, RichardI is not a class.

### 1.5.1.3 Instance?

To test if a variable is an instance, you can use the macro **Instance?**:

Macro:        Instance?  
Arguments:    X, a possible instance.  
Return:       Returns T, if X is an instance;  
               otherwise, NIL.

For example, is RichardI an instance of a class?

```
{ 2/15+ (Instance? RichardI)  
T
```

So, RichardI is an instance of a class.

### 1.5.1.4 AnnotatedValue

AnnotatedValue is a class that allows an annotatedValue to be treated as an object. An annotatedValue was an Interlisp-D data type that wrapped each ActiveValue instance. AnnotatedValue will be described in Section 7.2.6).

To test if a variable is an annotatedValue, you can use the macro **annotatedValue?**:

## Medley LOOPS: The Basic System

Macro:        AnnotatedValue?  
Arguments:    X, a possible annotatedValue.  
Return:       Returns T, if X is an annotatedValue;  
              otherwise, NIL.

```
2/27+ (AnnotatedValue? RichardI)
NIL
```

So, RichardI is not an annotatedVallue.

### 1.5.1.5 Understands

To test if an object will respond to a *self* message, you can use the macro **Understands**:

Macro:        Understands  
Arguments:    <object>, an instance or a class.  
              <message>, a method name.  
Return:       T, if self is a class or an instance of a class that  
              understands the message; otherwise, NIL.

```
2/22+ (SEND RichardI Understands 'GetValue)
NIL
```

RichardI does not understand GetValue, because GetValue is a function, not a message. An alternative test is:

```
2/32+ (SEND ($ RichardI) GetValue 'Father)
(GetValue #,($& Person (YZaB=7X1.0.0.1+; . 4)) Father) not possible.
```

## Medley LOOPS: The Basic System

### *1.5.2 Assigning Names to LOOPS Objects*

To manipulate Lisp objects, one can also assign a “LOOPS name” to it, whence it can be referenced by that name. A name can be assigned to an Interlisp object via the message **SetName**.

Message:	SetName
# Arguments:	1
Arguments:	1) <name>, the name to be assigned to the object.
Value:	Sets the LOOPS name <name> to refer to the Interlisp object.

LOOPS names are unique in a LOOPS environment. A LOOPS environment establishes a name space to partition the total name space of all possible name strings. The global variable *CurrentEnvironment* specifies a description of the current environment.

An attempt to assign a name that is already in use within the current environment generates an error if the **ErrorOnNameConflict** is set to T.

If **ErrorOnNameConflict** is NIL, and an object already has the specified name, the name is unassigned from the existing object and assigned to the new object, without generating an error.

For example, suppose ILV1 is a Lisp variable. You can assign a LOOPS Name to the Interlisp object whose handle is its value via:

```
(<- ILV1 SetName 'SHKFoo)
```

## Medley LOOPS: The Basic System

Thereafter, the user can refer to this object as ( $\$$  SHKFoo), which will return the handle for the Lisp object. Here is an example:

```
2/47+ (SETQ Person (DefineClass 'Person NIL ($ Class)))
#,$($C Person)
2/48+ (← Person SetName 'Person)
NIL
2/49+ Person
#,$($C Person)
2/50+
```

The LOOPS name Person was assigned as the name of the LOOPS class object Person. Line 47 of the example shows that Person is a class, indicated by  $\$C$ . Once a LOOPS name is assigned to the class object in Line 48, it can now be referenced by its LOOPS name as seen in Line 49.

The user can use a computed LOOPS name. For example, let lisp variable X have the atom RichardI. Using the form ( $\$!$  <expr>), then ( $\$!$  X) is translated as ( $\$$  RichardI). We can then set the value of RichardI's Father via:

```
2/45+ (SETQ X RichardI)
#,$($& Person (|YZaB=7X1.0.0.1+;| . 4))
2/46+ (PutValue ($! X) 'Father 'Tom)
Tom
```

### *1.5.3 Class Objects and LOOPS Names*

Class objects are automatically given LOOPS names when they are created.

## Medley LOOPS: The Basic System

### *1.5.4 NamedObject*

Any LOOPS object can be named. The class `NamedObject`, usually used as a superclass, allows a LOOPS object - either class or instance - to have a name. `NamedObject` has only one instance variable, **name**.

`GlobalNamedObjects` are named in the global name table. They are named independently of the environment they reside in, whereas `NamedObjects` are only known in their local environment. The names in one local environment may be reused in another local environment without conflict.

### *1.5.5 DatedObject*

`DatedObject` has active values associated with its instance variables, so that they are filled in when an object is created:

**created**, the date and time of creation of the object

**creator**, the USERNAME of the creator of the object

A LOOPS object should have `DatedObject` as a super when the environment in which it resides is shared by multiple users, so that the individuals who created objects can be identified. This is useful when an individual is responsible for objects that she or he creates.



## **1.6 System Variables and Functions**

When LOOPS is loaded, several LOOPS system variables are set by the LOADLOOPS function. LOOPS system variables are described in Table 1-2a and LOOPS directory variables are described in Table 1-2b.

## Medley LOOPS: The Basic System

**Table 1-2a. LOOPS System Variables**

Variable	Description
LoopsVersion	<p>Specifies the current release of LOOPS.</p> <pre>2/46← <b>LoopsVersion</b> Lyric/Medley</pre>
LoopsDate	<p>The date when LOADLOOPS was executed to create the current instance of LOOPS.</p> <pre>2/47← <b>LoopsDate</b> "17-Oct-2023 12:20:29"</pre>
*FEATURES*	<p>LOADLOOPS added the symbol LOOPS to this variable.</p> <pre>2/84← *FEATURES* (LOOPS :LOOPS LOOPS :LOOPS :CLOS :XEROX-MEDLEY :INTERLISP :XEROX EE-FLOATING-POINT :MEDLEY)</pre>
LoadLoopsForms	<p>A list of forms that were evaluated when LOOPS was loaded. Initialized to NIL using the File Manager command INITVARS.</p>
LispUserFilesForLoops	<p>A list of files required by LOOPS.</p> <pre>2/88← <b>LispUserFilesForLoops</b> (GRAPHER)</pre>
OptionalLispUserFiles	<p>A list of files that is loaded when LOOPS is loaded. Initialized using the File Manager command INITVARS.</p>

## Medley LOOPS: The Basic System

**Table 1-2b. LOOPS Directory Variables**

Directory Variable	Description
LOOPSDirectory	Initialized to the directory from which the file LOADLOOPS is loaded using the File Manager command INITVARS. Depends on where LOOPS is installed in the user's system.
LOOPSLIBRARYDIRECTORY	The directory where the LOOPS library files reside. Depends on where LOOPS is installed in the user's system.  <pre> 2/80+ LOOPSLIBRARYDIRECTORY {DSK}&lt;home&gt;Steve&gt;loops-main&gt;LIBRARY </pre>
LOOPUSERSDIRECTORY	The directory where the LOOPS User's Modules reside. Depends on where LOOPS is installed in the user's system.  <pre> 2/82+ LOOPUSERSDIRECTORY {DSK}&lt;home&gt;Steve&gt;loops-main&gt;USERS </pre>
LOOPUSERSRULESDIRECTORY	The directory where the LOOPS Rules User Module resides.  <pre> 2/86+ LOOPUSERSRULESDIRECTORY {DSK}&lt;home&gt;Steve&gt;loops-main&gt;USERS&gt;RULES </pre>
LoopsPatchFiles	A list of files passed to FILESLOAD that is used during the loading of LOOPS. Initialized to NIL.
LOOPFILES	The list of LOOPS files loaded by LOADLOOPS when building a LOOPS sysout.
ClearAllCatches	A list of forms each of which is evaluated within a call to the function ClearAllCatches. Initially set to NIL

## Medley LOOPS: The Basic System

LOOPSFILES contains the list of files loaded by LOADLOOPS.

```
2/88+ LOOPSFILES  
(LOADLOOPS LOOPSSITE BLOCKLOOKUP LOOPSSPEEDUP LOOPSDATATYPES LOOPSSTRUC LOOP  
SPRINT LOOPS-FILEPKG LOOPSSACCESS LOOPSSUID LOOPSEEDIT LOOPSMETHODS LOOPSKERNEL  
  LOOPSSACTIVEVALUES LOOPSSUTILITY LOOPSSINSPECT LOOPSSWINDOW LOOPSSBROWSE LOOPSSDE  
BUG LOOPSSUSERINTERFACE LOOPSS-TTYEDIT INSPECT-PATCH)
```

## Chapter Two

# Object-Oriented Programming in LOOPS

This chapter describes how to do object-oriented programming in LOOPS.

Note: Entities enclosed in “[ ]” are considered optional arguments.

Convention: In the following examples, the “\_” that appears in the LOOPS Manual is represented by the solid left arrow that appears in the Medley Interlisp Exec window.

### 2.1 Creating a New Class

LOOPS provides several methods for creating a new class that provide the user with control over the definition details.

#### *2.1.1 Creating a New Class via New*

The method for creating a new class is to send the message `New` to a metaClass. The metaClass ‘Class’ is used to define new classes. The format is:

```
(_ <metaClass> New <className> [<supersList>])
```

## Medley LOOPS: The Basic System

where: <metaClass> is either 'Class' or an existing class in the workspace.

**New** is the message to create a new class object.

<className> is the name of the new class.

<supersList> is a list of superclasses in the class hierarchy  
<metaClass>.

As an example, create a new class named 'Person':

```
2/8+ (SETQ Person (← ($ Class) New 'Person))
#,$C Person)
2/10+
```

Notice that the left arrow in the example above is used to send message to an object. However, in Word, it appears as an “\_”. This is due to the font differences between Word and Interlisp.

LOOPS returns a handle to the Person object of the form #,\$C Person) where the \$C indicates a class. By storing this handle in a Lisp atom, we can reference it later. The new class is a subclass of Class.

If the <supersList> is NIL or not specified then then the superclass of the new class is set to its <metaClass>.

The form (\_ <object> <selector> <arg1 ... arg2) is a compiler MACRO that is expanded into a function call of the form:

```
(APPLY* (FetchMethodOrHelp <object> '<selector>)
        <object>
        <selector>
        arg1 ... argN)
```

## Medley LOOPS: The Basic System

You can also use this form within a program if you want to edit the selector and arguments to the selector for several selectors associated with the object.

### *2.1.2 Creating a New Class with NewClass*

Another approach is to use the LOOPS function **NewClass**, which creates a class of the given name. Its format is:

Function:           NewClass  
Arguments:         <classname>, the name for the class.  
                    <metaclass>, the parent class of the new class.  
Return:            The class record.

This function does not check for an existing definition of the class.

```
2/3+ (SETQ PERSON (NewClass 'Person ($ Class))
)
#, ($C Person)
```

### *2.1.3 Creating a New Class with DefineClass*

Another approach is to use the LOOPS function **DefineClass**, which creates a class of the given name. Its format is:

Function:           DefineClass  
Arguments:         <classname>, the name for the class as a  
                    littatom.  
                    <supers>, a list of superclasses or NIL.  
                    <object>, the parent class of the new class.

## Medley LOOPS: The Basic System

Return:           The new class handle.

```
2/47+ (SETQ Person (DefineClass 'Person NIL ($ Class)))
#,($C Person)
2/48+ (← Person SetName 'Person)
NIL
2/49+ Person
#,($C Person)
2/50+
```

If <supers> is not NIL, it is a list of class designators each atom of which is a class designator.

If <supers> is a list of classes, the class has multiple superclasses.

If some superclass is not yet a class, then DefineClass asks the user to correct the list.

```
2/35+ (SETQ APERSON (DefineClass 'aPerson (LIST 'NotAClass) ($ Class)))
Should NotAClass be defined as a new class ? No
NotAClass must be replaced in definition, or defined as a class.
```

```
INTERLISP-ERROR
```

```
In ERROR:
(NotAClass) is a bad supers list
```

```
2/36+
```

The default for supers is (Object) if <object> is Class or is (Class) if <object> is MetaClass or one of its subclasses.

```
2/32+ (SETQ BPERSON (DefineClass 'bPerson NIL ($ MetaClass)))
#,($C bPerson)
```



## Medley LOOPS: The Basic System

```
2/34+ (PP bPerson)
CLASSES definition for bPerson:

(DEFCLASSES bPerson)
(DEFCLASS bPerson
  (MetaClass MetaClass Edited%:  **COMMENT** )
  (Supers Class))

NIL
```

**NOTE: DefineClass yields two different declarations if one specifies (\$ Class) as an <object> without specifying <supers> as a NIL. For example:**

```
2/73+ (DefineClass 'State '(Class))
#,$C State)
2/74+ (DefineClass 'State2 NIL '(Class))
#,$C State2)
2/75+ (PP State)
VARS definition for State:

(RPAQQ State #,$ State))
CLASSES definition for State:

(DEFCLASSES State)
(DEFCLASS State
  (MetaClass Class Edited%:  **COMMENT** )
  (Supers Class))

NIL
2/76+ (PP State2)
CLASSES definition for State2:

(DEFCLASSES State2)
(DEFCLASS State2
  (MetaClass (Class)
    Edited%:  **COMMENT**
  )
  (Supers Object))

NIL
```

**In line 73, "State" is defined as a subclass of "Class", whereas in line 74, "State2" is defined as a subclass of "Object". This cause considerable confusion concerning how to access "State" in a program.**

The class is built with the Edited: property containing the date and time it was created and the value of the variable INITIALS. To track which users create new classes in a multiuser application, you should set the variable INITIALS to those of the person creating new classes.

The new class has no class variables, instance variables, or methods associated with it.

The variable LASTWORD is set to <classname>, which is added to USERWORDS for spelling escape completion. LASTWORD tracks the last object defined by a user.

#### *2.1.4 LispClassTable*

LispClassTable is a hash table, which is a list of classes based on object type. For example,

```
2/88+ (PP LispClassTable)
VARS definition for LispClassTable:
(RPAQQ LispClassTable #<Hash-Table @ 137,106452>)
NIL
```

## **2.2 Creating an Instance of a Class**

## Medley LOOPS: The Basic System

We can create an instance of a class by sending the message **New** to the class. The format of this method is:

Method:    **New**  
Arguments: <class>, the handle of the class.  
          <name>, a LOOPS Name for the new class.  
          <supers>, a list of classes.  
          <init1>,  
          <init2>,  
          <init3>.  
Return:    The handle of the new class.

Upon return, the new class is sent the message `newClass` with the arguments <init1>, <init2>, <init3>.

### *2.2.1 Simple Form*

For example, the simplest form of creating a new class to use the **NEW** message:

```
2/6+ (SETQ HenryII ($ Person) NEW 'HenryII)
HenryII
2/7+ (PP HenryII)
VARS definition for HenryII:
(RPAQQ HenryII #,($ Person))
NIL
```

Note that specifying `HenryII` as the name of the new class only allows you to access the new class via `($ HenryII)`. By setting the handle to `HenryII` (the variable), you can now access it as shown above.

## Medley LOOPS: The Basic System

The initial values of the instance variables of the instance are usually taken from the instance variables as defined in the class definition.

We can set the gender of HenryII to Male using PutValue (see Section 3.5.2.1):

```
2/33+ (PutValue HenryII 'Gender 'Male)
Male
2/34+ (pp HenryII)
pp {in EVAL} -> PP ? yes
VARS definition for HenryII:
(RPAQQ HenryII #,($ HenryII))
INSTANCES definition for HenryII:
(DEFINST Person (HenryII (DZ%↑V9TaLU1.0.0.9V7 . 15))
 (Gender Male))
```

Note that Interlisp is case sensitive. PP is the name of a function that prettyprints the definition of an object. In line 34, 'pp' was specified, which is not the name of a function. Interlisp, using DWIM, asks the user to correct the function name. When the user types 'yes', it uses PP.

### *2.2.2 Creating a New Class as a Subclass*

To create a new class as a subclass, you can provide a value for the <supers> argument:

```
(SETQ myWindow (SEND ($ Class) New myClass '(Window))
```

which sets myWindow with the handle of the new class. myWindow is a subclass of Window. After it is created, it is sent the message newClass.

## Medley LOOPS: The Basic System

### *2.2.3 Initializing a New Class*

After creating a new class, it can be initialized using the `init` variables provided in the method format. After the class is created, it is sent the message `newClass` with the three `init` variables as arguments.

First, create a new class with `DefineClass` as follows:

```
(DefineClass 'SHKClass NIL '(Class))
```

Second, assign the method `newClass` to `myClass` as follows:

```
(DefineMethod ($ SHKClass)
  'newClass
  '(<init1> <init2> <init3>)
  '(PROGN
    (PutClass self <init1> 'prop1) Self)
)
```

Third, send `SHKclass` the message `newClass`:

```
(SEND ($ SHKclass)
  New
  'testClass
  NIL
  "steve's class"
)
```

### 2.3 Creating an Instance of a Class Using SEND

An alternate method is to use the function SEND to send the message to the class. Thus, we can create a new instance as follows:

```
(SETQ HenryII (SEND Person New 'HenryII))
```

```
2/19+ (SETQ HenryII (SEND Person New 'HenryII))
#,$(& Person (DZ%↑V9TaLU1.0.0.9V7 . 15))
```

We can assign a name to HenryII (the instance) of HenryII (the name) as:

```
#,$(& Person (DZ%↑V9TaLU1.0.0.9V7 . 15))
2/31+ (SEND HenryII SetName 'HenryII)
#,$(& Person (DZ%↑V9TaLU1.0.0.9V7 . 15))
2/32+ (PP HenryII)
VARS definition for HenryII:
(RPAQQ HenryII #,$ HenryII))
INSTANCES definition for HenryII:
(DEFINST Person (HenryII (DZ%↑V9TaLU1.0.0.9V7 . 15))
)
```

Note that the handle for an instance is more complex than a class. Users do not need to interpret the meaning of a handle in their use of Interlisp as it is translated by the runtime system into an address in memory as needed.

## 2.4 Instance Variables and Properties

An instance has two types of variables:

- Its *private instance variables*, and
- The *class variables* that it shares with all instances of the class.

When a class is defined, it specifies the instance variables for each instance of a class. An instance variable in a class may have a value. When accessing an IV through an instance, if the IV is not defined in the instance, LOOPS looks up the class hierarchy for it in one of the superclasses and uses the value found there.

By *private instance variables*, we mean IVs that have a value specific to that instance of a class. A private IV, then, is a copy of the IV defined in the class, but has a, perhaps, unique value for that instance. Since searching up the hierarchy required additional operations, performance could be improved by creating a copy of the IV in the instance.

A *class variable* was defined in the class and had the same value for all instances of the class. Accessing a CV required a search up the class hierarchy to find its definition. CVs could be cached in instances in order to improve access performance.

### 2.4.1 Instance Variable Operations

There are two types of operations upon these variables:

- Getting operations to retrieve a value of an instance variable, or
- Putting operations that set the value of an instance variable.

## Medley LOOPS: The Basic System

LOOPS provides a wide variety of functions for these operations which provides the programmer with substantial flexibility in manipulating the values of these instance variables. Section 3.5.2 describes the operations.

For Person, we can specify an instance variable (IV) named “Birthdate” and a method that computes the person’s age from the current date and their birthdate. But, this method should also check to see if DeathDate is defined, and then compute the person’s age at death by subtracting the birthdate from the deathdate. Both birthdate and deathdate should be private IVs because they should have unique values for each instance.

### 2.5 LOOPS Names

LOOPS maintains a separate name space for LOOPS objects from the Interlisp name space. Names are stored in a separate object name table for LOOPS, which is distinct from the object name table for Interlisp objects.

### 2.6 Instance Names

Instances are not created with names. To access them, one needs to keep a handle to reference an instance. One way to do this is to supply a name when the instance is created by assigning the handle to a variable. For example:

```
2/73← (SETQ window 1 (← ($ Window) New))
#,$& Window (DL%0.UC1.0d5.Ji9 . 22)
```



## Medley LOOPS: The Basic System

which creates an instance of the class **Window** and stores its handle in the Interlisp variable **window1**. A program can use **window1** to reference the instance.

```
2/75← (PP ($ window1))
EXPRESSIONS definition for ($ window1):
($ window1)
NIL
```

A second approach is to use a LOOPS name. One can assign a LOOPS name when the instance is created as follows which allows the instance to be referred to be the LOOPS form (**\$ Window2**). Similarly, one can also assign the handle to a variable named **window2**:

```
2/69← (← ($ Window) New 'Window2)
#,$& Window (DL%0.UC1.0d5.Ji9 . 21))
```

Alternatively, one can use the message **SetName** to assign a LOOPS name to an instance if you have a pointer to that object. For example:

```
2/78← (SEND ($ Window2) SetName 'Window2)
#,$& Window (DL%0.UC1.0d5.Ji9 . 21))
```

which allows the program to reference the instance as **Window2**.

### 2.6.1 Working with LOOPS Names

There are several forms for working with LOOPS names as described in Table 2-1.

**Table 2-1. LOOPS Forms for Name Manipulation**

Form	Type	Description
\$	Nlambda and macro	Specifies that the LOOPS name will be used; does not evaluate its argument.
#!	Function	Specifies that the LOOPS name will be used; evaluates its argument.
SetName	Method	Assigns a LOOPS name to an instance.
UnSetName	Method	Removes a name from an instance.
Rename	Method	Changes the name of an instance.
GetObjectNames	Function	Returns the names of an instance, including its UID.

Some examples are:

```

NIL
2/3+ ($ EdwardIII)
#,$& Person (FE%nPGZU1.0.0.n\< . 13))
    
```

```

}2/11+ ($! EdwardIII)
}#,$& Person (FE%nPGZU1.0.0.n\< . 13))
    
```

The variable **ErrorOnNameConflict** causes a break when an attempt is made to assign a LOOPS name to an instance that already has a LOOPS name. The initial value is NIL.

## 2.7 Editing a Class

Once a new class has been defined, its structure can be further elaborated using the editing capability in LOOPS. The format is:

## Medley LOOPS: The Basic System

(\_ (\$ <className>) Edit)

We can use the function EC to invoke the structure editor as:

```
SEdit Account Package: INTERLISP
((MetaClass Class Edited%:                ; Edited 25-May-2022 13:57 by
                                           ; root
      )
 (Supers Object)
 (ClassVariables)
 (InstanceVariables)
 (MethodFns))
```

For example, here is the Edit window for State:

```
SEdit State Package: INTERLISP
((MetaClass Class Edited%:           ; Edited 5-Jun-2024
                                     ; 09:55 by Steve
   )
 (Supers Class)
 (ClassVariables)
 (InstanceVariables
  (Description "A component of the United States" doc
   (* IV added by STEVE))
  (StateCapital NIL doc (* IV added by STEVE))
  (PartOf NIL doc (* IV added by STEVE))
  (Population NIL doc (* IV added by STEVE))
  (DateOfPopulation NIL doc (* IV added by STEVE))
  (Cities NIL doc (* IV added by STEVE))
  (Counties NIL doc (* IV added by STEVE)))
 (MethodFns))
```

## 2.8 The Class Record

A class is described by a *class record* in which several fields store data about CVs and CIVs:

- *cvNames* is a list of all the class variables;
- *cvDescrs* is a list of the descriptors for each of the CVs; and
- *localIVs* is a list of instance variables local to this class.

A function fetches *cvNames* of a class to access the list of class variable names or fetch *cvDescrs* to access the list of class variable descriptors. *GetSourceCV* is used to fetch the class variables defined in the class.

## Medley LOOPS: The Basic System

```
2/68+ fix 67
2/68+ (GetSourceCVs ($ Window))
((TitleItems NIL doc "special items to be done if in title part of window") (LeftB
uttonItems ((Update (QUOTE Update) "Update window to agree with object IVs")) doc
"Items to be done if Left button is selected in main window") (ShiftLeftButtonitem
s NIL doc "Items to be done if Left button is selected in main window with SHIFT k
ey down.") (MiddleButtonItems NIL doc "Items to be done if Middle button is select
ed in main window") (ShiftMiddleButtonItems NIL doc "Items to be done if Middle bu
tton is selected in main window with SHIFT key down.") (RightButtonItems ((Close (
Close (Close Destroy))) Snap Paint Clear Bury Repaint (Hardcopy (Hardcopy (Hardcop
yToFill HardcopyToPrinter))) Move Shape Shrink) doc "Items to be done if Right but
ton is selected"))
2/68+
```

A function uses the function **GetSourceIVs** to access a list of local instance variable descriptors.

```
2/39+ (GetSourceIVs ($ State))
((Description "A component of the United States" doc (* IV added by STEVE)) (S
tateCapital NIL doc (* IV added by STEVE)) (PartOf NIL doc (* IV added by STEV
E)) (Population NIL doc (* IV added by STEVE)) (DateOfPopulation NIL doc (* IV
added by STEVE)) (Cities NIL doc (* IV added by STEVE)) (Counties NIL doc (*
IV added by STEVE)))
```

### 2.8.1 Object Functions

Several functions take the name of an object and return its handle. These are described in Table 2-2. NLambda functions do not evaluate their arguments. If no object by name exists, \$ and \$! returns NIL.

**Table 2-2. LOOPS Object Functions**

Function	Type	Usage
\$	NLambda	Returns the handle of the object given its name.
\$!	Lambda	Returns a handle after evaluating the argument as the name to yield an object.
\$C	NLambda	Returns the class record.

For example, to retrieve the class record for 'Country', use:

## Medley LOOPS: The Basic System

```
2/42+ ($ State)  
#,$C State)
```

```
2/43+ (! State)  
#,$C State)  
2/44+ ($C State)  
#,$C State)
```

As an example, A NewClass does not exist, so \$! Attempts to evaluate it and gets an error:

```
2/78+ (! ANewClass)  
ANewClass is an unbound variable.
```

If no object exists by then name, \$C will attempt to create the class using the name.

## Chapter Three

### Class Messages and Functions

LOOPS includes a variety of functions for manipulating objects in the LOOPS environment. In LOOPS, as noted, there are three types of objects:

- *Instances*, which represent entities in the domain and are described by a template – a class;
- *Classes*, which define a set of instances by specifying a data structure and operators on the data structure; and
- *Metaclasses*, which specify a set of like classes.

An *inheritance hierarchy* of classes specifies a set of classes representing a sequence of refinements from an initial class to a class whose members are instances. Each class can have a list of one or more superclasses from which it inherits instance variables, class variables, and methods.

#### 3.1 Sending a Message to an Object

In Chapter Two, sending the **New** message to **Class** to create a new class was demonstrated. Sending the message **New** to a class to create an instance of a class was also demonstrated. These were examples of the more general capability of sending a message to an object to cause it to perform some action.

The general form of sending a message is:

## Medley LOOPS: The Basic System

(**←** <object> <message> <arg1> ... <argN>)

(SEND <object> <message> <arg1> ... <argN>)

where: <object> is a LOOPS object

<message> is a selector for a message handler embedded in the object (or one of its superclasses)

<arg1> is the first argument for the <message>

<argN> is the nth argument to the <message>.

The symbol **←** is used operator to send a message to a LOOPS object. Alternately, SEND is a macro which is expanded to the function to send the message to a LOOPS <object>.

Typically, the selector usually has the same name as the method which will handle the message. For example, we can create a new instance as follows:

```
(SEND ($ Person) New 'Stephen)
```

which creates a new instance whose name is 'Stephen'.

Note: Your keyboard may not be able to generate a **←**, so it is advised to use the SEND form. In test files, it will be easier to read.

Note: Send is defined in LOOPSMETHODS.



## 3.2 Checking Objectivity

To manipulate a LOOPS object, it is necessary to have a handle for that object. We can determine if an object is a LOOPS object using the function `Object?`, whose format is:

Function:     `Object?`  
Arguments:    An object.  
Return:       T, if a LOOPS object; NIL, otherwise.

```
2/21+ (Object? Arthur)
T
2/22+ ^
```

## 3.3 Class Operations

This section will describe the basic class and instance operations provided by LOOPS. Subsequent sections will look at advanced functions and methods.

### *3.3.1 Creating a New Class*

We create a new class by sending the message **New** to the metaclass `Class`. The format is:

Message:       New  
#Arguments:    <className> is the new class name.  
                  <superClassList> is a list of superclasses  
                  of the class.  
Return:        A handle for the new class.

### 3.3.1.1 Using DEFINECLASS

Let us define a Person with some attributes:

```
(*: "Define a generic person as a template")
(SETQ Person (DefineClass 'Person NIL ($ Class)))

(* ; "add instance variables to class Person")

(* ; "Family Relationships")
(PutCIVHere Person 'Father NIL 'doc)
(* ; "father of the person")
(PutCIVHere Person 'Mother NIL 'doc)
(* ; "mother of the person")
(PutCIVHere Person 'Sisters NIL 'doc)
(* ; "sister(s) of the person")
(PutCIVHere Person 'Brothers NIL 'doc)
(* ; "brother(s) of the person")
(PutCIVHere Person 'Spouses NIL 'doc)
(* ; "Spouses of Person")

(* ; "Attributes of a Person")
(* ; "gender of the person - male or female")
(PutCIVHere Person 'Gender NIL 'doc)

(* ; "The birthdate and deathdate of the person as a list")
(PutCIVHere Person 'Birthdate NIL 'doc)
```

## Medley LOOPS: The Basic System

```
(PutCIVHere Person 'Deathdate NIL 'doc)
```

```
(* ; "Country of Residence")
```

```
(PutCIVHere Person 'CitizenOf NIL 'doc)
```

```
(* ; "To check, describe the person")
```

```
(PP Person)
```

We can check the definition of Person using the PP function:

```
(RPAQQ Person #,($ Person))
CLASSES definition for Person:

(DEFCLASSES Person)
(DEFCLASS Person
  (MetaClass Class Edited:  **COMMENT** )
  (Supers Object)
  (InstanceVariables (Father #,NotSetValue doc NIL)
    (Mother #,NotSetValue doc NIL)
    (Sisters #,NotSetValue doc NIL)
    (Brothers #,NotSetValue doc NIL)
    (Spouses #,NotSetValue doc NIL)
    (Gender #,NotSetValue doc NIL)
    (Birthdate #,NotSetValue doc NIL)
    (Deathdate #,NotSetValue doc NIL)
    (CitizenOf #,NotSetValue doc NIL)))
```

### 3.3.1.2 Using SEND

You can use SEND to also create a new class as shown in the following example:

### 3.3.1.3 The DC Function

An alternate form is the function DC to define a class, whose format is:

## Medley LOOPS: The Basic System

Function: DC  
Arguments: <className>, the name of the new class.  
<superClassList> is a list of the superclasses  
of the class.  
Return: A handle for the new class.

Note: DC seems to operate differently from the specification in the LRM. Here is an example:

```
.....  
2/74+ (DC Town (LIST State County))  
Town has no FILES definition.  
NIL
```

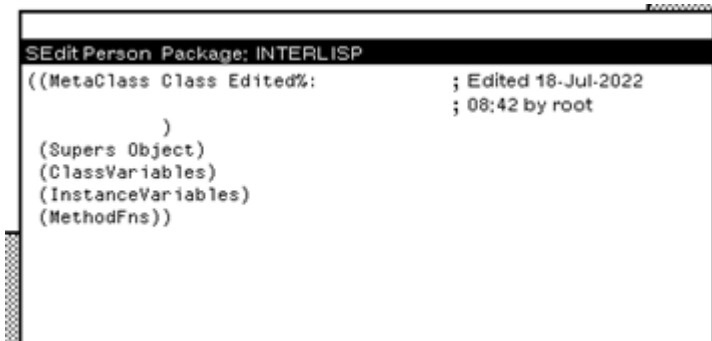
Apparently, DC is only usable with the SEDIT structure editor.

### 3.3.2 Editing a Class

Once a new class has been defined, its structure can be defined by editing the class by sending the message **Edit** to the new class. The format for the message sent to the new class is:

Message: Edit  
Arguments: None  
Return: An edited object.

Interlisp opens a new window using SEdit as shown in Figure 3-1.



```
SEdit Person Package: INTERLISP
((MetaClass Class Edited%:           ; Edited 18-Jul-2022
                                     ; 08:42 by root
   )
 (Supers Object)
 (ClassVariables)
 (InstanceVariables)
 (MethodFns))
```

Figure 3-1. Editing a Class Definition

### 3.3.3 Editing a Method

A method may be edited using the function **EM**, which invokes the Interlisp Editor to edit the method:

Function:       EM  
Arguments:      <className>, the name of the new class.  
                  <methodName>, the name of the method.  
Result:         ??

```
2/79* (EM County)
EM is an undefined function.
```

**Note: This function seems to be undefined in Medley. It is being investigated.**

You may also use the LOOPS browser to edit a method. This will be described in *Medley Loops: Tools and Utilities*.

## Medley LOOPS: The Basic System

### 3.3.4 Naming an Object

We can give an object a LOOPS name by sending it the message **SetName** as follows:

```
2/19+ (← Arthur SetName 'Arthur')
#,$(& Person (UL%↑VSGiKT1.0.0.c+; . 2))
2/20+ ($ Arthur)
#,$(& Person (UL%↑VSGiKT1.0.0.c+; . 2))
2/21+
```

We can inspect the definition of Arthur using **PP**:

```
2/29+ (← Arthur SetName 'Arthur')
#,$(& Person (UL%↑VSGiKT1.0.0.c+; . 2))
2/30+ (PP Arthur)
VARS definition for Arthur:
(RPAQQ Arthur #,$( Arthur))
INSTANCES definition for Arthur:
(DEFINST Person (Arthur (UL%↑VSGiKT1.0.0.c+; . 2))
)
```

An instance variable, *name*, receives the <name> specified for the message **SetName**. *name* is an instance variable of the class **NamedObject**, which would be specified as a superclass of the class.

We also create another person named **Bertram**.

```
2/17+ (SETQ Bertram (← Person New))
#,$(& Person (UL%↑VSGiKT1.0.0.c+; . 3))
2/18+ A
```

We can test if **Bertram** is an object in our environment via the **Object?** function.

```
2/18+ (Object? Bertram)
T
```

And, we can set Bertram's name just as we did with Arthur.

### 3.4 Accessing Supers

The *superclasses* of a class can be obtained using the function **Supers**, whose format is:

Function:	Supers
Arguments:	<class record>, the handle for the class.
Return:	A list of the superclasses.

For example, we defined Person using DefineClass. To find Person's supers, we can use:

```
2/8+ (Supers ($ Person))
| (#, ($C Class) #, ($C Object) #, ($C Tofu))
```

```
2/116+ (Supers NorthAmerica)
| (#, ($C GeoRegion) #, ($C Countries&Regions) #, ($C Object) #, ($C Tofu))
2/117+ *
```

---

which shows that Class and Object are its superclasses. *Tofu* is an internal name, supposedly meaning “Top of the Universe”, that was used by the PARC staff. It is embedded in the LOOPS source code.

### 3.5 Accessing Variables

There are different types of variables and properties that are associated with a class. Many of the functions referring to these variables and properties are associated with getting or putting their

## Medley LOOPS: The Basic System

values. LOOPS provides many general functions for performing these operations. LOOPS also introduced a compact programming notation for accessing variables and properties. The general functions are discussed in the section and the compact programming notation in a following section. Figure 3-2 depicts where variables are stored.

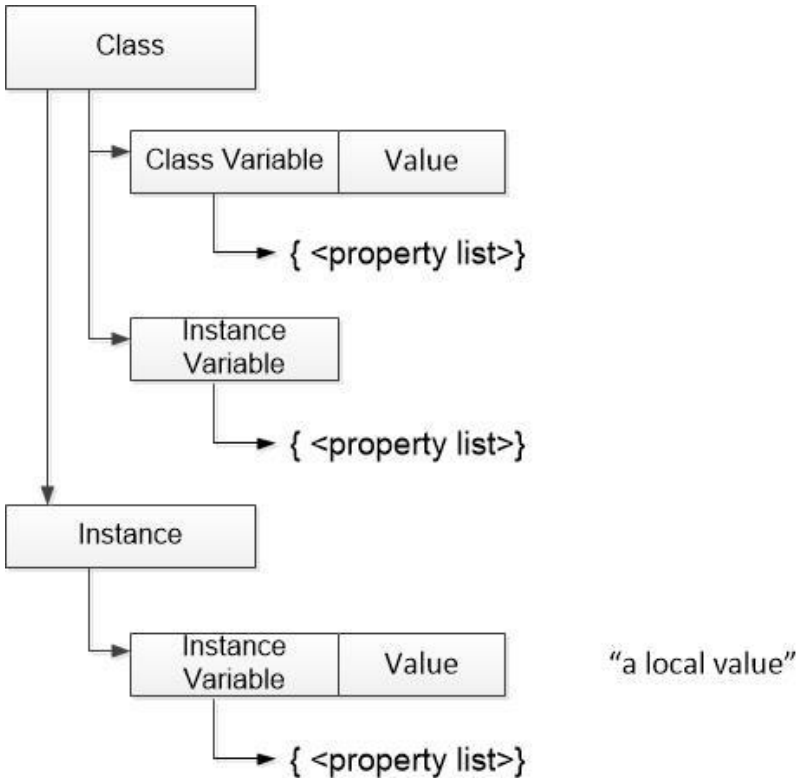


Figure 3-2. Variable Locations



## Medley LOOPS: The Basic System

Figure 3-2 depicts the relationship of variable storage to LOOPS objects.

A class may have both *class variables* and *instance variables*. A class variable may have a value or NIL. If it has a value, that value is passed down to each instance as an initial value. If no local variable for a class variable is specified in an instance, then the value of the class variable is fetched from the class, unless the program has specified a local value for the class variable. Each class variable may also have a <property list>.

An instance may have instance variables each of which may have a local value set by the program. Although instances have handles which distinguish them one from another, from the program perspective, instances are distinguished by the different values of their instance variables. Each instance variable may also have a property list.

A class may inherit class variables from its <superClassList>. If the class does not set a local value for an inherited class variable, the value of the most immediate predecessor in the <superClassList> that has set a value for the class variable is cached in the class.

To demonstrate some of these functions, a class from TRUCKIN from the directory <home>steve>LOOPS-MAIN>TRUCKIN-src was loaded. Prettyprinting it yields:

## Medley LOOPS: The Basic System

```
2/20+ (PP 'TruckinParameters)
CLASSES definition for TruckinParameters:

(DEFCLASSES TruckinParameters)
(DEFCLASS TruckinParameters
  (MetaClass GameClass Edited%: **COMMENT** doc "Used for
    Setting/resetting Truckin parameters")
  (Supers GameParameters)
  (ClassVariables (CopyCV NIL))
  (InstanceVariables (banditCount 2 goodVal NUMBERP exp banditCount
    doc "Number of Bandits in game")
    (timeTrace NIL goodVal (T NIL)
      exp timeTrace doc "If T then prints time taken
        by each player after each request")
    (debugMode T goodVal (T NIL)
      exp debugMode doc "If T then rule violations
        bring up RuleExec")
    (gameDebugFlg NIL goodVal (T NIL)
      exp gameDebugFlg doc "If T then prints some
        extra diagnostic messages")
    (truckinLogFlg NIL goodVal (T NIL)
      exp truckinLogFlg doc "If T then keeps a log of all
        Game Printout in Status window")
    (truckDelay 0 goodVal NUMBERP exp truckDelay doc "Controls
      speed at which trucks move.
        Higher delay means slower motion"))))
NIL
```

Since TruckinParameters is a class, an instance was created called My Parameters via:

```
2/25+ (SETQ MyParameters (← ($ TruckinParameters) New))
#,($& TruckinParameters (YW%+VDh;ET1.0.0.T84 . 531))
```

### 3.5.1 Getting Variable and Property Values

Two functions are used to get the value of a variable or a property. If the value of the variable or property is an active value, then the associated getFn is invoked on the value.

#### 3.5.1.1 Getting a Variable Value

**GetValue** returns the value of a variable or a property of an sometimes, IV from an instance of a class. The format is:

## Medley LOOPS: The Basic System

Function:            GetValue  
Arguments:         <object>, the handle of a LOOPS instance  
                      object.  
                      <varName>, the name of a variable in the  
                      instance.  
                      <propName>, the name of a property  
                      associated with <varName>.  
Return:             A value or NIL.

If <propName> is NIL, then GetValue returns the value of <varName>. <varName> may be an instance variable.

If <varName> is an IV, then the value is a local value resident in the instance object.

If <varName> is a CV and no local value was set, then the value returned is the default value of the variable resident in the class.

If there is no IV or CV by that name, then a break occurs.

So, we can fetch the value of CopyCV from MyParameters:

```
1111+
2/32+ (GetValue MyParameters 'CopyCV NIL)
NIL
```

which is NIL, because its value is not defined in MyParameters and its value is NIL in TruckinParameters. Note that MyParameters is a handle of the object.

We can try to fetch the value of 'banditcount' from MyParameters:

```
2/35+ (GetValue MyParameters 'banditCount NIL)
2
```

## Medley LOOPS: The Basic System

Finally, let us fetch the value of the doc property of banditCount:

```
2/36+ (GetValue MyParameters 'banditCount 'doc)
"Number of Bandits in game"
```

If <propName> is not NIL, GetValue returns the value of the <propName> from the property list of the IV.

If no value is found in the local property list, **GetValue** returns the default value for the property for the IV found in the class or one of its superclasses.

If no property value was found in any of the superclasses, the value returned is that of the global variable **NotSetValue**, which is initially set to ‘?’.

If there is no property by the name <propName>, GetValue returns the value of the variable **NoValueFound**.

*Note:* Returning the value of **NotSetValue** by LOOPS is different from Interlisp, which returns NIL.

*Note:* It is an error to try to use GetValue to fetch the property of an instance variable in a class.

### 3.5.1.2 Getting a Class Variable Value

**GetClassValue** returns the value of a class variable or property of a CV for a class object. The format is:

Function:	GetClassValue
Arguments:	<object>, the handle of a LOOPS class object. <varName>, the name of a class variable in the class.

## Medley LOOPS: The Basic System

<propName>, the name of a property associated with <varName>.

Return: A value or NIL.

Class variables may be inherited from superclasses or defined within a class definition. They are shared by all instance of a class. All instances of a class could see the value of a CV.

If <object> is an instance of a class, then:

- If the class of the instance has a variable <varName> and it has a value, then the value is returned.
- If the <varName> is not found in the class, LOOPS searches for the class upward through the class hierarchy until it finds <varName> in a superclass and returns the value associated with <varName> from that superclass.
- If <varName> is not found in any superclass, NIL is returned.

Search was not thought to be an expensive operation since a class hierarchy was not expected to be very deep.

We can fetch the value of the CV CopyCV from MyParameters”

```
| 2/39+ (GetClassValue MyParameters 'CopyCV)  
| NTI
```

Now, let us try fetching the value of banditCount, which we will pretend we do not know is not a CV:

## Medley LOOPS: The Basic System

```
2/40+ (GetClassValue MyParameters 'banditCount)
Help! LoopsHelp: banditCount not a CV of #,($C TruckInParamet
Help! LoopsHelp: banditCount not a CV of #,($C TruckInParamet
Parameters)
2/41+:
```

We see that Interlisp opens an error window and informs us that `banditCount` is not a CV of `MyParameters` and displays a prompt to possibly take remedial action. One such action might be to create an instance of bandit count in the instance of the class.

### *3.5.2 Putting Variable and Property Values*

Two functions are used to put (set) the value of a variable or a property. If the value of the variable or property was an active value, then the associated `<putFn>` is invoked on the value.

### 3.5.2.1 Putting a Variable Value

**PutValue** sets the value of a variable or property of an instance variable for an instance of a class. The format is:

Function:           PutValue  
Arguments:         <object>, the handle of a LOOPS object  
                    <varName>, the name of a variable in the  
                    instance.  
                    <newValue>, the value for the variable or  
                    property.  
                    <propName>, the name of a property associated  
                    with <varName>.  
Return:             <newValue> or NIL.

If <propName> is NIL, then PutValue stores <newValue> as the value of <varName>. <varName> may be an IV or a CV. The value returned depends on whether <varName> is an IV or a CV:

- If <varName> is an IV, then the value is a local value resident in the instance object.
- If <propName> is not NIL, PutValue sets the value of the <propName> in the property list of the IV.

Let us change the value of gameDebugFlg from NIL to T:

```
2/46+ (PutValue MyParameters 'gameDebugFlg T NIL)  
T  
2/47+ (GetValue MyParameters 'gameDebugFlg NIL)  
T
```

*Note:* It is an error to try to use PutValue to fetch the property of an IV in a class.

## Medley LOOPS: The Basic System

```
2/51* (PutValue ($ TruckinParameters) 'gamDebugFlg NIL 'doc)
gamDebugFlg
is not a local IV, so cannot be changed in #,($C TruckinParameters)
```

### Extended Example:

This example puts a value to an IV of EdwardIII where the IV is specified in the Person class. Since EdwardIII is an instance of Person, LOOPS cannot find it in EdwardIII as originally created since 'Son' was added to Person after EdwardIII was created. So, it looks in the Supers of EdwardIII, which is Person, and finds 'Son' as a CIV. It creates the IV locally in EdwardIII.

Here is the source code for the test:

```
(* ; "Set Son as an IV of Person")
(PutCIVHere ($ Person) 'Son)

(* ; "Set an IV in EdwardIII")

(SETQ result (PutValue ($ EdwardIII) 'Son 'Edward))
(PRIN1 "Son of ")
(PRIN1 ($ EdwardIII))
(PRIN1 " is ")
(PRINT result)

(SETQ result (PutValue ($ EdwardIII) 'Son 'Edward))
```



## Medley LOOPS: The Basic System

```
(PRIN1 "Checking Son of ")
(PRIN1 ($ EdwardIII))
(PRIN1 " is ")
(PRINT (GetValue ($! 'EdwardIII) 'Son))

(* ; "Set the value of the doc property for Son.")
(PutValue ($ EdwardIII) 'Son 'doc "The Black Prince")
(PRIN1 "EdwardIII:Son doc is ")
(SETQ doc1 (GetValue ($ 'EdwardIII) 'Son 'doc))
(PRINT doc1)
STOP
```

and, here is the results from executing the tests:

```
{DSK}<home>Steve>loops-tests>plantagenets>TestEdwardIII.;1
Son of #,($ EdwardIII) is Edward
Checking Son of #,($ EdwardIII) is Edward
EdwardIII:Son doc is (* IV added by STEVE)
VARS definition for EdwardIII:

(RPAQQ EdwardIII #,($ EdwardIII))
INSTANCES definition for EdwardIII:
(DEFINST Person (EdwardIII (FA%0.UG1.0d5.U85 . 13))
  (Father EdwardII)
  (Mother IsabellaCapet)
  (Spouses PhilippadAvesnes)
  (Gender Male)
  (Birthdate (11 13 1312))
  (Deathdate (6 21 1377))
  (Son Edward "The Black Prince" doc "The Black Prince" doc
    "The Black Prince" doc "The Black Prince" doc))
```

### An Error Example

## Medley LOOPS: The Basic System

If we try to put the value of a class IV in an instance of a class, when the CIV has not been declared in the class, we get an error as depicted in the following:

```

#,$ Ireland)
#,$ Ireland)
"An island country of Europe"
Europe
70273
HELP! LoopsHelp: Population not an IV of #,$& Country (L+K9@
Help! LoopsHelp: Population not an IV of #,$& Country
(L+K9@TV1.0.0.[15 , 15))
HELP
LoopsHelp
Object_IVMissing
Put_IVProp
PutValue
SIMT+UNWIND-PROTECT 2/20+;A
LOAD
LOAD
FAULTEVAL
EVAL
EVAL-INPUT
-- --
```

Looking back at the definition of Ireland's Super class, we see that Country does not have 'Population' declared as a CIV:

```
(* ; "Country - a Geographic Area")
(SETQ Country
  (DefineClass 'Country
    '(GeographicArea)
    ($ Class)
  )
(SEND ($ Country) SetName 'Country)
(PutCIVHere ($ Country)
  'Description
  "A component of a Geographic Area"
)
```

```
(PutCIVHere ($ Country) 'Part NIL)
(PutCIVHere ($ Country) 'Provinces NIL)
(PutCIVHere ($ Country) 'States NIL)
(PP Country)
```

which causes the error. So, we must declare ‘Population’ in Country for the put to succeed. We must do the same for CapitalCity.

### 3.5.2.2 Putting a Class Variable Value

**PutClassValue** sets the value of a class variable or property of a class variable for a class. The format is:

Function:	PutClassValue
Arguments:	<object>, the handle of a LOOPS object. <varName>, the name of a class variable in the class. <newValue>, the newValue to store. <propName>, the name of a property associated with <varName>.
Return:	<newValue>.

Class variables may be inherited from superclasses or defined within a class definition. They are shared by all instance of a class. All instances of a class could see the value of a CV.

If <object> is an instance of a class, then:

- If the class of the instance has <varName> and it has a value, then it sets the value of the variable.

## Medley LOOPS: The Basic System

- If the <varName> is not found in the class, LOOPS sets the value of the variable in the first class in the class hierarchy for which <varName> occurs.
- If <varName> is not found in any superclass, NIL is returned.

Let us change the value of CopyCV in TruckinParameters from NIL to T.

```
2/54* (PutClassValue ($ TruckinParameters) 'CopyCV T NIL)
T
2/55* (GetClassValue ($ TruckinParameters) 'CopyCV NIL)
T
```

According to the LRM, search was not thought to be an expensive operation since a class hierarchy was not expected to be very deep.

Another example. Set CitizenOf as a CV for Person, then attempt to put a value for that CV. We defined the CV via:

```
(PutCVHere Person 'CitizenOf NIL 'doc)
```

and here we see it when we prettyprint Person:

```
(DEFCLASSES Person)
(DEFCLASS Person
  (MetaClass Class Edited:  **COMMENT** )
  (Supers Object)
  (ClassVariables (CitizenOf NIL doc  **COMMENT** ))
  (InstanceVariables (Father #,NotSetValue doc NIL)
    (Mother #,NotSetValue doc NIL)
    (Sisters #,NotSetValue doc NIL)
    (Brothers #,NotSetValue doc NIL)
    (Spouses #,NotSetValue doc NIL)
    (Gender #,NotSetValue doc NIL)
    (Birthdate #,NotSetValue doc NIL)
    (Deathdate #,NotSetValue doc NIL)
    (Title #,NotSetValue doc NIL)))
```

Then, we put value of the CV via:

```
2/7* (PutClassValue Person 'CitizenOf 'England)  
England
```

and, we try to retrieve it via:

```
(PutClassValue Person 'CitizenOf 'England)
```

```
(PROG NIL
```

```
  (PRIN1 "CitizenOf ")
```

```
  (PRIN1 ($ Person)
```

```
  (PRIN1 " ")
```

```
  (PRINT (GetClassValue Person 'CitizenOf))
```

```
)
```

```
(PRINT " ")
```

```
(* ; "Get the same class value from EdwardIII")
```

```
(PROG NIL
```

```
  (PRIN1 "CitizenOf ")
```

```
  (PRIN1 ($ EdwardIII))
```

```
  (PRIN1 " ")
```

```
  (PRINT (GetClassValue Person 'CitizenOf))
```

```
)
```

```
-- --
2/25+ (LOAD 'TestGetPut.txt)

{DSK}<home>steve>LOOPS-MAIN>TestGetPut.txt;1
CitizenOf #,($ Person)  England
NIL
CitizenOf #,($ EdwardIII)  England
{DSK}<home>steve>LOOPS-MAIN>TestGetPut.txt;1
```

So, we see that we can get the class value from Person, and when we reference EdwardIII, the CV is retrieved from its parent class.

### 3.5.2.3 Pushing Values onto Variables

Two functions – **PushValue** and **PushClassValue** - push a new value onto the front of a list which is the value of an IV or a class variable. Their format is:

Function:	PushValue PushClassValue
Arguments:	<object>, the handle of a LOOPS object. <varName>, the name of a variable in the instance or class object. <newValue>, the newValue to store. <propName>, the name of a property associated with <varName>.
Return:	<newValue>.

The value of <varName> or <propName> of an IV must be a list.  
We can push a new value onto 'Son' of EdwardIII:

```
(* ; "Get the value of Son of EdwardIII")
(SETQ result1 (GetValue ($ EdwardIII) 'Son))
```

## Medley LOOPS: The Basic System

```
(PRIN1 "Son of ")
```

```
(PRIN1 ($ EdwardIII))
```

```
(PRIN1 " is ")
```

```
(PRINT result1)
```

```
(* ; "If value of Son is an atom, make it a list.")
```

```
(COND
```

```
  ((ATOM result1)
```

```
    (PutValue ($ EdwardIII) 'Son (LIST result1))
```

```
    (PP EdwardIII)
```

```
  )
```

```
)
```

```
(* ; "Try to Push a Value - Edmund onto Son.")
```

```
(SETQ result2 (PushValue ($ EdwardIII) 'Son 'Edmund))
```

```
(PRIN1 "Son of ")
```

```
(PRIN1 ($ EdwardIII))
```

```
(PRIN1 " is ")
```

```
(PRINT result2)
```

```
| 2/29←
```

```
| 2/29← (IL:PushValue ($ EdwardIII) 'Son 'Edmund)
```

```
| PushValue is an undefined function.
```

<<Note: Why is this undefined? Need to search source code.>>

## Medley LOOPS: The Basic System

If the value of either <varName> or <propName> is an active value, then when the list is fetched, its <getFn> is invoked. After the <newValue> has been stored on the list, the <putFn> will be triggered when the list is stored.

PushClassValue performs like PutClassValue for class variables.

### 3.5.2.4 Adding a Value to a Variable

A value can be added to the end of a variable list - either the value of the variable or to a property list using the function **AddValue**. It takes the format:

Function:           AddValue  
Arguments:        <object>, the handle of a LOOPS object.  
                  <varName>, the name of a class variable in the  
                  instance.  
                  <newValue>, the newValue to store.  
                  <propName>, the name of a property  
                  associated with <varName>.  
Return:            <newValue>.

Let us try to add a value to the end of Son of EdwardIII. If Son is not a list, LOOPS first makes it so.

```
(* ; "Push a new value onto an IV of an instance.")
```

```
(* ; "If the value is not a list, make it so")
```

```
(* ; "Get the value of Son of EdwardIII")
```

```
(SETQ result1 (GetValue ($ EdwardIII) 'Son))
```

```
(PRIN1 "Son of ")
```



## Medley LOOPS: The Basic System

```
(PRIN1 ($ EdwardIII))
```

```
(PRIN1 " is ")
```

```
(PRINT result1)
```

```
(* ; "If value of Son is an atom, make it a list.")
```

```
(COND
```

```
  ((ATOM result1)
```

```
    (PutValue ($ EdwardIII) 'Son (LIST result1))
```

```
    (PP EdwardIII)
```

```
  )
```

```
)
```

```
(* ; "Try to add a value to the end of Son.")
```

```
(SETQ result3 (AddValue ($ EdwardIII) 'Son 'Edmund))
```

```
(PRIN1 "Son of ")
```

```
(PRIN1 ($ EdwardIII))
```

```
(PRIN1 " is ")
```

```
(PRINT result3)
```

```
(* ; "Try to Push a Value = Edmund onto Son.")
```

```
(SETQ result2 (PushValue ($ EdwardIII) 'Son 'Edmund))
```

```
(PRIN1 "Son of ")
```

```
(PRIN1 ($ EdwardIII))
```

```
(PRIN1 " is ")
```

```
(PRINT result2)
```

## Medley LOOPS: The Basic System

STOP

```
2/38< (LOAD 'TestPushValue.txt)
{DSK}<home>steve>LOOPS-MAIN>TestPushValue.txt;1
Son of #,($ EdwardIII) is (Edward)
AddValue is an undefined function.
```

<<Note: Need to check source code why AddValue is not defined.>>

### 3.5.3 Non-triggering Get and Put

Although a value of a variable or a property list may have an active value associated with it, there are cases where one needs to access the value without triggering the active value. These functions take the format:

Function:	<b>GetValueOnly</b> <b>GetClassValueOnly</b>
Arguments:	<object>, the handle of a LOOPS object. <varName>, the name of an instance or class variable in instance <propName>, the name of a property
Result:	A value.

These functions access the specified value only without invoking the active value functions.

**GetValueOnly** accesses and returns the default value from a superclass if none exists for a class variable in the instance.

Similar functions for putting a value directly to a variable or a property have the format.

## Medley LOOPS: The Basic System

Function:       **PutValueOnly**  
                  **PutClassValueOnly**

Arguments:      <object>, the handle of a LOOPS object.  
                  <varName>, the name of an instance or class  
                  variable in instance.  
                  <newValue>, the new value for the variable or  
                  Property.  
                  <propName>, the name of a property

Result:         <newValue>.

GetClassValueOnly and PutClassValueOnly will only take class objects as arguments.

### *3.5.4 Local Get Functions*

You may need to determine if a value or property is set in a class or instance without inheriting any information or triggering active values. Two functions, **GetIVHere** and **GetCVHere**, allow you to do this. Their format is:

Function:       GetIVHere  
                  GetCVHere

Arguments:      <object>, the handle of a LOOPS object.  
                  <varName>, the name of an instance or class  
                  variable in instance.  
                  <propName>, the name of a property.

Result:         The <varName> or the <propName>, if it is  
                  non-NIL.

## Medley LOOPS: The Basic System

If the value of <varName> or <propName> was not yet stored in the <object>, the value of the variable **NotSetValue** is returned.

```
2/53← (SETQ T1 (← ($ Window) New 'w1))
#,( $& Window (|SY+VzAOnT1.0.0.0d:| . 2))
2/54← (GetIVHere T1 'left)
# ,NotSetValue
2/55← A
```

Now, if we PutValue of “left” on T1:

```
2/58← (PutValue T1 'left T NIL)
T
2/59← (GetIVHere T1 'left)
T
2/60←
```

We can get the value of the class CV Project for Person via:

```
2/63← (GetCVHere Person 'Project)
Kinship
2/64← A
```

---

There was no need (so far, according to the 1991 LRM) to have local put functions since all put functions were local to the class or instance. The necessary effect can be achieved by using PutValueOnly and PutClassValueOnly.

### *3.5.5 Accessing Class and Method Properties*

Several of the functions in the preceding sections only worked with instances of classes. Two functions, **GetClassIV** and

## Medley LOOPS: The Basic System

**PutClassIV**, access the default value or property value of an instance variable which is stored in the class.

Function:           GetClassIV  
Arguments:        <class>, the name of a class.  
                  <varName>, the name of a variable defined in  
                  the class.  
                  <propName>, the name of a property of the  
                  instance variable in the class.  
Return:            The default value.

Function:           PutClassIV  
Arguments:        <class>, the name of a class.  
                  <varName>, the name of a variable defined in  
                  the class.  
                  <newValue>, the new value to be stored.  
                  <propName>, the name of a property of the  
                  instance variable in the class.  
Return:            <newValue>.

PutClassIV stores <newValue> as the value of an instance variable or its property. The variable must be local to the class. For example:

```
(DefineClass 'FireEngine '(Class))  
(SEND ($ FireEngine) SetName 'FireEngine)
```

## Medley LOOPS: The Basic System

```
2/85+ (PP FireEngine)
CLASSES definition for FireEngine:

(DEFCLASSES FireEngine)
(DEFCLASS FireEngine
  (MetaClass Class Edited%:  **COMMENT** )
  (Supers Class))

2/88+ (← ($ FireEngine) AddIV 'Color 'red)
Color
2/89+ (pp FireEngine)
```

Sending the message AddIV to the class with the proper arguments allow us to add an IV.

```
2/90+ (PP FireEngine)
CLASSES definition for FireEngine:

(DEFCLASSES FireEngine)
(DEFCLASS FireEngine
  (MetaClass Class Edited%:  **COMMENT** )
  (Supers Class)
  (InstanceVariables (Color red doc  **COMMENT** )))

NIL
.....
```

### 3.5.6 Accessing Class Properties

LOOPS classes can have property lists for themselves and for methods of classes. One use of this feature is to document both the class and its methods. There are several methods for access these property lists. **GetClass**, **GetClassOnly**, and **GetClassHere** return a value of property on the property list of a class. Their format is:

## Medley LOOPS: The Basic System

Function:        GetClass  
                  GetClassHere  
                  GetClassOnly


Arguments:       <class>, the name of a class.  
                  <propName>, the name of a property of the  
                  class.

Return:          The value of <propName> of the class.



If <propName> was NIL, GetClass returned the metaclass of a class.

Class properties are inherited like class variables,. If <propName> was not in <class>, LOOPS searched the superclasses of <class> for <propName>, If it was not found, NIL was returned.

A class property could be an active value, in which case its getFn was triggered by GetClass. GetClassOnly did not trigger an active value. An example of GetClass returning the doc property of its superclass::

```
2/56+ (GetClass Person 'doc)
(* * This is the default metaClass for all classes)
2/57+ 
```

However, if we stored a value for doc in the class itself, we would see:

```
 2/66+ (GetClass Person 'doc)
"the template for Person"
2/67+ 
```

## Medley LOOPS: The Basic System

GetClassHere returned the local value of <propName> in <class>, If <propName> was not found in <class>, it returned the value of the global variable **NotSetValue**.

**PutClass** and **PutClassOnly** are used to store a new value into a class property. Their format is:

Function:	PutClass PutClassOnly
Arguments:	<class>, the name of a class. <newValue>, the value to be stored. <propName>, the name of a property of the class.
Return:	<newValue>.

PutClass sets the value of <propName> in <class> to <newValue> If <propName> was NIL, then it set the metaclass of <class> to <newValue>. PutClassOnly did not trigger the putFn of <propName> if it was an active value.

```
2/65+ (PutClass Person "the template for Person" 'doc)
      "the template for Person"
2/66+ A
```



## Medley LOOPS: The Basic System

Here is an example using `PutClassOnly`:

```
2/196+ (GetValueOnly UnitedStates 'States)
NIL
2/197+ (PutValueOnly UnitedStates 'States (LIST 'NewYork))
(NewYork)
2/198+ (PP UnitedStates)
VARS definition for UnitedStates:

(RPAQQ UnitedStates #,($ UnitedStates))
CLASSES definition for UnitedStates:

(DEFCLASSES UnitedStates)
(DEFCLASS UnitedStates
  (MetaClass Class Edited%:  **COMMENT** )
  (Supers Country)
  (InstanceVariables (Description "A set of 48 states, Alaska, and
                        Hawaii")
                     (Type Country doc  **COMMENT** )
                     (States (NewYork))))
NIL
```

### *3.5.7 Adding Variables to a Class*

LOOPS provides two functions for adding variables to a class: **PutCVHere** and **PutCIVHere**. These methods add the variable of the given type locally to the class and record it in the class record.

#### **3.5.7.1 Putting a Class Variable Locally**

`PutCVHere` adds a CV locally to a class, whether or not it is defined in a superclass. Its format is:

## Medley LOOPS: The Basic System

Function: PutCVHere  
Arguments: <class>, the name of the class.  
<varName>, the name of the variable.  
<value>, the initial value of the variable.  
Return: The value of the variable.

PutCVHere uses AddCV to add the variable to the class locally, if it is not already defined locally, and check it using GetValue.

```
2/81+ (PutCVHere Person 'Project 'Kinship)
Kinship
2/82+
```

After adding some CVs to Person, we can use PP to print the class description:

```
(DEFCLASSES Person)
(DEFCLASS Person
  (MetaClass Class Edited%:  **COMMENT** )
  (Supers Object)
  (InstanceVariables (Father #,NotSetValue doc NIL)
    (Mother #,NotSetValue doc NIL)
    (Sisters #,NotSetValue doc NIL)
    (Brothers #,NotSetValue doc NIL)
    (Gender #,NotSetValue doc NIL)))
NTI
```

### 3.5.7.2 Putting an Instance Variable Locally

PutCIVHere locally adds an IV, whether it is defined in a superclass or not, to a class. Its format is:

## Medley LOOPS: The Basic System

Function: PutCIVHere  
Arguments: <class>, the name of the class.  
<varName>, the name of the variable.  
<value>, the initial value of the variable.  
<prop>, the name of a property.  
Return: The value of the variable.

PutCIVHere uses AddCIV to add the variable to the class locally, if it is not already defined locally. If <prop> is NIL, then AddCIV adds the variable with <value> to the local IVs. If <prop> is non-NIL, then it used GetClassIV to retrieve the value of the IV from the class or its superclasses and add the that value to the property.

```
2/50+ (PutCIVHere Person 'Father NIL NIL)
NIL
```

```
2/52+ (GetValue Person 'Father)
NIL
2/53+
```

We can create the IV population in Maryland and initialize it to NIL:

```
2/201+ (PutCIVHere State 'Population NIL)
```

We can assign the value of the population of Maryland to the IV Population (circa 2022):

## Medley LOOPS: The Basic System

```
2/200* (PutValue Maryland 'Population 6165000)
6165000
2/200* (PP Maryland)
VARS definition for Maryland:
(RPAQQ Maryland #,($ Maryland))
INSTANCES definition for Maryland:
(DEFINST State (Maryland (SL%GJKbV1.0.0.S15 . 53))
  (Description "An Eastern State of the United States")
  (Name Maryland)
  (Population 6165000)
  (CapitalCity Annapolis))
```

If you use DefineClass to create the new class, then you need to use SetName to attach a name before you can use the name in these functions.

### 3.5.7.3 AddCIV

The AddCIV function adds a new class IV to a class if it does not exist locally within the class. Its format is:

Function:	AddCIV
Arguments:	<class>, the name of the class. <varName>, the name of the variable. <value>, the initial value of the variable. <prop>, the name of a property.
Return:	The value of the variable.

Let us add “Dinosaur” as an IV of a State because some states do indeed have state dinosaurs:

```
2/212* (AddCIV State 'Dinosaur NIL)
Dinosaur
```

## Medley LOOPS: The Basic System

Maryland is one of the states that has a state dinosaur, so we can specify that fact as:

```
2/214+ (PutValue Maryland 'Dinosaur 'AstrodonJohnstoni)
AstrodonJohnstoni
2/215+ (PP Maryland)
VARS definition for Maryland:
(RPAQQ Maryland #,($ Maryland))
INSTANCES definition for Maryland:
(DEFINST State (Maryland (SL%GJKbv1.0.0.SI5 . 53))
  (Description "An Eastern State of the United States")
  (Name Maryland)
  (Population 6165000)
  (CapitalCity Annapolis)
  (Dinosaur AstrodonJohnstoni))
```

### *3.5.8 Generalized Get and Put Functions*

Generalized Get and Put functions accept a type argument, which is used to select a more specialized function – such as the ones described in the previous sections – to perform a get or put operation.

#### **3.5.8.1 Generalized Get Functions**

Three generalized Get functions were provided: **Get It**, **GetItOnly**, and **GetItHere**. The <type> argument could have one of the values IV, CV, CLASS, or METHOD.

## Medley LOOPS: The Basic System

Function:           Get It  
                      GetItOnly  
                      GetItHere

Arguments:         <object>, the handle of a LOOPS object.  
                      <varOrMethod>, the name of the variable or  
                      method to be invoked.  
                      <propName>, the name of a property of the  
                      method.  
                      <type>, the type of function to be perform.

Return:             The value of <propName>, if non-NIL.

If <type> is NIL, IV is assumed. <varOrMethod> was interpreted as a variable.

If IV or CV was specified by <type>, <varOrMethod> was an IV or CV name.

If <type> was METHOD, <varOrMethod> was a message name.

If <type> was CLASS, <varOrMethod> was ignored.

The functions invoked the functions previously described:

(GetIt ... 'IV) → (GetValue ...)

(GetIt ... 'CV) → (GetClassValue ...)

(GetIt ... 'CLASS) → (GetClass ...)

(GetIt ... 'METHOD) → (GetMethod ...)

## Medley LOOPS: The Basic System

This approach allows the user to parameterize the ‘get functions based on the application and the task to be performed. As an example:

```
2/223+ (GetItOnly Maryland 'Dinosaur)
AstrodonJohnstoni
2/224+ (GetItHere Maryland 'Dinosaur)
AstrodonJohnstoni
2/225+ (GetIt Maryland 'Dinosaur)
AstrodonJohnstoni
```

Another Example:

```
(DEFCLASSES France)
(DEFCLASS France
  (MetaClass Class Edited%: **COMMENT** )
  (Supers Country Europe)
  (InstanceVariables (Description "A country of Europe")
    (PrimaryLanguage French)
    (CapitalCity Paris)))
NIL
```

```
2/58+ (GetItHere France 'CapitalCity)
Paris
```

```
2/62+ (GetIt France 'CapitalCity NIL 'IV)
Paris
```

Note: If not fetching the value of a property, that argument must be set to NIL.

### 3.5.8.2 Generalized Put Functions

Two generalized Put functions were provided: **PutIt** and **PutItOnly**. The <type> argument could have one of the values IV, CV, CLASS, or METHOD.

Function:	PutIt PutItOnly
Arguments:	<object>, the handle of a LOOPS object. <varOrMethod>, the name of the variable or method to be invoked. <newValue>, the new value to be stored. <propName>, the name of a property of the method. <type>, the type of function to be perform.
Return:	The value of <propName>, if non-NIL.

If <type> is NIL, IV is assumed. <varOrMethod> was interpreted as a variable.

If IV or CV was specified by <type>, then <varOrMethod> was an IV or CV name.

If <type> was METHOD, then <varOrMethod> was interpreted as a message name.

If <type> was CLASS, then the function was ignored.

These functions act in a manner like the GetIt functions except they store the <newValue> as described in previous sections.



## Medley LOOPS: The Basic System

### 3.5.9 Putting IV Value and Property

LOOPS provides functions for setting the value of an IV or the property of an IV: **PutIVValue** and **PutIVProp**.

Function: PutIVValue  
Arguments: <class>, the name of class having the IV.  
<varname>, the name of the instance variable.  
<newvalue>, the new value to be assigned to the IV.  
Return: <newvalue>

PutIVValue uses PutIt to assign the new value to the IV in the IVDescr. Here is an example:

```
2/230+ (PutCIVHere State 'Flower NIL)
NIL
2/231+ (PutIVValue Maryland 'Flower 'BlackeyedSusan)
BlackeyedSusan
```

```
-----
2/232+ (PP Maryland)
VARS definition for Maryland:
(RPAQQ Maryland #,($ Maryland))
INSTANCES definition for Maryland:
(DEFINST State (Maryland (SL%GJKbV1.0.0.SI5 . 53))
 (Description "An Eastern State of the United States")
 (Name Maryland)
 (Population 6165000)
 (CapitalCity Annapolis)
 (Dinosaur AstrodonJohnstoni)
 (Flower BlackeyedSusan))
```

If the IV is missing, e.g., not found in the list of IVDescrs, it uses PutValue to create the IV in the class and assign the value to it.

## Medley LOOPS: The Basic System

Function: PutIVProp  
Arguments: <class>, the name of class having the IV.  
<varname>, the name of the instance variable.  
<newvalue>, the new value to be assigned to the  
IV's property.  
<propname>, a property of the IV.  
Return: <newvalue>

PutIVProp uses PutIt to assign the new value to the property of the IV in the IVDescr.

If the property is missing on the IVs list of propDescrs, then it uses PutValue to create the property in the propDescrs of the IV and assigns newvalue to its property.

### *3.5.10 Dual Use of Get and Put Functions*

Some of the Get and Put functions have dual usage in that they set either an CV/IV value or a property of one of these. This may be confusing to some users until they determine the conditions under which they want to set values to either of these entities.

## **3.6 Accessing Methods**

*Methods* are Lisp functions which respond to a message sent to an object. Methods are defined in a class. When a method is sent to an instance, the method is located in its parent class or superclasses and invoked with the parameters in the message.

## Medley LOOPS: The Basic System

### *3.6.1 Accessing Method Properties*

LOOPS defines several functions for getting the values of methods defined for classes.

#### **3.6.1.1 Getting Methods**

Three functions are used to retrieve a method or the value of its properties: **GetMethod**, **GetMethodOnly**, and **GetMethodHere**. Their format was:

Function:	GetMethod GetMethodOnly GetMethodHere
Arguments:	<class>, the name of a class. <method>, the name of the method to be invoked. <propName>, the name of a property of the method..
Return:	The name of the <method> or the value of <propName>, if non-NIL.

GetMethod returns the method's Interlisp function name, which implemented the method, if <propName> was NIL. Since method properties are inherited, if the <propName> did not have a value in the <class>, LOOPS searches the superclasses to find the <method> and <propName> to retrieve the value.

GetMethodOnly does not trigger an active value if one was associated with the <propName>.

GetMethodHere returns the value of <propName> if it was defined locally; otherwise, it returned the value of **NotSetHere**.

## Medley LOOPS: The Basic System

Note: These functions work only on classes, not on instances.

### 3.6.3.2 Putting Methods

Two functions are used to add a new value for a method property: **PutMethod** and **PutGetMethodOnly**. Their format is:

Function:	PutMethod PutMethodOnly
Arguments:	<class>, the name of a class. <method>, the name of the method to be invoked. <newValue>, the new value to be stored. <propName>, the name of a property of the method.
Return:	<newValue>.

PutMethod stores <newValue> as the implementing function of <method>, if <propName> is NIL. Otherwise, it sets the value of <propName> associated with <method>.

PutMethodOnly does not invoke the putFn of an active value if one was associated with <propName>, but sets the value directly.

If a <method> or <class> is inherited, the value is changed in the class in which the <method> is defined, not the method of the class presented as an argument.

Note: These functions work only on classes, not on instances.

### 3.7 Delete Functions

LOOPS provides two functions for deleting variables within a class definition: **DeleteCV** and **DeleteCIV**:

Function:	DeleteCV DeleteCIV
Arguments:	<class>, the name of the class. <varname>, the name of the variable. <prop>, the name of a property of the class.
Return:	The <varname>, if deleted; otherwise, NIL.

DeleteCV fetches *cvNames* from the class record and searches it for the name of the variable. If found, it fetches the *cvDescrs* from the class record, locates the variable's descriptor from the record, deletes the class variable's name from *cvNames*, and sets the new values into the class record.

DeleteCIV works similarly but uses the value of the field *LocalIVs* in the class record, to search for the IV name, and delete it, if found. Here is an example:

```
2/227+ (PutCIVHere State 'Flower NIL)
NIL
2/228+ DeleteCIV State 'Flower)
NIL
```

If <prop> is non-NIL, the <varname> is the name of a property which is removed from either the *cvDescrs* or the *ivDescrs*.

### 3.8 Destroying Classes

You may also destroy classes using the following methods.

#### *3.8.1 Removing a Class*

The **Destroy** method removes a class from the LOOPS system. Its format is:

Method:	Destroy
Arguments:	<class>, the handle of the class.
Return:	NIL.

This method sends the method **DestroyClass** to the metaclass of the specified class.

For example”

```
(* ; "This data set is used for testing the features of LOOPS")
(* ; "Prepared by Steve Kaisler")
(* ; "Assumes that plantagenet.txt has been loaded.")
```

```
(* ; "Create a new class, Joker, of Person.")
(SETQ Joker (SEND Person New))
(SEND Joker SetName 'Joker)
(PP Joker)
```

```
(* ; "Now destroy the subclass Joker.")
(SETQ result (SEND Joker Destroy))
```

## Medley LOOPS: The Basic System

(PP Joker)

(PRIN1 "Joker has been destroyed: ")

(PRINT result)

And, loading TestClass.txt:

```
2/18+ (LOAD 'TestClass.txt)
{DSK}<home>steve>LOOPS-MAIN>TestClass.txt;1
VARS definition for Joker:
(RPAQQ Joker #,($ Joker))
INSTANCES definition for Joker:
(DEFINST Person (Joker (F%[%VHRaU1.0.0.VP7 . 15))
)

VARS definition for Joker:
(RPAQQ Joker #,($& DestroyedObject (92 . 65192)))
Joker has been destroyed: NIL
```

Note: users should be careful in destroying a class, especially one that is in the midst of a class hierarchy. Before destroying a class, users should use the Classbrowser to check where the class is positioned in the class hierarchy. Destroying a class in the midst of a class hierarchy without forethought may invalidate downstream classes and cause portions of the system to fail.

### *3.8.2 Destroying a Class*

The method **DestroyClass** is not generally used by user programs, but is sent by **Destroy** to perform the actions of destroying a class. Its format is:

## Medley LOOPS: The Basic System

Method: DestroyClass  
Arguments: <class>, the metaclass of the class to destroy.  
<classToDestroy>, the class to destroy.  
Return: NIL.

This method performed the following functions within the LOOPS system:

- Removed <classToDestroy> from any files on FILELST.
- Sends the Destroy! message to all methods associated with <classToDestroy>.
- Removed <classToDestroy> from any subclass data contained within its <supers>.
- Changes the class name to “aDestroyedClass”.
- Changes the supers list of <classToDestroy> to DestroyedObject and Object.
- Changes the metaclass of <classToDestroy> to DestroyedClass.
- Sets all fields of the internal class data structure to NIL.

This class can be specialized to change the way classes are destroyed. For example, if the user program wants to preserve some data in the class to be destroyed before it is actually removed from the system.



## Medley LOOPS: The Basic System

### 3.8.3 Ensuring Removal of Subclasses

The method **Destroy!** destroys a class and all subclasses. Its format is:

Method:        Destroy!  
Arguments:    <class>, the handle of the class to destroy.  
Return:        NIL.

Recursively sends the **Destroy** message to *self* and its subclasses. This allows users to remove entire branches of the class hierarchy.

Note: Users should be very careful in using this method as it generally wipes out a subbranch of the class hierarchy. Unless the user has saved the commands for creating each subclass and setting its attributes, the user will not be able to recover the subbranch of the class hierarchy.

## 3.9 Inheritance

Classes exist in a class-subclass hierarchy. In each class, the *supers* list defines where the class is in the hierarchy. When a class is created as a subclass of one or more classes, it contains the IVs of all the class in the *supers* list, and all IVs of the classes up the hierarchy to the metaclass.

The highest class in the LOOPS hierarchy is called **Tofu**, which stands for *Top of the Universe*. This class is very simple. It has no instance variables and three defined messages:

- MessageNotUnderstood
- MessageNotFound
- SuperMethodNotFound

Table 3-1 describes these three messages (Xerox 1991b).

**Table 3-1. Tofu Message Descriptions**

Message	Description
MessageNotUnderstood	Provides an error handling mechanism for when a message is sent to an object which cannot respond to the message.
MessageNotFound	Provides a mechanism for intermediate checking before sending the message MessageNotUnderstood.
SuperMethodNotFound	Provides a mechanism for intermediate checking before sending the message MessageNotUnderstood.

Tofu has two specializations as indicated in Figure 3-1:

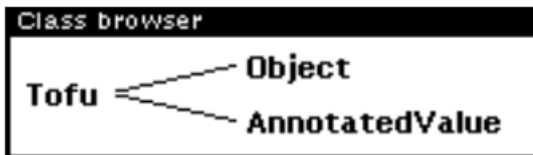


Figure 3-1. Tofu Specializations

Source: Xerox 1991b

The Object class is the root of most of the other LOOPS classes. AnnotatedValue is the root used with Active Values. It is recommended that **Tofu** only be specialized for some necessary conditions such as a new capability.

## Medley LOOPS: The Basic System

Consider the following example:

```
(* ; "Demonstrate Tofu Messages, Section 3.9.1")
(SETQ result (SEND Person WhatsMyLine))
(PRIN1 "Person does not have method: ")
(PRINT result)
```

The result is:

```
((+ #,($C Person) WhatsMyLine --) not understood
```

The result “not understood” is the result of sending the MessageNotUnderstood to Tofu.

### 3.10 Compact Forms for Accessing Data

LOOPS provides compact forms as macros that, when expanded, yielded data access functions as previously described in this chapter. These compact forms used the ‘@’ character as an element of a function call in a method. Table 3-2 describes the compact forms.

**Table 3-2. Compact Access Forms**

Access Form	Description
@	Yielded GetValue and GetClassValue functions.
@*	Yielded GetValue functions.
_@	Yielded PutValue and PutClassValue forms for assigning a new value.

### 3.10.1.1 @ Form

The @ form took an argument of an *access path* to a variable. The access path could consist of one to three arguments:

- One argument: self is assumed to be the object and the access path specifies an instance variable. For example, the form (`@ iv3`) translates to (`GetValue self 'iv3`).
- Two arguments: the first argument is an object and the second argument is an IV. For example, (`@ ($ w) center`) would translate to (`GetValue ($ w) 'center`).
- Three arguments: the first argument is an object, the second argument is an IV, and the third argument is a property. For example, (`@ ($ w) menu 'DontSave`) would translate to: (`GetValue ($ w) 'menu 'DontSave`).

Now, we can get the value of Spouses from EdwardIII using the GetValue function:

```
2/9+ (GetValue ($ EdwardIII) 'Spouses)
PhilippadAvesnes
```

But, if we use the '@' notation, we have:

```
2/13+ (SETQ result2 (@ ($ EdwardIII) 'Spouses))
Invalid form in access expression:
(QUOTE Spouses)

2/14+ (SETQ result2 (@ ($ EdwardIII) Spouses))
PhilippadAvesnes
```

because '@' is an Nlambda form, which does not evaluate the arguments. Thus, the second argument does not need to be quoted.

### 3.10.1.2 @\* Form

The @\* form generates GetValue forms. It takes an <access path> followed by a list of IV names. For example,

(@\* (\$ foo) a b c)

(GetValue (GetValue (GetValue (\$ foo) 'a) 'b) 'c)

### 3.10.1.3 @\_ Form

The @\_ form is used to assign a new value to an IV. It takes an <access path> followed by a new value. For example:

<example>

### 3.10.1.4 Testing @ Forms

Consider the definition of EdwardIII from the Plantagenet data set:

```
#,($& Person (92 . 65104))
#,$ EdwardIII)
(11 13 1312)
(6 21 1377)
Male
EdwardII
IsabellaCapet
"Loaded EdwardIII"
"Loaded EdwardIII"
"Philippa d'Avesnes"
#,$& Person (92 . 65096))
#,$ PhilippadAvesnes)
Female
(6 24 1311)
(8 15 1369)
PhilippadAvesnes
EdwardIII
```

And, here is the complete set of tests:

## Medley LOOPS: The Basic System

```
(* ; "Testing the @ Access Forms")
(PRINT "Testing the @ Form")

(SETQ atResult (@ ($ EdwardIII) Father))
(PRIN1 "atResult = ")
(PRINT atResult)

(PRINT "Testing the @* Form")
(SETQ atResult2 (@* EdwardIII Father))
(PRIN1 "atResult2 = ")
(PRINT atResult2)

(PRINT "Testing the _@ Form")
(SETQ atResult3 (_@ EdwardIII 'Spouses NIL))
(PP EdwardIII)
```

Note: the ‘\_@’ symbol is interpreted by Interlisp as ‘+@’

```
{DSK}<home>steve>LOOPS-MAIN>testataccess.txt;1
"Testing the @ Form"
atResult = EdwardII
"Testing the @* Form"
atResult2 = EdwardII
"Testing the +@ Form"
'Spouses {in (PutValue EdwardIII (QUOTE 'Spouses) NIL)} -> Spouses ? y
es
VARS definition for EdwardIII:

(RPAQQ EdwardIII #,($ EdwardIII))
INSTANCES definition for EdwardIII:
(DEFINST Person (EdwardIII (FM%e3k\U1.0.0.e07 . 13))
  (Father EdwardII)
  (Mother IsabellaCapet)
  (Spouses NIL)
  (Gender Male)
  (Birthdate (11 13 1312))
  (Deathdate (6 21 1377)))
```

As we see, the Spouses of EdwardIII is now NIL.

### 3.10.2 IV Delimiters

A “:” is a delimiter that indicated instance variable access. Table 3-3 depicts the delimiters and describes their meaning. Each of these delimiters is followed by the name of a variable

**Table 3-3. Instance Variable Delimiters in Compact Forms**

Delimiter	Usage
:	Accesses the value of the IV.
::	Accesses the value of the CV.
;	Accesses the value of the property.
.	Sends a message to the object with the selector.
!	Evaluates the next expression.
\	Specifies the next symbol is a Lisp symbol.
\$	Specifies the net object is a LOOPS object.

Here are some examples that demonstrate the use of these delimiters. These delimiters can be tested using:

(Parse@ (List <access path>) 'IV)

(@ foo)

(Parse@ (List 'foo) 'IV)

is translated to



```
2/48+ (ParsePut@ (LIST 'foo '1234) 'IV)
(PutValue self (QUOTE foo) 1234)
2/48+ A
```

### 3.11 Class Method Operations

LOOPS provides several types of operations for the methods of a class.

#### *3.11.1 Defining a Method*

We can define a method for a class using the LOOPS function **DefineMethod**, which has the format:

Function:	DefineMethod
Arguments:	<className>, the name of the new class. <methodName>, the name of the method. <args>, a list of arguments. <expr>, an expression defining the function. <file>, the file where the method is located. (optional)
Return:	Varies according to the arguments.

For Person, we can define both put and get methods to access the values of these attributes within an instance of the class. But, we define these methods in the class definition so every instance has access to them. The following examples defines some of these methods:

## Medley LOOPS: The Basic System

```
(* ; "Define put methods for person")
(DefineMethod ($ Person) 'PutFather '(person newValue)
  '(PutValue person 'Father newValue)
)

(DefineMethod ($ Person) 'PutMother '(person newValue)
  '(PutValue person 'Mother newValue)
)

(* ; "Define get methods for Person")
(DefineMethod ($ Person) 'GetFather '(person)
  '(GetValue person 'Father)
)

(DefineMethod ($ Person) 'GetMother '(person)
  '(GetValue person 'Mother)
)
```

To check the definition of the methods, we can use PP again to print data about the method:

## Medley LOOPS: The Basic System

```
2/14* (PP Person.PutFather)
FNS definition for Person.PutFather:
(DEFINEQ
  (Person.PutFather
    [LAMBDA (self person newValue)
      (CL:COMPILER-LET ((*ArgsOfMethodBeingCompiled* '(self person
                                                             newValue))
                       (*ClassNameOfMethodOwner* 'Person)
                       (*SelectorOfMethodBeingCompiled* 'PutFather)
                       (*SelfOfMethodBeingCompiled* 'self))
        (PutValue person %'Father newValue])
      )
  )
METHOD-FNS definition for Person.PutFather:
(Method ((Person PutFather) self person newValue) "Method
documentation"
(PutValue person %'Father newValue))
METHODS definition for Person.PutFather:
(\BatchMethodDefs)
(METH Person PutFather (person newValue)
 "Method documentation" (category (Person)))

(Method ((Person PutFather) self person newValue) "Method
documentation"
(PutValue person %'Father newValue))
(\UnbatchMethodDefs)
```

### 3.11.1.1 Extended Example

We will define a method called SetDeathDate for Person and then set the death date for HenryII.

```
(DefineMethod Person 'SetDeathdate '(day month year)
  '((PROG NIL
    (COND
      ((NOT (AND (> day 0) (< day 32))))
      (PRIN1 day)
      (PRINT "is not in range [1...31].")
      (RETURN NIL)
    )
  )
)
(COND
  ((NOT (AND (> month 0) (< month 13))))
  (PRIN1 month)
  (PRINT "is not in range [1...12].")
  (RETURN NIL)
)
)
(COND
  ((NOT (AND (> year 0) (< year 2100))))
  (PRIN1 month)
  (PRINT "is not in range [1...2100].")
  (RETURN NIL)
)
)
```

## Medley LOOPS: The Basic System

```
(* ; "Set Deathdate")
  (SETQ Deathdate (LIST day month year))
  (RETURN Deathdate)
)
)
2/31 ← (SEND ($ HenryII) SetDeathdate 06 07 1189)
(6 7 1189)
```

A similar method has been defined for setting the birthdate as well.

Note: The function definition should be enclosed in a PROG.

A second example shows that an external function may be called from the function definition of the method. Here, the function ComputeAge is called from the method to compute the age of the person.

## Medley LOOPS: The Basic System

```
(* ; "*****")
(* ; "Person Methods")
(* ; "Compute Age of a Person given deathdate and birthdate")
(DefineMethod ($ Person) 'Age '(person)
  '(PROG (age)
    (PRIN1 'Computing age of ")
    (PRINT person)
    (SETQ age (ComputeAge person))
    (PRIN1 "The age of ")
    (PRIN1 person)
    (PRIN1 " is ")
    (print age)
    (RETURN age)
  )
)
```

If we execute the function (PP Person.age), we see:

## Medley LOOPS: The Basic System

```
|METHOD-FNS definition for Person.Age:
(Method ((Person Age) self person) "Method documentation"
  (PROG (age)
    (PRIN1 "Computing age of ")
    (PRINT person)
    (SETQ age (ComputeAge person))
    (PRIN1 "The age of ")
    (PRIN1 person)
    (PRIN1 " is ")
    (PRINT age)
    (RETURN age)))
METHODS definition for Person.Age:
(\BatchMethodDefs)
(METH Person Age (person)
  "Method documentation" (category (Person)))

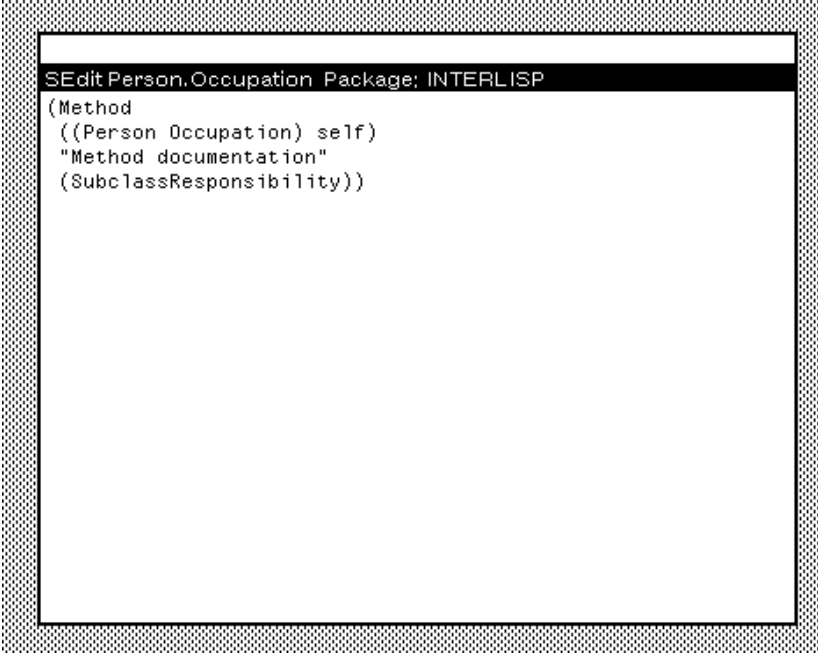
(Method ((Person Age) self person) "Method documentation"
  (PROG (age)
    (PRIN1 "Computing age of ")
    (PRINT person)
    (SETQ age (ComputeAge person))
    (PRIN1 "The age of ")
    (PRIN1 person)
    (PRIN1 " is ")
    (PRINT age)
    (RETURN age)))
(\UnbatchMethodDefs)
```

### 3.11.1.2 Invoking the Editor

If the <args> and the <expr> are NIL, then Interlisp invokes the editor to define the function and its arguments.

```
2/83← (DefineMethod Person 'Occupation NIL NIL)
```

## Medley LOOPS: The Basic System

A screenshot of a Medley LOOPS editor window. The window has a title bar at the top that reads "SEdit Person.Occupation Package: INTERLISP". Below the title bar, the editor contains the following code:

```
(Method  
  ((Person Occupation) self)  
  "Method documentation"  
  (SubclassResponsibility))
```

The code is displayed in a monospaced font. The editor window is surrounded by a thick, textured border.



## Medley LOOPS: The Basic System

### *3.11.2 Defining a Method by a Definer*

A variant allows the user more control over defining a method for a class using a definer, **Method**, which takes the form:

Definer:	Method
Arguments:	<type>, specifies The:FUNCTION-TYPE (optional). <class>, the class to which the method is attached. <message>, the new method's selector. <object>, this argument must be first in the list. <args>, a list of arguments. <body>, the body of the method.
Return:	The name of the method function.

The :FUNCTION-TYPE specifies the type of function:

- :IL, the body of the function uses Interlisp syntax, including CLISP, or
- :CL, the body of the function uses Common Lisp syntax.

As an example, consider the following (LRM 1993):

## Medley LOOPS: The Basic System

```
(Method :FUNCTION-TYPE:CL
  ((Window myWindow)
    self bar
    &Optional baz
    &REST glorp)
  (CL:FORMAT T
    T "Bar -s baz glorp -s%%"
    Bar baz glorp)
)
```

The method can be invoked for the class using the `<methodName>.<form>`, which is interpreted as:

- If `<form>` is non-nil, then `<argsOrFnName>` is interpreted as a list of arguments for the function and `<form>` is the body of that function.
- If `<argsOrFnName>` and `<form>` are NIL, the Definer creates a skeleton definition for a function and then invokes the Interlisp editor.
- If `<form>` is NIL, then `<argsOrFnName>` is interpreted as the name of an Interlisp function to be used as the implementation of the method.

The structure of the function is specified as

```
(LAMBDA <argsOrFnName> . <form>).
```

If the first element of `<argsOrFnName>` is not **self**, then **self** is inserted at the front of the list by LOOPS prior to executing the method.

## Medley LOOPS: The Basic System

The Definer creates a function name as the concatenation of <className>, '.' (period), and <methodName>. The function would appear as:

```
(DEFINEQ
  (<className>.<methodName>)
  (LAMBDA (self) <comment>
    (@:myValue (ADD1 (@:myValue)))
  )
)
```

The interpretation of this syntax was discussed in Section 3.10 on accessing variable values.

## Medley LOOPS: The Basic System

### 3.11.3 Defining A Method by Message

A method may also be defined by sending the message **DefMethod** to the class:

Message.	DefMethod
Arguments:	<className>, the name of the new class. <methodName>, the name of the method. <argsOrFnName>, a list of arguments or an Interlisp function name. <form>, a function implementing the method.
Return:	Catenated method name of <class>.<method>.

As an example, consider:

```
2/15+ (SEND Person DefMethod 'putMother '(person newValue) '(PutValue  
person 'Mother newValue))  
Person.putMother
```

And, we can see the result:

```
Local properties of Class Person  
MetaClass (Class Edited%: (* ; "Edited 16-Jan-  
Supers (Object)  
IVs (Father Mother Sisters Brothers Spot  
CVs NIL  
Methods (putFather putMother putSister)
```

## Medley LOOPS: The Basic System

Alternatively, here is another example:

```
(* ; "Using DefMethod to define a method in a class.")
(SEND Person
  DefMethod      'SetTitle
                '(person title)
                '(LAMBDA (person title)
                  (PutValue person 'Title title)
                )
)
```

which we can view with (PP Person.SetTitle):

```
FNS definition for Person.SetTitle:
(DEFINEQ
  (Person.SetTitle
    [LAMBDA (self person title)
      (CL:COMPILER-LET ((*ArgsOfMethodBeingCompiled* '(self person title))
                       (*ClassNameOfMethodOwner* 'Person)
                       (*SelectorOfMethodBeingCompiled* 'SetTitle)
                       (*SelfOfMethodBeingCompiled* 'self))
        (LAMBDA (person title)
          (PutValue person 'Title title))]
    )
)
METHOD-FNS definition for Person.SetTitle:
(Method ((Person SetTitle) self person title) "Method documentation"
  [LAMBDA (person title)
    (PutValue person 'Title title)])
METHODS definition for Person.SetTitle:
(\BatchMethodDefs) . . . . .
```

## Medley LOOPS: The Basic System

### *3.11.4 Deleting a Method*

A method may be deleted from a class using the **Delete Method** function, which is defined as follows:

Function: DeleteMethod  
Arguments: <class>, the class from which the methods is to be deleted,  
<method>, the name of the method to be deleted,  
<prop>, T if the function definition is to be deleted;  
otherwise, NIL.  
Return: NIL.

### *3.11.5 Editing a Method*

A method may be edited using the Structure Editor by sending a class the message **EditMethod**:

Class Method: EditMethod  
Arguments: <class>, handle of the class,  
<method>, the name of the method,  
<commands>, a list of editf commands,  
<okCategories>, atom or list specifying valid categories.  
Result: TBD

## Medley LOOPS: The Basic System

The behavior of this method varies with the arguments:

- If <method> was NIL, a menu of methods of the class was presented using the message PickSelector in okCategories. This was used to restrict the methods that a user might be able to delete;
- If <method> was non-NIL and was not a method defined in the class, the user was asked whether the method should be created in the class or not;
- If <method> could not be found, the spelling corrector was invoked to find a correct local method. If it can be corrected, the local method was edited, or an inherited method was made local and edited. EDITF was invoked with the argument <commands>.

So, send EditMethod to Person:

```
2/19+ (SEND Person EditMethod)  
NTI
```

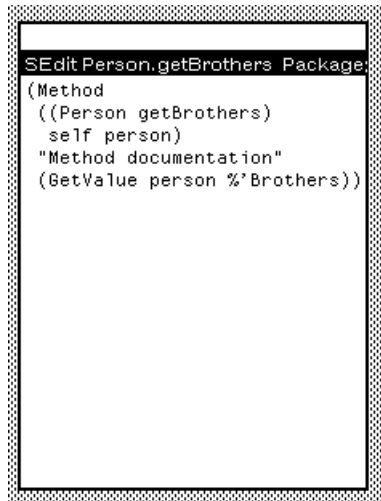
which pops up a window asking which method we would like to edit:



Figure 3-4. Method Edit Menu

## Medley LOOPS: The Basic System

If we click on `getBrothers` in menu in the window, Interlisp prompts us to define a SEdit window in which it displays the `getBrothers` method:

A screenshot of a window titled "SEdit Person.getBrothers Package". The window contains the following Lisp code:

```
(Method
 ((Person getBrothers)
  self person)
 "Method documentation"
 (GetValue person %'Brothers))
```

Figure 3-5. Method Display

If `EditMethod` cannot find the method selector in the specified class, it opens the spelling corrector to find the a local selector whose spelling might be corrected. If it can be corrected, the local method is used or an inherited method from a superclass is used. When the method name is determined, `EDITF` is invoked with *commands* passed as the second argument.

The `ClassInheritanceBrowser` could also be used to edit the method.



## Medley LOOPS: The Basic System

### *3.11.6 SubclassResponsibility*

If the result of either `DeleteMethod` or `EditMethod` was to define a new method in the class, then as part of creating the new method, a template is displayed which included `SubclassResponsibility` as an entry.

### *3.11.7 Alternatives to Executing Methods*

Alternative functions were available to execute methods to the messaging syntax. These functions allowed the programmer to determine the method to be dynamically applied to an instance based on the current state of the program.

#### **3.11.7.1 Executing a Method**

**DoMethod** computed an action which should be a method, associated a class, and applied it to an object and arguments, which were defined as follows:

Function:	DoMethod
Arguments:	<object>, an instance of a class to which the action is to be applied; <method>, name of the method to be execute, <class>, the class in which the method resided, or NIL, <args>, the arguments for the method.
Return:	Value returned by the method.

## Medley LOOPS: The Basic System

All arguments were evaluated. If <class> was NIL, DoMethod used the class of <object>. If the <method> did not exist in the class, an error was generated.

The TestDoMethod file has the following definition:

```
(* ; "Test DoMethod")  
(DoMethod EdwardIII 'Age Person '(EdwardIII))  
STOP
```

Note: That STOP must be the last statement in any Interlisp file to be loaded.

And the result of running it is:

```
2/20/84 (LOAD 'TestDoMethod.txt)  
{DSK}<home>Steve>loops-tests>Plantagenets>TestDoMethod.txt;1  
Computing age of (EdwardIII)  
"ComputeAge"  
The age of (EdwardIII) is 0  
{DSK}<home>Steve>loops-tests>Plantagenets>TestDoMethod.txt;1
```

DoMethod allows the user to dynamically select a method to be applied to a class based on criteria set by the program state.

### 3.11.7.2 Applying a Method

**ApplyMethod** applies the specified method to the already evaluated arguments of an instance; otherwise, it operates the same as DoMethod. It was defined as:

## Medley LOOPS: The Basic System

Function:      ApplyMethod  
Arguments:     <object>, the instance to which the action is to be applied,  
                  <method>, the method to be applied,  
                  <arglist>, the argument for the method,  
                  <class>, the class containing the method.  
Return:        The value returned by the method.

The definition of TestApplyMethod is:

```
(* ; "Test ApplyMethod")  
(ApplyMethod EdwardIII 'Age '(EdwardIII) Person)  
STOP
```

and, the result is:

```
2/94← (LOAD 'TestApplyMethod.txt)  
{DSK}<home>Steve>loops-tests>Plantagenets>TestApplyMethod.txt;1  
Computing age of EdwardIII  
"ComputeAge"  
The age of EdwardIII is 0  
{DSK}<home>Steve>loops-tests>Plantagenets>TestApplyMethod.txt;1
```

## Medley LOOPS: The Basic System

The following example, taken from the LRM 1991, demonstrates ApplyMethod with MessageNotUnderstood.

```
(Method
  ((DwimObject MessageNotUnderstood)
    self
    selector
    messageArguments
    superFig)
  (LET ((correctSelector (FixSelectorSpelling selector)))
    (COND
      ((correctSelector
        (ApplyMethod correctSelector messageArguments)
        ))
      (T (_Super)))
    )
  )
)
```

### 3.11.7.3 Executing a Method in the Class Hierarchy

**DoFringeMethods** is a function that either executes the method specified for the instance or searches up the class hierarchy to execute the method in a superclass. It is defined as:

## Medley LOOPS: The Basic System

Function: DoFringeMethods  
Arguments: <object>, an instance of a class,  
<method>, a method in the class or one  
of its superclasses,  
<arguments>, a list of arguments to the method.  
Return: NIL.

All the arguments were evaluated. If the <method> in the class of the <object> is defined in that class (not through inheritance), the local method was invoked.

If there was no local method, it searches the superclass hierarchy for the definition of the method and executes it in each superclass.

Note: this may result in the method being executed several times with the most specific version of the method executed first.

### 3.12 Manipulating Methods Across Classes

Several functions and methods were provided to move methods, instance variables, and class variables between classes.

#### *3.12.1 Renaming a Method*

A method in a class could be renamed using the function **RenameMethod**, as defined below:

## Medley LOOPS: The Basic System

**Function:** RenameMethod  
**Arguments:** <class>, the name of the class in which the method is defined,  
<oldMethodName>, the old name of the method before this function is called,  
<newMethodName>, the new name of the method after this function is called.  
**Return:** If successful, returns <newMethodName>.

We can rename `Person.Age` to `Person.GetAge` as follows:

```
2/95+ (RenameMethod Person 'Age' 'GetAge')
Person.Age is not broken.
Person.GetAge
```

<<Not sure why it prints `Person.Age is not broken`>>

It is likely rare that a user would rename a method. But, one case where this is appropriate is when a method has evolved over time with additional code. To make it easier to understand, the user might want to split the method into a major method and minor methods. The major method is invoked by the program, perhaps externally, and the minor methods are invoked from the major method privately.

### *3.12.2 Moving a Method between Classes*

A method may be move from one class to another with deletion from the old class, possibly with renaming. **MoveMethod** is defined as:

## Medley LOOPS: The Basic System

Function:        MoveMethod  
Arguments:     <oldClassName>, the class containing the method  
                 before this function is called,  
                 <newClassName>, the class containing the method  
                 after the method is moved,  
                 <method>, the name of the method to be moved,  
                 <newMethod>, if non-NIL the name of the method  
                 in the new class,  
                 <files>, a list of files in which the change is to be  
                 made.  
Return:         <newClassName, if specified;  
                 otherwise, <oldClassName>.

When the method was moved to the new class, it was deleted from the old class.

### *3.12.3 Alternate Moving a Method*

An alternate approach to using a function to move a method was use the method version of **MoveMethod**. It is described as:

Method:        MoveMethod  
Arguments:     <object>, the handle of the class from which the  
                 method will be moved,  
                 <newClassName>, the class to which the method  
                 will be named,  
                 <method>, the name of the method to be moved.  
Return:         <newClassName>.

### *3.12.4 Moving Methods to a File*

**MoveMethodsToFile** moved a method to a file if it had the same name as a method in the file. It is defined as:

Function:      MoveMethodsToFile  
Arguments:    <filename>, the name of the file to which  
                  the methods are moved.  
Return:        NIL, if the method does not have a corresponding  
                  entry in the file; otherwise, T.

This method is used when the method has been edited and then its code was saved to a file to ensure the source code was not lost.

During a long editing session, perhaps stretching over several interactive sessions, this was useful for ensuring that the method code was updated with exiting the interaction session.

### *3.12.5 Getting Functions Called from a Class Set*

It is often useful to find all functions that are called from one or more classes using the function **CalledFns**, which is described as:

Function:      CalledFns  
Arguments:    <classes>, a list of classes to search,  
                  <definedFlg>, either NIL, 1, or T.  
Return:        A list of functions; otherwise, NIL.

The user can determine what functions are called from the Window class as depicted in Figure 3-5:



## Medley LOOPS: The Basic System

```
2/45+ (CalledFns '(Window) T)
(Window.AfterMove Window.AfterReshape Window.AttachLispWindow Window.Blin
k Window.Bury Window.ButtonEventFn Window.ChoiceMenu Window.Clear Window.
ClearMenuCache Window.ClearPromptWindow Window.Close Window.ClosePromptWi
ndow Window.CreateWindow Window.CursorInside? Window.Destroy Window.Detac
hLispWindow Window.GetMenuItems Window.GetPromptWindow Window.GetProp Win
dow.Hardcopy Window.HardcopyToFile Window.HardcopyToPrinter Window.HasLis
pWindow Window.Invert Window.ItemMenu Window.LeftChoice Window.LeftSelect
ion Window.MiddleChoice Window.MiddleSelection Window.MousePackage Window.
MouseReadable Window.Move Window.Move1 Window.Open Window.Open? Window.
Paint Window.PromptEval Window.PromptForList Window.PromptForString Windo
w.PromptForWord Window.PromptPrint Window.PromptRead Window.Repaint Windo
w.RightButtonFn Window.RightSelection Window.ScrollWindow Window.SetOuter
Region Window.SetProp Window.SetRegion Window.Shade Window.Shape Window.S
hape1 Window.Shape? Window.Shrink Window.Snap Window.TitleSelection Windo
w.ToTop Window.Update Window.WhenMenuItemHeld)
2/46+ ▲
```

Figure 3-6. Functions Called From Window

### 3.13 Methods Concerning the Class of an Object

Given an instance, we can find its class and determine if it is an instance of a specified class.

#### 3.13.1 Finding the Class of an Object

We can determine the class of an object. There are two cases:

- If the object is an instance of a LOOPS class, or
- If the object is an instance of a Lisp class.

Macro:            Class

Arguments:        self, a pointer to an object.

Return:            Value depending on the argument.

If *self* is a LOOPS object, it returns the class of the object.

If it is not a LOOPS object, e.g., a Lisp object, the function (GetLispClass self) is evaluated to return the Lisp class.

## Medley LOOPS: The Basic System

For example,

```
2/15← (Class EdwardIII)
#,$C Person)
2/16← (SEND EdwardIII Class)
#,$C Person)
```

Note that the message `Class` sent to a LOOPS object yields the same result as calling the Macro `Class` on the object.

Another example is shown below (from the `OntologyCase` file):

```
2/55← (Class 'Ireland)
#,$C Tofu)
```

where `Ireland` is a subclass of `Tofu`, the top-level LOOPS class.

Suppose we define a function as `Add1`:

```
2/17← (DEFINEQ (Add1
                (LAMBDA (x)
                  (IPLIS x 1)
                )
              ))
(Add1)
2/18← (Class Add1)
Add1 is an unbound variable.
```

where `Add1` is not a LOOPS or Lisp class.

### *3.13.2 Getting the Class Name*

The name of a class can be found using **`ClassName`**.

## Medley LOOPS: The Basic System

Function:      ClassName  
Arguments:     <class>, the name of an object.  
Result:        Depends on the type of object.

If <class> is a LOOPS class, it returns the name of the class. If it is an instance of a class, it returns the name of the parent class of the instance. EdwardIII is an instance of Person.

```
2/82+ (ClassName Person)
Person
2/83+ (ClassName EdwardIII)
Person
2/84+
```

If self is neither a class or an instance of a class, the function GetLispClass was called with self as an argument. If it returns NIL, the function LoopsHelp was called with self and “has no class name”.

```
2/22+ (ClassName Add1)
Add1 is an unbound variable.

2/23+ (SETQ X (IPLUS X 1))
X is an unbound variable.

2/24+ SETQ X 20)
20
2/25+ (ClassName x)
x {in EVAL} -> X ? yes
Tofu
```

where, as previously, Tofu was the top of the class hierarchy for Lisp classes.

Similar results can be obtained by sending the message ClassName to self.

## Medley LOOPS: The Basic System

```
2/26+ (SEND EdwardIII ClassName)  
Person
```

### 3.13.3 Determining an Instance of a Class

An object can be determined to be an instance of a class by sending it the message **InstOf**.

Method: InstOf  
Arguments: self, a pointer to an object.  
            <class>, a pointer to a class or its symbolic name.  
Result: T or NIL.

Consider the following different forms:

```
2/31+ (SEND EdwardIII InstOf 'Person)  
T
```

Showing both forms of sending a message to a LOOPS object.

```
2/38+ (+ ($ EdwardIII) InstOf ($ Person))  
T  
2/39+ (SEND ($ EdwardIII) InstOf ($ Person))  
T
```

A variant of this function, **InstOf!**, determines if self is an instance of a class or any of the classes' subclasses.

### 3.13.4 Copying Instances

Two methods can be used for copying instances: deep copying or shallow copying.

## Medley LOOPS: The Basic System

Deep copying Creates a new instance of the same class as *oldInstance*. **CopyDeep** fills the instance variables of the new instance with copies of lists, active values, and instances pointed to by the *oldInstance*.

Method: CopyDeep  
Arguments: <oldInstance>, a pointer to an instance.  
<newObjList>, an association list.  
Return: The handle of the new list.

Here is an example from the LRM:

```
2/95+ (SETQ newEdward (CopyDeep EdwardIII))
CopyDeep is an undefined function.
```

**Note: CopyDeep seems to be an undefined function in Medley LOOPS. This is being investigated.**

## Chapter Four

### Instance Functions and Methods

LOOPS provides a diverse set of functions and methods for defining and manipulating instances of classes.

#### 4.1 Defining a New Instance

There are several ways to create new instances of a class. When an instance is created by sending the **New** message to a class, the default behavior for **Class.New** is to send the message **NewInstance** to the newly object that was created.

**NewInstance** can be specialized if special or additional operations are required at the time that the new instance is created. The specialization of **NewInstance** should return self.

##### *4.1.1 Sending the Class the Message NEW*

A new instance may be defined by sending a class the message **NEW**, which has the format:

## Medley LOOPS: The Basic System

Method:        New  
Arguments:     <class>, the name of the class for the new instance.  
                  <name>, the name to be assigned to the new  
                  instance.  
                  <arg1> ... <argN>, arguments which are passed to  
                  **NewInstance** when it is sent to the new object.  
Return:        The handle of the new instance of the <class>.

After the new instance of the class is created, it is sent the message **NewInstance** with the arguments <arg1> ... <argN>.

```
(SEND ($ <class> NEW '<instanceName>')  
(SEND <className> NEW '<instanceName>')
```

The latter case applies if you have assigned the class record to the Interlisp variable via SETQ as in:

```
(SETQ <instanceName> (SEND ($ <class> NEW '<instanceName>'))  
(SEND <instanceName> SetName '<instanceName>')
```

This works because the Lisp name space and the LOOPS name space are separate name spaces.

For example, using Person, let us create an instance for EdwardI from Plantagenets:

```
(* ; "EdwardI")  
(SETQ EdwardI (SEND Person New))  
(SEND EdwardI SetName 'EdwardI)  
  
(PutValue EdwardI 'Birthdate (LIST 06 17 1239))
```

## Medley LOOPS: The Basic System

```
(PutValue EdwardI 'Deathdate (LIST 07 07 1307))
```

```
(PutValue EdwardI 'Gender 'Male)
```

```
(putFather EdwardI 'HenryIII)
```

```
(putMother EdwardI 'EleanorBerenger)
```

```
(putSpouse EdwardI 'EleanorOfCastile)
```

```
(PRINT "Loaded EdwardI")
```

In the default case, the **New** method uses the default values for the IVs in the new instance.

```
2/87← (PP EdwardI)  
VARS definition for EdwardI:  
(RPAQQ EdwardI #, ($ EdwardI))  
INSTANCES definition for EdwardI:  
(DEFINST Person (EdwardI (UV%ho2@X1.0.0.8j6 . 9))  
  (Father HenryIII)  
  (Mother EleanorBerenger)  
  (Spouses EleanorOfCastile)  
  (Gender Male)  
  (Birthdate (6 17 1239))  
  (Deathdate (7 7 1307)))
```

### 1.4.1.1 Instance Handles

In order to manipulate a LOOPS object, we have to be able to access it. In LOOPS, we need to assign a *handle* representing the object to an Interlisp variable that allows us to reference the object after it has been created.



### Handle versus Pointer

In the LRM and other references, the term ‘pointer’ is used to specify a way to access an object. In other programming languages, such as C or C++, ‘pointer’ often is interpreted as an address in the program’s memory that allows direct access to the object. Such pointers can be subject to operations, such as arithmetic or indexing, This violates the notion of an object as an entity. We use the term ‘handle’ to specify a LOOPS object. The Interlisp virtual machine translates a handle into an address in the Interlisp virtual memory. It is not an address.

#### 1.4.1.2 Computed LOOPS Names

Suppose we attempt to define a new LOOPS object using the form:

```
2/98+ (SETQ X1 (SEND ($ Class) New))
NIL must be a LITATOM to be a class name.
```

When we send the selector New to Class, it requires a name for the class. So, we must use:

```
2/102+ (SETQ X1 (SEND ($ Class) New 'X1))
#, ($C X1)
```

X1 contains the handle of the new object. LOOPS enters the handle of the object into an internal table.

Suppose we assign a name to a new LOOPS object using the form:

## Medley LOOPS: The Basic System

```
2/103+ (SEND ($ X1) SetName 'Michael)
#,$($C Michael)
2/104+ (PP X1)
VARS definition for X1:
(RPAQQ X1 #,$($ Michael))
NIL
```

The object X1 can be referred to by the expression (\$ Michael). So, comparing them:

```
2/105+ (EQUAL ($ X1) ($ Michael))
T
```

We see that the LOOPS object can be referred to by both names. This is sometimes useful to have an internal name in a program and an external name for public use.

Also, suppose we set the variable Y to Michael:

```
2/106+ (SETQ Y 'Michael)
Michael
2/107+ ($! Y)
#,$($C Michael)
```

Then, we can use the form (\$! Y) = (\$ Michael) to reference the object.

### *4.1.2 Using NewInstance Message*

The **NewInstance** message sent to a newly created instance of a class allows the program to specialize the initialization of the new object. It has the format:

## Medley LOOPS: The Basic System

Method:        NewInstance  
Arguments:     <object>, evaluates to the handle of a class.  
               <name>, the LOOPS name for the new instance.  
               <arg1> ... <arg5>, optional arguments to be used  
               by user written code that specializes an instance  
               of NewInstance.  
Return:        LOOPS name of newly created instance.

A class may have the method self defined for it using the Assignment @ form:

```
(DefineMethod     ($ <class>)  
                  'New  
                  '(self name <arg1> ... <arg5>)  
                  '(_@    (_ self NewInstance name)  
                          <arg1>  
                          ...  
                          <arg5>)  
                  )
```

This sets the name of the new instance to <name> when it is created. The list after NEW are the arguments to be passed to **NewInstance**.

The default case is to use the default values for the instance variables to initialize the instance variables values in the new instance. The default values are determined from the definition of the instance variables in the class.

## Medley LOOPS: The Basic System

We can use **NewInstance** as follows:

```
2/123+ (SETQ Sam (SEND Person NewInstance 'Sam))

New METHOD-FNS definition for Sam.SetTitle.
New METHOD-FNS definition for Sam.GetAge.
#,$C Sam)
2/124+ (PP Sam)
VARS definition for Sam:

(RPAQQ Sam #,$ Sam))
INSTANCES definition for Sam:
(DEFCLASS Sam
  (MetaClass Class Edited%:  **COMMENT** )
  (Supers Object)
  (ClassVariables (CitizenOf England doc  **COMMENT** ))
  (InstanceVariables (Father #,NotSetValue doc NIL)
    (Mother #,NotSetValue doc NIL)
    (Sisters #,NotSetValue doc NIL)
    (Brothers #,NotSetValue doc NIL)
    (Spouses #,NotSetValue doc NIL)
    (Gender #,NotSetValue doc NIL)
    (Birthdate #,NotSetValue doc NIL)
    (Deathdate #,NotSetValue doc NIL)
    (Title #,NotSetValue doc NIL)))
```

where the initialization of the new instance is performed by the class rather than the metaclass. As noted in the LRM, subclasses of **Object** should have a **\_Super** form within the method to allow the execution of the default behavior.

### *4.1.3 Creating an Instance with Initial Values*

A new instance of a class may be created and the initial values of the instance variables specified using the message **NewWithValues**.

## Medley LOOPS: The Basic System

Message:           NewWithValues  
Arguments:        <class>, the handle of a class.  
                  <valDescriptionList>, a list of varNames with  
                  values and, possibly, associated properties and  
                  their values for <varName>.  
Return:            The handle of the new object.

<valDescriptionList> is a list of value descriptions having the following form:

```
((<varName1> <value1> <prop1> <propValue1> ...)
 (<varName2> <value2> <prop2> <propValue2> ...)
 ...
 )
```

where: <varName<sub>i</sub>> is the name of an instance variable of the class.  
      <value<sub>i</sub>> is the value associated with <varName<sub>i</sub>>  
      <prop<sub>j</sub>> indicates the jth property of the variable  
      <propValue<sub>j</sub>> is the value of <prop<sub>j</sub>>.

NewWithValues does not invoke the **New Instance** method, which means the new instance is not recognized by the File Handler. To be recognized, the new instance must be assigned a name via the method SetName.

### *4.1.4 Creating an Instance with Immediate Messaging*

A new instance can be created and have an immediate message sent to it within one form, using the **\_New** macro.

## Medley LOOPS: The Basic System

Macro:            \_ New  
Arguments:        <class>, the handle of the parent class for the  
                  new instance.  
                  <message>, name of the message to be sent  
                  to the new instance.  
                  <args>, the arguments to be sent with the  
                  message.  
Return:           The new instance handle.

The new instance is created and the message <message> is immediately sent to it. As an example,

```
2/107← (SETQ W1 (←New ($ Window  
                  Open))  
#, ($& Window (FI%0.UC1.0d5.NL5 . 22))
```

which creates an instance of window and sends it the message Open to pop-up a new window.

Then, assign W1 as the name of the new window:

```
2/109← (SEND W1 SetName 'W1)  
#, ($& Window (FI%0.UC1.0d5.NL5 . 22))
```

Then, shape W1 to give it visibility on the desktop:

```
2/115← (← W1 Shape)  
(160 86 408 323)
```

The first thing to do after naming the window is to send a SHAPE message to expand the window something visible. Since there are no

# Medley LOOPS: The Basic System

arguments to Shape, the cursor appears with a ghost image that allows you to locate the window and shape it on the desktop. We drew the shape of the window in the lower left corner of the desktop as shown in Figure 4-1.

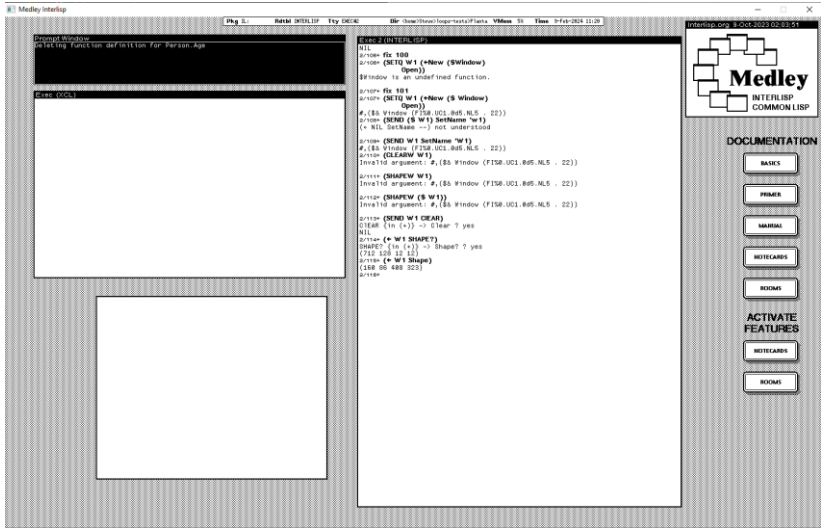


Figure 4-1. Shaping a Window After Creating it.

## 4.2 Data Storage for New Instance

When an instance is created, the value of the variable **NotSetValue**, which is an active value, is assigned to its instance variables. Trying to access an instance variable with this active value triggers the method **IVValueMissing**.

Data was stored in instances on all puts and on **GetValue** methods when the value was an active value, but **NotSetValue**.

## Medley LOOPS: The Basic System

When reading the value of an instance variable that is not stored in the instance, changes in the variable at the class level are seen when the variable is read.

However, if an *initForm* property is specified in a class description, then the value is stored at the time of creation.

Testing for whether a value is stored locally can be performed in two ways:

- Through the user interface in local mode, which returns `#,NotSetValue` for values not locally stored; or
- Via the **GetIVHere** function.

### *4.2.1 IVValueMissing*

The function **IVValueMissing** is triggered by an active value when the value of the variable **NotSetValue** is found in an instance variable. **IVValueMissing** is described as follows:

Function:	IVValueMissing
Arguments:	<varName>, instance variable name. <propName>, a property name for the instance variable <varName>. <typeFlg>, used internally to indicate the type of access. <newValue>, if a Put ration, the value to be stored.
Return:	Depends on behavior.

The behavior varied with the function that invoked it as described in Table 4-1.



**Table 4-1. IVValueMissing Behavior**

Invoking Function	Description
GetValueOnly	Returns the default value of the instance variable stored in the class.
GetValue	Returns the default value of the instance variable stored in the class if it is not an active value. If the value is an active value, then a copy was made of the active value, stored in the instance, and sent the message <b>GetWrappedValue</b> .
PutValueOnly	Stored the new value in the instance.
PutValue	Stored the new value in the instance, unless the default value was an active value. Then, a copy of the active value was made, stored in the instance, and sent the <b>PutWrappedValue</b> message.

#### 4.2.2 NotSetValue

The function **NotSetValue** determined if its argument was equal to the value of the variable `NotSetValue`. It is defined as:

Function:        `NotSetValue`  
 Argument:      `<arg>`, any value.  
 Return:        `NIL` or `T`.

```
2/66+ (NotSetValue)
NIL
2/67+
```

After loading Plantagenet data set, we can see that the variables of Sam are:

## Medley LOOPS: The Basic System

```
2/123← (SETQ Sam (SEND Person NewInstance 'Sam))
New METHOD-FNS definition for Sam.SetTitle.
New METHOD-FNS definition for Sam.GetAge.
#,$C Sam)
2/124← (PP Sam)
VARS definition for Sam:
(RPAQQ Sam #,$ Sam))
INSTANCES definition for Sam:
(DEFCLASS Sam
  (MetaClass Class Edited%:  **COMMENT** )
  (Supers Object)
  (ClassVariables (CitizenOf England doc  **COMMENT** ))
  (InstanceVariables (Father #,NotSetValue doc NIL)
    (Mother #,NotSetValue doc NIL)
    (Sisters #,NotSetValue doc NIL)
    (Brothers #,NotSetValue doc NIL)
    (Spouses #,NotSetValue doc NIL)
    (Gender #,NotSetValue doc NIL)
    (Birthdate #,NotSetValue doc NIL)
    (Deathdate #,NotSetValue doc NIL)
    (Title #,NotSetValue doc NIL)))
```

We can test the value of Sam's Mother as follows:

```
2/125← (NotSetValue (GetValue Sam 'Mother))
T
```

### 4.2.3: *initForm*

The argument **:initForm** is an IV property, which allowed an instance variable to be initialized at the time the instance was created. The **:initForm** and its value are used in the class definition.

Its value was evaluated when the instance was created. The evaluated form's value was stored in the IV when the new instance was created. As an example:

## Medley LOOPS: The Basic System

```
2/55+ (DefineClass 'tetclas)
#, ($C tetclas)
2/58+ (IL:AddCIV ($ testclas) 'dte NIL '[] :initForm| (DATE)))
(← NIL ListAttribute --) not understood
```

### 4.2.4 Changing the Number of IVs in an Instance

An instance can contain more IVs than are defined within its parent class. It is an error to attempt to remove an IV from an instance that was defined in a parent class. In this case, the `IVMissing` method is invoked. LOOPS provides several functions and methods for managing IVs in an instance.

#### 4.2.4.1 Adding an IV

The function **AddIV** added an instance variable to an instance. It is defined as:

Function:	AddIV
Arguments:	<object>, the handle of the instance. <name>, the name of the instance variable to be added. <value>, the value to be assigned to the new IV. <propName>, a property name for the IV, but may be NIL.
Return:	Used for side effects only.

If <propName> is not NIL and <name> already exists, it is added to the IV specified by <name>.

If <name> already exists and <propName> is NIL, the value of the IV is set to <value>.

## Medley LOOPS: The Basic System

If IV <name> does not exist and <propName> is non-NIL, then an IV with <name> is added to the instance and assigned the value NotSetValue. It is given the property <propName> with specified <value>.

If both <name> and <properName> exist, the value was assigned as the value of <propName>.

The reasons for adding an IV to an instance that is not defined in the parent class are 1) to add a descriptor to an instance that is unique to that instance, and 2) to hide certain IVs from casual inspection by users.

Add the IV “Children” to EdwardIII:

```
2/71+ (AddIV EdwardIII 'Children)
NIL
2/72+ EdwardIII
#,$(& Person (UV%Qn3@X1.0.0.ah7 . 13))
2/73+ (PP EdwardIII)
VARS definition for EdwardIII:
(RPAQQ EdwardIII #,$(EdwardIII))
INSTANCES definition for EdwardIII:
(DEFINST Person (EdwardIII (UV%Qn3@X1.0.0.ah7 . 13))
  (Father EdwardII)
  (Mother IsabellaCapet)
  (Spouses PhilippadAvesnes)
  (Gender Male)
  (Birthdate (11 13 1312))
  (Deathdate (6 21 1377))
  (Children NIL))
NIL
```

We see that “Children” has been added to EdwardIII, but not to his parent, EdwardII:

## Medley LOOPS: The Basic System

```
-----
2/74+ (PP EdwardII)
VARS definition for EdwardII:
(RPAQQ EdwardII #,($ EdwardII))
INSTANCES definition for EdwardII:
(DEFINST Person (EdwardII (UV%Qn3@X1.0.0.ah7 . 11))
  (Father EdwardI)
  (Mother EleanorOfCastile)
  (Spouses IsabellaCapet)
  (Gender Male)
  (Birthdate (4 25 1284))
  (Deathdate (9 21 1327)))
NIL
```

Thus, we can specialize individual instances of a class without not affecting all instances of the class.

## Chapter Five

### Metaclass Functions and Methods

A metaclass serves as a template for one to many different classes, which may differ minimally or a lot from each other. The primary metaclass is `Object`.

Classes are described by metaclasses. *Metaclasses* may define methods that are inherited by each subclass (although, we might refer to these as ‘instance classes’, which is not to be confused with instances of a class). Sending a message to a class invokes a method in a metaclass.

One method defined by `MetaClass` is **New** which creates a new instance of the metaclass. Since *ActiveValue* is a metaclass, sending **New** to *ActiveValue* would create a new active value. A particular subclass of a metaclass may override the definition of **New** and specialize it for that class.

A classes’ metaclass is assigned when the class is created.

#### 5.1 Base Metaclasses

LOOPS has three defined base metaclasses as depicted in Figure 5-1. These are described in Table 5-1.

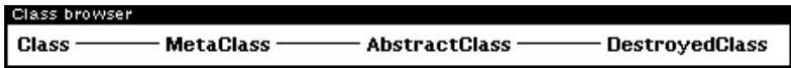


Figure 5-1. Base Metaclasses

Source: LRM 91

**Table 5-1. Base Metaclass Descriptions**

Metaclass	Description
<b>Class</b>	This is the default metaclass for all classes defined within LOOPS. When a class receives the message New, it creates an instance of itself.
<b>AbstractClass</b>	If a class's metaclass is AbstractClass, then it cannot be instantiated and a warning message will be printed in the Exec window. Only subclasses can be created for an AbstractClass.
<b>DestroyedClass</b>	This is a class or metaclass that has been sent the message Destroy or Destroy!. Trying to instantiate a destroyed class causes an error. Attempts to destroy a DestroyedClass have no effect.

### 5.1.1 Abstract Classes

An *abstract class* should be used to define a class which should not have any instances. An abstract class can be used to create a template for a set of classes that have a common set of CVs and methods. The instance of an abstract class has these common CVs and methods, but can then be specialized or extended by adding CVs and methods specific to the use of the class.

Mixins are always used with another class to create a subclass. Instances are created from the new class that has a mixin as one of its parents.

## 5.2 Pseudoclasses

LOOPS provides an interface to Interlisp objects through pseudoclasses. A pseudoclass associates a class with a Lisp data type. When messages are sent to Lisp objects, they are actually passed to the associated pseudoclass. Lisp objects are then considered to be pseudoinstances of the class.

Pseudoclasses provides two mechanisms for handling messages to Lisp objects:

1. A message can be sent to a list whose first element is a class, which is used to lookup the methods; or
2. A message to a Lisp data type.

In the second approach, the function `GetLispClass` is used to locate the class. It searches an internal Lisp table based on the type name of the data type. If no associated class is found, it is assumed to be **Tofu**. If an associated class is found, the data type is considered a pseudoclass.

### *5.2.1 Pseudoclass Functions*

To obtain the Lisp class of a Lisp object, use the function **GetLispClass**, whose format is:

Function:	<code>GetLispClass</code>
Argument:	<code>&lt;object&gt;</code> , a Lisp object.
Result:	The pseudoclass of the data type of the <code>&lt;object&gt;</code> .



## Medley LOOPS: The Basic System

GetLispClass uses the LispClassTable to map type names of Lisp objects to pseudoclasses. LispClassTable is a hash table using EQ hashing to map a name as key to return a pseudoclass, NIL, or a function to be applied.

GetLispClass gets the hash value for the name using (TYPENAME <object>) from an internal hash table. There are three cases:

1. If the hash value is NIL, (\$ Tofu) is returned.
2. If the hash value is not NIL, and it is a pseudoclass, it is returned.
3. Otherwise, the hash value is a function which is applied to <object> and the result is returned.

For example, let us check the Lisp Class of EdwardI:

```
2/16+ (GetLispClass EdwardI)
#,$C Tofu)
```

whereas:

```
2/17+ (GetClass EdwardI)
(+ #,($& Person (FL?cV\U1.0.0.?E8 . 9)) GetClassProp --) not understood
```

Here, EdwardI is a LOOPS class, not a Lisp class.

<<Not sure why GetClassProp doesn't work!!!>>

### 5.3 Metaclass Functions

LOOPS provides multiple functions to create and manage metaclasses.

### *5.3.1 Defining a New Metaclass*

To create a new Metaclass instance, you send the message **New** to **MetaClass**. The format is:

```
(_ ($ MetaClass) New <metaClassName> <supers>)
```

This statement will instantiate a new metaclass with **MetaClass** as its metaclass. <metaClassName> must be a symbolic name.

If <supers> is not specified, the default will be (**Class**); otherwise, <supers> must evaluate to a list of classes.

The result is the name of the new metaclass.

## Chapter Six

### Sending Messages Alternatives

Objects in LOOPS communicate with each other by sending messages. In Chapter 3, we saw the basic forms for sending messages to objects. LOOPS provided some advanced forms for sending messages to objects, which will be discussed in this chapter.

**Note:** In the following sections, the term `<args>` stands for a sequence of arguments, usually represented as `<arg1> ... <argN>`.

**Note:** In the following sections, Times New Roman does not have a character representation for `←` which is equivalent to `SEND` for message sending. We will use “\_” as an alternate indicator for `←` in this text.

#### 6.1 Sending A Message to a LOOPS Object

As a recap, the syntax for sending a message to a LOOPS object appears as:

```
2/78← (← ($ Class) New 'X)  
#,$C X)
```

`←` is implemented as a macro in LOOPS with the definition:

## Medley LOOPS: The Basic System

```
(DEFMACRO _____  
_ (self selector &REST args)  
  `(!_ ,self ',selector ,@args))
```

where the boxes are style indicators (using Notepad++) to view the source code. As we see, “self” is inserted as the reference to the LOOPS object.

The equivalent form for SNED, which is also a macro, is:

```
(DEFMACRO _____  
SEND (self selector &REST args)  
  `(_ ,self ,selector ,@args))
```

We can see that these are equivalent definitions, but for two macro expansions.

The definition for “\_!” is:

```
(DEFMACRO _____  
_! (self selector &REST args)  
  [Once-Only (self)  
    `(APPLY* (FetchMethodOrHelp ,self ,selector)  
              ,self  
              ,@args)])
```

FetchMethodOrHelp searches up the supers chain for the selector/message name, and then calls FetchMethod.

This is offered by way of showing how a method is fetched from class or up the class hierarchy.

## 6.2 \_!

The form `_!` Sends a message to an object self after it has evaluated all arguments. It is defined as:

Function: `_!`  
Arguments: `<object>`, a handle for an object.  
`<methodName>`, a method, which is not evaluated.  
`<args>`, a sequence of arguments for the method  
Return: The value returned based on arguments.

Let us create an instance of a city with name Frederick:

```
2/61+ (←! City 'SetName 'Frederick)  
#,$C Frederick)
```

```
2/62+ (PP Frederick)  
CLASSES definition for Frederick:  
(DEFCLASSES Frederick)  
(DEFCLASS Frederick  
  (MetaClass Class Edited%: **COMMENT** )  
  (Supers State County)  
  (InstanceVariables (Description "A component of a State/County of the  
                        United States")  
                      (PartOf NIL)  
                      (Population NIL)))  
NIL
```

## 6.3 \_IV

The form `_IV` invokes the method stored in an instance of a class specified by `<object>`. It is defined as:

## Medley LOOPS: The Basic System

Function: `_IV`  
Arguments: `<object>`, a handle for an object.  
`<IVName>`, an instance variable name which is not evaluated.  
`<args>`, a sequence of arguments to the function.  
Return: The value of the function; otherwise, breaks.

`_IV` calls a method, `IVFunction`, to determine if the method is accessible by the instance. If not, it returns the message "No iv function".

### 6.4 `_Try`

The form `_Try` invokes the method in self if it exists. It is defined as:

Function: `_Try`  
Arguments: `<object>`, a handle for an object.  
`<methodName>`, the name of a method to try.  
`<args>`, a sequence of arguments for the method.  
Return: The value computed by the method, if it exists;  
Otherwise, NIL.

Normally, when a message is sent to an object to invoke a method, if the method does not exist in the object or its parent class, the system breaks. This function avoids catching this 2break, and just returns NIL. As an example:

```
2/64+ (SETQ ANNAPOLIS (←Try City 'New 'Annapolis))
NotSent
```

The New message is not sent to City because it is defined in a superclass, not the object City.

## 6.5 \_Super

**\_Super** invokes a method in the superclass of an instance by searching up the class hierarchy for the first occurrence of the method. It is defined as:

Function:     \_Super  
Arguments:    <object>, the handle of an object.  
              <methodName>, the name of the method to  
              invoke.  
              <args>< a sequence of arguments to the method.  
Return:       ??

As examples:

```
2/59+ (←Super EdwardIII 'GetMother)
Selector to ←Super does not match method selector
(QUOTE GetMother)
```

GetMother is not a method defined in either EdwardIII or its superclass, Person.

**\_Super** cannot be called directly. Rather, it must be embedded in another method in a class to invoke a method in a parent class of the

## Medley LOOPS: The Basic System

class of the instance. To do so, it searches up the class hierarchy to locate the definition of the method.

If no arguments were provided, it used the arguments of the method from which it was called.

If `<methodName>` does not exist in a superclass, then a break was initiated.

### *6.5.1 `_Super?`*

A variant, `_Super?`, uses the single most next general method. It does not break if there is no occurrence of `<methodName>` in the superclass. As with `_Super`, it must appear in the body of a method.

### *6.5.2 `_SuperFringe`*

Another variant, **`_SuperFringe`**, invokes `<methodName>` from each of the classes on the super's list of the class. That is, if the class is inheriting from multiple classes, the `<methodName>` is invoked, if it occurs, in each of those superclasses in which it is defined.

## **6.6 `_New`**

This form `_New` creates an instance of a class, then sends a message with arguments to that instance. It is defined as:



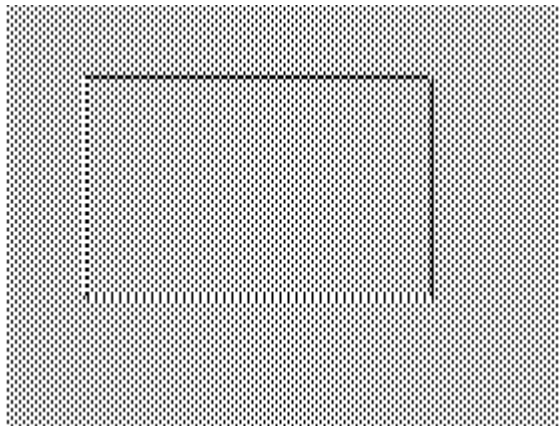
## Medley LOOPS: The Basic System

Function: `_New`  
Arguments: `<class>`, the class for which an instance is to be created.  
`<methodName>`, the method to be invoked, which is not evaluated.  
`<args>`, a sequence of arguments to the method.  
Return: A handle for the new instance.

We can create a new window, then shape it using the following statement:

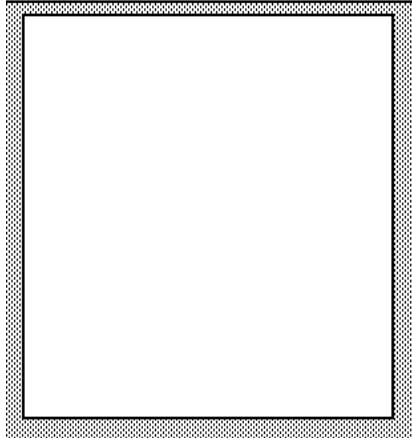
```
2/78+ (SETQ W1 (←New ($ Window) Shape))  
|#, ($& Window (FM%0.UC1.0d5.CR8 . 15))
```

Since no arguments are provided to `Shape`, the user is prompted with a ghost of a rectangular window to draw the window. Which yields a ghost image as:



## Medley LOOPS: The Basic System

The cursor is located at the lower right corner of the ghost image. Dragging it defines the coordinates for the window.



### 6.7 FetchMethod

**FetchMethod** fetches the function name of the message that is sent to the class. The function can be found in the class or its superclasses. It is defined as:

Method:        FetchMethod  
Arguments:    <class>, the handle of the class to which the  
                  message is sent.  
                  <methodName>, the method, which is evaluated.  
Return:        The function name; otherwise, NIL.

If the <methodName> is not found in the class or any of its supers, FetchMethod returns NIL.

## Medley LOOPS: The Basic System

A common mistake is to specify an instance of a class as seen below:

```
2/72← (FetchMethod Maryland 'States?)
ARG NOT class
#,$& State (|NY0.UC1.0d5.Ca:| . 16))

2/73← (PP Maryland)
VARS definition for Maryland:

(RPAQQ Maryland #,($ Maryland))
INSTANCES definition for Maryland:
(DEFINST State (Maryland (|NY0.UC1.0d5.Ca:| . 16))
  (Description "An Eastern State of the United States")
  (StateCapital Annapolis))
```

Note, however, what is returned is the class of Maryland, which is State.

## Chapter Seven

### Introduction to

### Data-Oriented Programming

In *object-oriented programming*, when a message is sent to an object, the method implementing the message may change the state of variables within the object.

In *data-oriented programming* or *access-oriented programming* (AOP), an action performed on data may be triggered as a side effect when the data is accessed. These data structures are called *active values* because the act of reading or writing a data item invokes some action upon the value of the data item or upon other data items. The action is a specified implicit procedure when the value of the variable is read or set. As Bobrow and Stefik (1986) noted, this mechanism is the dual of messages, which tell objects to perform operations that can change the value of variables as a side effect.

In AOP, for any variable of an object, a procedure can be specified that is invoked when the variable is accessed for reading or writing. In LOOPS, this structure was called an active value, because the act of reading or writing invoked (e.g., caused to be executed) the associated procedure. As the LRM notes, this mechanism was dual to the concept of sending a message to perform an operation, which could change the value of a variable as a side effect.

One aspect of AOP is to provide for “hidden variables” e.g., variables not directly accessible by the primary methods associated

## Medley LOOPS: The Basic System

with messages, but through an indirect action, e.g., the effect of the active value. Such hidden variables could be changed only through the action upon another, perhaps “public” variable.

As Bobrow and Stefik (1986) noted, an active value can serve as the “glue” which connects to subsystems together, but preserves their independence in terms of programming and functionality. It can also be used to allow one process to monitor another and to maintain constraints among data in a system.

Active values enable one process to monitor the behavior of another process. Figure 7-1 presents the abstract class Activevalue and its specializations.

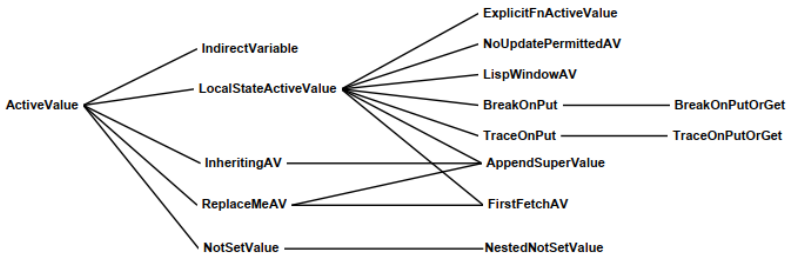


Figure 7-1. ActiveValue Specializations

Source: LRM 1991

**Note: ActiveValues are widely used in the Truckin’ game, which demonstrates many of the features of LOOPS.**

## 7.1 Specifying an Active Value

In LOOPS, an active value is specified as follows:

```
#(<localState> <getFn> <putFn>)
```

where <localState> is the variable that stores the data

<getFn> is the name of a function invoked when a program accesses the active value, and

<putFn> is the name of a function invoked when a program sets the active value.

This notation is converted by a read macro to the Interlisp data type **activeValue**. So, we can picture an activeValue in this way:

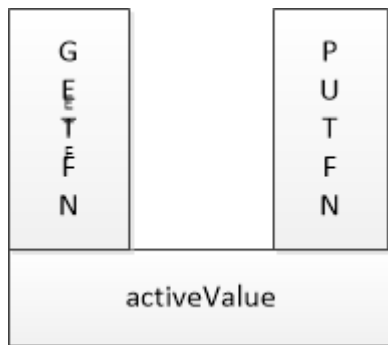


Figure 7-2. activeValue Structure

## Medley LOOPS: The Basic System

### *7.1.1 getFn and putFn*

The functions are defined with standard arguments as shown below. An active value need not specify both a <getFn> and a <putFn> functions, although typically both are specified.

The default values for the functions operate as follows:

- If <getFn> is NIL, the current value of the <localState> is returned.
- If <getFn> is non-NIL, then the function is applied to the value of the variable before it is returned to the calling function.
- If <putFn> is NIL, the value of <localState> is replaced with the argument provided.
- If <putFn> is non-NIL, then it is invoked with the new value before that value was stored at the variable.

Behind the scenes, when a message is sent to an active value, either **GettingWrappedValue** on reads or **PuttingWrappedValue** if a write is sent to the active value object with the originating message. The originating message may or may not trigger side effects as a result of receiving that message.

### *7.1.2 Defining an Active Value*

To define an active value, the following steps are suggested.

1. Start with the definition of a Thermometer as shown below:

## Medley LOOPS: The Basic System

```
2/50+ (LOAD 'Thermometer.txt)
{DSK}<home>Steve>loops-tests>Loopstests>Thermometer.txt;1
VARS definition for Thermometer:
(RPAQQ Thermometer #,($ Thermometer))
CLASSES definition for Thermometer:
(DEFCLASSES Thermometer)
(DEFCLASS Thermometer
 (MetaClass Class Edited:  **COMMENT** )
 (Supers Object))
VARS definition for T1:
(RPAQQ T1 #,($ T1))
INSTANCES definition for T1:
(DEFINST Thermometer (T1 (FP%0.UG1.0d5.fs8 . 2))
 )
{DSK}<home>Steve>loops-tests>Loopstests>Thermometer.txt;1
.....
```

Thermometer is a class that can define many different instances of thermometers. One instance is named T1. The code to define a basic thermometer is below.

```
(* ; "Thermometer Example")
```

```
(SETQ Thermometer (DefineClass 'Thermometer NIL ($ Class)))
```

```
(SEND ($ Thermometer) SetName 'Thermometer)
```

```
(PP Thermometer)
```

```
(* ; "Define an instance of a thermometer")
```

```
(SETQ T1 (SEND ($ Thermometer) New 'T1))
```

```
(SEND ($ T1) SetName 'T1)
```

```
(PP T1)
```



## Medley LOOPS: The Basic System

To measure the temperature, we need to define a `LocalActiveStateValue` variable which will be defined as an active value.

The template for the `LocalActiveStateValue` is:

```
#(<localactivestatevalue <getFn> <PutFn>)
```

The definition for this is:

```
2/51← (PutCIVHere Thermometer # (Temperature getTemperature  
putTemperature))  
NIL
```

which appears as:

```
2/53← (pp Thermometer)  
pp {in EVAL} -> PP ? yes  
VARS definition for Thermometer:  
  
(RPAQQ Thermometer #,($ Thermometer))  
CLASSES definition for Thermometer:  
  
(DEFCLASSES Thermometer)  
(DEFCLASS Thermometer  
  (MetaClass Class Edited%: **COMMENT** )  
  (Supers Object)  
  (InstanceVariables (#<ARRAY T (3) @ 142,2140> NIL doc **COMMENT** )  
  )  
)  
NIL
```

To get the value registered by the `Thermometer`, we can do:

```
2/54← (GetValue Thermometer 'Temperature)  
#,NotSetValue
```

Because a value has not been set to `Temperature`.

## Medley LOOPS: The Basic System

### *7.1.3 Nested Active Values*

Active values could be nested to allow multiple access functions to be applied to variable values. As noted in Bobrow and Stefik (1983), one might want to have two processes - a debugging process and a display process – to monitor the state of some variable.

Nested active values store the innermost active value as the `localState` of the outermost active value. For example, we could have:

```
#(_      #(XPos NIL UpdateDisplay)
  GettingTracedVar
  SettingTracedVar)
```

The `putFns` are invoked from the outmost AV to the innermost AV. So, `SettingTracedVar` would be called with the new value for `XPos`. But, it would call the function `PutLocalState` to set its own `localState` which it the innermost AV. The innermost AV `putFn` – `UpdateDisplay` – is called with the new value to update the display and set the value of `XPos`.

Nested get operations worked in the reverse order with the innermost AV being called and the progressing outwardly. The returned value is the value provided by the outermost `getFn`.

### *7.1.4 Using Active Values*

Active values are a powerful mechanism, although simple in implementation, that allow the programmer control over the getting and setting of instance variable values and property values.

Consider an instance variable `IV` in an instance `I` of a class `C`. The default value of `IV` has been declared to be the active value `AV`. Assume that `AV` has never been set. The first time a `(PutValue I IV <expr>)` is invoked, a copy of `AV` is made and inserted into `I` as the value of `IV`. The `putFn` is invoked with

## Medley LOOPS: The Basic System

the copy of AV, which provides a place where the localState of the AV can be stored private to I.

The following example is taken from Bobrow and Stefik (1983). Define a class SUM with three IVs: top, bottom, and sum. IVs top and bottom are initialized to zero. IV sum will be computed when asked for. The LOOPS code for SUM is:

```
(DEFCLASS SUM
  (Metaclass <class>)
  (Supers Object)
  (InstanceVariables
    (top 0)
    (bottom 0)
    (sum   #(Shared ComputeSum NoUpdatePermitted))
  )
  (ClassVariables)
  (Methods
    (printOn printColumn)
  )
)
```

*Note: I have modified the indenting to make the code more readable.*

The structure of the AV is specified as follows:

- This IV is an instance of some <class>;
- The <class< is an instance of Object;
- It has three instance variables, which are enumerated;
- There are no class variables specified for <class>, since we did not show its declaration; and
- It has two methods, presumably inherited from <class>.

The programmer would define ComputeSum to calculate the sum.

## Medley LOOPS: The Basic System

**NoUpdatePermitted** is a LOOPS kernel function which does not allow the AV to update the sum IV. Since no updating of the IV is allowed, Shared allows this AV to be shared with other IVs, rather than being copied anew for each IV.

**NOTE: SEdit does not make copies of active values. If AVs are copied in SEdit, they will share structure, which means that one AV is modified, all AVs of that type will be modified.**

### 7.2 Active Value Functions

Several functions are provided to manipulate active values, which are described in this section.

#### *7.2.1 FirstFetch*

**FirstFetch** is a standard getFn that expects the AV's localState to be an expression to be evaluated. The first time that a Get function is performed on the AV, the expression is evaluated and it becomes the initial value of the AV.

As an example, consider the following class declaration:

```
(DEFCLASS TestDatum
  (MetaClass Class)
  ( ... )
  (InstanceVariables
    (sample #((RAND 0.0 100.0) FirstFetch <putFn>))
  )
)
```

## Medley LOOPS: The Basic System

When a Get function is executed for the AV, the expression (RAND 0.0 100.0) is executed to generate a random number between 0.0 and 100.0. This value replaces the expression in the AV.

Let us define Germany with Capital City Munich and an active value using FirstFetch:

```
(SETQ Germany (DefineClass 'Germany '(Country) ($ Class)))
(SEND ($ Germany) SetName 'Germany)
(PutCIVHere ($ Germany)
             'CapitalCity
             #(Munich FirstFetch 'GetNewCity)
             NIL)
(PP Germany)
```

```

(DEFCLASSES Germany)
(DEFCLASS Germany
  (MetaClass Class Edited%: **COMMENT** )
  (Supers Country)
  (InstanceVariables (CapitalCity #<ARRAY T (3) @ 134,54250>)))
NIL
```

Now, trying to fetch CapitalCity

```
2/84+ (GetIt Germany 'CapitalCity NIL 'IV)
#<ARRAY T (3) @ 134,54250>
```

## Medley LOOPS: The Basic System

The definition of `GetNewCity` is:

```
(* ; "GetNewCity")
(DEFINEQ (GetNewCity
  (LAMBDA NIL
    (PRINT "Enter new city name:")
    (SETQ NewCity (RATOMS T))
    (PRINT NewCity)
    (RETURN NewCity)
  )
)
```

### 7.2.1.1 FirstFetchAV

**FirstFetchAV** is a specialization of **localStateActiveValue** and the **ReplaceMeAV**, which has an expression as the value of the **localstate**. On the first put access, the expression was evaluated. The resulting value replaced the **FirstFetchAV** so that the value of the variable was no longer an active value. This AV is often used as the default value of the variable which allows the actual value to be replaced at run time.

### *7.2.2 Indirect Access*

LOOPS provides a mechanism for accessing a value stored in another IV through an IV using one of the indirect functions: **GetIndirect** and **PutIndirect**.

## Medley LOOPS: The Basic System

Function:       GetIndirect  
                  PutIndirect  
Arguments:      A localState which is a nested AV specifying  
                  an IV in that AV.  
Return:         The value of executing the function.

<<Example>>

### *7.2.3 ReplaceMe*

In some cases, a programmer will want to set a default value for a variable in a class using an AV, but replace it by a value provided when the program sets the value of the variable. **ReplaceMe** accomplishes this:

Function:        ReplaceMe  
Arguments:  
Return:         The value generated to replace the AV.

Consider the following example:

```
 #(NIL ComputeValue ReplaceMe)
```

NIL is the default value for the localState of the AV. When a Get function is given the variable, it returns the value computed by ComputeValue. When a Put function is given the variable, the value provided is set as the value of the active variable.

## Medley LOOPS: The Basic System

### 7.2.4 User-Defined Function

The *getFn* and *putFn* functions associated with an active value can be defined by the user. They take a standard set of arguments:

Function:	getFn putFn
Arguments:	<object>, the object containing the active value . <varName>, the name of a variable containing the active value; NIL if not stored in a variable. <oldOrNewValue>, for a getFn, this is the AV's localState; for a putFn, the new value to be stored in the AV. <propName>, the name of a property associated with the AV; if NIL, the is associated with <varName>. <activeVal>, the AV in which this getFn was found. <type>, where the AV is stored: NIL for an instance variable; CV for a class variable; CLASS for a class property; or METHOD for a method property.
Return:	For getFn, the value returned by the Get operation. For putFn, it may make changes to the local state using the function PutLocalState.

When **PutLocalState** is used, it may trigger any embedded active values.



## Medley LOOPS: The Basic System

### 7.2.5 Local State Functions

Two functions can be used to retrieve or update the `localState` of an active value: **GetLocalState** and **PutLocalState**.

Function	GetLocalState PutLocalState
Arguments:	<activeValue>, the handle for the active value. [<newValue>, <i>for putFn only</i> , the new value to be stored.] <object>, the object containing the active value . <varName>, the name of a variable containing the active value; NIL if not stored in a variable. <propName>, the name of a property associated with the AV; if NIL, the is associated with <varName>. <type>, where the AV is stored: NIL for an instance variable; CV for a class variable; CLASS for a class property; or METHOD for a method property.
Return:	see below.

For **GetLocalState**, it returns the value of the `localState` of the <activeValue>.

For **PutLocalState**, it stores the <newValue> in the `localState` of the <activeValue> and returns <newValue>.

If the `localState` of an AV is itself an Av, then its `getFn` will be triggered to return the value of the embedded AV. For a `putFn`, an embedded Av will only be triggered when `PutLocalState` is invoked.

### 7.2.5.1 Alternate Functions

Alternative functions **GetLocalStateOnly** and **PutLocalStateOnly** will only retrieve or store a new value in the `localState` of the referenced active value. Their format is:

Function: `GetLocalStateOnly`

Arguments:

Function: `PutLocalStateOnly`

Arguments:

### 7.2.5.2 Using LocalState

Most `ActiveValue` subclasses are specializations of `LocalStateActiveValue`, which used an instance variable, `localState`, in the `ActiveValue` to hold the value.

### *7.2.6 Annotated Values*

Interlisp uses a special data type called an *annotatedValue* data type, to wrap each instance of an active value, when it is installed in an object. The `annotatedValue` contains the **activeValue** instance. Thus, **GetValue** and **PutValue** can use Interlisp's type checking mechanism to see if the value in instance variable should be processed normally or via the active value mechanism. This mechanism is transparent to application programs.

In case the user forgets about the distinction between a `annotatedValue` and an **activeValue**, Interlisp has a class,

## Medley LOOPS: The Basic System

annotatedValue, to mediate when the user program attempts to treat an annotatedValue as an activeValue.

Each annotatedValue contains a field named annotatedValue. This field contained an ActiveValue object.

### 7.2.6.1 AnnotatedValue

The **AnnotatedValue** class was equivalent to the Lisp data type annotatedValue. It is an abstract class that cannot be instantiated. Its superclass is the LOOPS class tofu. Instances of this class are Lisp data type instances.

LOOPS provides several macros for explicitly controlling annotatedValues as presented in the following sections.

### 7.2.6.2 fetch

The **fetch** macros retrieve the contents of the annotatedValue field of an annotatedValue instance.

Macro:            fetch  
Arguments:        <value>, an annotatedValue instance.  
Return:            The contents of the field annotatedValue.

It is coded as:

(fetch **annotatedValue** of <value>).

## Medley LOOPS: The Basic System

### 7.2.6.3 replace

The **replace** macro replaced the contents of the annotatedValue field of an **annotatedValue** instance.

Macro:            replace  
Arguments:        <value>, an annotatedValue instance.  
Return:            The contents of the field annotatedValue.

It is coded as:

(replace **annotatedValue** of <value> with <object>).

### 7.2.6.4 create

The **create** macro created a new instance of the data type annotatedValue.

Macro:            create  
Arguments:        <value>, an annotatedValue instance.  
                  <object>, an ActiveValue object to be stored in the field.  
                  No type checking is done on the instance.  
Returns:           The contents of the field annotatedValue.

It was coded as:

(create **annotatedValue** \_ <object>).

### 7.2.6.5 type?

The **type?** Macro performed a type check for an instance of the Lisp data type annotatedValue.

## Medley LOOPS: The Basic System

Macro:           type?  
Arguments:       <value>, the value to check as to type..  
Return:           T, if value is an instance of the data type  
                  annotatedValue; otherwise NIL.

It was coded as:  
(type? **annotatedValue** <value>).

### 7.2.6.6 **\_AV**

The **\_AV** macro sent a message to the `ActiveValue` object wrapped in an `annotatedValue`.

Macro:            **\_AV**  
Arguments:        <av>, an instance of an `annotatedValue`.  
                  < method>, a method of the enclosed  
                  **ActiveValue**.  
                  <args>, arguments to be passed to the method.  
Returns:           The result of executing the method using  
                  the arguments.

It is coded as:  
(**\_AV** <av> <method> . <args>).

### 7.2.6.7 **MessageNotUnderstood**

This **MessageNotUnderstood** macro forwarded a message intended for the wrapped `ActiveValue` to that object.

## Medley LOOPS: The Basic System

Macro:            MessageNotUnderstood  
Arguments:        <object>, the object containing the ActiveValue  
                    instance.

It is coded as:

(\_ <object> MessageNotUnderstood).

### *7.2.7 Managing Annotated Values*

LOOPS provides several methods for managing active values. These methods are defined in the class `ActiveValue` and inherited by every instance of `ActiveValue`.

#### **7.2.7.1 Printing ActiveValue Instances**

The method **AVPrintSource** prints a description of an `ActiveValue` instance. Its format is:

Method:            AVPrintSource  
Arguments:        <object>, an instance of **ActiveValue**.  
                    <classname>< the name of the class of the  
ActiveValue.  
                    <avNames>, a list of names of self, the last being  
                    the unique identifier (UID) of self.  
Return:            A form to be used by DEFPRINT, where the  
                    form has the format (form1 . form2).

## Medley LOOPS: The Basic System

The LOOPS Manual notes that the default form is something like:

```
("#, " $AV <classname>
  <avnames   (ivname value proptime value ...)
              (ivname value proptime value ...)
              ....
)
```

The list (ivname value proptime value ...) describes the current state of the instance variable of the `ActiveValue` instance.

Including the UID in the print form allows the identity of the **ActiveValue** instance to be recovered. This allows different annotatedValues to share the same **ActiveValue**, and to allow this sharing to be preserved across saving and reloading into a Lisp environment. An example from the LOOPS Manual:

```
#, ($AV IndirectVariable
   (HeightFromWidth (NCV0.OX: .SD7 .KR . 8))
   (object #. ($ SquareWindow))
   (varname width)
   (proptime NIL)
   (type IV)
)
```

Note: formatting added.

### 7.2.7.2 \$AV

**\$AV** is used to reconstruct an annotatedValue that was saved to a file. Its format is:

Function:     **\$AV**  
Arguments:    <classname>, the name of the class of **ActiveValue**.  
              <avNames>, a list of LOOPS names of **ActiveValue**  
              instances.  
              <ivForms>, a list describing the state of the instance  
              variables of the **ActiveValue**.  
Return:       a new annotatedValue whose **ActiveValue** is  
              reconstructed from the second and third arguments  
              above.

One could construct an active value by typing a form such as:

```
($AV     <activeValueClassName>  
      NIL  
      (<ivname> <value> <propName> <value>)  
      (<ivname> <value> <propName> <value>)  
      ....  
)
```

None of the arguments were evaluated because **\$AV** was an NLambda, Nospread function.

Alternatively, a user could also use the functions **PutClassIV**, **PutClassValue**, **PutClassValueOnly**, **AddCIV**, **AddcCV**, or other methods to modify or add class and instance variables.



## 7.3 The ActiveValue Class

The **ActiveValue** class defines the protocol for interaction followed by all active values. The basic functionality of activeValues was defined here and inherited by each of its subclasses. **ActiveValue** itself is an abstract class, which is a placeholder in the class hierarchy, and cannot be instantiated.

### *7.3.1 Using Active Values*

LOOPS specified several guidelines for using active values:

- Before you start coding, decide what functionality you want the active value to provide:
  - Will it cause a side effect?
  - Will it maintain/enforce constraints between two pieces of data?
  - Will it transform the value provided in the access to some internal form?
  - Will it transform an internal value to an external form to be deliver to the calling object?
  - Will its contents need to be initialized?
- Determine which activeValue subclass that you want to specialize, if necessary, to achieve the functionality specified above.
- Create an instance of the activeValue subclass you have chosen or specialized.
- Initialize the contents of the activeValue subclass instance, if necessary.
- Install the active value on the data that you want to become active using **AddActiveValue**.

7.3.2 Specializing an Active Value

To use active values, you need to make instances of some subclass of **activeValue** or create your own specializations in order to create instance. Figure 7-1 shows activeValue and its specializations.

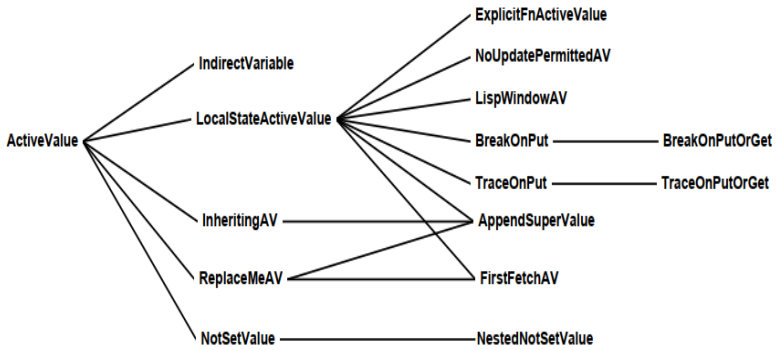


Figure 7-1. **ActiveValue** and its specializations

The following sections describe the specializations of **ActiveValue**.

### 7.3.2.1 IndirectVariable

This specialization acted as indirect addressing by returning the value of another variable as its value. Consider a variable that is directly addressable by a Get or Put method, while you want to hide the actual variable from the view of the caller. You can use this subclass to create instances that perform the actual Get or Put on the variable referenced by this AV, called the *tracked variable*. Any transformation methods would be associated with the AV, which allow the tracked variable to have a canonical format. Then, different instances of IndirectVariable could perform different transformations on it depending how the result would be used. Table 7-1 describes its instance variables.

**Table 7-1. IndirectVariable Instance Variables**

Instance Variable	Description
object	An instance of a class containing varName.
varName	The name of the tracked variable which is referenced by the AV.
propName	If non-NIL, a property associated with the IV.
Type	Type of variable being referenced. Valid values are CV, IV, or NIL. Default is an IV.

A specialization of **IndirectVariable** is created so as not to establish equality between the two variables. Thus, you also need to specialize **GetWrappedValue** and **PutWrappedValue**.

The definition for GetWrappedValue is:

"Fetch the value wrapped in the active value without triggering any side-effects."

(SELECTQ (@ type)

## Medley LOOPS: The Basic System

```
((NIL IV)
  (GetValueOnly (@ object)
                (@ varName)
                (@ propName)
  )
)
(CV
  (GetClassValueOnly (@ object)
                     (@ varName)
                     (@ propName)
  )
)
(HELPCHECK "Invalid type" (@ type))
)
```

The definition for Put WrappedValue is:

"Replace the value wrapped in the active value without triggering any side-effects."

```
(SELECTQ (@ type)
  ((NIL IV)
    (PutValueOnly (@ object)
                  (@ varName)
                  newValue
                  (@ propName)
    )
  )
)
(CV
  (PutClassValueOnly (@ object)
                     (@ varName)
                     newValue
                     (@ propName)
  )
)
```

## Medley LOOPS: The Basic System

```
)  
)  
  (HELPCHECK "Invalid type" (@ type))  
)
```

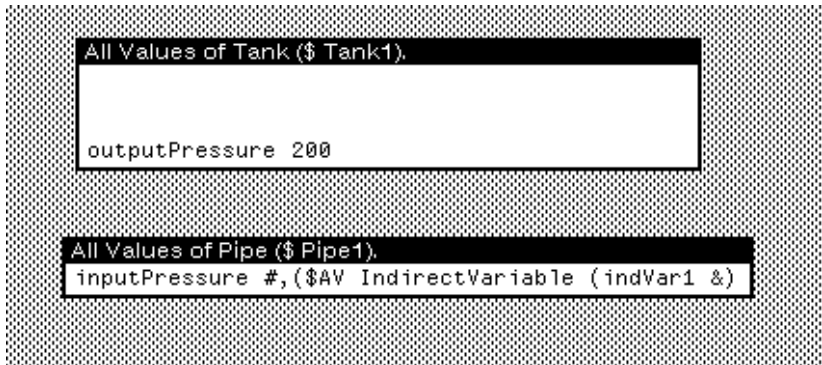
The function **\_Supers** ensured that the default behavior of **IndirectVariable** is used to retrieve or store the data in the tracked variable. TestAV.txt is an adaptation of a test sequence in the LRM. The source code is in Appendix C.1. Here is the result of loading TestAV.txt.

## Medley LOOPS: The Basic System

```
Exec 2 (INTERLISP)
(IL:LOAD 'TestAV.txt)
{DSK}<home>Steve>loops-tests>LoopsTests>TestAV.txt;1
EXPRESSIONS definition for ($ Tank):
($ Tank)
EXPRESSIONS definition for ($ Pipe):
($ Pipe)
VARS definition for Tank:
(RPAQQ Tank NIL)
CLASSES definition for Tank:
(DEFCLASSES Tank)
(DEFCLASS Tank
  (MetaClass Class Edited:  **COMMENT** )
  (Supers Object)
  (InstanceVariables (outputPressure NIL doc  **COMMENT** )))
(SEND ($ Pipe) AddIV 'inputPressure)"
VARS definition for Pipe:
(RPAQQ Pipe NIL)
CLASSES definition for Pipe:
(DEFCLASSES Pipe)
(DEFCLASS Pipe
  (MetaClass Class Edited:  **COMMENT** )
  (Supers Object)
  (InstanceVariables (inputPressure NIL doc  **COMMENT** )))
EXPRESSIONS definition for ($ Tank1):
($ Tank1)
EXPRESSIONS definition for ($ Pipe1):
($ Pipe1)
"(+ ($ IndirectVariable) New 'indVar1)"
EXPRESSIONS definition for ($ indVar1):
($ indVar1)
"Assign object and varName"
"(+@ ($ indVar1) object ($ Tank1))"
"(+@ ($ indVar1) varName 'outputPressure)"
EXPRESSIONS definition for ($ indVar1):
($ indVar1)
"Installing ActValue instance."
"(+ ($ indVar1) AddActiveValue ($ Pipe1) 'inputPressure)"
EXPRESSIONS definition for ($ indVar1):
($ indVar1)
"inputPressure {in (PutValue Pipe1 (QUOTE 'inputPressure) 100)} ->
inputPressure ? yes
'outputPressure {in (PutValue Tank1 (QUOTE 'outputPressure) 200)} ->
outputPressure ? yes
*** End of TestAV ***"
{DSK}<home>Steve>loops-tests>LoopsTests>TestAV.txt;1
2/83*^
```

## Medley LOOPS: The Basic System

At the end of the test file, we inspect the two entities: Tank1 and Pipe1.



Here is another example of using IndirectVariable.

```
2/77+ (LOAD 'NewTestAv.txt)
{DSK}<home>Steve>loops-tests>LOOPSTests>NewTestAV.txt;1
VARS definition for 3FeetAbove:

(RPAQQ 3FeetAbove NIL)
CLASSES definition for 3FeetAbove:

(DEFCLASSES 3FeetAbove)
(DEFCLASS 3FeetAbove
  (MetaClass Class Edited:  **COMMENT** )
  (Supers IndirectVariable))

The height of Bin1 is 0
The height of Bin1 is 0
Setting heights of Bin1 and Conveyor1.
The height of Conveyor1 is 15
Rset the height of Conveyor1
The height of Conveyor1 is 21
The height of Bin1 is 21
{DSK}<home>Steve>loops-tests>LOOPSTests>NewTestAV.txt;1
2/78+
```

And, the variables of 3FeetAbove are:

```
All Values of 3FeetAbove ($ 3fa1).
object #,($ Bin1)
varName height
propName NIL
type NIL
```

The code for NewTestAv is found in Appendix C.2.

### 7.3.2.2 LocalStateActiveValue

**LocalStateActiveValue** contains the instance variable **localState**, which is used to store the value of the tracked variable. It is useful when you need an activeValue that produces a specific side effect in your application.

Thus, you also need to specialize **GetWrappedValue** and **PutWrappedValue**. Table 7-2 describes its instance variables.

**Table 7-2. LocalStateActiveValue Instance Variables**

Instance Variable	Description
<b>localState</b>	A variable that holds the actual value of the variable which is wrapped as an active value.

Applying **GetWrappedValueOnly** to an instance of **LocalStateActiveValue** results in calling `(GetValueOnly self (QUOTE localState))`. Similarly, applying **PutWrappedValueOnly** results in calling `(PutValueOnly self (QUOTE localState) (if (NotSetValue newValue) then NestedNotSetValue else newValue))`.

When a **LocalStateActiveValue** was used as the default value for an instance variable in a class, it must be copied to each instance of the class; otherwise, every instance of the class would share a single



## Medley LOOPS: The Basic System

**localState.** This copying is done automatically by LOOPS at the first instance of accessing the instance variable.

Once the copying is completed, every instance of the class has its own version of **localstate**. The copying operation was performed by the method **CopyActiveValue**.

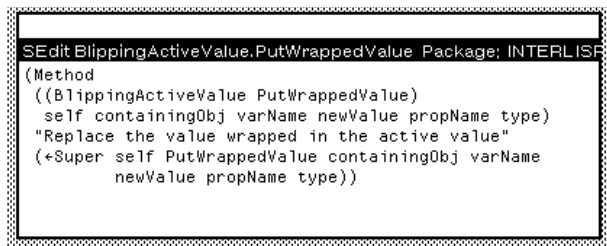
Method: CopyActiveValue  
Arguments: <object>, the ActiveValue instance.  
<annotatedValue>, the value to be copied.  
Return: A new annotatedValue wrapped round a copy of the ActiveValue <object>.

Try this example from the LRM 1991.

```
2/99+ (DefineClass 'BlippingActiveValue '(LocalStateActiveValue))
#,$($C BlippingActiveValue)
```

```
2/101+ (+ ($ BlippingActiveValue)
SpecializeMethod
'PutWrappedValue)
```

which pops up a SEdit window:



```
SEdit BlippingActiveValue.PutWrappedValue Package: INTERLISP
(Method
((BlippingActiveValue PutWrappedValue)
 self containingObj varName newValue propName type)
"Replace the value wrapped in the active value"
(+Super self PutWrappedValue containingObj varName
newValue propName type))
```

Closing the SEdit window yields:

## Medley LOOPS: The Basic System

```
2/101← (← ($ BlippingActiveValue)
SpecializeMethod
'PutWrappedValue)
BlippingActiveValue.PutWrappedValue
```

Similarly for GetWrappedValue:

```
2/102← (← ($ BlippingActiveValue) SpecializeMethod 'GetWrappedValue)
```

which also pops up a SEdit window:

```
SEdit BlippingActiveValue.GetWrappedValue Package: INTERLISP
(Method
((BlippingActiveValue GetWrappedValue)
 self containingObj varName propName type)
"Fetch the value wrapped in the active value"
(←Super self GetWrappedValue containingObj varName
propName type))
```

Closing the SEdit window yields:

```
2/102← (← ($ BlippingActiveValue) SpecializeMethod 'GetWrappedValue)
BlippingActiveValue.GetWrappedValue
```

Make window1 an instance of Window:

```
2/111← (← ($ Window) New 'window1)
#,$(& Window (|L↑[ZYLX1.0.0.k0<| . 9))
```

We can see what the default description BlippingActiveValue GetWrappedValue) is via:

```
2/115← (PRINTOUT PPDefault "!!")
"!"
```

In Exec2, the default TTY we see:

```
(Method ((BlippingActiveValue GetWrappedValue)
 self containingObj varName propName type)
"Fetch the value wrapped in the active value"
(←Super
 self GetWrappedValue containingObj varName propName type))!!
```

## Medley LOOPS: The Basic System

Now, let's set the height of window1:

```
2/118+ (←@ ($ window 1) height 100)
100
```

And, add an active value for 'height to winow1:

```
2/118+ (←New ($ BlippingActiveValue) AddActiveValue ($ window 1) 'height)
#,$& BlippingActiveValue (|L↑[ZYLX1.0.0.k0<| . 10))
.....
```

Now, let's change the height to 300.

```
2/118+ (←@ ($ window 1) height 300)
300
2/120+ (@ ($ window 1) height)
300
```

### 7.3.2.3 ExplicitFnActiveValue

**ExplicitFnActiveValue** emulates an active value that was used in the previous Buttruss version of Interlisp and found in earlier versions of Truckin. It was also used in LOOPSBACKWARDS, which will be described in *Medley Loops: Rule-based Systems*.

**Note:** *The Medley implementation recommends that users do not use this AV in new projects.*

### 7.3.2.4 NoUpdatePermittedAV

**NoUpdatePermittedAV** is a subclass of **activeValue** that does not allow the variable to be updated. This AV is used to effectively create a constant variable in a class or an instance. When the AV is created, the current state is stored in the local State. **GetValue** will return the value of the variable, but **PutValue** will cause a break with the message **NoUpdatePermitted!**. Table 7-3 describes its instance variables.

**Table 7-3. NoUpdatePermittedAV Instance Variables**

Instance Variable	Description
localState	A variable that holds the actual value of the variable which is wrapped as an active value.

### 7.3.2.5 LispWindowAV

**LispWindowAV** is a subclass used by LOOPS to ensure that the **window** instance variable within a **LOOPSWindow** contains an Interlisp window. It is a specialization of **LocalStateActiveValue**.

**LispWindowAV** is installed on the **window** variable of a subclass of **LOOPSWindow**. It checks to see if **localState** is a window and assures that the other instance variables are set correctly. See Medley LOOPS Advanced Topics for further details.

**Note: Medley LOOPS suggests this class provides functionality required by the LOOPS system, and should not generally be used by LOOPS users.**

### 7.3.2.6 InheritingAV

**InheritingAV** is an abstract class that is used as a mixin to add the **InheritedValue** method to a class. It is also used as a super class of **AppendSuperValue** to provide incremental menus in various parts of LOOPS.

This specialization is used as a mixin to add the **InheritedValue** method to a class. This method allows an instance to access the value of an instance variable defined in the parent class as if there was no value assigned to the IV in the **localState** of the instance. It is added as mixin to other specialization of **activeValue** in the instance.

### 7.3.3 Breaking and Tracing Active Values

To facilitate the breaking and tracing active values, LOOPS defines some specializations of **localStateActiveValue** for debugging of LOOPS applications. All breaks and traces occur before the variable is read or written. Table 7-4 describes the instance variables found in these subclasses.

**Table 7-4 Subclass Instance Variables**

<b>Instance Variable</b>	<b>Description</b>
object	An instance of a class containing varName.
varName	The name of the tracked variable which is referenced by the AV.
propName	If non-NIL, a property associated with the IV.

### 7.3.3.1 BreakOnPut

**BreakOnPut** breaks when an attempt is made to put a new value to a variable.

### 7.3.3.2 BreakOnPutOrGet

**BreakOnPutOrGet** breaks when a variable is read or written.

### 7.3.3.3 TraceOnPut

**TraceOnPut** traces attempts to put new value to a variable.

### 7.3.3.4 TraceOnPutOrGet

**TraceOnPutOrGet** traces an attempt to read or write a variable.

## *7.3.4 Appending to a Super Value*

A instance variable may have a value both in the parent class of an instance and defined locally in the instance itself. It is sometimes useful to know both values to determine which one to use in an application. Alternatively, the local value may be a refinement of the value of the instance variable stored in the parent class.

When **AppendSuperValue** is installed on a variable of an instance, Get- references return the value of the variable in the **localState** of the instance appended to the value of the variable in its parent class (e.g., the value that is inherited).

Any **PutValue** replaced the active value as well as the **localState** value.

*Note: Medley LOOPS notes that `appendSuperValue` was designed for use in class variables where replacement is infrequent. [It is not clear why this is so?]*

### 7.3.5 *InheritedValue*

**InheritedValue** allows a user program to access the value of an IV defined in the parent class or a superclass in the class hierarchy that the instance would have inherited if the IV had no value in the **localState**. **InheritedValue** has the format:

Method: `InheritedValue`  
Arguments: `<object>`, InheritingAV instance.  
`<object>`, the class or instance containing the Variable.  
`<varName>`, the name of the variable.  
`<propName>`, Name of an IV or CV property to be viewed.  
`<type>`, one of IV, CV, or NIL.  
Return: The value which should have been inherited if the local instance had no value.

### 7.3.6 *ReplaceMeAV*

This specialization sets **PutWrappedValue** to simply replace itself on the first Put access. It is an abstract class not intended for instantiation. It is used as a mixin with another specialization to add its functionality to the subclass. No variables were defined in this class.

## Medley LOOPS: The Basic System

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

### *7.3.7 NotSetValue*

This specialization of **ActiveValue** is used to implement instance variable inheritance. It has no instance variable to hold a local value and is replaced after the first **PutValue** instance.

When an instance of a LOOPS object is created, all of its IVs were initialized to contain the value of the variable **NotSetValue**. Its value in an IV is replaced by the initialization procedures with another value generated by the initialization procedures of new instances that were invoked by the methods **NewWithValues** and **NewInstance**. By initializing all new instance variables in a new instance, LOOPS speeds up the initialization process.

The annotated value **#,NotSetValue** is bound to the Lisp variable **NotSetValue**. It always had to be on the inside of any set of nested values. Its **WrappingPrecedence** method returns **NIL**.

#### **7.3.7.1 NestedNotSetValue**

This is a subclass of **NotSetValue** that is used by the internal code of LOOPS to solve the problem of using active values as default values. **It should not be used by the user.**

### *7.3.8 User Specializations of Active Values*

When new specializations of the class **ActiveValue** were defined, the methods **GetWrappedValueOnly** and **PutWrappedValueOnly** might



need to be specialized. The user may choose to specialize the methods described in Table 7-5.

**Table 7-5. User Specialization of Selected AV Methods**

Method	Description
AVPrintSource	Prints data regarding an <b>ActiveValue</b> instance.
GetWrappedValue	The method for retrieving an activeValue.
PutWrappedValue	The method for putting an activeValue.
WrappingPrecedence	Returns T, NIL, a number to specify the order of activeValue nesting.
CopyActiveValue	The method for copying an annotatedValue and its wrapped activeValue.

### 7.4 Active Value Methods

The class **ActiveValue** has numerous methods for implementing its behavior. These methods fall into several categories which are described in the following sections.

#### *7.4.1 Adding and Deleting Active Values*

This section describes methods for installing, deleting, and replacing active values in LOOPS objects.

##### **7.4.1.1 AddActiveValue**

The **AddActiveValue** method installs an active value by first wrapping it in an **annotatedValue** and then placing the annotated Value as the value of a variable. Its format is:

## Medley LOOPS: The Basic System

Method:       AddActiveValue  
Arguments:    <object>, the handle of an instance of ActiveValue.  
              <object>, the handle of the object containing the  
              variable.  
              <varName>, the name of the variable receiving the AV.  
              <propName>, the name of an IV or CV property to be  
              transformed into an active value.  
              <type>, one of IV, CV, CLASS, METHOD or NIL.  
              <annotatedValue>, an **AnnotatedValue** object that  
              will contain the **ActiveValue** or NIL. If NIL, a new  
              **AnnotatedValue** is created.  
Result:       <annotatedValue>.

An example, from the example in Section 7.3.2.2, is:

```
2/118* (←New ($ BlippingActiveValue) AddActiveValue ($ window 1) 'height)
#,$($ BlippingActiveValue (|L+[ZYLX1.0.0.k0<| . 10))
.....
```

### 7.4.1.2 Wrapped Precedence

The **WrappedPrecedence** method returned a value which determined how to nest the active value associated with *self*. It's format is:

Method:       WrappedPrecedence  
Arguments:    <object>, an **ActiveValue** instance.  
Result:       T, NIL, or a number.

If it returned T, the AV associated with *self* was installed outside any other AV.

If it returned NIL, the AV was installed as the innermost AV.

## Medley LOOPS: The Basic System

If a number, indicated which layer (level of precedence) this AV would occupy in a set of nested AVs.

If an AV has an IV **localState**, then the original AV is inserted into the **localState** of the new AV to be installed.

### 7.4.1.3 DeleteActiveValue

The **DeleteActiveValue** method deleted an AV from a containing object. It's format is:

Method: DeleteActiveValue

Arguments: <object>, the handle of an ActiveValue instance.  
<object>, the LOOPS object containing the variable where the AV is stored.  
<varName>, the name of the variable receiving the AV.  
<propName>, the name of an IV or CV property to be transformed into an active value.  
<type>, one of IV, CV, CLASS, METHOD or NIL.

Return: The deleted AV, if it was found in <varName>.

### 7.4.1.4 ReplaceActiveValue

The **ReplaceActiveValue** method replaced an AV in a LOOPS object variable with a new AV. It was also used to replace an existing AV with an updated version. Its format is:

## Medley LOOPS: The Basic System

**Method:** ReplaceActiveValue

**Arguments:** <object>, the handle of an ActiveValue instance.  
<newVal>, the new value used to replace self.  
<object>, the LOOPS object containing the variable where the AV is stored.  
<varName>, the name of the variable receiving the AV.  
<propName>, the name of an IV or CV property to be transformed into an active value.  
<type>, one of IV, CV, or NIL.

**Return:** The value of <newVal>.

### *7.4.2 Wrapped Value Methods*

As noted, the value of a variable is wrapped in an **ActiveValue**, usually in the instance variable **localState**. This is done by specifying the behavior of new **ActiveValue** specializations by specializing the methods **GetWrappedValue** and **PutWrappedValue**. They bypass the active value mechanism.

#### **7.4.2.1 Getting Wrapped Values**

LOOPS provides two methods for getting wrapped value: **GetWrappedValue** and **GetWrappedValueOnly**. Their format is:

## Medley LOOPS: The Basic System

Method:        GetWrappedValue  
                  GetWrappedMethodOnly

Arguments:    <object>, the ActiveValue instance  
                  <object>, the LOOPS object containing the variable  
                  where the AV is stored.  
                  <varName>, the name of the variable receiving  
                  the AV.  
                  <propName>, the name of an IV or CV property  
                  to be transformed into an active value.  
                  <type>, one of IV, CV, CLASS, METHOD or NIL.

Return:        The value returned from the actions performed by  
                  the GetWrappedValue method.

The method contains the code to be triggered when a get reference has been made to an active value.

**GetWrappedValueOnly** allowed the ActiveValue mechanism to deal with nested active values. Users generally do not have specialize it, since other instances are available.

### 7.4.2.2 Putting Wrapped Values

LOOPS provided two mechanisms for putting wrapped values: **PutWrappedValue** and **PutWrappedValueOnly**. Their format is:

## Medley LOOPS: The Basic System

Method:       PutWrappedValue  
              PutWrappedValueOnly

Arguments:    <object>, the ActiveValue instance  
              <object>, the LOOPS object containing the variable  
              where the AV is stored.  
              <varName>, the name of the variable receiving  
              the AV.  
              <newValue>, the new value to be stored.  
              <propName>, the name of an IV or CV property  
              to be transformed into an active value.  
              <type>, one of IV, CV, CLASS, METHOD or NIL.

Return:       The value of <newValue>.

The method contains the code to be triggered when a put reference has been made to an active value.

**PutWrappedValueOnly** allowed the ActiveValue mechanism to deal with nested active values. Users generally do not have specialize it, since other instances are available.

### 7.5 Annotated Properties

In an active value, property annotations can be used to implement useful, but subsidiary, values. Such properties might include data about precision, about reliability, about accuracy of the value of the variable.

A reasoning or other system could store data about the value without changing the value itself. In the Truckin' application, we will see how gauges can be used to change the display of certain data by inspecting the data.

## 7.6 Defensive Programming

As we have seen, AVs are a powerful mechanism for managing access to IVs/CVs of classes and instances. A particular use is called *defensive programming*, which attempts to develop structures that prevent the programmer from doing damage to the application through inadvertent use of methods and functions. The basic idea in defensive programming is to wrap each IV/CV in an AV with the appropriate getFn and putFn methods that can:

1. Check incoming data for the right type and value;
2. Convert incoming data, as necessary, to accurately reflect the required;
3. Reformat outgoing data from internal representation to a format expected by external classes;
4. Mask all or parts of the outgoing data that should not be shared by external classes, but are necessary for internal computation within the class.

Defensive programming requires additional computational time, but it is a mechanism for trying to eliminate some of the proximal causes for errors in programs.

## 7.7 Active Value Uses

Bobrow and Stefik (1986) described several uses of active values. In the LOOPS debugging package, AVs were attached to variables that allowed the changes in the variables values to be traced, including out of range values for the variables.

Another use in the Gauges package is to change the display in the gauge whenever the value of a variable changed to which the gauge is attached. In *Volume II: Tools and Utilities*, Gauges will be discussed along with other tools and utilities. In either case, there is no change to code of the monitoring process.

In *Volume III, Designing Rule-Based Systems with LOOPS*, this capability will be described in more detail when we discuss the Truckin' game.



## Chapter 8

### Introduction to

## Rule-Oriented Programming

In *rule-oriented programming*, the behavior of a system is specified by sets of condition-action pairs. Typically, these have been symbolically represented by “if...then” statements, as seen in predicate logic, which are called *rules*. Rules are organized into rule sets, which capture an aspect of the behavior of the system. The total behavior is captured in the collection of rule sets.

Rules are selected by patterns in the data, which may be largely independent of each other. Typically, however, a collection of rule sets provides a capability for describing flexible responses to a wide and varying range of events that may occur in the domain of interest.

This section will provide a brief description of some aspects of rule-oriented programming in LOOPS. It is provided here only for completeness of the description of the paradigms incorporated in LOOPS.

Volume III of LOOPS documentation, *Medley LOOPS: Rule-based Systems*, will describe advanced aspects of rule-oriented program and how to write rule-based systems using the LOOPSRULES functions. It will also describe the Truckin’ game developed by several researchers at Xerox PARC to demonstrate how rule-based systems should be developed within the LOOPS environment.

## 8.1 RuleSets and Rules

A RuleSet contains specific control structures for selecting and executing rules.

Rule-oriented programming was the basis for many early Artificial Intelligence (AI) projects or building expert systems. LOOPS has incorporated lessons learned from those early systems to provide a powerful, flexible capability for rule-oriented programming.

Major features of LOOPS rule-oriented programming include:

1. Rules are organized into RuleSets each of which can have its own control structures for selecting and executing rules.
2. RuleSets are the building blocks for organizing reasoning programs in LOOPS.
3. LOOPS provides a workspace for rules, which contains the name space for rule variables.
4. Rule-oriented programming is integrated with the other programming paradigms discussed in this manual.
5. RuleSets can be accessed by sending a message to an object or triggered through an active value as well as being invoked directly from Lisp programs or other rules.
6. Rules leave an automatic audit trail that can be used to determine how a program reached its results.
7. RuleSets can be embedded within tasks that enhances the variety of control mechanisms.
8. A debugging facility allows users to debug their rule sets.

## 8.2 Organizing a Rule-based System

A Rule-Based System (RBS) can be organized in many ways:

- it can just be a collection of rules;
- it might be a RuleSet with a collection of rules; or
- it might be a collection of RuleSets.

The complexity of the program increases as we move from the top of the list to the bottom of the list.

### 8.3 RuleSet

Early on, most RBSs were organized as a collection of rules. This was a relatively simple structure, but as the number of rules increased it became much harder to organize them into groups in accordance with the domain structure. This difficulty meant it was often hard to grasp which rules applied to which conditions within the problem domain.

The concept of a RuleSet is that all rules pertaining to an object or subproblem within the domain are collected together in one place. This makes it easier to locate rules focused on a particular aspect of problem solving within the domain. It also make it easier to add new rules because the rule are contained within the RuleSet.

## Medley LOOPS: The Basic System

We can think of RuleSets in the following way:

1. The RuleSet provides a name space such that rules with a RuleSet must have unique names. However, those names can be used across different RuleSets.
2. The Rule Set makes it easy to add new rules to the program because new rules can be checked within a smaller set of rules, not all rules within the system.
3. A RuleSet makes it easy to select a set of rules to execute when a specific condition is detected rather having to check all rules, which depending on the number of rules and complexity of their IF parts, can be time-consuming.
4. RuleSets allow for different types of control structures, which provides more flexibility in structuring the program to reason within a problem space.

### *8.3.1 RuleSet Class Definition*

A **RuleSet** is a class which has the following definition:

```
(DEFCLASS RuleSet
  (MetaClass RuleSetMeta
    doc "A RuleSet is a set of rules, together with methods for
        interpreting them."
    Edited%: (* dgb%: "27-Aug-84 17:33"))
  (Supers NamedObject Perspective Method)
  (InstanceVariables
    (compiledRules NIL doc "Name of Lisp Function for
        Rules.")
    (workSpace NIL doc "name of class for work space.")
    (args NIL doc "arguments to the RuleSet other than self."))
```

## Medley LOOPS: The Basic System

```
(tempVars NIL doc "temporary variables.")
(taskVars NIL doc "Task variables.")
(debugVars NIL doc "variables to be printed during a trace
or break.")
(numRules NIL doc "Number of Rules in RuleSet.")
(controlStructure doAll doc "control structure for rules.")
(whileCondition NIL doc "while condition for RuleSet.")
(compilerOptions NIL doc "Compilation options.")
(auditClass #,($C StandardAuditRecord) doc "name of class
for audit records.")
(metaAssignments NIL doc "RuleSet specific meta
assignment statements.")
(ruleClass #,($C Rule) doc "name of class for rule objects.")
(taskClass NIL doc "Name of class used for tasking.")
(perspectiveNode #,($C RuleSetNode) myViewName RuleSet
)
)
```

These fields will be described in more detail in *Volume III: Rule-based Systems*.

### *8.3.2 RuleSetSource*

A **RuleSetSource** is a class that stores a list of rule numbers that contain the rule source code. Its definition is:

```
(DEFCLASS RuleSetSource
  (MetaClass Template
    doc "Source code for a RuleSet. Contains editing
    information about the RuleSet, and an indexed list of rule
```

## Medley LOOPS: The Basic System

```
objects."  
  Edited%: NIL  
)  
(Supers NamedObject Perspective Method)  
(InstanceVariables  
  (compiledRules NIL doc "name of Lisp function for  
  Rules")  
  (workspace NIL doc "name of class for workspace")  
  (args NIL doc "arguments to RuleSet other than self")  
  (tempVars NIL doc "temporary variables")  
  (taskVars NIL doc "task variables")  
  (debugVars NIL doc "variables to be printed during a  
  trace or break")  
  (numRules NIL doc « number of rules in the RuleSet")  
  (controlStructure doAll doc "control structure for rules")  
  (whileCondition NIL doc "while condition for RuleSet")  
  (compilerOptions NIL doc "compiler options")  
  (auditClass #,($C StandardAuditRecord) doc "name for  
  audit records")  
  (metaAssignments NIL doc "RuleSet specific meta  
  assignment statements")  
  (ruleClass #,($C Rule) doc "name of class for rule  
  objects")  
  (taskClass NIL doc "name of class used for tasking")  
  (perspectiveNode #,($C RuleSetNode) myViewName  
  RuleSetNode)  
)  
)  
)
```

## Medley LOOPS: The Basic System

### *8.3.3 RuleSet Structure*

At a high-level, the organization of a rule set appears something like this (example extracted from Stefik and Bobrow 1983):

RuleSet Name:	CheckWashingMachine
Workspace Class:	WashingMachine
Control Structure:	while1
While Condition:	ruleApplied

### *8.3.4 RuleSet Methods*

LOOPSRULES defines numerous methods for a RuleSet. Many of these are used internally within the LOOPSRULES code, although all are accessible to the programmer. However, this manual discusses just those methods that seem most useful to the programmer defining and using a RuleSet.

### *8.3.5 Invoking RuleSets*

RuleSets can be executed by invoking them from rules. A simple double-dot notation can be used to invoke a RuleSet as follows:

```
RS1..ws1
```

where RS1 is a variable bound to a RuleSet and the variable ws1 is its workspace. The value returned is that returned from executing the RuleSet.

## Medley LOOPS: The Basic System

A RuleSet can also be invoked by its LOOPS object name using the \$ notation as in:

```
$SHKRules..ws1
```

which invokes the RuleSet object with the LOOPS name SHKRules.

It is possible to nest the calling of Rules as if we calling a nested sequence of subroutines. Thus A..B..C has A invoking B which invokes C. C could also invoke A recursively.

The programmer must beware of coding a recursive invocation that does not have a termination condition else the recursion will continue until memory is exhausted.

### 8.4 RuleSetMeta

**RuleSetMeta** is the MetaClass for RuleSets. Its structure is:

```
(DEFCLASS RuleSetMeta
  (MetaClass Shkreli's doc "MetaClass for RuleSets" Edited%:
  (* <editor-name>: <data string>))
  (Supers Template)
)
```



## 8.5 RuleSetNode

A **RuleSetNode** is a Node for RuleSet Perspectives. Its structure is:

```
(DEFCLASS RuleSetNode
```

```
  (MetaClass Template doc "Node for RuleSet perspectives"  
  Edited%: (* <editor-name>: <data string>))
```

```
  (Supers Node Object)
```

```
  (InstanceVariables
```

```
    (perspectives NIL source #,($C RuleSetSource) Ruleset #,($C  
    RuleSet)
```

```
  )
```

```
)
```

As noted in the LRM 1983, perspectives provided different views of an entity based on how it is used in an RBS. An example might be viewing a man through different role lens such as father, employee, or traveler. A *perspective* provides access to the information in different ways.

**NOTE:** According to the LRM 1983, perspectives were not implemented in that version of LOOPS. We are researching this capability in technical documentation from Xerox PARC.

## 8.6 RuleSetSource

A **RuleSetSource** contains the source code for a RuleSet. Its structure is:

```
(DEFCLASS RuleSetSource
  MetaClass Template doc "Source code for a Ruleset, including
  editing information and an indexed list of rule objects."
  Edited%: (* <editor-name>: <data string>))
(Supers DatedObject Perspective)
(InstanceVariables
  (perspectiveNode #,($C RuleSetNode) myViewName
  source)
  (edited NIL doc "Date last edited.")
  (editor NIL doc "last user to edit the rules.")
  (ruleList NIL doc "a list of the rule objects")
)
)
```

## 8.7 Rule

A **Rule** describes one or more actions to be taken when a specified set of one or more conditions are satisfied. A rule has three major parts:

1. The left hand side (LHS)
2. The right hand side (RHS)
3. The meta description (MD).

## Medley LOOPS: The Basic System

Without the meta description, we typically write a rule as:

```
LHS -> RHS
IF LHS THEN RHS
```

which are equivalent syntactic representations.

A rule may have no conditions, whence it can be written as:

```
→ RHS
IF T THEN RHS.
```

A rule can be preceded by a meta description which is enclosed in curly braces as depicted below:

```
{MD} LHS -> RHS.
```

### *8.7.1 Rule Class Definition*

A Rule is a class which has the following definition:

```
(DEFCLASS Rule
  (MetaClass Class doc "Class for describing rules as objects.
  Instances of this class (rule objects) are created as a side-effect
  when RuleSets are compiled in audit mode."
    Edited%: * mjs%: "12-FEB-83 12:19"))
  (Supers Object)
  (InstanceVariables
```

## Medley LOOPS: The Basic System

(source NIL doc "string that was the source of the rule in  
The RuleSet.")

(edited NIL doc "person who edited the rule.")

(editor NIL doc "time and date of the editing.")

(ruleNumber 0 doc "sequence number of the rule in the  
RuleSet at the time of editing.")

(ruleSet NIL doc "RuleSet to which this rule belongs.")

)

)

### *8.7.2 Variables Used in Rules*

LOOPS supports multiple types of variables that are using within  
LOOPS programs. Table 8-1 presents these types of variables.

## Medley LOOPS: The Basic System

**Table 8-1. Types of LOOPS Variables**

Variable Type	Usage
Class Variables	These variables are descriptive of a class and are inherited by all instances of the class – either directly, in which case their value can be overridden in an instance, or indirectly, through a search of the class hierarchy.
Instance Variables	These variables are descriptive of a particular instance of a class.
RuleSet arguments	All RuleSets have the variable <b>self</b> as their workspace.
Temporary Variables	These variables are allocated when a RuleSet is invoked and deallocated when the Ruleset completes its execution
Audit Record Variables	These variables are used in the meta-assignment statements in the Meta-Description part of a rule. They describe data to be saved in audit records, which can be used to create side-effects of rules.
Rule Variables (REVIEW!!)	These variables hold descriptions of the rules themselves and are used only in the Meta-Descriptions of the rules. They specify data saved in the Rule Object when a rule is compiled. They are declared indirectly as the instance variables of a Rule Class declaration.
Interlisp Variables	These variables are defined as global in the Interlisp environment and available to any function, method, or rule.
Reserved Words	Several variables have specific uses in the LOOPS environment and a READ-ONLY. Table 8-2 describes these.
Other Literals	Literals can refer to Interlisp functions, LOOPS objects, and message selectors as well as strings and quoted constants.

## Medley LOOPS: The Basic System

**Table 8-2. Reserved Word Usage**

<b>Reserved Word</b>	<b>Usage</b>
self	The current workspace.
rs	The current RuleSet.
task	The Task representing a current invocation of a RuleSet.
caller	The RuleSet that invoked the current RuleSet (rs).
ruleApplied	Has value T if a rule was applied in this cycle of a while condition.
ruleObject	This variable represents the rule itself when a RuleSet is being executed.
ruleNumber	This variable is bound to the sequence number of a rule in a RuleSet when it is executing.
ruleLabel	This variable is bound to the label of a rule, if specified, or NIL.
reasons	This variable is bound to a list of audit records supporting an instance variable of an LHS of the rule.
auditObject	This variable is bound to an object on which a reason record will be attached at run time.
auditVarName	This variable is bound to the name of a variable on which the reason will be a property.

### *8.7.3 Infix Operators Used in Rules*

Rules can be written in several formats. LOOPS provides several infix operators that can enhance the readability of rules. Table 8-3 presents these operators.

**Table 8-3. Infix Operators in LOOPS Rules**

Operators	Usage
+, -, *, /	Arithmetic Operators
++, --	Arithmetic Operators Module 4
>, <, >=, <=, =, ~=	Relational Operators
==	EQUAL
<<	Member of a list (FMEMB)

Two unary operators were also supported:

‘-‘ (Minus)

‘~‘ (Not)

The precedence of operators followed the standard conventions.

#### *8.7.4 Use of Interlisp Functions in Rules*

Interlisp functions may be invoked in LOOPS rules by enclosing the function and its arguments, if any, in parentheses. The function name is the first literal followed by the arguments as literals. Functions may also be invoked to produce the values of arguments as in the following example:

(Display <arg1> <arg2> (Cursor x y))

## Medley LOOPS: The Basic System

### 8.7.5 Use of LOOPS Objects and Message Selectors

LOOPS classes and other named objects may be referenced in rule using the  $\$<class>$  notation. As noted above (Section??), messages may be sent to LOOPS classes using message selectors in the following format:

```
<var> _(<_ $<class> <selector> [<arg1> ... <argN>])
```

In a rule this might appear as:

```
IF cell _ (<_ $LowCell Occupied? 'Heavy')
THEN (<_ cell Move 3 'North');
```

For unary messages, e.g., messages where only the selector is specified, an assumption of an implicit **self** allows the following format:

```
tile.Type='BlueGreenCross command.Type='Slide4 >- ...
```

where *Type* is the unary message sent to the *tile* instance variable in the workspace. *tile* must be a LOOPS object at run-time else an error results.

We could also refer to a LOOPS object whose name is *Tile* as follows:

```
$Tile.Type='BlueGreenCross;
```



## Medley LOOPS: The Basic System

We can access a variable in named LOOPS object using the colon notation:

```
$Tile:type='BlueGreenCross ...
```

where the `type` instance variable of the LOOPS object `Tile` is accessed.

Double colon notation can be used to access a class variable of a LOOPS object, such as:

```
Truck::MaxGas<45 ::ValueAdded>600 -> ...
```

where `MaxGas` is a class variable of an object bound to `truck` and `ValueAdded` is a class variable of **self**.

Two additional examples demonstrate the colon-comma notation:

```
wire::capacitance>5 wire:voltage:,support='simulation -> ...
```

In the first expression, `wire` is an instance variable of the workspace and `capacitance` is a property of that variable.

In the second expression, the value of variable `wire` is a LOOPS object which has an instance variable of `voltage` whose property is `support`, which receives the value `'simulation`.

LOOPS provides flexibility of expression, but it takes some time getting used to the different forms. Thus, it is helpful to read these expressions twice to ensure you understand what action will be taken and value produced.

## 8.8 Running RuleSets

A RuleSet can be executed using the **RunRS** function whose format is depicted below:

Function:	RunRS
Arguments:	<RuleSet>, the Rule Set to be run. <workspace>, a LOOPS object to be used as a workspace. <arg1>...<argN>, arguments to the RuleSet.
Result:	The value returned by the RuleSet.

## 8.9 Using RuleSets as Methods

A ruleset can be used as a method by making it the implementation of a method for a class. As an example (Stefik, Bobrow, and Mittal 1983):

```
(DEFCLASS WashingMachine
  (MetaClass Class doc (* comment) ...)
  (InstanceVariables (Owner ...))
  (Methods
    (Simulate SimulateWMRules)
    (Check RunCheckWMRules
      doc (* Rules to check WM)
    )
  )
)
```

```
)  
...  
)
```

When a Simulate message is sent to an instance of WashingMachine, the SimulateWMRules RuleSet will be run with the instance as its workspace.

### *8.9.1 Defining A RuleSet as a Method*

The function **DefRSM** (“define RuleSet as a Method”) can be used to specify a RuleSet as a method. It takes the form:

Function:	DefRSM
Arguments:	<ClassName>, the name of a class. <Selector>, the name of the message to invoke the RuleSet, <RuleSetName>, the name of the RuleSet to be installed as a method.
Result:	The name of the RuleSet.

If the RuleSetName is NIL, DefRSM creates a RuleSet object, opens the Editor for the user to enter rules, compiles the rules into a Lisp function, and installs the RuleSet as the target of the <selector> in the class Methods.

## **8.10 Control Structures for Selecting Rules**

## Medley LOOPS: The Basic System

A control structure is associated with a rule set. It determines which rules are executed given that the conditions (the if-part) must be satisfied using the rules in the workspace. Different values for variables in rules will lead to different rules being executed. Table 8-4 depicts the control structures.

**Table 8-4. RuleSet Control Structures**

<b>Control</b>	<b>Description</b>
Do1	<p>Execute the first rule in the RuleSet whose conditions are satisfied. The value of the RuleSet is the value of the rule.</p> <p>It is used to specify a set of mutually exclusive actions. Specific rules should be placed before general rules in the RuleSet.</p>
DoAll	<p>For each rule in the RuleSet, every rule whose conditions are satisfied is executed. The value of the RuleSet is the value of the last rule executed. If no rule is executed, the value is NIL.</p> <p>It is used when many aspects of a situation can be carried out independently, but should all be carried out in one invocation of the RuleSet.</p>
While1	<p>Iterating over the rules of the RuleSet, one rule is executed in each iteration, and then selection begins anew from the beginning of the Rule Set.</p> <p>The value of the RuleSet is the value of the last rule executed.</p> <p>If no rule is executed, then the value of the RuleSet is NIL.</p> <p>Iteration can be terminated by placing a STOP statement in the action part of a rule.</p>
WhileAll	<p>Iterating over the rules of the RuleSet, if the while condition is satisfied, every rule for which it is satisfied is executed.</p>

### *8.10.1 Singleton Rule Execution*

In some cases, such as initializing a problem, we want to execute a rule only once when the Ruleset is first examined. This is termed a One-Shot Rule, which corresponds to the singleton case in Design Patterns (Gamma, Johnson, Helms, and Vlissides 1985).

In the Singleton design pattern, a variable was set to false (or 0) to indicate an action had not yet occurred, then set to true or 1 when the action was performed. The variable was tested each time for 0 when the action was to be performed. Thereafter, if the value of 1, the action was never performed again. Figure 8-5 depicts how this might be done:

Control Structure: While1

Temporary Vars: ruleXApplied;

.....

IF ~ruleXApplied <condition<sub>1</sub>> < condition<sub>2</sub>>

THEN ruleXApplied \_ T <action<sub>1</sub>>;

LOOPS provides a shorthand notation, which expresses the same intent:

{1} IF <condition<sub>1</sub>> < condition<sub>2</sub>> THEN <action<sub>1</sub>>;

where the in the braces indicates the number of times the rule is to be executed

## References

Bobrow, D.G., and Stefik, M. J. 1986. Perspectives on Artificial Intelligence Programming. *Science* 231:4741, pp. 951-956.

-Reprinted in Rich, C. & Waters R.C. (Eds.) *Readings in Artificial Intelligence and Software Engineering*, pp. 581-587, Los Altos: Morgan Kaufman Publishers, 1986.)

Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. 1986. "CommonLoops: Merging Lisp and Object-Oriented Programming. *OOPSLA '86: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 17-29, Portland, Oregon, September 29 – October 2, 1986, Edited by Normal Meyrowitz, Special Issue of SIGPLAN Notices 21:11, November 1986.

(Reprinted in Peterson, G.E. (ed), *Object-Oriented Computing*, Volume 1: Concepts, IEEE Computer Society Press, pp. 169-181, 1987.

Bobrow, D. G. & Stefik, M. J. 1982. LOOPS: Data and Object Oriented Programming for Interlisp. *European AI Conference*, Orsay, France.

Bobrow, D. G., Stefik, M. 1983. The Loops Manual. Knowledge-Based VLSI Design Group Memo KB-VLSI-81-13.

Cannon, H. Flavors: a non-hierarchical approach to object-oriented programming, *Symbolics, Inc.*, 1982.

Kaisler, S. 1985. *Interlisp: The Language and Its Usage*, John Wiley & Sons, Inc., New York, NY

Copyright reverted to author; scanned and converted to PDF – 2021. Available at [Interlisp.org](http://Interlisp.org).

## Medley LOOPS: The Basic System

Kaisler, S. Unpublished. *Medley Interlisp: The Interactive Programming Environment*,

Revised and extended 2021. Available at [Interlisp.org](http://Interlisp.org).

Kaisler, S. Unpublished. *Medley Interlisp: Tools and Utilities, Revised and extended 2021*. Available at [Interlisp.org](http://Interlisp.org).

Malone, T.W., K.R. Grant, and F.A. Turbak. 1986. "The Information Lens: An Intelligent System for Information Sharing in Organizations", CHI'86 Proceedings.

Stefik, M. 1979. "An examination of a frame-structured representation system", *Proceedings of the International Joint Conference on Artificial Intelligence*, Tokyo, Japan, pp. 845-852.

Stefik, M., D.G. Bobrow, S. Mittal, and L. Conway. 1983. Knowledge Programming in LOOPS: Report on an Experimental Course, *AI Magazine*.

Stefik, M. and Bobrow, D.G., and Kahn, K. 1986. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3:1, pp. 10-18.

-Reprinted in Peterson, G.E. (ed). 1987. *Object-Oriented Computing, Volume 2: Implementations*, IEEE Computer Society Press, pp. 170-179.

-Also reprinted in Richer, M.H. (ed.) *AI Tools and Techniques*, pp. 47-63, Ablex Publishing Corporation, Norwood, New Jersey.

Stefik, M. and Bobrow, D.G., and Kahn, K. 1986. "Access-oriented programming for a multiparadigm environment", *Proceedings of the Hawaii International Conference on System Sciences*.

## Medley LOOPS: The Basic System

(Note: This paper won the best paper award out of 80 papers for the conference. An expanded version of this paper appeared by invitation in IEEE Software. This paper has also been reprinted in various books.)

Stefik, M. and Bobrow, D.G. 1986. Object-oriented programming: Themes and Variations. *AI Magazine* 6:4, pp. 40-62.

-Reprinted in Peterson, G.E. (ed). 1987. *Object-Oriented Computing*, Volume 1: Concepts, IEEE Computer Society Press, pp. 182-204.

-Also reprinted in Richer, M.H. (ed.) *AI Tools and Techniques*, pp. 3-45, Ablex Publishing Corporation, Norwood, New Jersey.)

Venue Corporation, San Carlos, CA:

(a) 1991. LOOPS Library Module Manual (LMM)

(b) 1991. LOOPS Reference Manual (LRM)

Xerox Corporation, Palo Alto, CA:

(a) 1986. Xerox LOOPS, A Friendly Primer, 3102242



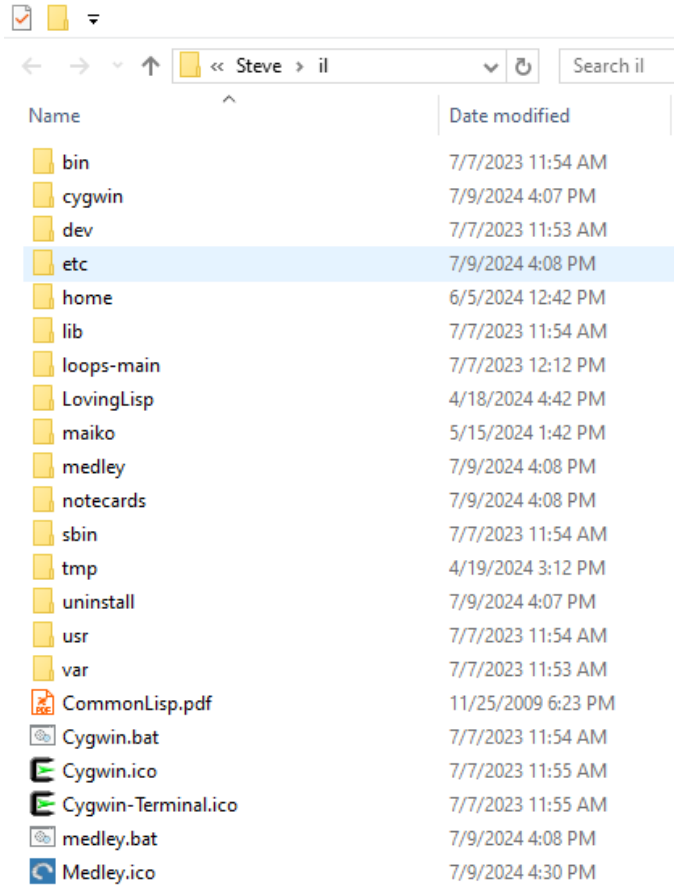
## Appendix A: Running MEDLEY

### A.1 Running Medley

Under Cygwin, my home directory is `/home/steve` and my user name is `steve`. Within `/home/steve`, the installer has established directories for `maiko` and `medley`. `Maiko` is the virtual machine for executing the Interlisp byte codes. `Medley` is the directory for Interlisp software – both source code and compiled code.

To run Medley on Windows 10/11, navigate to your home directory, which should be `c:/users/<name>/il`. It should look something like this:

## Medley LOOPS: The Basic System



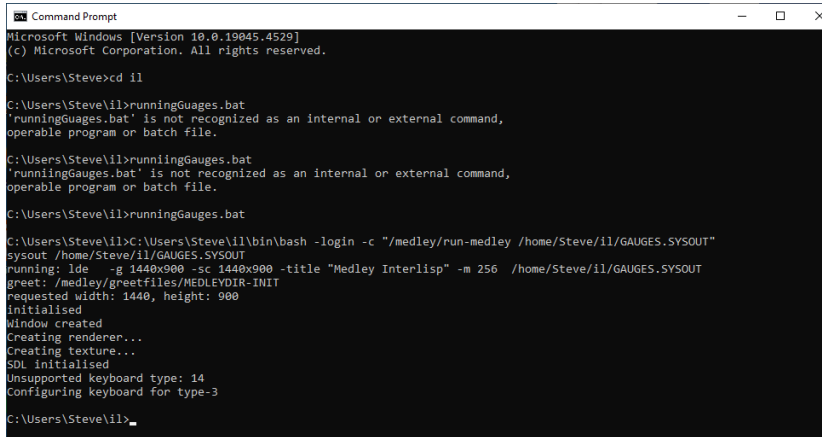
1. Navigate to you home directory, thence to subdirectory /il.

The key directories you are interested in are “loops-main”, where you will find subdirectories for Medley LOOPS and “home”, where you will find your home directory, <name>.

2. Enter “medley.bat” at the command prompt.

## Medley LOOPS: The Basic System

1. Open a Command window via cmd. You should see something like this:



```
Microsoft Windows [Version 10.0.19045.4529]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Steve>cd il

C:\Users\Steve\il>runningGauges.bat
'runningGauges.bat' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Steve\il>runningGauges.bat
'runningGauges.bat' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Steve\il>runningGauges.bat

C:\Users\Steve\il>C:\Users\Steve\il\bin\bash -login -c "/medley/run-medley /home/Steve/il/GAUGES.SYSOUT"
sysout /home/Steve/il/GAUGES.SYSOUT
running: lde -g 1440x900 -sc 1440x900 -title "Medley Interlisp" -m 256 /home/Steve/il/GAUGES.SYSOUT
greet: /medley/greetfiles/MEDLEYDIR-INIT
requested width: 1440, height: 900
initialised
window created
Creating renderer...
Creating texture...
SDL initialised
Unsupported keyboard type: 14
Configuring keyboard for type-3

C:\Users\Steve\il>
```

Instead of runningGauges, you should see a runningMedley message.

2. Go to directory il in your home directory, here c:\Users\Steve\il which initiates a Medley Interlisp environment as seen below:



## Medley LOOPS: The Basic System

```
Command Prompt
C:\Users\Steve\il\home\Steve\ATN>dir
Volume in drive C has no label.
Volume Serial Number is 02CE-FFDA

Directory of C:\Users\Steve\il\home\Steve\ATN

07/17/2024  11:34 AM  <DIR>          .
07/17/2024  11:34 AM  <DIR>          ..
03/26/2003  02:32 PM           762 atn-compile
12/18/2002  01:28 PM           259 atn-compile-mac.l
07/17/2024  08:07 AM      4,594 atn-cstate.l
09/11/2004  04:21 PM      4,061 atn-gsem.l
07/17/2024  07:54 AM           542 atn-load
07/17/2024  07:54 AM           542 atn-load.txt
09/12/2004  06:51 PM      2,794 atn-pstate.l
03/08/2004  09:37 AM      3,968 atn-record.l
02/15/2007  02:20 PM      6,133 atn-semantics.l
```

4. Create a LISP.SYSOUT by entering the following command in the Interlisp window (right window):

```
>(SYSOUT 'LISP.SYSOUT)
```

Now you have a LISP sysout that can be loaded each time you inoke medley.

I have created .bat files to run various versions of Medley sysouts (a) without LOOPS, (b) with just LOOPS, or (c) with LOOPSRULES and GAUGES. I tend to run with LOOPSRULES and GAUGES>

## Appendix B

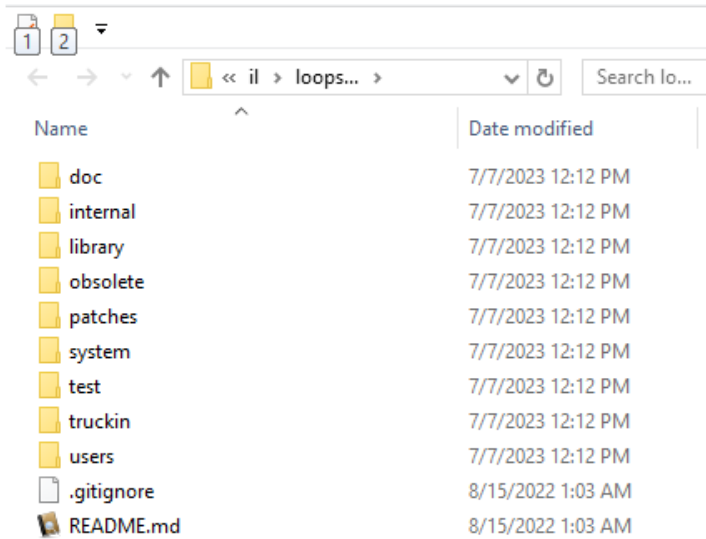
### Installing and Running LOOPS

In order to run LOOPS, you must have a Medley Interlisp distribution that runs on Windows 10 or higher, a Linux variant, or MacOS release. Medley Interlisp assumes that an Xwindows environment is available for implementing a graphical user interface (GUI). Medley Interlisp for Windows uses Cygwin, which is included in the release (see [www.Interlisp.org](http://www.Interlisp.org)).

LOOPS code is loaded into the directory “loops-main” in the Interlisp directory accessed by `c:\Users/Steve/il`.

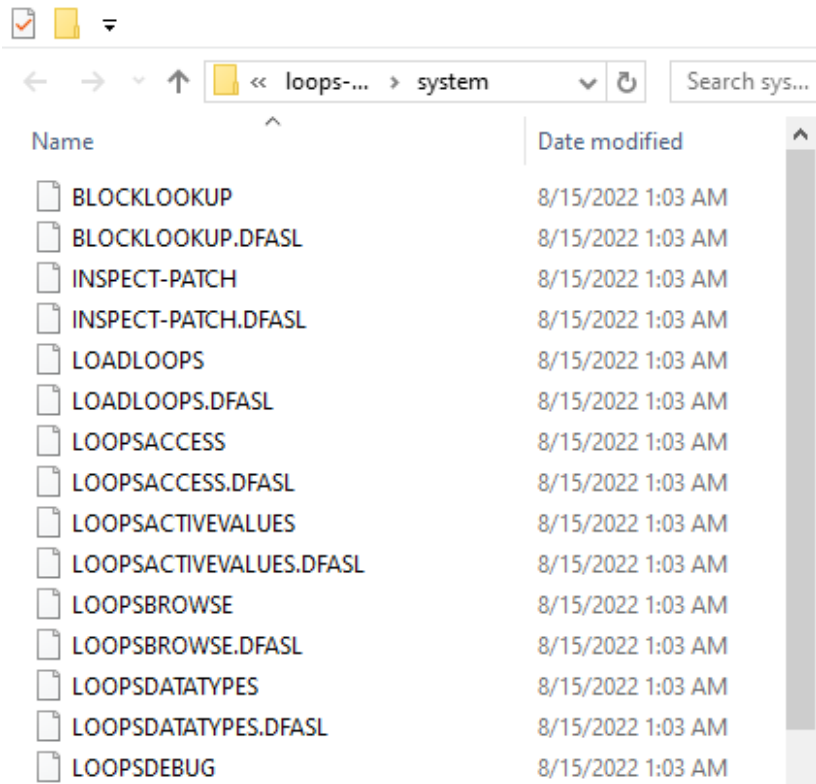
1. Click on loops-main to open this directory. You should see something like this.

## Medley LOOPS: The Basic System



2. Navigate to the “system” directory. You should see something like this (partial):

## Medley LOOPS: The Basic System



3. In the Interlisp Window (right window), enter the commands:

```
(CNDIR "your/loops/system")
```

```
(FILESLOAD LOADLOOPS)
```

```
(LOADLOOPS)
```

These commands load the compiled version of the LOOPS system from loops-main/system.

These commands will NOT work in the XCL window (left window).



## Medley LOOPS: The Basic System

You may also want to load the Masterscope enhancements for LOOPS via:

```
>(CNDIR "your/loops/library")
>(FILESLOAD LOOPSMS)
```

At this point, you may want to consider creating a LOOPS sysout that consists of the LOOPS infrastructure. This will allow you to perform object-oriented programming, which extends the imperative/functional programming model of Interlisp. To do so, enter the following command at the prompt:

```
>(SYSOUT 'LOOPS.SYSOUT)
```

### 4. Loading Gauge. Gauges are described in Volume II: Medl

Medley LOOPS supports graphical widgets called *gauges*. To load gauges into your sysout, you need to navigate to loops-main/library via:

```
>(CNDIR "your/loops/library")
>(FILESLOAD GAUGELOADER
>(LOADGAUGES)
```

to load gauges. Gauges are described in Volume II: *Medley LOOPS: Tools and Utilities*.

## Medley LOOPS: The Basic System

```
Exec 2 (INTERLISP)
NIL
2/134+ (LOADLOOPSRULES T)
"LOADLOOPSRULES"
"Loading LOOPUSERSFILES"
"Loaded LOOPUSERSFILES"
"Loading LOOPSLIBRARYFILES"
"Loaded LOOPSLIBRARYFILES"
{DSK}<home>steve>LOOPS-MAIN>
2/135+
```

```
Exec 2 (INTERLISP)
NIL
2/136+ (SYSOUT 'LOOPSRULES)
"{DSK}<home>steve>LOOPS-MAIN>LOOPSRULES.SYSOUT;1"
2/137+
```

Now, when you want to run with this sysout, you can do so using this form of run-medley from the operating system prompt:

Figure B-2. Starting Medley with the LOOPSRULES Sysout

The Interlisp Exec appears as shown in Figure B-3.

```
Exec 2 (INTERLISP)
NIL
2/136+ (SYSOUT 'LOOPSRULES)
****ATTENTION USER ROOT:
this sysout is initialized for user STEVE.
To reinitialize, type GREET()
(NIL)
2/137+
```

## Medley LOOPS: The Basic System

Figure B-3. LOOPSRULES Sysout Running

Note that the title bar will reflect the user as root, but you can change this by clicking on **User** and typing the appropriate name, e.g. steve, in the pop-up window.

### B.2 Loading LOOPSRULES and GAUGES

### B.3 Setting System Variables

Interlisp uses several system variables to access code and data. Two of these are:

- DIRECTORIES:
- DISPLAYFONTDIRECTORIES:

You should set these so that so the sysout can find your Medley library and font files. Figure B-x depicts an example.

```
39> (SETQ DIRECTORIES (CONS DIRECTORIES '/home/steve.LOOPS/src/library))
({"{DSK}<home>steve>medley>library}" "{DSK}<home>steve>medley>lispusers}" "{DSK}
}<home>steve>medley>internal>library}" "{DSK}<home>steve>medley>sources}" ) . /h
ome/steve.LOOPS/src/library)
40> DISPLAYFONTDIRECTORIES
({"{DSK}<home>steve>medley>fonts>displayfonts}" "{DSK}<home>steve>medley>fonts>a
ltofonts}" )
41>A
```

Figure B-x. Directory Variables

The Medley sysout predefines some locations for directories as indicated in the figure.

## Medley LOOPS: The Basic System

### *B.3.1 Connect to the LOOPS System Directory*

You connect to the LOOPS system directory using the CNDIR function as depicted below.

```
42> (CNDIR '/home/steve/LOOPS/src/system)  
{DSK}<home>steve>LOOPS>src>system>  
43>
```

## Appendix C

### Testing LOOPS Installation

LOOPS Medley includes a directory with test modules for testing LOOPS functionality and timing. These are (on my system):

- /home/steve/LOOPS-MAIN/test/loops
- /home/steve/LOOPS-MAIN/test/timing

```
steve@SHKLaptop:~/LOOPS-MAIN/test$ ls -lsa
total 20
4 drwxr-xr-x  5 root  root 4096 Aug 22 10:57 .
4 drwxr-xr-x 10 steve root 4096 Feb 14 10:25 ..
4 drwxr-xr-x  3 root  root 4096 Aug 22 10:57 from1.1
4 drwxr-xr-x  2 root  root 4096 Aug 22 10:57 loops
4 drwxr-xr-x  2 root  root 4096 Aug 22 10:57 timing
```

On your system, the home directory will be different.

We'll discuss and executes the loops tests first, then move on to the timing tests.

#### C.1 LOOPS 1.1 Tests

LOOPS testing should begin with a fresh sysout. LOOPS-SETUP.TEDIT describes the testing operations.

The first action is to set the system font directories. I did this during the installation process (refer to Appendix B), I will just display the font directories:

## Medley LOOPS: The Basic System

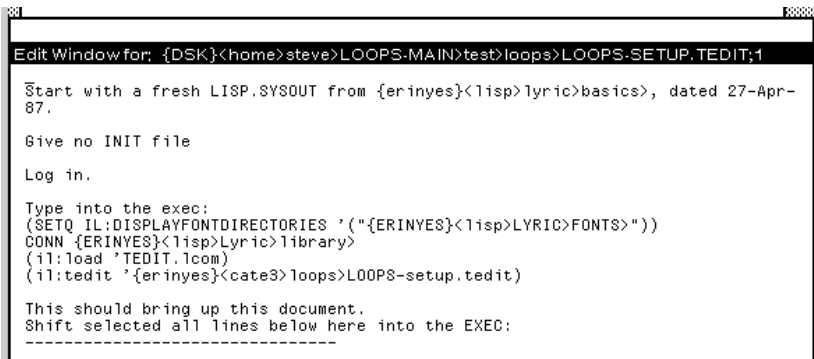
```
2/12* IL:DISPLAYFONTDIRECTORIES  
("{DSK}<home>steve>medley>fonts>displayfonts)" "{DSK}<home>steve>medley>  
fonts>altfonts)" "{DSK}<home>steve>medley>fonts>adobe")
```

1. If you want to load SETUP document from within Medley, you may do so with:

```
2/10* (TEDIT 'LOOPS-SETUP.TEDIT)  
#<Process TEdit/134,73614>
```

Note: Our standard MEDLEY sysout already has TEDIT loaded into it.

When you run the command above, you are prompted to frame a TEDIT window which then displays the setup document:



```
Edit Window for: {DSK}<home>steve>LOOPS-MAIN>test>loops>LOOPS-SETUP.TEDIT;1  
  
Start with a fresh LISP.SYSOUT from {erinyes}<lisp>lyric>basics>, dated 27-Apr-87.  
  
Give no INIT file  
  
Log in.  
  
Type into the exec:  
(SETQ IL:DISPLAYFONTDIRECTORIES '("{ERINYES}<lisp>LYRIC>FONTS"))  
CONN {ERINYES}<lisp>Lyric>library  
(il:load 'TEDIT.lcom)  
(il:tedit '{erinyes}<cate3>loops>LOOPS-setup.tedit)  
  
This should bring up this document.  
Shift selected all lines below here into the EXEC:  
-----
```

2. Next, load the FILEBROWSER using (LOAD'FILEBROWSER). This loads the file browser functions from the MEDLEY Library.

```
{DSK}<home>steve>medley>library>FILEBROWSER.;1
```

## Medley LOOPS: The Basic System

3. Next, we load WHO-LINE for the Lisp Users packages:

```
2/14+ (IL:LOAD 'WHO-LINE)
{DSK}<home>steve>medley>lispusers>WHO-LINE.;1
File created 26-Mar-2021 11:01:59
WHO-LINECOMS
New fns definition for INSTALL-WHO-LINE-OPTIONS.
New fns definition for WHO-LINE-USERNAME.
New fns definition for WHO-LINE-CHANGE-USER.
New fns definition for WHO-LINE-USER-AFTER-LOGIN.
New fns definition for WHO-LINE-HOST-NAME.
New fns definition for CURRENT-TTY-PACKAGE.
New fns definition for SET-PACKAGE-INTERACTIVELY.
New fns definition for SET-TTY-PACKAGE-INTERACTIVELY.
New fns definition for CURRENT-TTY-READTABLE-NAME.
New fns definition for SET-READTABLE-INTERACTIVELY.
New fns definition for SET-TTY-READTABLE-INTERACTIVELY.
New fns definition for WHO-LINE-TTY-PROCESS.
New fns definition for CHANGE-TTY-PROCESS-INTERACTIVELY.
New fns definition for WHO-LINE-CURRENT-DIRECTORY.
New fns definition for SET-CONNECTED-DIRECTORY-INTERACTIVELY.
New fns definition for WHO-LINE-VMEM.
New fns definition for WHO-LINE-SAVE-VMEM.
New fns definition for WHO-LINE-TIME.
New fns definition for WHO-LINE-SET-TIME.
New fns definition for WHO-LINE-SHOW-ACTIVE.
Warning: \UPDATE-WHO-LINE-ACTIVE-FLAG may be unsafe to redefine
-- continue? ...No
\UPDATE-WHO-LINE-ACTIVE-FLAG not redefined
```

This also generates an error pane as shown below:

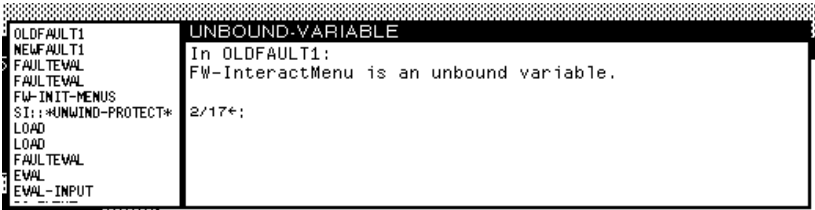
<pre>? OLDFAULT1 ? NEWFAULT1 ? FAULT-EVAL ? LET* ? WHO-LINE-VMEM ? EVAL ? SI::*UNWIND-PROTECT* ? UPDATE-WHO-LINE ? SI::*UNWIND-PROTECT* ? PERIODICALLY-UPDATE-WHO-L ? T</pre>	<pre>UNDEFINED-FUNCTION In OLDFAULT1: .VMEM.CONSISTENTP. is an undefined function. BACKGROUND/15(debug)</pre>
---	---

## Medley LOOPS: The Basic System

*NOTE: At this time, we will continue, but once we have tested LOOPS, we will return to diagnose and correct this problem.*

4. Next, we load FILEWATCH, which generates an error pane as shown below:

```
2/18< (IL:LOAD 'filewatch)
{DSK}<home>steve>medley>lispusers>FILEWATCH.;1
File created 13-Jun-2021 08:41:07
FILEWATCHCOMS
```



Close the error pane by clicking on the top bar and selecting 'Close'.

*NOTE: At this time, we will continue, but once we have tested LOOPS, we will return to diagnose and correct this problem.*

5. Next, we load CROCK.

```
2/18< (IL:LOAD 'Crock)
{DSK}<home>steve>medley>lispusers>CROCK.;1
File created 2-Apr-87 00:37:46
CROCKCOMS
{DSK}<home>steve>medley>lispusers>CROCK.;1
```



## Medley LOOPS: The Basic System

6. Next, we load do-test.

```
2/19+ (IL:LOAD 'do-test)
2/21+ CONN internal/library
Non-existent directory
internal/library
```

We have not located do-test in the medley internal and/or library directories. We will explore reloading the Lyric directories to locate it.

*NOTE: At this time, we will continue, but once we have tested LOOPS, we will return to diagnose and correct this problem.*

7. Set a default compiler option:

```
23> (SETQ IL:*DEFAULT<LEANUP<COMPILER* 'cl:COMPILE-FILE)
COMPILE-FILE
```

Since there were errors in a number of the attempted loads, I created a new environment loading only:

- FILEBROWSER
- Filewatch
- Crock

8. Save as LOOPSTEST.SYSOUT in <home>steve>LOOPS-MAIN/LOOPSTEST.SYSOUT

## Medley LOOPS: The Basic System

```
2/11← (IL:SYSOUT 'LOOPSTEST.SYSOUT)  
LOOPS/MEDLEY 23-Jan-2023 ...  
Hello, SirOrMadam.  
(NIL)  
2/12←
```

This preserves the files loaded above so can run the tests for LOOPS.

## Appendix C: Test Applications

We developed several test to test the LOOPS code. These are briefly described in the sections below. For purposes of discussion the source code is also included.

### C.1 Source Code for TestAV.txt

This test was copied from the LRM 1991.

```
(* ; "Define the classes")
(DefineClass 'Tank)
(SETQ Tank (SEND ($ Tank) SetName 'Tank))
(PP ($ Tank))

(DefineClass 'Pipe)
(SETQ Pipe (SEND ($ Pipe) SetName 'Pipe))
(PP ($ Pipe))

(* ; "Add outputPressure as IV to Tank")
(SEND ($ Tank) AddIV 'outputPressure)
(PP Tank)

(* ; "Add inputputPressure to Pipe")
(PRINT "(SEND ($ Pipe) AddIV 'inputPressure)")
(SEND ($ Pipe) AddIV 'inputPressure)
(PP Pipe)
```

## Medley LOOPS: The Basic System

```
(* ; "Create subclass of Tank and Pipe named Tank1 and Pipe1")
(SETQ Tank1 (_ ($ Tank) New (QUOTE Tank1)))
(PP ($ Tank1))

(SETQ Pipe1 (_ ($ Pipe) New (QUOTE Pipe1)))
(PP ($ Pipe1))

(* ; "Create an instance of IndirectVariable")
(* ; "Initialize its contents to point to the Tank's pressure")
(PRINT "(_ ($ IndirectVariable) New 'indVar1)")
(SETQ indVar1 (_ ($ IndirectVariable) New (QUOTE indVar1)))
(_ indVar1 SetName (QUOTE indVar1))
(PP ($ indVar1))

(PRINT "Assign object and varName")
(PRINT "(_@ ($ indVar1) object ($ Tank1))")
(_@ indVar1 object Tank1)

(PRINT "(_@ ($ indVar1) varName 'outputPressure)")
(_@ ($ indVar1) varName 'outputPressure)
(PP ($ indVar1))

(* ; "Install the active value instance as the pipe's input pressure")
(PRINT "Installing ActValue instance.")
(PRINT "(_ ($ indVar1) AddActiveValue ($ Pipe1) 'inputPressure)")
(_ ($ indVar1) AddActiveValue ($ Pipe1) 'inputPressure)
(PP ($ indVar1))

(* ; "Accesses to either pipe's input pressure or tank's output pressure")
```

## Medley LOOPS: The Basic System

```
(@ Pipe1 inputPressure)
(_@ Pipe1 'inputPressure 100)
(@ Tank1 outputPressure)
(_@ Tank1 'outputPressure 200)
(@ Pipe1 inputPressure)

(* ; "Show Inspector Window on Tank1 & Pipe1")
(_ Tank1 Inspect NIL)

(_ Pipe1 Inspect NIL)

(PRINT "*** End of TestAV ***")
STOP
```

### C.2 Source Code for NewTestAv.txt

This test was copied from the LRM 1991.

```
(* ; "*** NewTestAV ***")
(* ; "From Section 8.2, Example 2 of the LRM ***")

(* ; "Create the Bin class for the Conveyor")
(DefineClass 'Bin)

(DefineClass 'Conveyor)

(* ; "Add IVs to describe Bin")
(SEND ($ Bin) AddIV 'height 0)
(SEND ($ Conveyor) AddIV 'height 0)
```

## Medley LOOPS: The Basic System

```
(* ; "Create a Bin instance.")
(SETQ Bin1 (SEND ($ Bin) New 'Bin1))
(SETQ Bin1 (SEND ($ Bin1) SetName 'Bin1))

(* ; "Create a Conveyor instance.")
(SETQ Conveyor1 (SEND ($ Conveyor) New 'Conveyor1))
(SETQ COnveyor1 (SEND ($ Conveyor1) SetName 'Conveyor1))

(* ; "Define 3FeetAbove as a class.")
(DefineClass '3FeetAbove '(IndirectVariable))
(SETQ 3FeetAbove (SEND ($ 3FeetAbove) SetName '3FeetAbove))
(PP 3FeetAbove)

(* ; "Create an instance of 3FeetAbove.")
(* ; "Initialize its contents to point to the bin's height.")
(SEND ($ 3FeetAbove) New '3fa1)

(_@ ($ 3fa1) object ($ Bin1))
(_@ ($ 3fa1) varName 'height)

(* ; "Install 3fa1 as the value of the conveyor's height.")
(SEND ($ 3fa1) AddActiveValue ($ Conveyor1) 'height)
(SEND ($ 3fa1) Inspect NIL)

(* ; "The height of Bin1 defaults to 0, but what is the height of conveyor?")

(PRINTOUT T      "The height of Bin1 is " (@ ($ Bin1) height) T)
(PRINTOUT T      "The height of Bin1 is " (@ ($ Conveyor1) height) T)
```

## Medley LOOPS: The Basic System

```
(* ; "Now, set Bin1's height or Conveyor1's height.")
(* ; "See how the track each other.")
(PRINTOUT T "Setting heights of Bin1 and Conveyor1." T)
(_@ ($ Bin1) height 15)

(PRINTOUT T "The hieght of Conveyor1 is " (@ ($ Conveyor1) height) T)

(PRINTOUT T "Rset the height of Conveyor1" T)
(_@ ($ Conveyor1) height 21)

(PRINTOUT T "The height of Conveyor1 is " (@ ($ Conveyor1) height) T)
(PRINTOUT T "THE height of Bin1 is " (@ ($ Bin1) height) T)

(* ; "Define subclass of LocalStateActiveValue.")
(* ; "Provide two IVs relative to height.")
(DefineClass 'WarningAV '(LocalStateActiveValue))
(SEND ($ WarningAV) AddIV 'lowTrigger 0)
(SEND ($ WarningAV) AddIV 'highTrigger 100)

STOP
```

## Index

- \$AV, 207
- <-New, 164, 183
- <-Super, 182
- <-Super?, 183
- access-oriented programming,
  - 187
- Access-Oriented
  - Programming, 16, 18
- active values*, 187
- ActiveValue
  - tracked variable*, 210
- ActiveValue, 208
- ActiveValue
  - IndirectVariable, 210
- ActiveValue
  - GetWrappedValue, 210
- ActiveValue
  - PutWrappedValue, 210
- ActiveValue
  - GetWrappedValue, 215
- ActiveValue
  - PutWrappedValue, 215
- AddActiveValue, 224
- AddCIV, 107
- AddIV, 170
- AddValue, 95
- AnnotatedValue, 121, 202
  - annotatedValue* data type, 201
- annotatedValue?**, 44
- AppendSuperValue, 220, 221
- ApplyMethod, 145
- AVPrintSource, 205
- CalledFns, 151
- class*, 23
- class variables*, 63
- Class?**, 43
- ClassInheritanceBrowser, 143
- ClassName, 153
- control structure, 251
- CopyActiveValue, 216
- CopyDeep, 156
- data-oriented programming,
  - 187
- DatedObject, 48
- DC, 74
- defensive programming*, 230
- DefineClass, 55
- DefineMethod, 128



## Medley LOOPS: The Basic System

DefMethod, 139  
DefRSM, 250  
Delete Method, 141  
DeleteActiveValue, 226  
DeleteCIV, 116  
DeleteCV, 116  
Destroy, 117  
Destroy!, 120  
DestroyClass, 118  
Directed Acyclic Graph, 35  
DoFringeMethods, 147  
DoMethod, 144  
Edit, 75  
EditMethod, 141  
EM, 76  
**ErrorOnNameConflict**, 46  
ExplicitFnActiveValue, 218  
FetchMethod, 185  
**FirstFetch**, 195  
FirstFetchAV, 197  
GetClass, 101  
GetClassHere, 101  
GetClassIV, 99  
GetClassOnly, 101  
GetClassValue, 83  
GetClassValueOnly, 97  
GetCVHere, 98  
GetIndirect, 197  
GetIt, 108  
GetItHere, 108  
GetItOnly, 108  
GetIVHere, 98  
GetLispClass, 175  
GetLocalState, 200  
GetLocalStateOnly, 201  
GetMethod, 114  
GetMethodHere, 114  
GetMethodOnly, 114  
GetSourceIVs, 68  
GetValue, 81, 201  
GetValueOnly, 97  
GetWrappedValue, 227  
GetWrappedValueOnly, 227  
GlobalNamedObjects, 48  
*handle*, 39  
*Inheritance*, 34  
InheritedValue, 222  
InheritingAV, 220  
*instance*, 23, 27  
**Instance?**, 44  
InstOf, 155  
InstOf!, 155  
IVProperty  
    initForm, 169  
IVValueMissing, 166, 167  
Lisp  
    data type, 175

## Medley LOOPS: The Basic System

- Lisp Object-Oriented Programming System, 15
- LispWindowAV, 219
- LocalStateActiveValue, 215
- LOOPS
  - @ form, 123
  - @\* form, 124
  - <-! form, 180
  - <-@ form, 124
  - <-IV form, 180
  - <-Try form, 181
  - abstract class, 174
  - AbstractClass, 31
  - class hierarchy, 30, 33
  - class record*, 68
  - class variable*, 23
  - class variables, 80
  - Classes, 70
  - CurrentEnvironment, 46
  - generic class description, 32
  - inheritance hierarchy, 70
  - inheritance network, 33, 34
  - instance variable*, 24
  - instance variables, 80
  - Instances, 70
  - LoopsHelp, 154
  - metaClass, 30
  - Metaclasses, 70
  - Methods, 113
  - name conflict resolution, 39
  - NotSetHere, 114
  - NotSetValue, 83, 99
  - NoValueFound, 83
  - property annotations, 229
  - property list, 80
  - pseudoclass, 175
  - RuleSet, 233, 234
  - superclass, 30
  - Tofu, 120
  - Variables, 23
- LOOPS Reference Manual.  
*See* LRM
- LoopsDate, 50
- LOOPSDirectory, 51
- LOOPSFILES, 51
- LoopsVersion, 50
- Macro
  - AV, 204
  - create, 203
  - fetch, 202
  - MessageNotUnderstood, 204
  - replace, 203
  - type?, 203
- McCarthy, John, 15
- MessageNotFound, 121
- MessageNotUnderstood, 121
- Metaclasses*, 173

- method, 29
- Method, 136
- Mixins, 174
- MoveMethod, 149, 150
- MoveMethodsToFile, 151
- NamedObject, 48
- New**, 70, 72, 177
- NEW, 157
- NewClass, 55
- NewInstance**, 158, 161
- NewWithValues, 163
- NotSetValue, 166, 168
- NoUpdatePermitted**, 195
- NoUpdatePermittedAV, 219
- Object?**, 42, 72
- object-oriented programming*, 187
- Object-Oriented Programming, 16, 18
- private instance variables*, 63
- Procedure-Oriented Programming, 15, 18
- PushClassValue, 93
- PushValue, 93
- PutCIVHere, 104, 105
- PutClass, 103
- PutClassIV, 100
- PutClassOnly, 103
- PutClassValue, 90
- PutClassValueOnly, 98
- PutCVHere, 104
- PutGetMethodOnly, 115
- PutIndirect, 197
- PutIt, 111
- PutItOnly, 111
- PutIVProp, 112
- PutIVValue, 112
- PutLocalState, 200
- PutLocalStateOnly, 201
- PutMethod, 115
- PutValue, 86, 201
- PutValueOnly, 98
- PutWrappedValue, 228
- PutWrappedValueOnly, 228
- RenameMethod, 148
- ReplaceActiveValue, 226
- ReplaceMe, 198
- Rule, 241
- rule-oriented programming*, 232
- Rule-Oriented Programming, 16, 19
- rules*, 232
- RuleSet, 235
- RuleSetMeta, 239
- RuleSetNode, 240
- RuleSetSource, 236, 241
- RunRS, 249

## Medley LOOPS: The Basic System

selector, 29

SetName, 46, 77

*subclasses*, 30

SuperMethodNotFound, 121

Supers, 78

*Tofu*, 78, 175

Understands, 45

WrappedPrecedence, 225

Xerox Palo Alto Research  
Center, 15