XEROX

# Artificial Intelligence Systems

## Xerox LOOPS, A Friendly Primer

Document Number 3102242

XEROX

# Loops:
# A Friendly Primer

March, 1987

# PREFACE

You are a tourist in a strange city. You would like to see the sights but you don't know where they are. You don't even know what they are. What do you do? You could look at a street map but it would have too much detail. You could pick a direction and go, hoping to run into something interesting. You could ask people on the street. What you probably would do though is buy a guide book. In it you would find just the kind of information you need to start learning about an unfamiliar city: simplified maps, descriptions of sections of the city, lists of tourist attractions, and so on.

Learning a new programming language is not unlike being lost in a strange city. Language manuals, including the *LOOPS Reference Manual*, are not meant to be used by beginners; they are intended for programmers already familiar with the language. Therefore, the information in reference manuals is not organized in a way that makes a programming language easy to learn.

This primer is the equivalent of a tourist's guide book. It shows you the "sights" but it leaves out a lot of detail. Once you are comfortable with the basic LOOPS programming concepts and procedures described here, you can use the *LOOPS Reference Manual* as it was intended and fully exploit the capabilities of LOOPS.

This primer was written with the beginner's viewpoint in mind. It addresses strategic considerations, introduces basic procedures and methods, and provides numerous examples and pictures. The material in each chapter is presented with step-by-step instructions.

While this primer does not assume you have any previous programming experience in LOOPS, it does assume you have a Xerox 1108/9 or a Xerox 1186 AI Workstation which is running the Cantilever version of LOOPS, and that you have experience with interlisp-D and its programming environment. If you are not familiar with Interlisp-D, its recommended that you start by working your way through *Interlisp-D: A Friendly Primer*. In particular, you should know how to use DEdit and how to interact with menus. If you have specific questions about Interlisp-D, look in the *Interlisp-D Reference Manual*.

Before you sit down at the computer with this primer, glance over the Table of Contents, read the first two chapters, and read the introductory statements at the top of the first page in each of the other chapters. Doing this familiarizes you with the task that lies ahead. Then, as you read this primer, actually enter the examples in each chapter. The chapters in the primer are meant to be worked through in order.

Chapters 1 and 2 provide an introduction to LOOPS. Chapter 1 introduces the concept of object-oriented programming in LOOPS. Chapter 2 is a glossary which provides an initial overview of LOOPS concepts. The glossary is also a useful reference.

Chapters 3, 4 and 5 introduce the basic information necessary for programming in LOOPS. Chapter 3 shows how to create the objects that form the basis of LOOPS programs. Chapter 4 shows how to make those objects interact with each other. Chapter 5 shows how to save LOOPS programs on files.

In Chapter 6, a LOOPS program is developed step-by-step using the concepts covered in previous chapters. After working through this example, you will be able to develop simple LOOPS programs.

Chapter 7 introduces some fundamental design strategies for organizing LOOPS programs.

The remaining chapters present more advanced topics. The material in these chapters enables you exploit the real power of the LOOPS language. Chapter 8 shows how to use specialization to add functionality to objects. Chapters 9 and 10 introduce other useful LOOPS tools -- active values and gauges. Chapter 11 covers more sophisticated uses of specialization to create LOOPS objects.

Chapter 12 demonstrates how to customize browsers. Browsers are graphical editing tools provided by LOOPS, and available for customization in your own programs. The example in this chapter also demonstrates further programming techniques.

Chapter 13 shows how to use Masterscope with LOOPS programs. Masterscope is an Interlisp-D utility for analyzing programs.

### Acknowledgments

The early inspiration and model for this primer came from the Intelligent Tutoring Systems Group of the Learning Research and Development Center (LRDC) at the University of Pittsburgh.

Lyn Ann Mears and Ted Rees of Computer Possibilities were the primary authors of this primer.

Many people from the Xerox Corporation deserve mention. The Knowledge Systems Area at the Xerox Palo Alto Research Center has been the driving force behind the development and productization of Loops since its inception. Special thanks goes to that group in general, and Danny Bobrow, Mark Stefik, Sanjay Mittal, and Stan Lanning in particular. At the Xerox Artificial Intelligence Systems, John Vittal was responsible for the primer project. Pablo Ghenis, Jairus Hihn and Joshua Stern were the primary reviewers. Rick Martin provided the technical interface to the authors.

# TABLE OF CONTENTS

# 1. INTRODUCTION - WHAT IS LOOPS?

Artificial intelligence (AI) programs must accomplish a widely varying set of tasks. For this reason, LOOPS integrates several programming styles, or paradigms, so that each part of a program can be written in a way that is best suited to the particular task it is supposed to accomplish.

The problem that an AI program is supposed to solve is often poorly understood at the start of a project. The very act of attempting to write the program leads to greater understanding and, most likely, to a redesign of the program. LOOPS facilitates this kind of *exploratory programming* by making it easy to construct and modify program elements and alter the way they interact.

The basic programming styles that LOOPS provides are described below:

*Procedure-oriented Programming*

This is the style that most widely known languages provide (eg., FORTRAN, Pascal). A procedure-oriented program consists of a set of procedures (functions, subroutines, main blocks, etc.). These procedures act upon a set of data which is (at least conceptually) separate from them. Interlisp-D is a procedural language and LOOPS is fully integrated into Interlisp-D. Any part of a program that can most profitably be written in a procedural style can be written in pure Lisp. No special steps need to be taken to access or to interface with Interlisp-D.

*Object-oriented Programming*

In this form of programming, there is an integration between functions and data. Each program element, each *object*, is a package containing some some functions, which are called *methods*, and some data, which are the values of its *variables*. In effect, each object is a specialized processor with its own private memory. The action in an object-oriented program is initiated by *message passing*. Objects send messages to other objects. Each message causes the receiving object to invoke the appropriate method to perform some operation, which often includes sending messages to other objects. Any programming problem whose solution can be viewed as a collection of similar objects that function by passing commands and results to each other is a good candidate for object-oriented programming.

*Access-oriented Programming*

In this paradigm, arbitrary actions are performed whenever a value is accessed. Access-oriented programming is very useful when certain values must be monitored or protected in some way. In a simulation, for example, the variation of the values of certain variables over time is the output of the program. These simulation variables tend to be accessed from many different places in a program making it difficult to ensure that changes are

noticed, or even appropriate. In access-oriented programming the value cannot be accessed without triggering an action because the trigger, which is called an *active value*, effectively surrounds the value.

*Inheritance*

Inheritance is an integral part of object-oriented programming. Inheritance simplifies the construction and modification of LOOPS programs. It is not necessary to construct each LOOPS object from scratch. A new object can be constructed by using an existing object as an example. Only those parts of the new object that are different from the existing object need to be specified. Whatever is the same can be inherited.

An example of inheritance is shown in Figure 1.1. The overall network of inheritance is called the *class lattice*. *Classes* are objects that describe collections of things. The solid boxes in the figure are classes. Particular members of a class are called *instances*, and they are created by using a class as a template. The dashed-line boxes in the figure are instances.

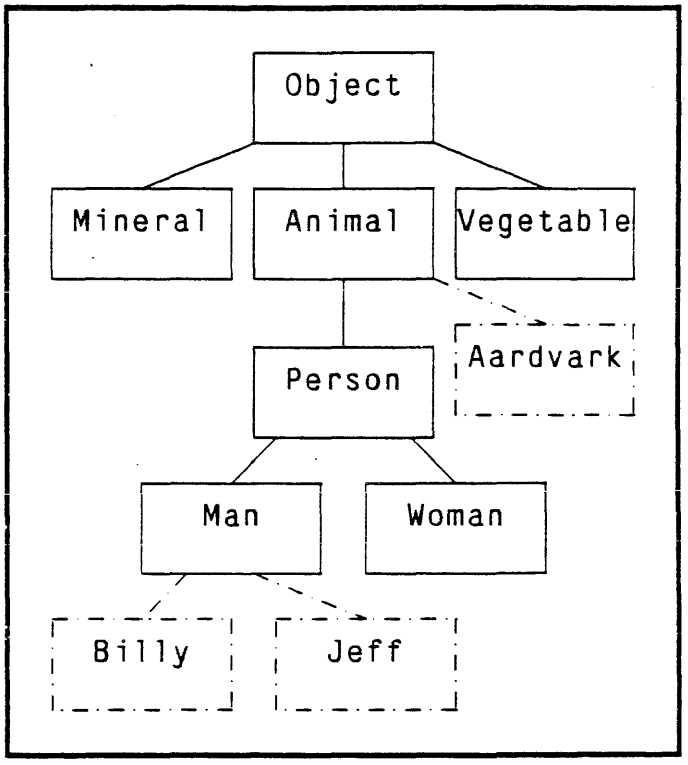



**Figure 1.1.** Figure illustrating LOOPS inheritance

The classes from which a class inherits are referred to as its *supers* (short for super classes). **OBJECT** is the most generic object and does not have any supers. Continuing with Figure 1.1, **OBJECT** is a super of **MINERAL**, **ANIMAL** and **VEGETABLE**; **ANIMAL** is a super of **PERSON**; and **PERSON** is a super of **MAN** and **WOMAN**. Every class automatically contains (inherits) all of the variables and methods of its supers, unless the new class is created with variables or methods with the same name as its supers. In that case, the local variables and methods override the ones that would have been inherited.

The notion of inheritance makes it very natural to think of an object's supers as being above it and this is the way Figure 1.1 is

drawn. However, as you will see, the LOOPS interface draws the class lattice horizontally with inheritance from left to right.

The classes below a class in the lattice are called its *subclasses* or *specializations*. In Figure 1.1, **MINERAL**, **ANIMAL**, and **VEGETABLE** are specializations of **OBJECT**; **PERSON** is a specialization of **ANIMAL**; and **MAN** and **WOMAN** are specializations of **PERSON**.

An *instance* of a class represents one specific thing. Instances are the objects that actually perform the work in a LOOPS program. All instances of a class have the same methods and variables as defined by their class. In Figure 1.1, **AARDVARK** is an instance of the class **ANIMAL** and **BILLY** and **JEFF** are instances of the class **MAN**.

Using an object-oriented language often feels unfamiliar at first. With a little experience, however, it becomes natural to think about problems in terms of communicating objects with shared behavior. Once you have mastered LOOPS, you will have at your disposal a versatile collection of modern programming tools, and you will be ready to attack complex and difficult problems.

# 2. A GLOSSARY OF TERMS

This Glossary covers basic programming elements and concepts that you will encounter in LOOPS. Examples for the terms are given in Figure 1.1, Page 1.2, and Figure 2.5.

Just skim this glossary when you first read through the primer. As you encounter new terminology in the chapters to follow, you will find this a handy reference section.

Because LOOPS concepts are interrelated, sometimes a concept is used before it has been well explained and illustrated. This glossary should help you get over the rough spots.

**access-oriented programming** — A programming paradigm in which fetching or storing data activates computations.

**active value** — The mechanism that implements access-oriented programming for variables in LOOPS. Active values can be thought of as probes placed on the variables of a LOOPS program. These probes can activate additional computations when data are fetched or stored.

**background menu** — The menu that is displayed when you click the right button and hold it while the mouse cursor is in the gray background area of the screen. A standard background menu is shown below.

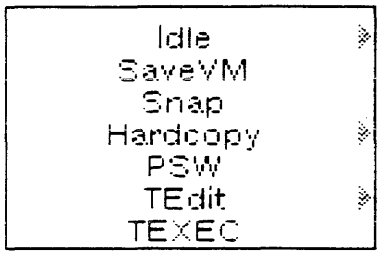



**Figure 2.1.** Example background menu

**browser** — A display that allows the user to examine, manipulate, and shift attention in a data structure. LOOPS provides browsers for the class lattice and for instances. An example of one of the browsers for the class lattice, a ClassBrowser, is shown in Figure 2.5.

**browser editing menu** — A menu accessed by pressing the middle mouse button on a class in a LOOPS class browser. The items in this menu allow you to create and edit classes and methods. (See Figure 2.NIL.)

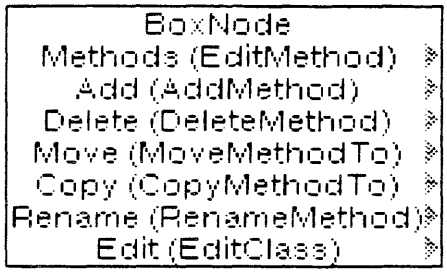



**Figure 2.2.** Browser Editing Menu

browser information menu — A menu accessed by pressing the left mouse button on a class in a LOOPS class browser. The items in this menu give information about the class lattice. (See Figure 2.NIL.)

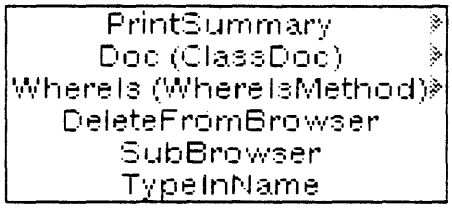



**Figure 2.3.** Browser Information Menu

browser manipulation menu — A menu accessed by pressing the left or middle mouse button in the title bar of a LOOPS class browser. The items in this menu allow you to make changes to the class lattice shown in the browser. (See Figure 2.NIL.)


**Figure 2.4.** Browser Manipulation Menu

class — A description of one or more similar objects. Classes provide a template for the objects they specify. Classes specify variables, values, and methods. In Figure 1.1, Page 1.2, **OBJECT**, **MINERAL**, **ANIMAL**, **VEGETABLE**, **PERSON**, **MAN** and **WOMAN** are all classes.

class inheritance — The means by which a class inherits variables, values, and methods from its super class. Class inheritance allows you to define a class as a specialization of another class. The newly defined class is called a "subclass" or a "specialization". The previously defined class is called a "super". A specialized class inherits much of its structure from a super. Class inheritance supports program modularity and facilitates the design process.

class lattice — The network of inheritance relations among classes. Usually class lattices in LOOPS are displayed left to right; that is, supers are to the left of their subclasses. Figure 2.5 shows the contents of Figure 1.1 as it would appear in a browser.

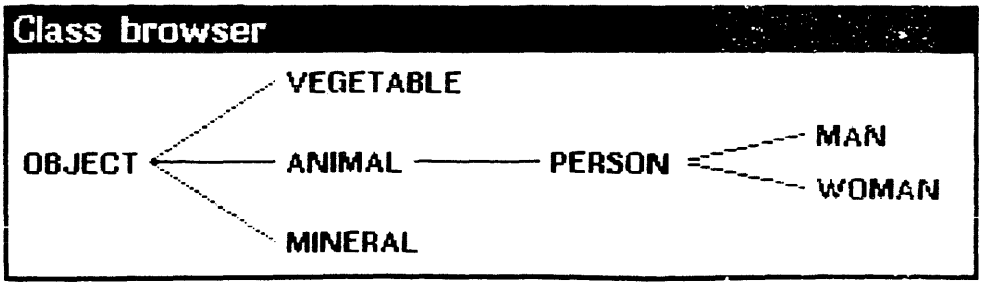



**Figure 2.5.** Standard LOOPS class lattice

class variable — The variables that store information shared by all instances of a class. For instance, in Figure 2.5, **PERSON** might have the class variables LEGS, with the value 2, and EYES, also with the value 2.

inspector — An interactive display program for examining and changing the parts of a data structure.

instance — An actual data object with its structure defined by a particular class. For example in Figure 1.1 **BILLY** is an instance of **MAN**. Note that instances do not give rise to further specializations within the lattice structure. Instances within a class share the same methods, class variables, and class variable values. All instances of a class have the same instance variables, but the values of these instance values may differ. These differing values will cause each instance to respond differently than its siblings, even though the instances are from the same class.

In Figure 1.1, **BILLY** and **JEFF** are instances of the class **MAN**, and **AARDVARK** is an instance of the class **ANIMAL**. **BILLY** inherits all variables and methods from **MAN**, as well as all variables and methods from **PERSON**, **ANIMAL** and **OBJECT**. However, **BILLY** and **JEFF** may have different values for their instance variables.

instance variable — A variable used to store information specific to an instance and, therefore, a "local variable" for that instance. Instance variables are defined in classes. When you specify an instance variable in a class, you assign to it a default value. This value is inherited as the default value down through the class lattice structure. For example, in Figure 2.5, the class, **PERSON**, might have the instance variable, **HATSIZE**, with default value 7. The class **MAN** would inherit this instance variable with the default value 7 and pass it on to **BILLY**. Each instance has its own copy of the instance variables, and the instance variable values can be changed independent of the values for other instances of the same class. So, for example, the instance **BILLY** can have 7.25 as the value of {HATSIZE} while the instance of **JEFF** has a **HATSIZE** of 7.75.

Interlisp-D executive window — The window in which Interlisp-D functions are entered.

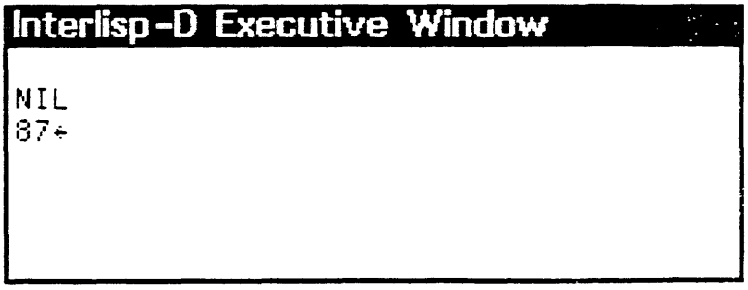



**Figure 2.6.** Top Level Interlisp-D executive window

Left buttoning inside the Top Level Interlisp-D Executive Window causes the type-in cursor to appear in the window.

join — A class with multiple specialization branches. See "*left to right, up to joins.*"

lattice — A directed graph without cycles. In LOOPS, the inheritance network is arranged in a lattice. While a tree allows each node to have only one parent, a lattice allows multiple parents (in

LOOPS, multiple supers). A lattice does not allow a class to have itself as a super class or ancestor class.

left buttoning — Pressing the left mouse button and then releasing it.

"left to right, up to joins" — The rule for inheritance in a lattice of objects. Each branch up the class lattice is searched, starting with the leftmost branch and working right. A class with several specialization branches is not searched until all the specialization branches have been searched. A class with multiple specialization branches is referred to as a join.

Masterscope — A program analysis tool. When told to analyze a program, Masterscope creates a data base of information about the program. In particular, Masterscope knows which functions call other functions and which functions use which variables. Masterscope can then answer questions about the program and display the information with a browser.

menu — A way of graphically presenting a set of options. There are two kinds of menus: pop up menus are created when needed and disappear after an item has been selected; permanent menus remain on the screen after use.

message — A command to an object to do something. A message activates a method defined in an object's class. For example, if **BILLY** has a method, **BRAG**, you can send a **BRAG** message to **BILLY** and have **BILLY** execute the code associated with the **BRAG** method.

method — The code stored in objects. Each method performs the actions needed to implement a particular message. A subclass inherits its super's methods.

middle buttoning — Pressing the middle mouse button and then releasing it. If your mouse does not have a middle button then press both the left and right mouse buttons together.

mixin — A class used to add some particular functionality to many other kinds of classes. Usually a mixin is a second super for a class. Mixins rarely have their own instances.

mouse — The little rectangular box connected to the computer. There are two or three buttons located on the top of the mouse. The buttons are referred to as the left, middle and right mouse buttons. Pressing the left and right button at the same time will simulate the middle button if your mouse does not have one.

mouse cursor — The small arrow on the screen that points to the northwest.

**Figure 2.7.** The mouse cursor

The mouse cursor moves as you move the mouse.

object — The main structures in object-oriented programming. They combine aspects of procedures -- for computation, -- and data, -- to describe their state. Classes and instances are both objects.

object-oriented programming — A programming paradigm in which structures are designed that contain both data and the methods for manipulating that data.

procedure-oriented programming — A programming paradigm in which programs are composed of functions and procedures. Data structures are separate objects

that get passed to functions and procedures. This is the best known programming paradigm and is supported in standard programming languages like FORTRAN and Pascal.

prompt window — The skinny black window at the top of the screen.

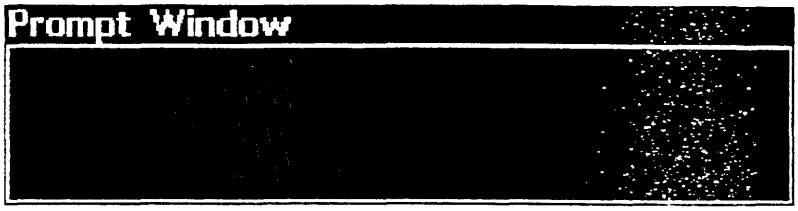



**Figure 2.8.** Prompt window

right buttoning — Pressing the right mouse button and then releasing it.

self — A method argument that represents the receiver of the message. All methods contain the argument self. Self is automatically bound to the object which received the message that invoked the method. Methods use self in order to access the variables and other methods of the object defining the method.

specialization — The process of creating a subclass from a class; or, the result of that process. In Figure 1.1, **MINERAL**, **ANIMAL**, and **VEGETABLE** are specializations of **OBJECT**; **PERSON** is a specialization of **ANIMAL**; and **MAN** and **WOMAN** are specializations of **PERSON**.

super — A class from which a given class inherits. In Figure 2.5, **OBJECT** is the super of **MINERAL**, **ANIMAL**, and **VEGETABLE**; **ANIMAL** is the super of **PERSON** and **AARDVARK**; **PERSON** is the super of **MAN** and **WOMAN**.

TEdit menu — Refers to the menu that is displayed when you middle hold button while pointing the mouse cursor at the title bar of a TEdit window.

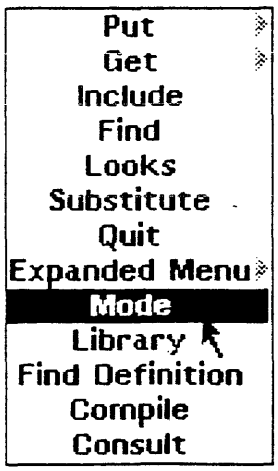



**Figure 2.9.** Example TEdit menu

window manipulation menu — A menu accessed by pressing the right mouse button in the title bar of a LOOPS browser. The items in this menu manipulate the window containing the browser. Figure 2.NiL shows the standard Window Manipulation Menu.

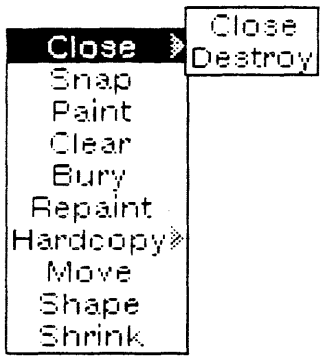



**Figure 2.10.** Window Manipulation Menu

# 3. CLASSES AND INSTANCES

This chapter introduces the steps for building a simple class lattice. You define classes by inserting documentation, class variables, default class variable values, instance variables and default instance variable values. You define subclasses and create instances of a class.

After you have become familiar with how classes inherit information from their supers, you learn how to manipulate this inheritance structure.

The easiest way to develop LOOPS programs is by using a browser. The browser displays the class lattice and provides menus of commands for building and manipulating this structure.

## 3.1 Creating a Class

Begin by getting LOOPS running on your Xerox AI workstation. LOOPS is generally installed in the form of a sysout because loading the individual files is very time-consuming. Xerox provides a sysout in the LOOPS software kit. If your machine is on a network, you should consult your local system administrator to find out where LOOPS is stored. If you have a stand-alone machine, you should have LOOPS on a series of floppy disks. A LOOPS sysout is installed using the same process used to install an Interlisp-D sysout. If you do not know how to do this, please refer to *Interlisp-D: A Friendly Primer* or the *User's Guide* that came with your machine.

Once you have a LOOPS sysout running, you should see the LOOPS icon, Figure 3.2. If you do not see this icon, you must bring it up by using the background menu. To bring up the background menu, hold the right mouse button while the cursor is in the grey background area of your screen. Select the phrase **Loops Icon** off the menu that pops up and release the mouse button. If you like, you can move the icon to a different location by selecting the **Move** option from the icon's right-button menu.

Before beginning to use the browser, you need to create a root for the class lattice structure. For our example, the root is the class **Animal**. Create this root by typing:

```
(DefineClass 'Animal)
```

```
Top level -- Connected to {DSK}<LISPFILE

10←(DefineClass 'Animal)
#.($ Animal)
11←
```

**Figure 3.1.** Creating the Root Class

As in Interlisp-D, LOOPS distinguishes upper and lower case letters. Thus, you should be sure to type things exactly as you see them. Also, you should notice that **Animal** is quoted so that it is not evaluated. **DefineClass** returns a pointer to the class it has just created. Such pointers are printed by the system as **#.($ ClassName)**, as you can see in Figure 3.1.

In order to begin working in a class browser, position the mouse cursor on the LOOPS icon, press the left mouse button, and select **Browse Class** as shown in Figure 3.2.

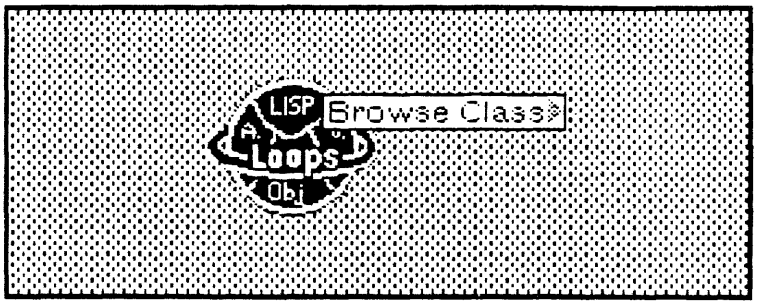



**Figure 3.2.** Accessing LOOPS Browser

When you see the prompt in the prompt window, type in the root object you wish to browse. In this case you should type **Animal**. A ghost image of a browser window appears near the cursor on the screen. You may position the window by moving the mouse cursor and pressing the left mouse button when the window is in the correct place. The result should be a browser window as shown in Figure 3.3.


**Figure 3.3.** Browser for the class **Animal**

## 3.2 Editing a Class

After creating a class, you need to add the following to it:

- documentation
- class variables
- instance variables
- methods

In this chapter, you learn how to add the first three. You learn about methods in Section 4.2.

### 3.2.1 Using the Browser Editing Menu

Each item in a class browser has two menus associated with it. One, presented later, contains informational commands. The other one, presented now, allows you to make alterations to classes and the class hierarchy. To access this menu, move the cursor over **Animal** and click the middle mouse button. The menu appears and remains visible when you release the button (see Figure 3.4). The first word of each item in the menu indicates the types of operations that are contained in its submenu. The part in parentheses indicates the command that results from selecting the main menu item. The submenu is accessed by pressing the left mouse button and sliding the cursor to the right over the grey arrow while continuing to hold the button down. Items are selected from the submenu by moving the mouse cursor until the desired item is hi-lighted and releasing the mouse button.

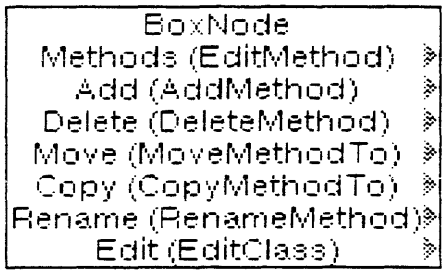



**Figure 3.4.** Browser Editing Menu

Class definitions are edited with DEdit the same way that functions and other entities are edited in Interlisp-D. To call DEdit, select **Edit(EditClass)**. A **DEdit** window opens with the skeleton class definition as shown below in Figure 3.5.

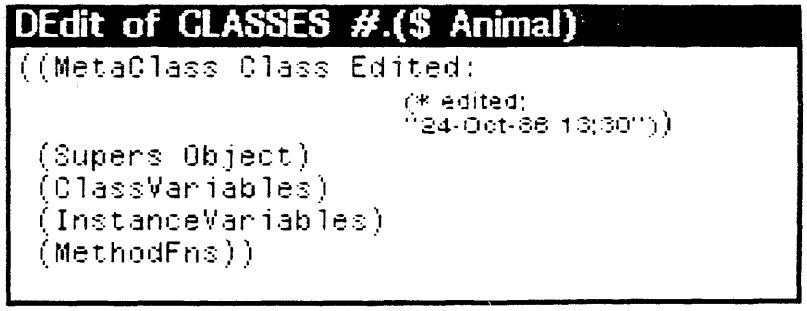



**Figure 3.5.** Editing of the class **Animal**

Our figures do not show the DEdit command menu. If this menu does not appear to the right of the DEdit window, move the cursor into the window and click the left button.

To preview the complete **Animal** class definition, look at Figure 3.7.

### 3.2.2 Documenting the Class

Effective documentation is just as important in LOOPS as in any other programming language. Both the class itself and the items within it can be documented. Each item of documentation consists of the symbol **doc** followed by a standard Interlisp-D comment.

(Note: **doc** is a property name, and your documentation is the value of the **doc** property.)

To document **Animal**, add the following:

```
doc (* definition of root object, Animal)
```

in between **Class** and **Edited**. You can save yourself a couple of keystrokes by enclosing **doc** and the comment in a list, inserting the list and then removing the parentheses. If you are unsure of how to do this operation, you should refer to *Interlisp-D: A Friendly Primer* and practice using DEdit before you continue. You will be making extensive use of DEdit throughout this primer. The edit session should now look like Figure 3.6.

```
DEdit of CLASSES #.($ Animal)
((MetaClass Class doc          (* definition of root
                                object, Animal)
        Edited:                (* edited:
                                "24-Oct-86 13:30"))
 (Supers Object)
 (ClassVariables)
 (InstanceVariables)
 (MethodFns))
```

**Figure 3.6.** Adding documentation to **Animal**

### 3.2.3 Inserting Class Variables, Values, and Properties

Now you can put in the class variables. They are inserted in the **ClassVariables** list. Each class variable is specified by giving its name, its default value, and its property names and their values. **Doc** should be the last property name:

*(VariableName Value Property1 Value1 Property2 Value2 ...*
**doc (* comment))**

In our example, we use the class variables **HasEyes** and **IsLiving**. Both of these variables should have the default value T in the class, **Animal**. You should add the following after **ClassVariables**:

```
(HasEyes T doc (* all animals have eyes))
(IsLiving T doc (* all animals are living))
```

### 3.2.4 Inserting Instance Variables, Values, and Properties

Now put in the instance variables. Instance variables have the same format as class variables and are inserted in the

**InstanceVariables** list. For our example, we use the instance variables **DateOfBirth** and **HeartRate**. Because the values of these variables are known only if we know which individual animal, that is, which instance, is referred to, we use the default value of 0 for both. Add the following after **InstanceVariables**:

```
(DateOfBirth 0 doc (* animals do not all have
the same birthday))
(HeartRate 0 doc (* different animals have
different heartrates))
```

When you are finished, your edit session should look like Figure 3.7.

```
DEdit of CLASSES #.($ Animal)
((MetaClass Class doc          (* definition of root
                                object, Animal)
        Edited:                (* edited:
                                "24-Oct-86 13:57"))
 (Supers Object)
 (ClassVariables (HasEyes T doc
                                (* all animals have
                                eyes))
  (IsLiving T doc               (* all animals are
                                living)))
 (InstanceVariables (DateOfBirth 0 doc
                                (* animals do not all
                                have the same
                                birthday))
  (HeartRate 0 doc              (* different animals
                                have different
                                heartrates)))
 (MethodFns))
```

**Figure 3.7.** Editing in variables for **Animal**

Exit **DEdit** as you normally would by selecting **Exit** from the edit command menu. If there is an error in syntax, such as omitting a **doc** before a comment, **DEdit** gives you some information in the prompt window. DEdit allows you to exit only after you have corrected all syntax errors.

## 3.2.5 Using the Browser Information Menu

In Section 3.2.1 the middle mouse menu is used to change a class definition. The left mouse menu contains informational commands. Bring up this menu by moving the cursor to **Animal** and clicking the left mouse button. As before, the menu appears and remains after you release the button. (See Figure 3.8.)

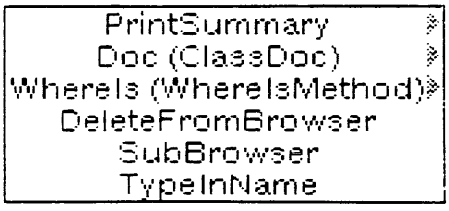



**Figure 3.8.** Information Menu

Selecting **PrintSummary** causes a summary of **Animal** to be printed in another window, as shown in Figure 3.9.

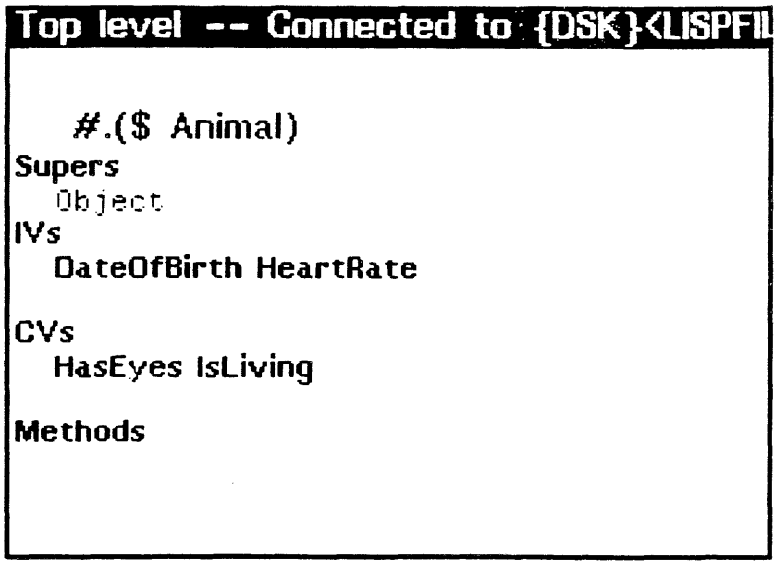



**Figure 3.9.** Summary of the class **Animal**

**PrintSummary** gives a high level summary of a class definition. The easiest way to see the structure in detail -- including values, property names and property values -- is by calling the editor. Now try bringing up the information menu again and selecting **Doc(ClassDoc)**. Any documentation you added is printed out.

## 3.3 Creating Subclasses

To continue developing our example, we specialize **Animal** to create a subclass, **Person**. We then add two subclasses, **Man** and **Woman** to **Person**.

To create the first subclass, bring up the editing menu and select **SpecializeClass** from the submenu of **Add(AddMethod)**. Then, type **Person** when you are prompted for the new subclass of **Animal** in the prompt window. The browser is automatically updated and looks like Figure 3.10.


**Figure 3.10.** Browser automatically updated to include **Person**

In order to define **Person**, repeat the same steps you followed for **Animal**. If you are unsure of the procedure, refer back to Section 3.2. When entering **DEdit**, notice that **Person**'s super was automatically set to **Animal**.

Add the following class variables and values:

- Legs 2
- Mammal T

and the following instance variables and values:

- Hatsize 7
- HairColor Brown

Note that 2, T, 7, and Brown are merely default values. Class variables can be changed by the actions of any instance. Similarly, instance variables can be set to appropriate values in each instance. Remember to add documentation to the class and its variables. When you are finished, your edit session should look like Figure 3.11 below.

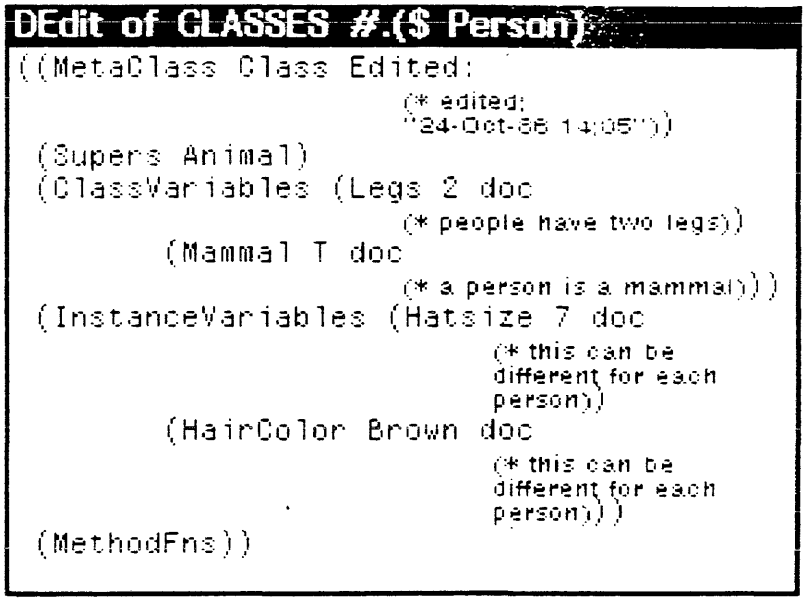



**Figure 3.11.** Editing in variables for **Person**

Now bring up the browser information menu for **Person** and select **PrintSummary**. You should notice a difference from the last time you printed a summary as in Figure 3.12.

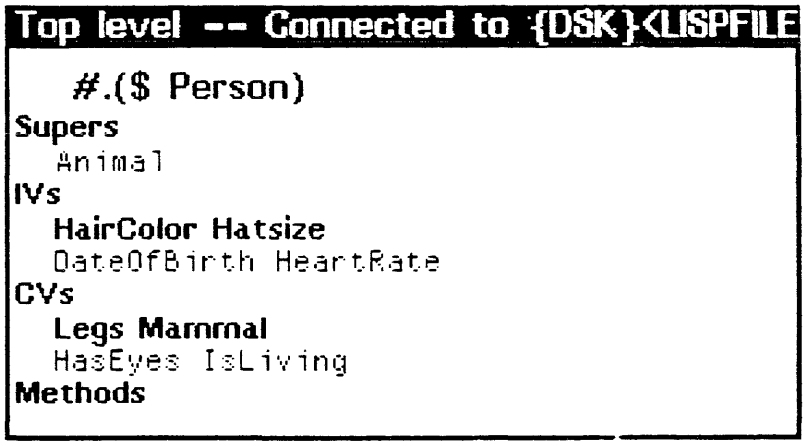



**Figure 3.12.** All variables and values for **Person**

Items which are defined in **Person** are printed in boldface while the items which are inherited from **Animal** are printed in a regular font.

Now create two specializations of **Person**: **Man** and **Woman**. When done, your class browser should look like Figure 3.13.

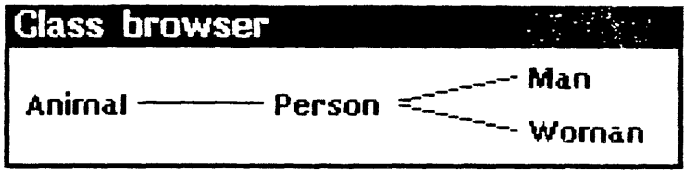



**Figure 3.13.** Browser with the classes **Man** and **Woman** added

Since all men are referred to by the pronoun, "he", give **Man** the class variable **Pronoun** with the value **He**.

Because individual men may or may not have beards and big muscles, the instance variables are **Muscles** and **Beard**, with the default values Big and T respectively.

## 3.4 Creating Instances

To create an instance of man, which we call **Billy**, type (at the top level):

**(← ($ Man) New 'Billy)**

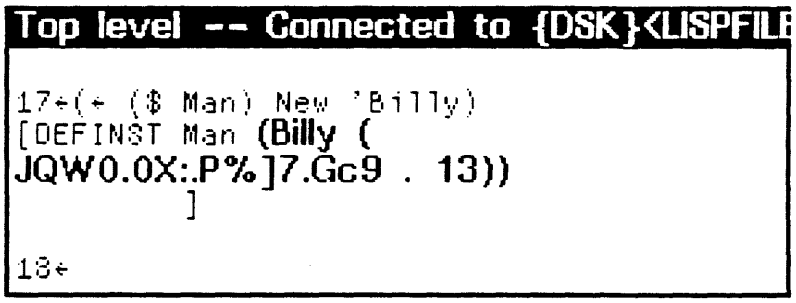



**Figure 3.14.** Creating an instance

Note that a pointer to the instance is returned just as when a class is created. What you see on your screen is somewhat different from Figure 3.14 because LOOPS creates a unique identifier for each instance. This identifier is the "JQWO...." gibberish after **Billy**. Unique identifiers ensure that different instances are not inadvertently confused with one another.

The **($ name)** notation is the way to reference classes and instances. Essentially, the **$** informs the system to use the LOOPS object with the specified name. The ← means *send a message*. **New** is a message which tells the class **Man** to create an instance named **Billy**. (Sending messages is discussed in Section 4.2.) Create a second instance of **Man**, with the name **Jeff**, by typing:

**(← ($ Man) New 'Jeff)**

Instances do not appear in the class browser window as part of the class lattice. However, the LOOPS inspector does allow the display of instances. The LOOPS inspector is a specialized version of the Interlisp-D Inspector. (See the *Interlisp-D Reference Manual* or *Interlisp-D: A Friendly Primer* for more information on the standard inspector.)

### 3.4.1 Inspecting an Instance

Type:

```
(INSPECT ($ Billy))
```

to get an inspector window for the instance **Billy**. This inspector window is shown in Figure 3.15.

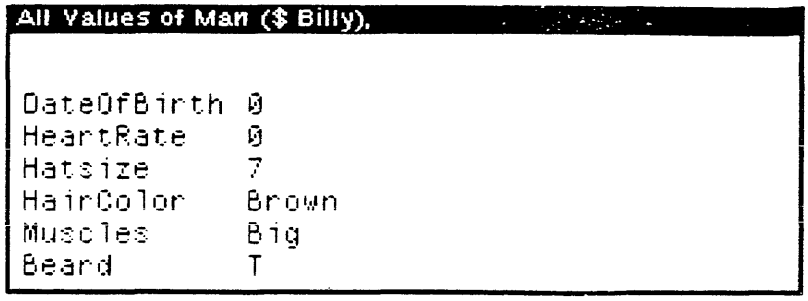



**Figure 3.15.** Inspecting **Billy**, an instance of **Man**

The inspector shows the structure of an instance. It also provides an easy way to alter the values of instance variables. Notice that the title bar says *All Values* and that the inherited instance values from **Animal**, **Person** and **Man** are present.

Create an inspector window for the instance **Jeff** in the same way. The result is an inspector window as shown in Figure 3.16.

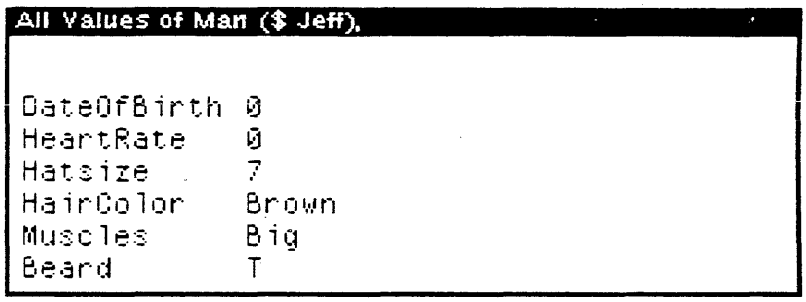



**Figure 3.16.** Inspecting **Jeff**, an instance of **Man**

### 3.4.2 Changing Instance Variable Values With the Instance Inspector

You can use the inspector to change instance variable values. Begin by pressing the middle mouse button in the title bar of the inspector window for **Billy** and holding it down. The inspector menu is displayed as shown in Figure 3.17.

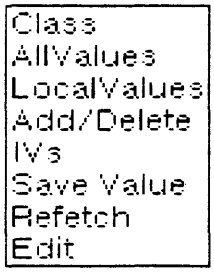



**Figure 3.17.** Inspector menu

Choose **LocalValues** by moving the mouse cursor over it and releasing the mouse button. The displayed values are changed as shown below in Figure 3.18.

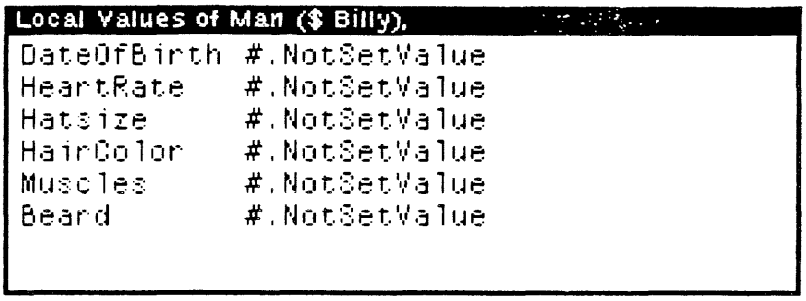



**Figure 3.18.** Local values of the instance **Billy**

The **#.NotSetValue** indicates that you have not set any local values in the instance; all of these values are defaults inherited from the super classes.

To illustrate the inspector changing values, we will alter **Billy's** HairColor to Blond. Begin by selecting the item to be changed by clicking the left mouse button over HairColor. Next bring up a command menu by holding down the middle mouse button (with the cursor inside the inspector window) and select **PutValue** from that menu. Type **'Blond** in the prompt window. This new value will be displayed. It is necessary to quote Blond because values which are entered with the Inspector are evaluated.

Now choose **AllValues** from the Inspector menu and notice that the default values are redisplayed. However, HairColor is now the local value, **Blond**, as shown in Figure 3.19.

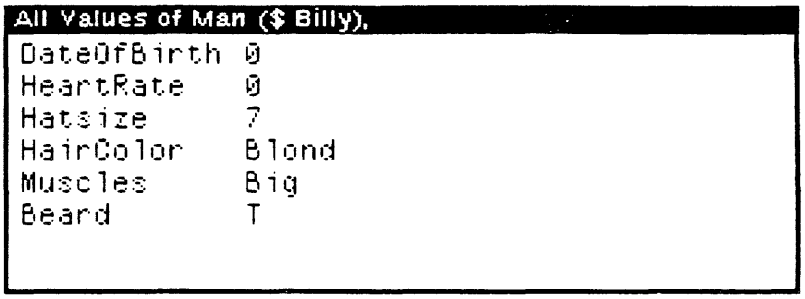



**Figure 3.19.** Inspector window of the instance **Billy** showing all values

## 3.5 Altering the Structure or the Class Lattice

It is possible to alter an existing class lattice by using the editing menu of the class browser. Classes can be moved in the hierarchy or removed completely. In order to make the browser look simpler, a class can also be removed from a browser without actually removing it from the lattice.

### 3.5.1 Moving a Class

To move a class in the class lattice you must change its super. As an example, you will move **Man** so that it is directly below

**Animal** in the lattice. The first step is to select the new super by "boxing" it. To do this, select **BoxNode** from the editing menu (middle button) on **Animal**. Then bring up the editing menu on the object to be moved, in this case **Man**. Select **MoveSuperTo** from the submenu of **Move(MoveMethodTo)**. A pop up menu appears which shows the current super, in this case, **Person**. To confirm the move, select this item by clicking the left button. After doing this, your browser should look Figure 3.20.

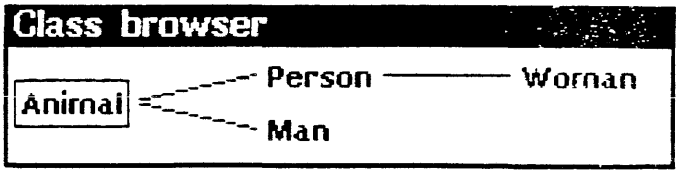



**Figure 3.20.** New lattice with **Man's** super changed to **Animal**

Since we don't really want the lattice to look like Figure 3.20, change **Man's** super back to **Person**.

## 3.5.2 Deleting and Restoring a Class from a Browser

You may want to remove a class you are not working with from the browser window. To remove **Man** from the browser window, bring up the information menu by clicking the left button on **Man**. Selecting **DeleteFromBrowser** causes **Man** to be deleted from the browser. The lattice in the browser window looks like Figure 3.21.

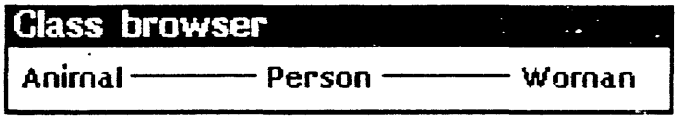



**Figure 3.21.** Browser with **Man** deleted

The class, **Man**, still exists; it simply does not appear in the browser. Note that deleting a super class from the browser will delete the class along with all of its subclasses.

Classes that have previously been deleted from a browser can be brought back. To do so, move the cursor into the browser's title bar and hold down either the left or middle mouse button.


**Figure 3.22.** Browser Manipulation Menu

Select **RemoveFromBadList** from the submenu of **AddRoot**, as shown in Figure 3.23.


**Figure 3.23.** **AddRoot** sub-menu

A pop up menu containing the items that have been deleted from the browser then appears as shown below in Figure 3.24.


**Figure 3.24.** Pop up menu for **RemoveFromBadList**

Select **Man** and it reappears in the browser.

### 3.5.3 Destroying a Class

A class may also be destroyed. That is, the class can be completely deleted from your LOOPS environment.

Create a subclass **Insect** which is a specialization of **Animal** (as explained in Section 3.3). **Insect** will be used to demonstrate how to destroy a class.

To destroy **Insect**, bring up the editing menu (middle button) on the class **Insect**. Select **DeleteClass** from the **Delete(DeleteMethod)** submenu. You must confirm before any class is actually destroyed. Confirmation is accomplished by using the pop up menu shown in Figure 3.25.


**Figure 3.25.** Pop up menu to confirm destruction of **Insect**

Select **Destroy Insect** to confirm that you wish to destroy the class. If you decide not to destroy the class, click any mouse button with the cursor outside of the pop-up menu. The lattice in the browser window now looks like Figure 3.21 again.

It is an error to attempt to destroy a class which has subclasses, since a subclass can not exist if its super does not exist. Such an attempt puts you in an Interlisp-D break window. If you type **OK** inside the break window, the class and all of its subclasses are destroyed. If you type ↑ in the break window the operation is aborted.

## 3.6 Destroying and Shrinking Browsers

If you are finished using a particular browser and want to get rid of it, you can destroy it. To do so, hold the right mouse button while in the browser window's title bar (see Figure 3.26).

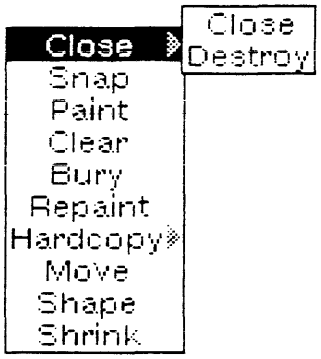



**Figure 3.26.** Window Manipulation Menu

The items in this menu are similar to those in the Interlisp-D window manipulation menu except for the **Close** item. Selecting **Destroy** from the submenu of **Close** closes the window and destroys the browser. If you select **Close**, the window closes, but the browser still exists and take up memory space. Since a browser whose window is closed is not easily accessible, it is usually better to destroy it.

Browser windows can also be shrunk using the window manipulation menu. Lattice browsers shrink to icons with the name of the root class as their title. Bring up the window manipulation menu on the example browser and select **Shrink**. The result is an icon as shown in Figure 3.27. As with all window icons, it can be expanded by positioning the mouse over it and clicking the middle button. It can be moved by positioning the mouse over it, holding down the left button, moving the icon to the desired spot and releasing the button.


**Figure 3.27.** Icon for Browser window of **Animal**

## 3.7 A Word about Notation

LOOPS uses several different notations to refer to classes and instances.

**Animal** When we refer to a class or an instance in the text we refer to it by its name, that is, the name it was given when it was created. If there is some possibility of confusion, we also state explicitly that we are referring to a class or an instance.

**($ Animal)** This is the way classes or instances are referred to in LOOPS code. The **$** causes the system to find and return a pointer to the internal data structure which embodies the class or instance.

**#.($ Animal)** This is the way the system prints out a class. For example, this is what you see if you type **($ Animal)** at the top level read-eval-print loop.

# 10. GAUGES: ACTIVE VALUES AND OBJECT HIERARCHIES IN ACTION

<NOTE TO XEROX: AS INSTRUCTED, WE HAVE NOT UPDATED THIS CHAPTER BECAUSE OF CONTINUING WORK ON GAUGES.>

In normal life we use gauges to track specific values. Typically, we use gauges where it is important to continually monitor a value. LOOPS provides a set of tools, called *Gauges*, which emulate those real life gauges we are familiar with. They are defined as LOOPS classes with active values providing the continuous monitoring. The class inheritance lattice for gauges, shown in Figure 10.1, shows how all of the sub-classes of **Gauge** are related. This structure is a combination of elision through inheritance and incremental specialization. (See Chapter 6). Notice that classes like **DigiMeter** and **DigiScale** have multiple supers.

There are two types of gauges: analog and digital. Analog gauges register changes in the value in a pictorial form, without registering the exact value. The **HorizontalScale**, **Meter**, **Dial**, **VerticalScale**, **HBarChart**, and **BarChart** as shown in Figure 10.2, are all examples of analog gauges. Digital gauges do provide the precise value, as shown by **LCD**, **DigiMeter**, and **DigiScale** in Figure 10.2.

Gauges are defined so that when they are attached to a value within your program, that value becomes an active value which has no effect on the program using that value. A change in the value causes the reading on the gauge to change. When they are detached from a value the value returns to its original state. LOOPS provides a simple way to incorporate instances of gauges into existing programs.

Gauges are very useful tools in their existing form, but they can also be customized. Existing classes in the gauge class lattice can be specialized and new classes added to the lattice.

In the following pages, we discuss basic use of LOOPS gauges. Chapter 9 shows some examples of customizing gauges.

If you are working in a new programming environment, load the file containing the Bank Account Example from Chapter 5 so that you may continue to work with the example in this chapter.

## 10.1 Object Hierarchies

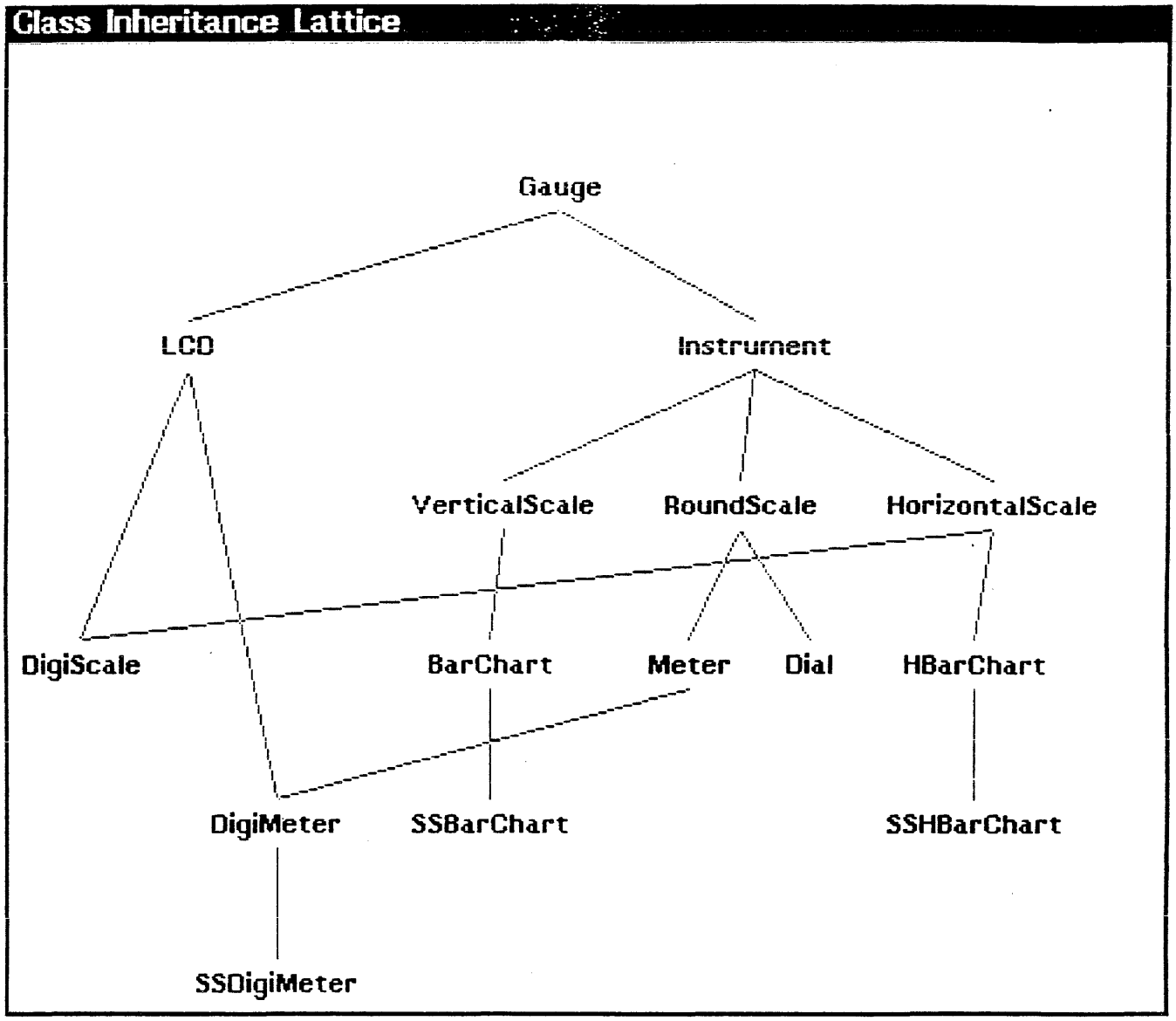



**Figure 10.1.** Class inheritance lattice for gauges

To view the class inheritance lattice for gauges, type:

```
(Browse $Gauge)
```

## 10.2 Examples of Gauges

Figure 10.2 shows some of the gauges available for you to use. If you would like to see a gauge, create an instance by typing:

```
(← gaugeClass New 'MyGauge)
```

Then, display the gauge by typing:

```
(← 'MyGauge Update)
```

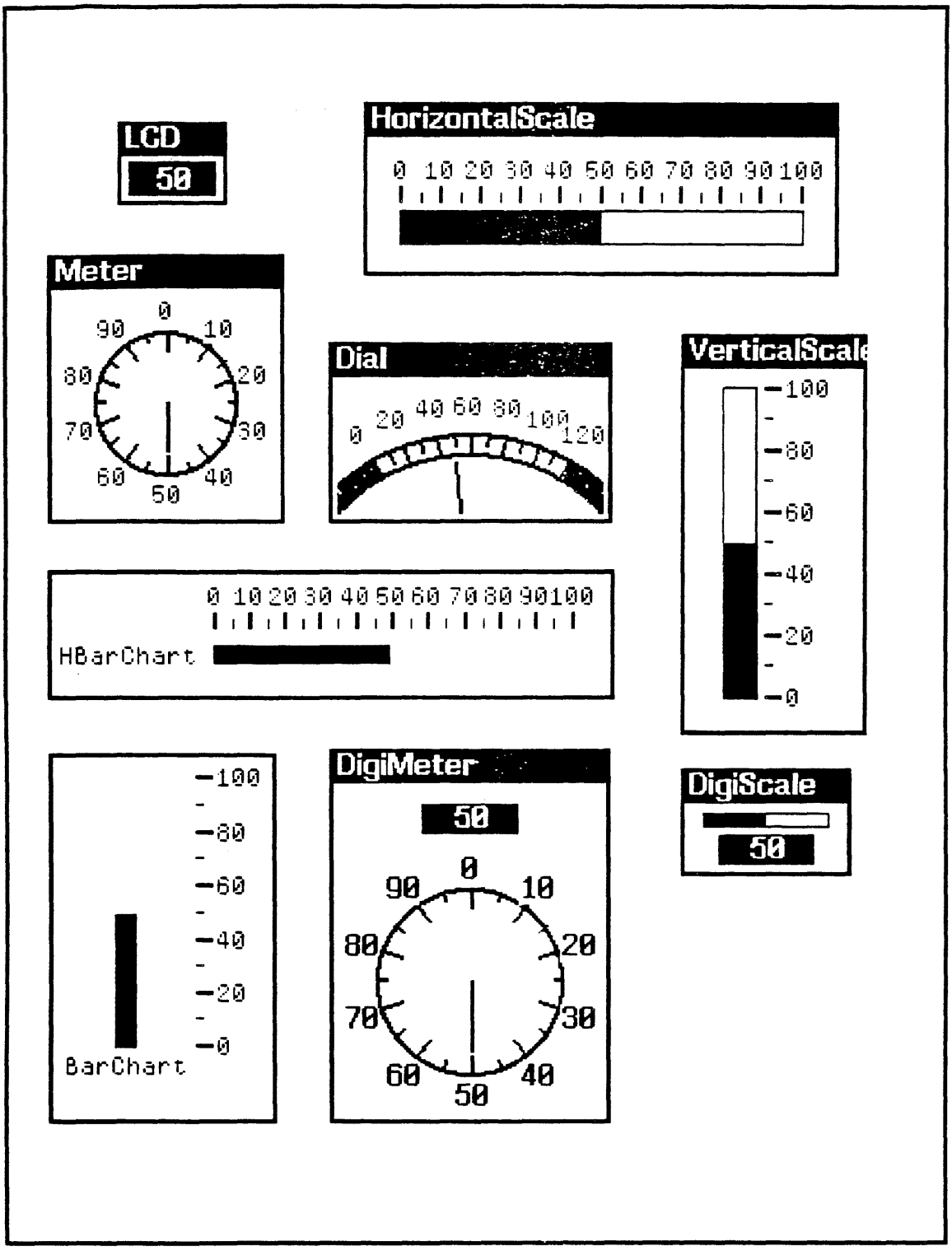



**Figure 10.2.** Some examples of gauges

## 10.3 Create Gauge Instances

You must create instances of the gauge classes to use in your programs. To see how the gauge class, **VerticalScale**, behaves create an instance of the class **VerticalScale**, named **MyVS** by typing:

```
(← $VerticalScale New 'MyVS)
```

To see the gauge send it the **Update** message by typing:

```
(← $MyVS Update)
```

A gauge will appear like the one shown in Figure 10.3.

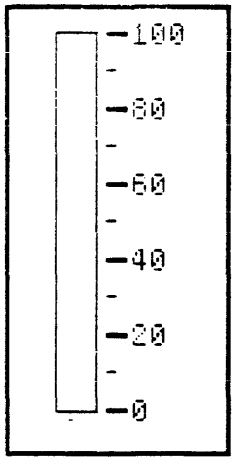



Figure 10.3. **VerticalScale** gauge

Create an instance of the class **Dial**, **MyDial**, by typing:

```
(← $Dial New 'MyDial)
```

Send it the **Update** message to display it by typing:

```
(← $MyDial Update)
```

It will look like Figure 10.4.

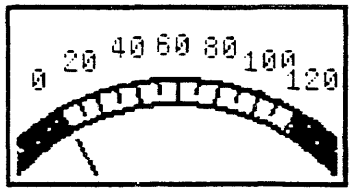



Figure 10.4. **Dial** gauge

Send both gauges the **Set** message with the argument, 50, by typing:

```
(← $MyVS Set 50)
(← $MyDial Set 50)
```

Notice that they are set as shown in Figure 10.5.

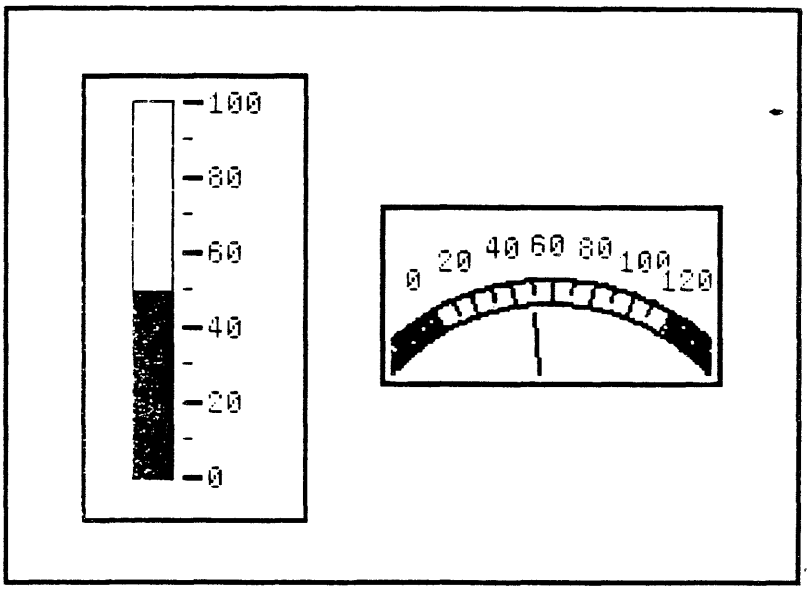



**Figure 10.5.** Setting gauges to value of 50

Try to set both of the gauges to 200. They will appear as in Figure 10.6. Notice the question marks in the upper left corners show that the value is off of the scale. We will show you how to change the scale below.

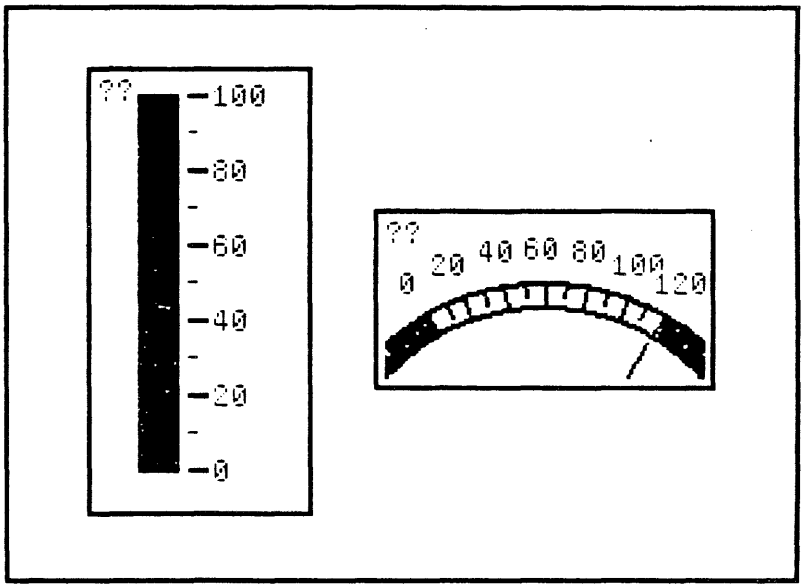



**Figure 10.6.** Gauges set above their scales

Create instances of some of the other types of gauges and experiment to become familiar with them.

## 10.4 Attaching Gauges

In this section, we discuss how to attach different gauges to the instance variable, Balance, from our bank account example.

## 10.4.1 VerticalScale

First, create an instance of **Savings**, **MySavings** by typing:

```
(← $Savings New 'MySavings)
```

Now, you can attach **MyVS** to the Balance instance variable of **MySavings**. The syntax is:

```
(← gaugeInstance Attach object varName)
```

*gaugeInstance* The name of the instance of a class of gauges you wish to use.

**Attach** Message sent to gaugeInstance.

*object* Object you wish the gauge to be attached to.

*varName* Variable in object that you want the gauge to display.

So type:

```
(← $MyVS Attach $MySavings 'Balance)
```

The **VerticalScale** will then appear as a ghost image prompting you to position it. Move the mouse cursor to a clear space on your screen and click the left mouse button to place the gauge there. Notice that it now has a title, *Balance*, which is the instance variable it is attached to.

When a gauge is attached to a value, that value becomes an active value. Inspect the instance, **MySavings**, by typing:

```
(INSPECT $MySavings)
```

You will see that the value of the instance variable, Balance, has changed to an active value, as shown in Figure 10.7.

**All Values of Savings $MySavings.**

```
CreditHistory  NIL
DebitHistory   NIL
Balance        #(50 NIL SendAVMessage)
User           "LYN"
DateOpened     "16-Jun-86 08:59:35"
TimeOpened     08:59:35
InterestRate   .05
```

**Figure 10.7.** Instance variable, Balance, as an active value after a gauge is attached to it

## 10.4.2 Dial

Attach **MyDial** to the Balance of **MySavings** following the same procedure as above.

Since you hope to have more than $100.00 in your account, change the scale on each of the gauges. The **SetScale** message sets the range on most gauges. Here we use it to set a scale from 0 to 10000 for **MyVS** and **MyDial**. To do this, type:

```
(← $MyVS SetScale 0 10000)

(← $MyDial SetScale 0 10000)
```

(Note: The complete set of messages available for use with gauges are listed in the *LOOPS Reference Manual*.)

The gauges will appear as shown in Figure 10.8. Notice the multiplication factor in the lower left corner of each gauge.

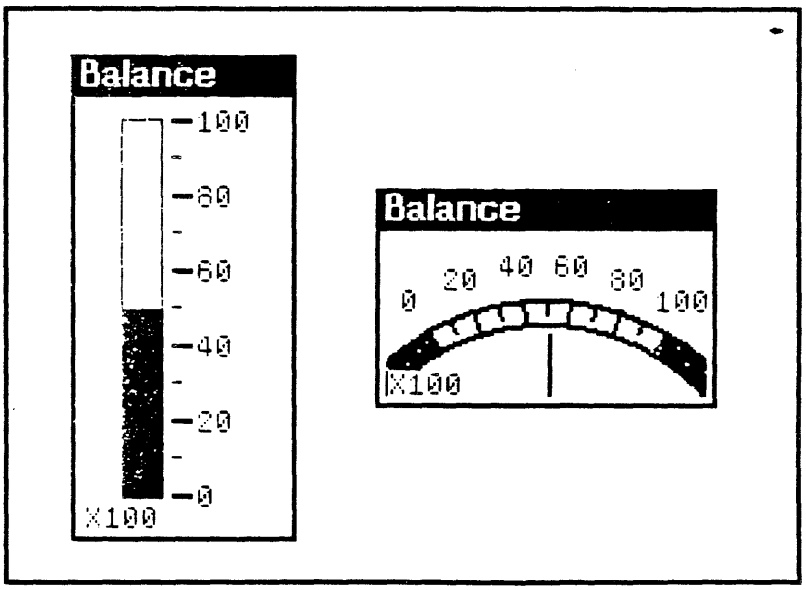



**Figure 10.8.** Result of changing scales on gauges

Send the **Credit** and **Debit** messages to **MySavings** so you can see how the **VerticalScale** and the **Dial** behave. Do this by typing:

```
(← $MySavings Credit 500)
(← $MySavings Credit 6000)
(← $MySavings Debit 2500)
(← $MySavings Debit 300)
```

(Once you are finished, leave the gauges where they are, and continue on with this chapter. However, if you prefer to have a clean screen, you may preview section 8.5, at the end of this chapter, on detaching gauges.)

## 10.4.3 DigiScale and DigiMeter

**DigiScale** and **DigiMeter** combine both digital and analog gauges. The power of multiple inheritance is shown in these two gauge classes. They have two supers, one an analog gauge and the other a digital gauge, whose functions and data are combined.

Create an instance of the class, **DigiScale**, with the name, **MyDS**, and an instance of the class, **DigiMeter**, with the name, **MyDM** by typing:

```
(← $DigiScale New 'MyDS)
(← $DigiMeter New 'MyDM)
```

Attach both gauges to the instance variable, Balance, of the instance, **MySavings**, and set their scales to the range 0 to 10000, as you did with **MyVS** and **MyDial** in sections 8.4.1 and 8.4.2. The result will be similar to that shown in Figure 10.9.

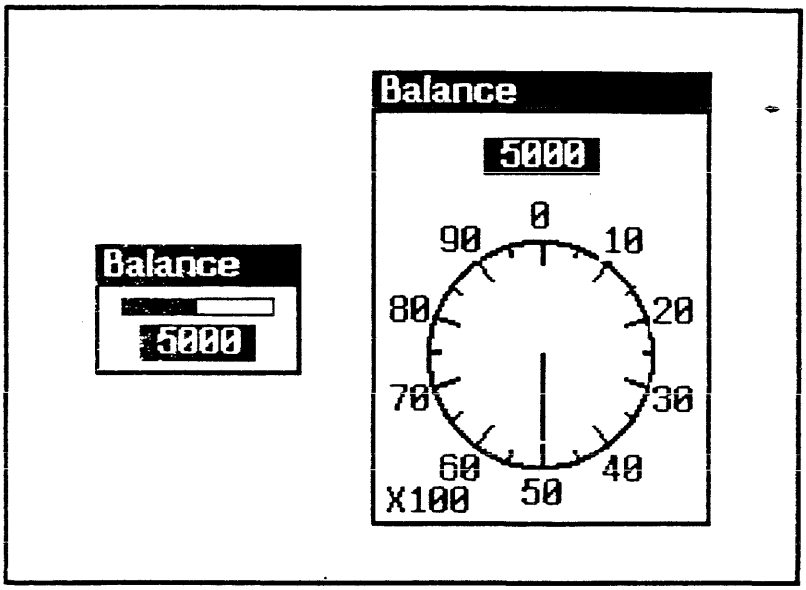



Figure 10.9. Instances of **DigiScale** and **DigiMeter** attached to Balance of **MySavings**

Send **Credit** and **Debit** messages to **MySavings** to see how **MyDS** and **MyDM** behave.

## 10.4.4 BarChart and HBarChart

**BarChart** and **HBarChart** display a number of values together on one chart. They are useful when you need to compare values; for instance, a bar chart will show the balances of two accounts so they may be compared.

Create two more instances of the class, **Savings**. Name them **JeffsSavings** and **BillysSavings**. Now create an instance of the class, **BarChart**, named **AccountBarChart**. You can use this bar chart to compare the balance of Jeff's savings account with the balance of Billy's savings account.

Attach it by typing:

```
(← $AccountBarChart Attach $JeffsSavings 'Balance)
(← $AccountBarChart Attach $BillysSavings 'Balance)
```

You will be prompted for a label for each. When prompted for the label for the one attached to **JeffsSavings**, type: Jeff, and when prompted for the label for the one attached to **BillysSavings**, type: Billy. The result will be a chart as shown in Figure 10.10.

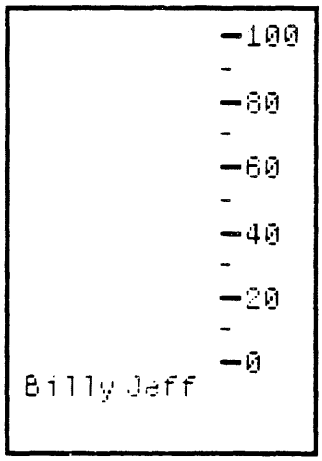



**Figure 10.10.** Instance of the class **BarChart** attached to balance of **JeffsSavings** and balance of **BillysSavings**

Set the scale on **AccountBarChart** to the range 0 to 10000. Now **Credit** and **Debit** both accounts to see how the class **BarChart** behaves. The result will be similar to Figure 10.11.

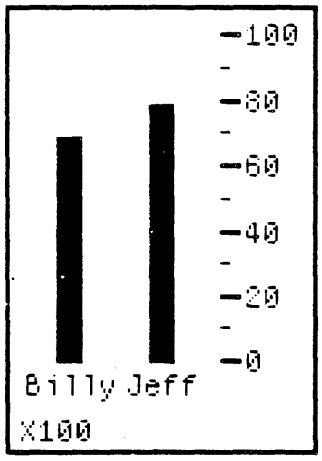



**Figure 10.11.** **AccountBarChart** with balances of **JeffsSavings** and **BillysSavings**

## 10.5 Detaching Gauges

The option, **Close**, in the window manipulation menu (right button) for gauge windows contains two sub-items: **Close** and **Destroy**, as shown in Figure 10.12.

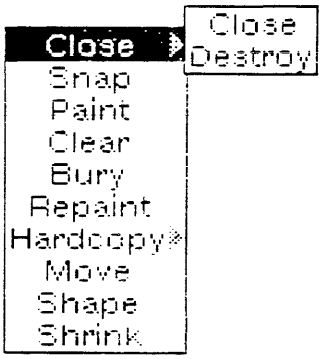



**Figure 10.12.** Window Manipulation menu for gauge window

If the **Close** sub-item is selected, the gauge will be detached from the instance and instance variable it is attached to. The gauge will still exist but it will not be attached to anything; you can see it by sending it the **Update** message.

If the **Destroy** sub-item is selected, the gauge will be detached from the instance and instance variable it is attached to and the gauge instance will be destroyed. **Destroy** should only be selected if the gauge will not be used again.

Now that you are familiar with gauges, experiment with some of the other classes in the gauge inheritance lattice.

# 11. MIXINS - INHERITANCE WITH MULTIPLE SUPERS

The inheritance order in a class lattice with multiple supers has not yet been discussed. This chapter addresses the full complexity of inheritance.

One very useful technique based on multiple inheritance is the use of "mixins." Mixins are classes that are specifically designed to be inherited along with a given class's main super. A mixin provides a package of methods and variables that can be added to many other classes to give them added functionality. The examples in this chapter demonstrate multiple inheritance with mixins.

## 11.1 Multiple Inheritance

As an example of multiple inheritance, consider the abstract class lattice shown in Figure 11.1. The names of the classes are indicated in bold face type and the names of the methods are represented in normal type. For classes with multiple supers, the left-most super in the figure is also the left-most super in the **Supers** list.

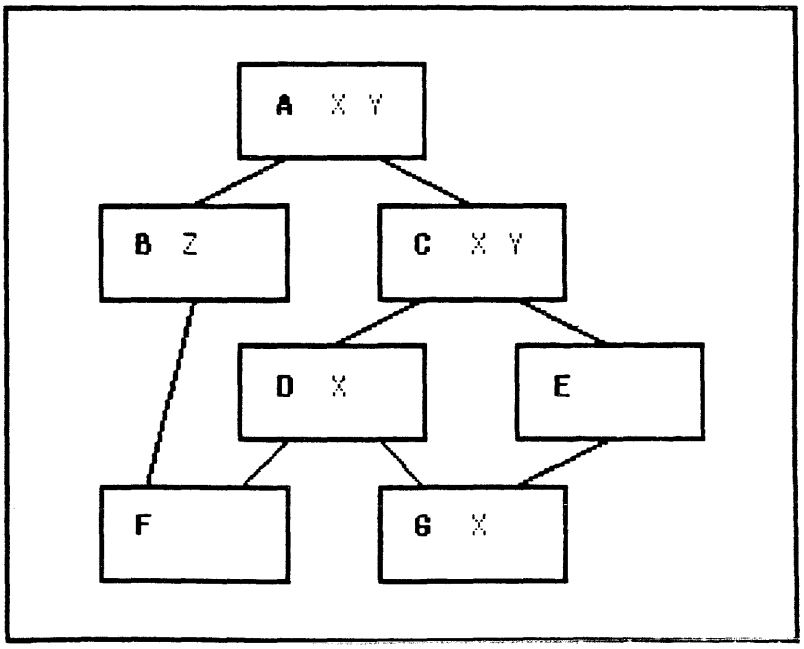



**Figure 11.1.** Diagram to show inheritance

In this discussion, methods are referred to by their full names. For example, the method **X** defined in the class **D** is referred to as **D.X**.

The rule for inheritance in LOOPS is "left to right, up to joins". Each branch up the lattice is searched, starting with the leftmost branch and working right. A class with several specialization branches is not searched until all of the specialization branches have been searched. A class with multiple specialization branches is referred to as a join. A few examples will make this clear.

In Figure 11.1, **F** has two immediate supers: **B** and **D**. These are the classes that appear on the supers list of **F**. **G** also has two immediate supers, **D** and **E**. **D** and **E** have the same immediate super, **C**. **B** and **C**'s immediate super is **A**. The order of inheritance for **F** is **F, B, D, C, A**.

Suppose message **X** is sent to **F**. Since **F** does not have a local method with the selector **X**, LOOPS searches for a method with that name in **F**'s supers. Its immediate supers are **B** and **D**. **B** is the left-most super, so it is checked first, then **D** is checked. In this case, the message **X** will be fielded by the method **D.X**.

Now, suppose that the method **D.X** contains a call to **←Super**. **D** has one immediate super, **C**, which does possess the method **X**. Therefore, **C.X** will be invoked as well.

Now assume the message **Y** is sent to **F**. As before, LOOPS checks **B** and **D**, but does not find the appropriate method. LOOPS next checks **D** and again does not find the appropriate method. The next class checked is **C** as the super of **D**. Note that **A**, which contains a method for **Y**, is not checked until all of its specializations have been checked. In this example, **A** is checked only after **B** and **C** are checked.

In Figure 11.1, **G** has two supers: **D** and **E**. The order of inheritance for **G** is **D, E, C, A**.

If the message **X** is sent to **G**, it is fielded by **G.X**. If **G.X** contains a call to **←Super**, that message is fielded by **D.X**.

Now, suppose **G.X** contains a call to **←SuperFringe**. If **X** is sent to **G**, the message is fielded by **G.X**. The call to **←SuperFringe** then sends the message to the classes on the supers list of **G**: **D** and **E**. The message sent to **D** is fielded by **D.X** and the message sent to **E** is fielded by **C.X**. In summary, the methods **G.X**, **D.X**, and **C.X** are all invoked.

For convenience, the above example focused on method inheritance. The example also applies for inheritance of variables.

## 11.2 An Existing Gauge Mixin

**<NOTE TO XEROX: AS INSTRUCTED, THIS SECTION HAS NOT BEEN UPDATED DUE TO CONTINUING WORK ON GAUGES.>**

The class, **SelfScaleMixin**, is a pre-defined mixin that automatically sets the scale on gauges. If you have a browser containing the gauge classes, **SelfScaleMixin** can be added to the lattice by selecting **AddRoot** from the browser manipulation menu and typing **SelfScaleMixin** when prompted for the item to be added.

The gauge classes, **SSBarChart**, **SSDigiMeter**, and **SSHBarChart**, have **SelfScaleMixin** as a super; other than this, they are the same as the classes, **BarChart**, **DigiMeter**, and **HBarChart**.

By itself, **SelfScaleMixin** is a useless class. The definition of **SelfScaleMixin** is shown below in Figure 11.2.

```
DEdit of CLASSES #,($C SelfScaleMixin)
((MetaClass Class Edited:
                          (* dgb: "10-JUN-83 02:32"))
 (Supers Object)
 (ClassVariables)
 (InstanceVariables (lowScaleFactor
                     5 doc
                     (* If maxCurrentReading
                     shrinks so that it will fit more
                     than lowScaleFactor times in
                     inputRange, the gauge
                     rescales)))))
```

**Figure 11.2.** The mixin, **SelfScaleMixin**

The method, **Set**, for **SelfScaleMixin** is shown in Figure 11.3.

DEdit of function SelfScaleMixin,Set

```
(Method
 ((SelfScaleMixin Set)
  self reading otherArg1 otherArg2)
                          (* RBGMartin
                          "11-Apr-86 14:41")
                          (* Check if reading is too
                          high or too low, and if so see
                          if gauge needs to rescale)
 (PROG (maxDiff (max (@ reading)))
       (COND
          ((IGREATERP
            (SETQ maxDiff
             (IDIFFERENCE max
                    (@ inputLower)))
            (@ inputRange))
                          (* If max is greater than
                          previous max then change
                          range to make current max
                          be 4/5 of full scale)
           (← self SetScale (@ inputLower
                                )
             (IPLUS (@ inputLower)
                    (IQUOTIENT
                     (ITIMES 5 maxDiff)
                     4))))
          ((AND
            (IGREATERP
             (@ inputRange)
             (ITIMES maxDiff
                    (@ lowScaleFactor)))
            (IGREATERP (@ inputRange)
                   10))

      (* If max is less than lowScaleFactor times
      range, and newMax would not be less than
      10, then change range to make current max
      be 4/5 of full scale)

           (←
            self SetScale (@ inputLower)
            (IPLUS
             (@ inputLower)
             (IMAX 10
                   (IQUOTIENT
                    (ITIMES 5 maxDiff)
                    4))))))
 (←Super
  self Set reading otherArg1 otherArg2))
```

**Figure 11.3.** The method, **Set**, for **SelfScaleMixin**

To see how self scale gauges work, create instances of the classes, **DigiMeter** and **SSDigiMeter**, with the names **MyDigiMeter** and **MySSDigiMeter** by typing:

```
(← $DigiMeter New 'MyDigiMeter)

(← $SSDigiMeter New 'MySSDigiMeter)
```

Attach both gauges to the instance variable, Balance, of the instance, **MySavings**, by typing:

```
(← $MyDigiMeter Attach $MySavings 'Balance)
(← $MySSDigiMeter Attach $MySavings 'Balance)
```

This is shown in Figure 11.4.

```
Top level -- Connected to {DSK}<LISPFILE

9←(← $DigiMeter New 'MyDigiMeter)

10←(← $SSDigiMeter New 'MySSDigiMeter)

11←(← $MyDigiMeter Attach
$MySavings 'Balance)

12←(← $MySSDigiMeter Attach
$MySavings 'Balance)

13←
```

**Figure 11.4.** Creating instances of **DigiMeter** and **SSDigiMeter** and attaching them to Balance

**Credit** and **Debit** the instance, **MySavings**, until you understand the behavior of the two gauges. Both gauges, **MyDigiMeter** and **MySSDigiMeter**, are shown in Figure 11.5 with a reading of 350.

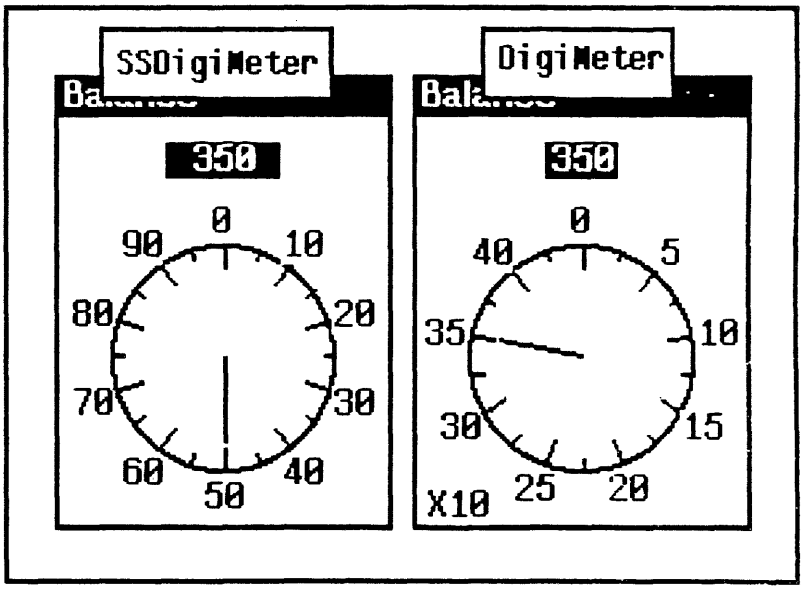



**Figure 11.5.** Instances of **DigiMeter** and **SSDigiMeter** with readings of 350

Notice that the analog meter on **MyDigiMeter**, does not give a correct reading but the analog meter on **MySSDigiMeter** does. The dial on **DigiMeter** went around 3 1/2 times. You would need to infer that from seeing that the gauge only goes as high as 100. On the other hand, **SSDigiMeter**, as a product of **SelfScaleMixin**, has included a x10 factor.

## 11.3 A New Gauge Mixin

<NOTE TO XEROX: AS INSTRUCTED, THIS SECTION HAS NOT BEEN UPDATED DUE TO CONTINUING WORK ON GAUGES.>

Now we create a new mixin **BlinkMixin** to use with the gauges. **BlinkMixin** causes a gauge to blink three times when it is set. The method **Blink** which gauges inherited from the class **Window**, can be used to do this. To see the class inheritance lattice for **Window**, type: **(Browse $Window)**. Create the mixin, **BlinkMixin** by typing:

```
(DefineClass 'BlinkMixin)
```

Add it to the gauge browser window by selecting **AddRoot** from the browser manipulation menu and typing **BlinkMixin** when prompted for the name of the item to be added.

You need to have the **Set** method for **BlinkMixin** do the same thing as the **Set** you have been using for gauges along with causing the gauge to blink 3 times. The new version of **Set** for **BlinkMixin** calls **Blink** to make the gauge blink 3 times and then uses **←Super** to invoke the normal **Set** operation which sets the gauge.

To do this, first select **Add(AddMethod)** from the editing menu on the class **BlinkMixin** to create a template for the method **Set**. Add *reading* to the argument list for the method after **self**. Replace **(MethodNeedsToBeSpecialized)** in the template with:

```
(← self Blink 3)
(←Super self Set reading)
self
```

The method looks like Figure 11.6 when you finish.

DEdit of function BlinkMixin.Set

```
(Method ((BlinkMixin Set)
         self reading)
                        (* edited:
                        "17-Jun-88 12:50")
                        (* method that causes gauge
                        to blink three times
                        whenever it is set)
        (← self Blink 3)
        (←Super
         self Set reading)
        self)
```

**Figure 11.6.** The method **Set** for **BlinkMixin**

Now, create a new class, **BlinkDigiScale**, which has as supers both **BlinkMixin** and **DigiScale**. Type:

```
(DefineClass 'BlinkDigiScale '(BlinkMixin
DigiScale))
```

Next, create an instance of the class **BlinkDigiScale** and name it **MyBlinkDS**. Display **MyBlinkDS** and send it the **Set** message to see **BlinkMixin** in action.

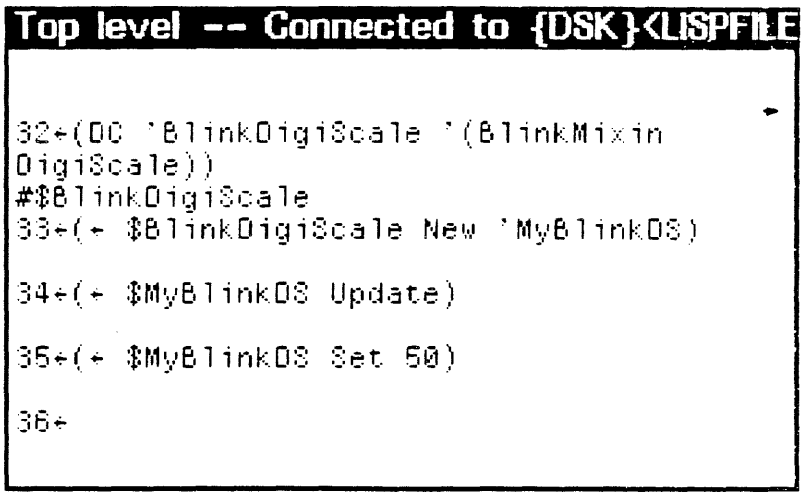



**Figure 11.7.** Creating class, **BlinkDigiScale**, and testing it

## 11.4 A Mixin for the Bank Account Example

This section explores class inheritance of methods from multiple supers in the context of the Bank Account example. If the Bank Account example is not already loaded, load it now. Also, if a browser for the Bank Account example does not currently exist, create one by browsing **GenericAccount**.

Some bank accounts allow you to withdraw more money than the account contains. The overdraft is treated as if the account holder took out a loan. A mixin can be created which, when combined with any of the classes of the Bank Account example, yields an account that allows overdrafts. When there is an overdraft, the account balance is set to 0 and the amount of the overdraft is recorded separately.

To implement this, create the **OverDraftMixin** class by typing:

```
(DefineClass 'OverDraftMixin)
```

Add the class **OverDraftMixin** to the class inheritance lattice for bank accounts. Select **AddRoot** from the browser manipulation menu and type **OverDraftMixin** when prompted for the name of the item to be added. The class inheritance lattice should look somewhat like Figure 11.8. The figures in this chapter do not show some classes that were created in previous chapters.

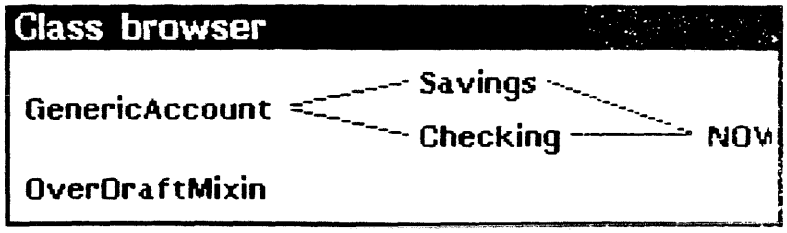



**Figure 11.8.** Class inheritance lattice for bank accounts after adding the class, **OverDraftMixin**

Edit **OverDraftMixin** so that it looks like Figure 11.9.

```
DEdit of CLASSES #.($ OverDraftMixin)
((MetaClass Class doc          (* this mixing allows
                                overdrafts to be made
                                from the account)
        Edited:                (* edited:
                                "22-Nov-86 13:03"))
 (Supers Object)
 (ClassVariables)
 (InstanceVariables (OverDraft 0 doc
                                (* this is the amount
                                of overdrafts)))
 (MethodFns))
```

**Figure 11.9.** Mixin **OverDraftMixin** which allows overdrafts on bank accounts

**OverDraftMixin** needs the methods **Credit** and **Debit**. They will be specializations of **GenericAccount.Credit** and **GenericAccount.Debit**. However, **OverDraftMixin** does not inherit these methods: they must be added from scratch.

**OverDraftMixin.Credit** needs to check to see if there is an overdraft. If there is, the amount of the credit should be applied to **OverDraft** and any money remaining after the overdraft is zeroed out should be added to **Balance**. If there is no overdraft, the credit amount is simply added to **Balance**.

The method **Credit** should look like Figure 11.10 when you finish.

```
DEdit of function OverDraftMixin.Credit
(Method
 ((OverDraftMixin Credit)
  self Amount)          (* edited:
                        "18-Jan-87 17:53")
                        (* Method to credit an
                        account that allows
                        overdrafts)
 (if (GREATERP Amount (@ OverDraft))
     then (←Super
           self Credit
           (DIFFERENCE Amount
                  (@ OverDraft)))
          (←@
           OverDraft 0)
   else (←@
         OverDraft
         (DIFFERENCE (@ OverDraft)
                Amount)))
 (@ Balance))
```

**Figure 11.10.** The method **Credit** for **OverDraftMixin**

**OverDraftMixin.Debit** needs to check to see if the amount of the debit is greater than the balance. If it is, **Balance** should be set to zero and the difference added to **Overdraft**. If the

debit does not exceed the balance, the amount is simply subtracted from the balance.

**Debit** should look like Figure 11.11 when you finish.

**DEdit of function OverDraftMixin.Debit**

```
(Method
 ((OverDraftMixin Debit)
  self Amount)          (* edited:
                        "18-Jan-87 17:54")
                        (* Method to debit an
                        account that allows
                        overdrafts.)
 (if (LEQ Amount (@ Balance))
     then (←Super
           self Debit Amount)
   else
   (←@
    .OverDraft
     (PLUS (@ OverDraft)
           (DIFFERENCE Amount
                  (@ Balance))))
   (←Super
    self Debit (@ Balance))))
```

**Figure 11.11.** The method **Debit** for **OverDraftMixin**

Notice the call to **←Super** in both of these methods. If you look back at Figure 11.9, you will see that the only super that **OverDraftMixin** has is **Object**. If **OverDraftMixin** were designed to be used alone, this would be an error since **Object** has no methods named **Credit** and **Debit**. However, because it is a mixin, **OverDraftMixin**'s calls to **←Super** are not a problem (as long as it is mixed in with a class which does have these methods).

Now create a class named **OverDraftChecking** with multiple supers, **OverDraftMixin** and **Checking**. First specialize **Checking**. Then add the super **OverDraftMixin** to **OverDraftChecking**. Be sure the mixin is first in **OverDraftChecking**'s supers list, because **OverDraftChecking** needs to have **Credit** messages fielded by **OverDraftMixin.Credit**, not by **GenericAccount.Credit**.

The class inheritance lattice for bank accounts should update automatically to look like Figure 11.12.

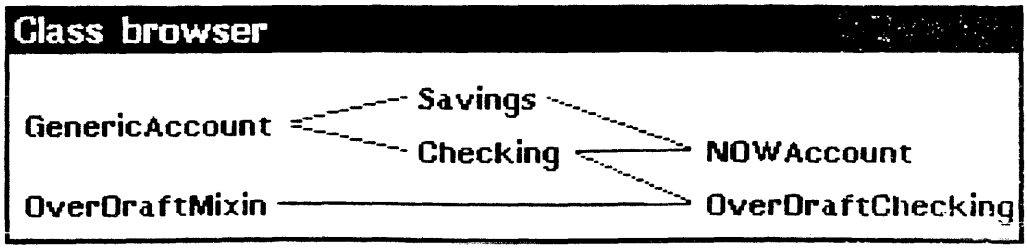



**Figure 11.12.** Class inheritance lattice for bank accounts with the class **OverDraftChecking** added

Now test that the methods of **OverDraftChecking** work properly. Create an instance of **OverDraftChecking** and send it various credit and debit messages. Use the inspector to see if **Balance** and **OverDraft** are updated correctly.

In this chapter, **Credit** and **Debit** were designed to return the resulting account balance. This is in keeping with the way that the other versions of these methods work. The returned value is not particularly informative when there is an overdraft because 0 is always returned. It might be interesting to think about ways to make the result more useful.

# 12. CUSTOMIZING LOOPS TOOLS

This chapter explores an advanced example that incorporates most of the techniques learned in previous chapters. Here, a specialization of one of the LOOPS browsers is created. This new browser is then used to display information about a budget. This exercise illustrates how to specialize the LOOPS system tools and provides more experience with building LOOPS programs.

If the Bank Account example from Chapter 6 is not loaded, you should load it now.

In this project, various budget categories, also called accounts, are represented by instances of the **AccountUse** class. Unlike instances from earlier examples, these instances form a tree. For example, the personal expense account might have entertainment and clothing subaccounts. Similarly, the entertainment subaccount might have restaurant and movie subaccounts.

As you know, the ClassBrowser does not show instances and the inspector shows only one instance at a time. In order to view an entire budget, a new kind of browser must be created.

## 12.1 Existing Browsers

Browsers are LOOPS objects. Each browser is an instance of one of the browser classes. The parent of all browsers is the **LatticeBrowser** class. The class inheritance lattice for the browsers is shown below in Figure 12.1.

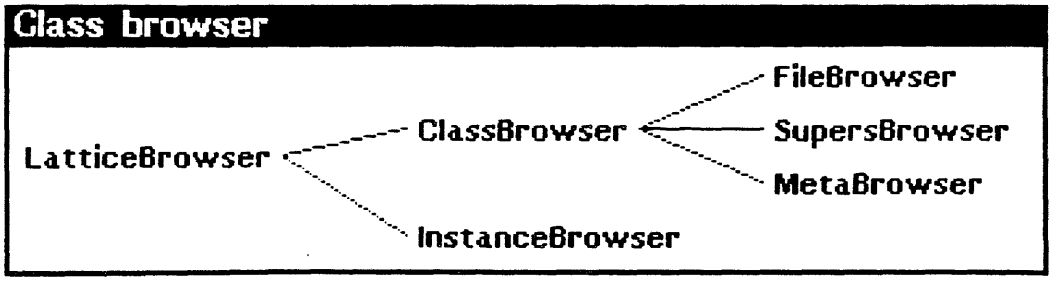



**Figure 12.1.** Class inheritance lattice for **LatticeBrowser**

Up to this point, examples have emphasized using the **ClassBrowser**. The **FileBrowser** was introduced in Section 5.2. The browser customized in this chapter is the **InstanceBrowser**. Unlike the **ClassBrowser**, which automatically depicts the inheritance relationships among classes, the **InstanceBrowser** requires explicit specification of

object links before it can display the lattice structure of the related objects.

## 12.2 Creating a Browser Subclass

Bring up a Lattice Browser on the class **LatticeBrowser**. Specialize **InstanceBrowser** to create the subclass, **AccountBrowser**.

**InstanceBrowser** contains an instance variable, **subIV**. The name stands for sublink instance variable. **subIV** contains the name of an instance variable in the class or classes of instances to be displayed. The instance variable named in **subIV** should be the variable whose value is used to link an instance to its subsidiary instance(s). When the browser is displaying instances, it looks in each one for an instance variable of this name. If found, the browser uses the value of the variable to find the children of each instance and displays them also.

The default value of **subIV** in **InstanceBrowser** is simply **NIL**. In order to use **InstanceBrowser** directly, it is necessary to fill in this value with the name of the instance variable. In the example, the instance variable **Child** is used to create a tree of instances. If, for example, instances **Y** and **Z** appear in the **Child** list of **X**, then **X** should appear higher than **Y** and **Z** in the browser tree display. The first part of the example makes **Child** the default value for the **subIV** instance variable.

In order for the child default value for **subIV** to appear in **AccountBrowser**, **subIV** must be local to **AccountBrowser**. It would be simple to use DEdit to add this instance variable. However, it is useful to know how to copy variables from one class to another. This operation is similar to the copying operations you already know. First, box **AccountBrowser**. Then, using **InstanceBrowser**'s edit menu, select **CopyIVTo** from the submenu of **Copy(CopyMethodTo)**. Finally, select **subIV** from the menu that pops up. Select **title** from the pop up menu, as well.

Now change the value of **subIV** to **Child** and the value of **title** to **"Account Browser"**. **AccountBrowser**, should look like Figure 12.2.

```
DEdit of CLASSES #.($ AccountBrowser)
((MetaClass Class Edited:
                          (* edited:
                          "22-Nov-86 17:05"))
 (Supers InstanceBrowser)
 (ClassVariables)
 (InstanceVariables (subIV Child doc
                          (* Name of instance variable
                          which provides names
                          and/or pointers to
                          subobjects))
        (title "Account Browser"))
 (MethodFns))
```

**Figure 12.2.** **AccountBrowser** after adding instance variable, **subIV**

## 12.3 Creating a Savings Subclass

The money in a savings account may be earmarked for a variety of uses, such as school, business, and personal. The money for personal uses may be further divided into money to be spent and money to be saved.

Below, an example budget is created and displayed in an instance of **AccountBrowser**.

Create a subclass of **Savings**, called **SpecialSavings**, with the instance variable **Child**. When in use **Child** holds a list of instances which represent the budget items for this account.

**SpecialSavings** should look like Figure 12.3.

```
DEdit of CLASSES #.($ SpecialSavings)
((MetaClass Class doc      (* budgeted savings
                           account)
        Edited:            (* edited:
                           "22-Nov-86 17:11"))
 (Supers Savings)
 (ClassVariables)
 (InstanceVariables (Child NIL doc
                           (* children of object
                           in brwser)))
 (MethodFns))
```

**Figure 12.3.** **SpecialSavings** with instance variable, **Child**, added

## 12.4 Creating an AccountUse Class

Create a class named **AccountUse** to be used for creating the instances that represent the various categories in a budget. **AccountUse** is not a specialization of any of the bank classes; use **DefineClass** and then use **AddRoot** to add it to the bank browser.

Give **AccountUse** the instance variable, **Balance**, with a default value of 0. **Balance** holds the amount of money budgeted for each item. Next, add the instance variable **Child** with a default value of NIL. This holds the list of subitems to be represented in **AccountBrowser**.

When you are finished, the **AccountUse** class should look like Figure 12.4.

```
DEdit of CLASSES #.($ AccountUse)
((MetaClass Class doc          (* class for instances
                                  which will represent
                                  budget items)
        Edited:                (* edited:
                                  "22-Nov-86 17:17"))
 (Supers Object)
 (ClassVariables)
 (InstanceVariables (Balance 0 doc
                                (* amount of money in
                                  budget item))
        (Child NIL doc          (* budget subitems)))
 (MethodFns))
```

**Figure 12.4.** The class **AccountUse** with its instance variables added

## 12.5 Setting Up the Budget Tree

In this section, a set of instances are created and linked through **Child** variables.

Fist, create an instance of **SpecialSavings** named **MySpecialSavings**. Now create a set of instances of **AccountUse** to represent budget items. Name them **Business**, **School**, **Personal**, **save**, and **spend**.

The main budget items of the account will be **Business**, **School**, and **Personal**. Therefore, the value of **Child** for **MySpecialSavings** should be a list containing pointers to these three instances. The browser uses these pointers to display the instances as subinstances of **MySpecialSavings**. The easiest way to put them in is to use the inspector. Open an inspector for **MySpecialSavings** and select **Child** with the left button. Then hold the middle button and select **PutValue**. When prompted, type:

```
(LIST ($ Business) ($ School) ($ Personal))
```

the value of Child should change to:

```
(#.($ Business) #.($ School) #.($ Personal))
```

Now use the inspector on **Personal** to make **save** and **spend** be subinstances of it. Its **Child** should look like:

```
(#.($ save) #.($ spend))
```

## 12.6 Creating a Browser Instance

An **AccountBrowser** instance can be created by typing:

```
(← (← ($ AccountBrowser) New) Browse ($ MyAccount))
```

A browser as shown in Figure 12.5 should appear.

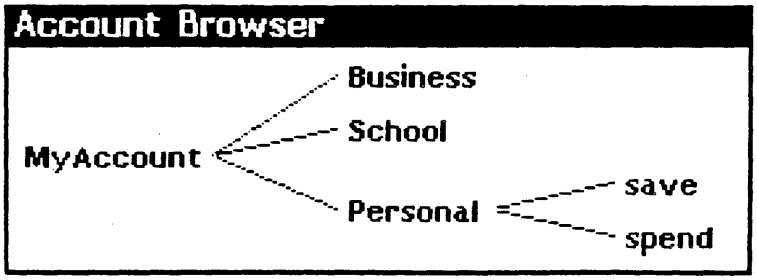



Figure 12.5. **AccountBrowser** for **SpecialSavings**

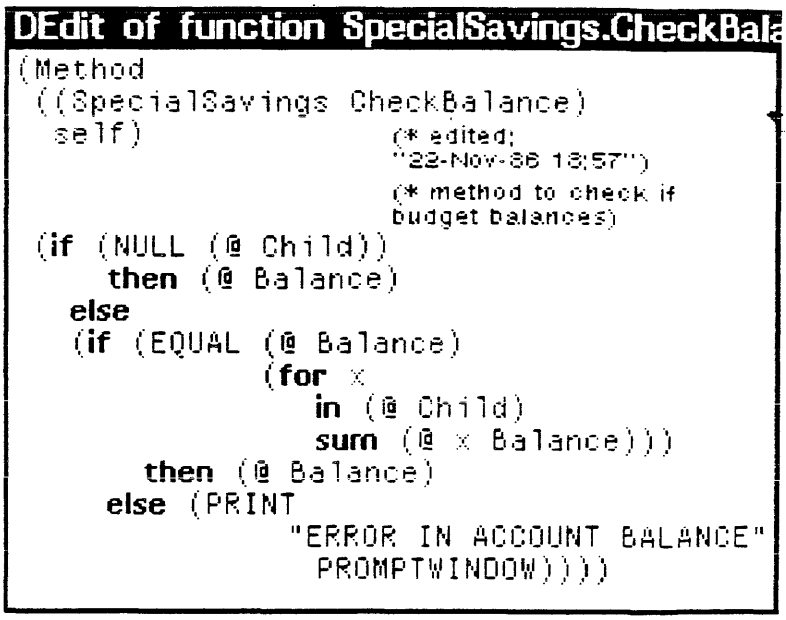
Remember, this lattice shows a series of instances, not classes. Unlike a ClassBrowser, this browser shows a lattice only if explicitly set up using **subIV**.

## 12.7 Using Your Lattice

The instances and the browser displayed are not very useful. They simply show the way money in **MyAccount** is budgeted. However, they can become useful if the method **CheckBalance** is added to the classes of the instances displayed. **CheckBalance** will check the values of **Balance** in each of the objects in **MyBrowser** and make sure that the values of **Balance** in any object's subobjects add up to the object's value of **Balance**. For instance, if **Personal** has a balance of $100, its subobjects, **save** and **spend**, should have balances that add to $100.

Because there are instances of two different classes in the budget tree, the **CheckBalance** method must belong to both **SpecialSavings** and **AccountUse**. Each one is simply given a copy. Think about how both might be made to inherit the same method.

Begin by creating the method **SpecialSavings.CheckBalance** as shown in Figure 12.6.



```
DEdit of function SpecialSavings.CheckBala
(Method
 ((SpecialSavings CheckBalance)
  self)                    (* edited:
                           "22-Nov-86 13:57")
                           (* method to check if
                           budget balances)
 (if (NULL (@ Child))
     then (@ Balance)
   else
    (if (EQUAL (@ Balance)
               (for x
                  in (@ Child)
                  sum (@ x Balance)))
        then (@ Balance)
      else (PRINT
                 "ERROR IN ACCOUNT BALANCE"
                  PROMPTWINDOW))))
```

Figure 12.6. The method **CheckBalance**

Copy this method from **SpecialSavings** to **AccountUse**.

Now the **CheckBalance** method can be tested. Give each budget item a balance. If an inspector is open for any of them, use that inspector. If not, click the middle button over the budget item names in the **AccountBrowser** and select the item **Edit** from the menu. This is a new way to alter the local values of instances. The same menu can be used to open up inspectors. This menu, and the others that are easily discovered, are inherited from **InstanceBrowser**.

Once values have been given to the budget items, send the message **CheckBalance** to any and all of them. Try creating balanced as well as unbalanced budgets to make sure both work properly.

The information displayed when the budget does not balance is not very useful. Consider altering **CheckBalance** to print out information about exactly what does not balance and where. Also consider how to set up a system in which the various budget items would actually be subaccounts. That is, credits and debits would be sent to a specific part of the budget. The balance would be updated at that part of the budget as well as in the overall balance of the account.

# 13. USING MASTERSCOPE WITH LOOPS

The Masterscope program, used to analyze programs in Interlisp-D, is also used with LOOPS programs. If you are not familiar with Masterscope, see the Masterscope chapter in *Interlisp-D: A Friendly Primer* and in the *Interlisp-D Reference Manual*.

As programs become larger or more complex, it can become difficult to keep track of which objects send messages, read the values of variables, or put the values of variables in other objects. Masterscope is a tool that allows you to examine the structure of programs. It is able to analyze LOOPS programs to see how objects interact with each other.

## 13.1 Masterscope Verbs for use with LOOPS

LOOPS adds a series of verbs to Masterscope so that relationships peculiar to LOOPS can be analyzed. Here is a selection of those verbs (see *The LOOPS Manual* for a complete list):

| | |
|---|---|
| **SEND** | sends the message |
| **SEND SELF** | sends the message to *self* |
| **SEND NOTSELF** | sends the message to other than *self* |
| **SPECIALIZE** | specializes the method |
| **GET** | gets the instance variable |
| **GET CV** | gets the class variable |
| **PUT** | sets the instance variable |
| **PUT CV** | sets the class variable |
| **USE IV** | gets or sets the instance variable |
| **USE CV** | gets or sets the class variable |
| **USEOBJECT** | references the named object |

## 13.2 An Example of using Masterscope

Masterscope commands are invoked by typing a period followed by a space followed by the command. If the banking example is

stored on a file, try Masterscope on it. The first step is to have Masterscope build up a data base of the relationships in the file by typing:

```
. ANALYZE FUNCTIONS ON BANK
```

Your screen looks like Figure 13.1 after doing this.

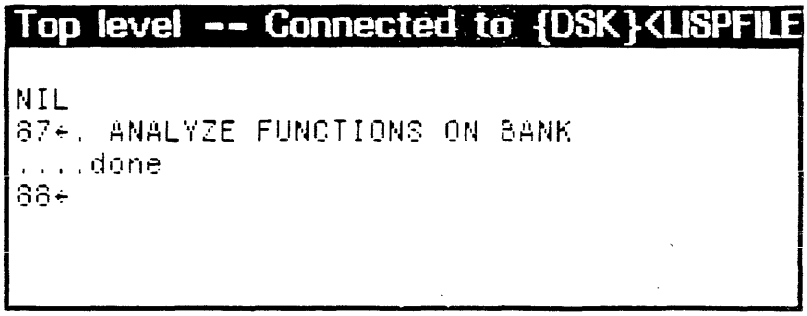



**Figure 13.1.** Using Masterscope to analyze the Bank Account example

To find out where **Credit** messages are sent from, type:

```
. WHO SENDS Credit
```

To find out where a **Balance** variable is referenced:

```
. WHO GETS Balance
. WHO USES IV Balance
```

Figure 13.2 shows the results of some Masterscope commands.

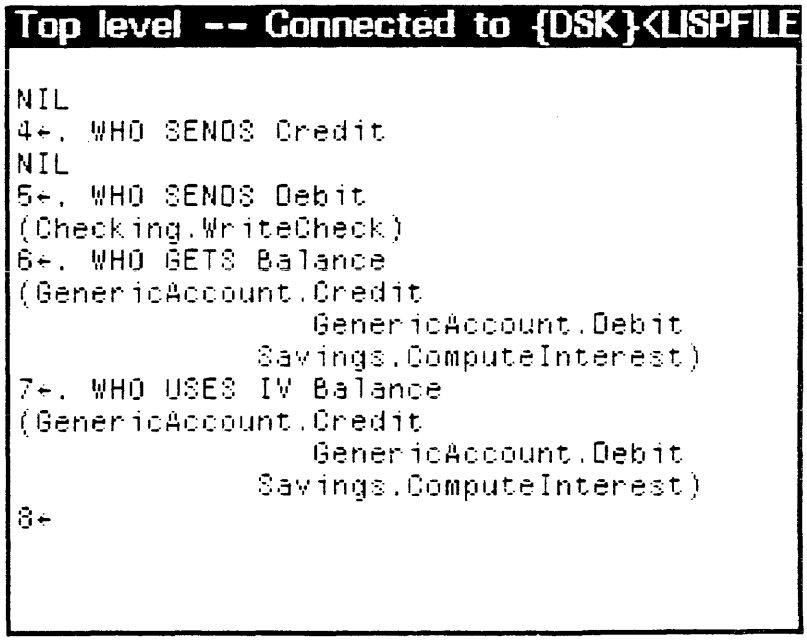



**Figure 13.2.** Using Masterscope on the Bank Account example

The Masterscope command **DESCRIBE** includes information about sending, getting, and putting. **DESCRIBE** can be used with LOOPS methods. For example, to get a description of the method **GenericAccount.Debit**, type:

```
. DESCRIBE GenericAccount.Debit
```

The results of calling the Masterscope command **DESCRIBE** are shown in Figure 13.3.

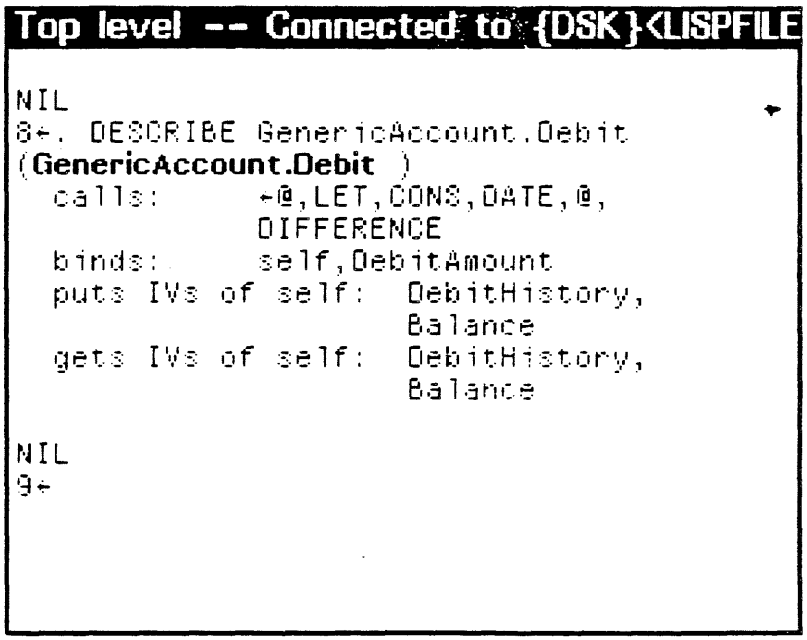



**Figure 13.3.** The Masterscope command DESCRIBE

# 14. SOME CLOSING WORDS

This primer is designed to tell just enough about LOOPS to get you started. Early chapters discuss the concepts of object oriented programming and the LOOPS implementation of those concepts. Later chapters present standard tools, design techniques, and methods for modifying the objects provided in the LOOPS environment.

It is now time to use this material on tasks that are more interesting and relevant than the Bank Account example. Here are some useful suggestions for next steps in LOOPS.

LOOPS lends itself to exploratory programming. We urge you to take the ideas in this primer and begin to develop preliminary versions of your system in LOOPS. There are development projects where LOOPS was used to implement 15 to 20 different versions of a system before it was exactly right. The LOOPS interface provides both a programming tool and a thinking tool. As you develop a new system, each preliminary version provides an object for thought and discussion. The preliminary versions are a crucial part of the design process.

As you gain some more experience with LOOPS, we suggest you skim the entire *LOOPS Reference Manual*. By becoming familiar with this manual you learn where to look when you need a feature that is too obscure or tricky to be covered in this primer.

The truly adventurous LOOPS user should also consider looking at the definitions for objects provided by the system. All the predefined LOOPS classes exist in the class lattice and can be inspected and browsed just like user defined classes.

For more information on how the LOOPS language was created and defined, we recommend the article:

"Object Oriented Programming: Themes and Variations", Mark Stefik and Daniel G. Bobrow, *Artificial Intelligence Magazine*, Winter 1986, Vol. 6, No. 4, pp 40 - 62.

For an interesting discussion of future trends in object oriented programming, we recommend the article:

"CommonLoops: Merging Common LISP and Object Oriented Programming"; Daniel G. Bobrow, Ken Kahn, Gregor Kaczales, Larry Masinter, Mark Stefik, and Frank Zdybel; Xerox Palo Alto Research Center, Intelligent Systems Laboratory Series, ISC-85-8; August 1985.

Good luck!

**(Animal (NTV0.0X:.P%]7.%[@>.4))** This is the way the system prints an instance. The unintelligible string of characters after the name is called a unique identifier or UID. Since instances are not required to have names, a UID is generated for each one.

# 4. VARIABLES, METHODS, AND MESSAGES

This chapter describes how to access the variables in LOOPS objects, how to create and move a method, and how to send messages.

By the end of this chapter you will know the basic information necessary to implement complete LOOPS programs.

## 4.1 Variables

In the previous chapter, you learned how to examine and set the values of variables using the inspector or the editor. In this section you will learn about four basic forms for reading and setting instance variables and class variables in a running program. Beginning with this chapter, we use *iv* for instance variable and *cv* for class variable.

### 4.1.1 Reading Instance Variables

The syntax for reading an instance variable is:

**(@** *object ivname*)

To see how variable access works, type:

```
(@ ($ Billy) Haircolor)
```

It should return Blond. Note that the *ivname* (**HairColor**) is not evaluated and thus should not be quoted.

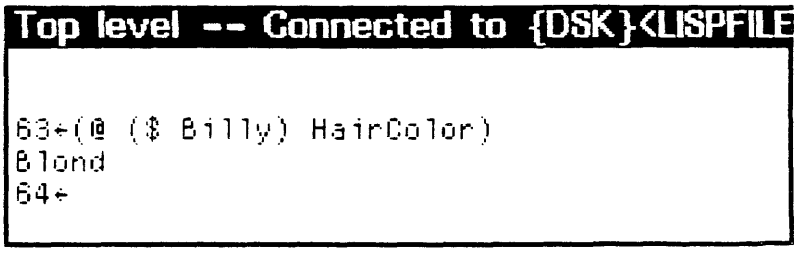



**Figure 4.1.** Using the **@** function

### 4.1.2 Setting Instance Variables

To set an instance variable, the syntax is:

(←@ *object ivname newvalue*)

Try changing the value of **Billy**'s Hatsize to 8 by typing:

```
(←@ ($ Billy) Hatsize 8)
```

If you still have your inspector window for **Billy** open, you may have noticed that the value of **HatSize** did not change. Inspectors do not automatically update themselves when values are changed. To see the change, select **Refetch** from the inspector's title bar left button menu, and **Billy**'s Hatsize value changes to 8 as shown in Figure 4.2.

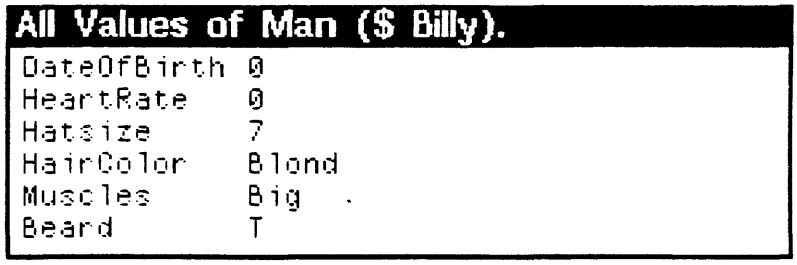



**Figure 4.2.** Instance variables and values for Billy

## 4.1.3 Reading Class Variables

The syntax for reading class variables is similar to that for instance variables:

(@ *object* ::*cvname*)

Note the double colon (::) prefixing *cvname*.

Try typing:

```
(@ ($ Person) ::Legs)
```

The value 2 should be returned as shown in Figure 4.3.

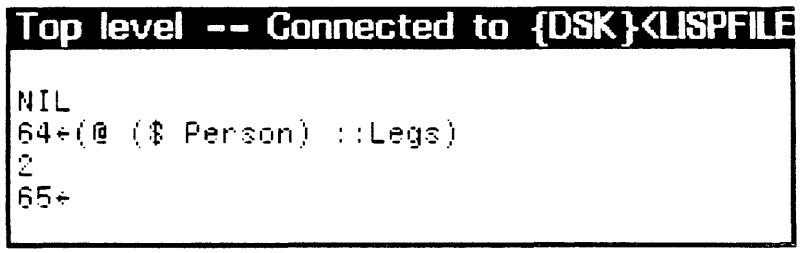



**Figure 4.3.** Fetching the value of a class variable

## 4.1.4 Setting Class Variables

The syntax for setting class variables is also very similar:

(←@ *object* ::*cvname newvalue*)

Change the value of **Person**'s class variable, Legs, to 4, by typing:

```
(←@ ($ Person) ::Legs 4)
```

Now change the value of **Person**'s Legs back to 2.

### 4.1.5 A Note of Caution

Much of the modularity of an object-oriented program is derived from allowing only an object's methods to read or change that object's internal variables. Typically, reading and setting is done through particular messages that enforce constraints to maintain consistency among an objects's variables. LOOPs does not enforce this modularity; any object may read or set the variables of another object. You should use this capability carefully. **@** and **←@** should be used mainly inside the methods of an object to access that object's variables -- including any variables inherited from its supers.

## 4.2 Methods

Each class has a set of methods which establish what that class can do. Just as with variables, a class inherits all of the methods from its supers. Because methods are defined in a class, all instances of the class have the same methods and behave in basically the same way. Any differences in behavior are due to differences in the values of the instance variables in each instance.

### 4.2.1 Creating a Method

For the example, add the method **WhoAmI** to the class, **Man**. Bring up the editing menu for **Man** and select **Add(AddMethod)** from the editing menu. You are prompted for the name of your method. Once you have typed in the name you are automatically put into DEdit with a method template to edit, as shown in Figure 4.4.

**DEdit of function Man.WhoAmI**

```
(Method ((Man WhoAmI)
         self)          (* edited;
                        "14-Nov-88 19:15")
                        (* New method template)
        (SubclassResponsibility))
```

**Figure 4.4.** New method template for **Man's** method **WhoAmI**

A method is a special kind of function defined by LOOPS. Note that the function type, instead of being **LAMBDA** or one of the other Interlisp-D function types, is **Method**. The first argument to a Method function is always a list containing the object in which the method is defined and the name of the method. The second argument is always *self*. *Self* is bound to the object that receives the message when the method is run. Methods use *self* to access the variables and other methods of the object. If a method has other arguments, they follow *self*.

First, document the method by replacing **(* New method template)** with:

**(* method to print a description of an instance of Man)**

In general, comments can be added to methods just as they are added to any other kind of function.

Replace **(SubClassResponsibility)** in the body of the method with:

```
(PRIN1 "I am ")
(PRINT self)
(PRIN1 "I was born on ")
(PRINT (@ DateOfBirth))
(PRIN1 "My haircolor is ")
(PRINT (@ HairColor))
(PRIN1 "My hatsize is ")
(PRINT (@ Hatsize))
```

The call to the function, **SubClassResponsibility**, is more than a place holder. If you forget to edit a method, it causes a warning if the method is run or compiled.

When finished, your defined method, **WhoAmI**, should look like Figure 4.5 below.

**DEdit of function Man.WhoAmI**

```
(Method ((Man WhoAmI)
         self)          (* edited:
                        "17-Nov-86 17:13")
                        (* Method to print a
                        description of an instance of
                        Man)
        (PRIN1 "I am ")
        (PRINT self)
        (PRIN1 "I was born on ")
        (PRINT (@ DateOfBirth))
        (PRIN1 "My haircolor is ")
        (PRINT (@ HairColor))
        (PRIN1 "My hatsize is ")
        (PRINT (@ Hatsize)))
```

**Figure 4.5. Man's method WhoAmI**

Did you notice something different about the variable access expressions? There is no *object* argument in them. They could have been written in this form:

**(@ self DateOfBirth)**

However, because variable accessing is meant to be done mainly from inside methods, the argument *object* is automatically assumed to be *self* if it is left out.

## 4.2.2 Moving a Method

As a LOOPS program is developed, it is often found that a method is in the wrong place in the class lattice. In our example, it is clear that the method **WhoAmI** applies to **Woman** as well as to **Man**. Therefore it should be moved up to **Person** where it is

inherited by both classes. Bring up the editing menu (middle button) on the class **Person** in the browser window and select **BoxNode**. This puts a box around **Person** to show where the method should go. Next, call the editing menu on **Man** and select **Move(MoveMethodTo)**. A pop up menu appears with **Man's** methods as its items. (See Figure 4.6.) To complete the move, select **WhoAmI** from this menu.

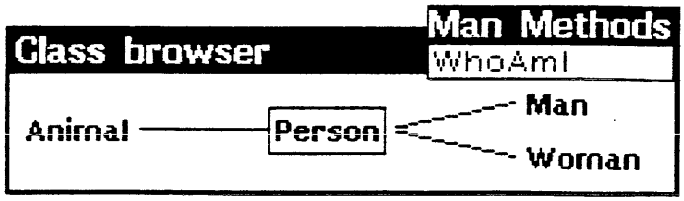



**Figure 4.6.** Pop-up menu with **Man's** methods

To check that **Person** has received the method **WhoAmI**, bring up the information menu (left button) on **Person** and select **PrintSummary**. The summary of the class **Person** is printed as shown in Figure 4.7. Under **Methods**, the method **WhoAmI** is printed in bold type to indicate it belongs to **Person**. If you PrintSummary for **Woman**, the **WhoAmI** method is listed, but not in bold because **Woman** inherits this method from **Person**.

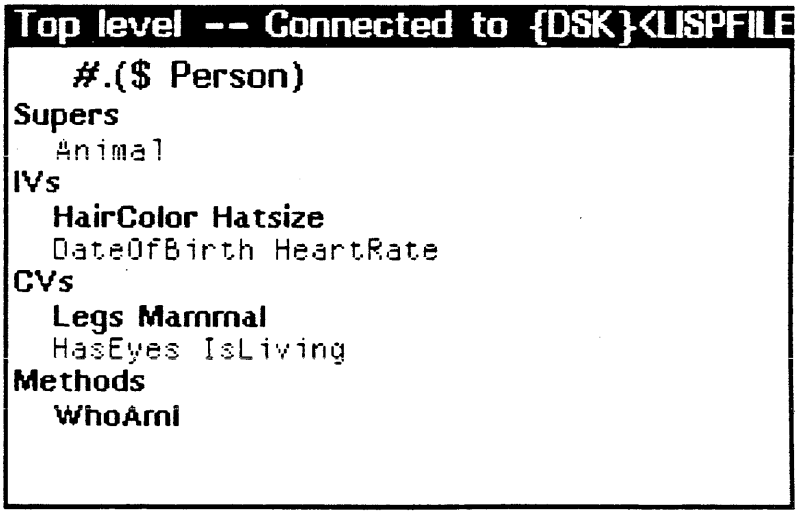



**Figure 4.7.** Summary of **Person**

## 4.3 Messages

Message passing -- objects sending messages to other objects --is the main activity of programs written in object oriented languages. When an object receives a message, it runs the appropriate method. After running the method, the object returns some value to the sending object. Generally the method causes some side effects to happen as well. Often these side effects include sending messages to other objects. You have, for example, already sent the message **New** to cause the method **New** to create instances.

## 4.3.1 Syntax of a Message

The syntax of a message is:

*(← object selector argument1 argument2 . . .)*

← activates, or sends, the message.

*object* is the object to which the message is sent. This object is bound to *self* in the body of the method definition. This argument is evaluated.

*selector* is the name of the method that is to be invoked by this message. This argument is not evaluated and should not be quoted.

*argument(n)* are bound to the corresponding arguments in the method function. These arguments are evaluated.

## 4.3.2 Sending a Message

To send a message with selector **WhoAmI** to the instance **Billy**, type:

```
(← ($ Billy) WhoAmI)
```

The result of this message is displayed in Figure 4.8

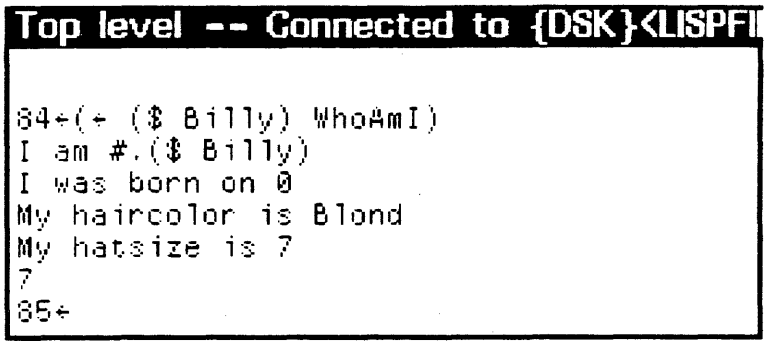



**Figure 4.8.** Output when message **WhoAmI** is sent to **Billy**

When an instance receives a message, it matches the selector of the message with the names of its own methods. These methods include those that are inherited from the supers of the instance's class. If a match is found, the method is run. If no match is found, an error occurs.

It is quite possible for different classes to have methods with the same name, but with very different method bodies. This allows objects to communicate with other objects in a standard way without the sender worrying about internal differences in those objects. For instance, there might be many different **WhoAmI** methods using different bodies. As long as they all print out a description of the object to which they are sent, they are all the "same" method as far as other objects are concerned.

# 5. SAVING LOOPS PROGRAMS

## 5.1 Using FILES? and MAKEFILE

All of the elements of a LOOPS program can be saved on files in the same way that work is saved in Interlisp-D. The function **Files?** is used to add newly created objects, methods and instances to files. The function **MAKEFILE** is used to write a file to a storage device.

When you type:

```
(FILES?)
```

any class definitions, methods, and instances which are not already associated with files are listed (along with standard Interlisp-D entities such as functions and variables). You are then asked if you want to specify their destination files. If you type **Y** (for yes), they are listed one at a time. After each, type the name of the file it is to go in. An example is shown in Figure 5.1.

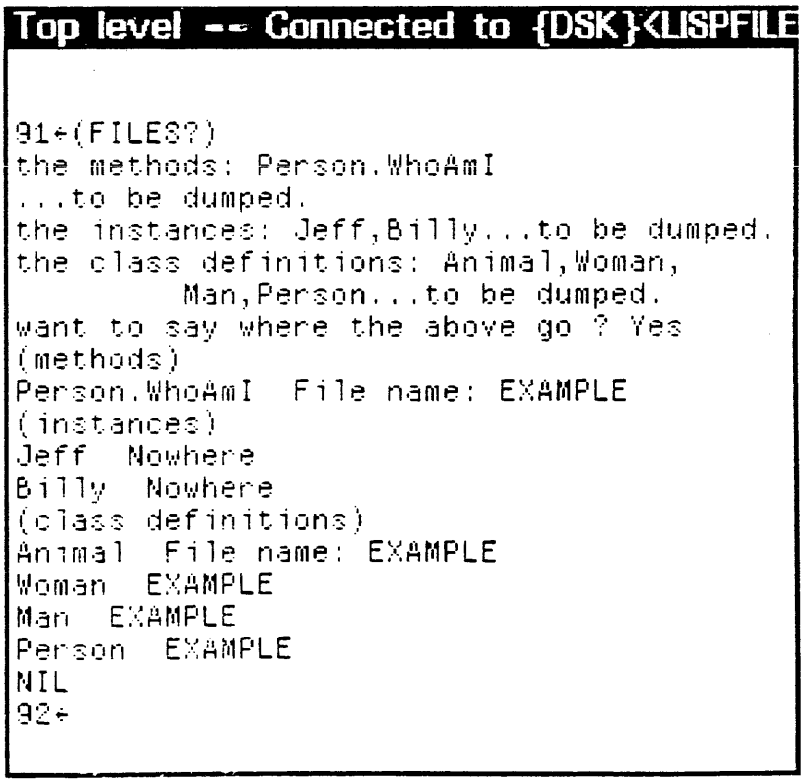



**Figure 5.1.** Using the function **FILES?**

Note that we chose not to save the instances **Jeff** and **Billy**, although they could have been saved as well. It is often just as easy to recreate instances from their classes as it it to save them on files. In some cases, instances may be the product of a particular run of a LOOPS program and should not be saved since the next run will produce different instances.

Note how methods are named. **Person.WhoAmI** is the **WhoAmI** method for the class **Person**. This naming convention is followed outside of the actual LOOPS code and LOOPS browsers.

To write out a file, you can use **MAKEFILE** (or **MAKEFILES**) as you would for any Interlisp-D file:

```
(MAKEFILE 'filename)
```

The file, EXAMPLE, which was created in Figure 5.1, is written to the hard disk in Figure 5.2.

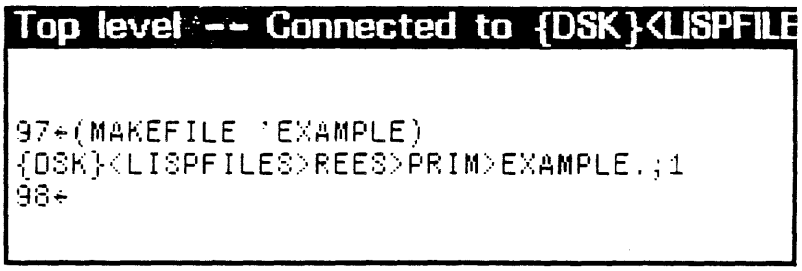



**Figure 5.2.** Saving an example file

## 5.2 Using the FileBrowser

After a long session of creating and editing LOOPS code, it can be rather tedious to have to inform the file package where each method, class and instance should go. Also, if the system you are developing is big enough to be stored in more than one file, it can be difficult to decide which objects go in which files. The convention is to store an entire sublattice in a file. Methods should go in the same file as the classes to which they belong, subclasses should go in the files of their supers and instances should go in the files of their classes. If things are not stored in this way, the files have to be loaded in a very particular order. A method can not be defined for a class that does not exist; a class can not be defined if its super does not exist, and so on. In the worst case, it is possible to create files which can not be loaded in any order.

LOOPS provides a file browser (FileBrowser) to simplify adding classes and methods to files. This browser should not be confused with the library package, FILEBROWSER. The LOOPS file browser is very similar to the class browser you have already been using. The FileBrowser can be used *instead* of the ClassBrowser whenever you want to create new items and save them in a file. It is also useful when you want to see what classes and methods are in a given file. The class browser operations

you have already used, such as creating and editing classes and methods, are performed in exactly the same way using the FileBrowser as using the ClassBrowser.

In order to illustrate the FileBrowser, you will create a file and put a class, a subclass and a method in it. Because the only reason for the existence of the classes and the method is to observe how the FileBrowser works, they will be "dummies" -- they will not do anything.

To begin, hold the middle button in the loops icon and select **Browse File**, as shown in Figure 5.3.

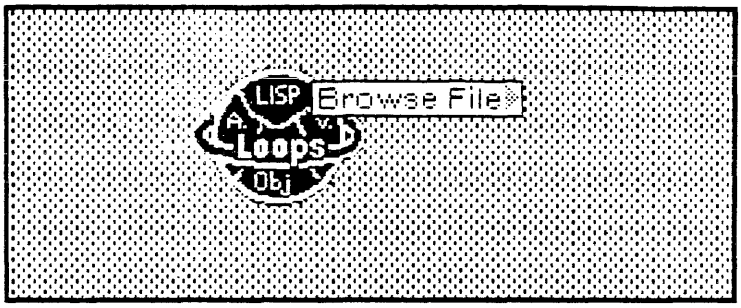



**Figure 5.3.** Accessing the FileBrowser

The File List appears as in Figure 5.4.


**Figure 5.4.** The menu from the Browse File selection.

Select ***newFile*** and then type a file name in the prompt window. We are using the name **FBEXAMPLE**, but it does not matter what name you use. Do not use the name of an existing file. A completely empty FileBrowser should appear as in Figure 5.5.

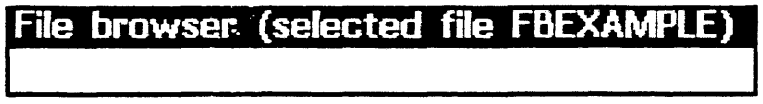



**Figure 5.5.** An empty FileBrowser

To browse a file that was already loaded, you could have selected its name from the File List menu. To load an existing file, you could have used ***loadFile*** from the submenu of ***newFile***.

Now you can create a root class for the FileBrowser. Unlike the ClassBrowser, the FileBrowser allows you to create the root class from a menu, so you do not have to use **DefineClass**. Click the middle button in the FileBrowser's title bar to get the menu shown in Figure 5.6. Note that the first three items are exactly the same as those in the ClassBrowser title bar menu. Select **AddRoot** and type in some class name at the prompt. In our example, the name is not important. We are using the name **AClass**. Because adding a root to a FileBrowser also adds it to a

file, you are asked to confirm this operation by clicking the left mouse button.


**Figure 5.6.** The FileBrowser title bar menu

Now use the class edit menu just as you have before (click the middle mouse button over the class name) to give your root class at least one subclass, and add at least one method to one of your classes. (We have named ours **ASubClass** and **OneMethod**.) For this example, it is not necessary to actually edit the classes or methods.

As you create items with a FileBrowser, they are automatically put into the file coms list. This is a data structure maintained by Interlisp-D that describes the contents of a file. To see the result of your work, bring up the title bar menu again and select **Edit File Coms**. You should see, as in Figure 5.7, that everything you have created with the browser is already in the file coms.

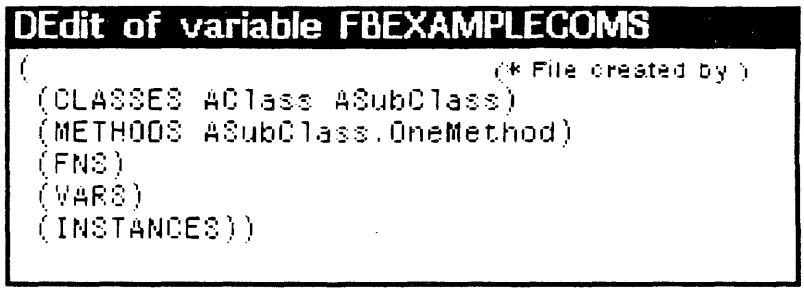



**Figure 5.7.** DEdit of File COMS variable

Now, type **(FILES?)**. It is still necessary to do this to make sure that instances and any auxiliary Interlisp-D functions are properly included in files. However, for this example, the only message you should see is that your file needs to be dumped. As always, the final step is to use **MAKEFILE** to write out any files that need to be dumped. For our example, type **(MAKEFILE 'FBEXAMPLE)**.

In the chapters that follow, we use the ClassBrowser in our figures. However, if you wish to save the examples you create, you can use the FileBrowser. Remember that while the FileBrowser has some extra menu items for dealing with files, the basic menu items for creating, modifying, and displaying LOOPS objects and the LOOPS lattice are exactly the same in the ClassBrowser and the FileBrowser.

# 6. THE BANK ACCOUNT EXAMPLE

In this chapter you write a program that integrates all that has been covered so far. This example is used again in Chapter 9. If you think you might not have time to cover both chapters in one session, use the FileBrowser (see Section 5.2) instead of the ClassBrowser to make it easier to save this program on a file.

## 6.1 Designing the Program

Your goal is to write a program that defines and keeps track of different types of bank accounts. The types of accounts to include are:

Savings Account — A savings account must contain a record of the balance as well as a history of deposits and withdrawals. It must also store the current interest rate and compute the interest earned.

Checking Account — A checking account must similarly contain a record of the balance and a history of deposits and withdrawals. It must also maintain a list of check numbers coupled with the amounts and dates of the checks written.

NOW Account — A NOW account is a combination of a savings account and a checking account. Checks can be written and interest is earned.

These accounts all have some operations and variables in common. You design your class lattice structure by determining the operations and variables common among the objects.

First, you define a class, **GenericAccount**, which consists of those variables and methods common to all of the accounts. Figure 6.1 shows a representation of the class lattice you are to develop. The classes are shown along with their instance variables.

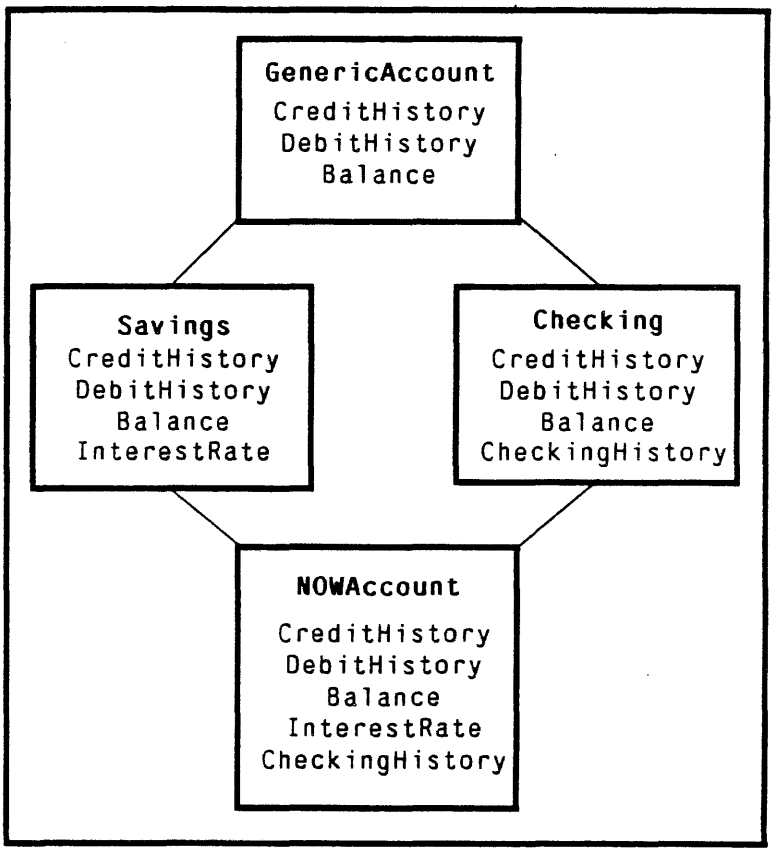



**Figure 6.1.** Inheritance lattice for Bank Account example

The class **GenericAccount** can add in deposits as credits, subtract out withdrawals as debits, and update the balance. The rest of the lattice structure comes naturally from the definitions of the accounts given above. Notice that we have given **NOWAccount** two supers: **Savings** and **Checking**.

## 6.2 Creating the Classes

First, create all of the classes you need. Then you can go back and fill in the class variables, instance variables, and methods.

To create the root node, **GenericAccount**, type:

```
(DefineClass 'GenericAccount)
```

Next open a browser for **GenericAccount**. Then, create two specializations of **GenericAccount**: **Savings** and **Checking**. Your browser window should look like Figure 6.2.

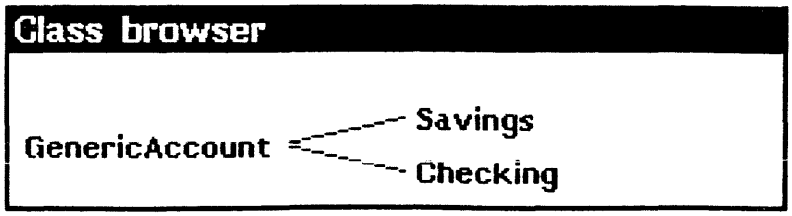


Figure 6.2. Lattice with **Savings** and **Checking** classes added

**NOWAccount** has 2 supers, so the process of creating it is a little different. First, create it by specializing **Savings**. Then bring up **NOWAccount**'s editing menu and select **AddSuper** from the submenu of **Add(AddMethod)**. Type **Checking** in the prompt window. Your browser should look like Figure 6.3.

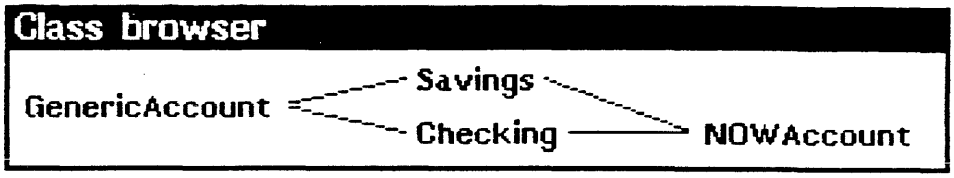



Figure 6.3. Browser including **NOWAccount**

Now that the entire lattice is created, you can edit each class.

## 6.3 Editing GenericAccount

Start your class definitions with **GenericAccount**. Bring up the editing menu (middle button) on **GenericAccount** in the browser window and select **Edit(EditClass)**. Document **GenericAccount** by adding the following between *Class* and *Edited* in the class definition template:

```
doc (* the generic type of bank account; defines the basic
things needed for all kinds of accounts)
```

### 6.3.1 Adding Variables, Values, and Documentation

**GenericAccount** has only one class variable since only one variable has a value that is the same for every account. Add the following after **ClassVariables** in the template:

```
(FDICInsured 100000 doc (* all accounts insured by federal
government to $100,000))
```

All of your accounts have a credit history, a debit history, and a balance. Since each account has different values for these variables, CreditHistory, DebitHistory, and Balance are all implemented as instance variables.

CreditHistory and DebitHistory will keep lists in the form of **(CreditAmount . date)** and **(DebitAmount . date)**. To insert these instance variables into your **GenericAccount** class, add the following variables, values, and documentation to the list starting with **InstanceVariables** in the template:

```
(CreditHistory NIL doc (* a list of (CreditAmount . date) to
tell the credit history))
(DebitHistory NIL doc (* a list of (DebitAmount . date) to
tell the debit history))
(Balance 0 doc (* the current balance of the account))
```

Note that **NIL** is the default value for CreditHistory and for DebitHistory. When you are finished editing, **GenericAccount** should look like Figure 6.4.

```
DEdit of CLASSES #.($ GenericAccount)
((MetaClass Class doc
                                (* the generic type of
                                bank account, defines
                                the basic things need
                                for all kinds of
                                accounts)
        Edited:                 (* edited;
                                "15-Nov-88 14:15"))
 (Supers Object)
 (ClassVariables (FDICInsured 100000 doc
                                (* all accounts
                                insured by the federal
                                government to
                                $100,000)))
 (InstanceVariables (CreditHistory NIL
                                  doc
                                (* a list of (
                                CreditAmount . date)
                                to tell the credit
                                history))
        (DebitHistory NIL doc
                                (* a list of (
                                DebitAmount . date) to
                                tell the debit history))
        (Balance 0 doc          (* the current balance
                                of the account)))
 (MethodFns))
```

**Figure 6.4.** Instance and class variables in **GenericAccount**

## 6.3.2 Defining Credit and Debit Methods

All of the accounts need methods to credit and debit the account. First, you define the method, **Credit**, which takes a deposit, adds it to the balance and updates the credit history. Begin by using the browser edit menu to add the method **Credit** to **GenericAccount**. When the DEdit window appears, add the argument, *CreditAmount*, after *self*. Add the appropriate documentation then type the following code in place of **(SubClassResponsibility)** in the body of the method:

```
(←@ CreditHistory
   (CONS
          (CONS CreditAmount (DATE))
          (@  CreditHistory)))
(←@  Balance
   (PLUS
          (@  Balance)
          CreditAmount))
(@ Balance)
```

The results should be as shown in Figure 6.5. Note that the method returns the new balance.

**DEdit of function GenericAccount.Credit**

```
(Method ((GenericAccount Credit)
         self CreditAmount)
                        (* edited:
                        "15-Nov-86 14:26")

      (* * Adds (CreditAmount , date) to
      CreditHistory and adds CreditAmount to
      Balance)

        (←@
         CreditHistory
         (CONS (CONS CreditAmount (DATE))
               (@ CreditHistory)))
        (←@
         Balance
         (PLUS (@ Balance)
               CreditAmount))
        (@ Balance))
```

**Figure 6.5.** The method **Credit** for **GenericAccount**

Now define the method **Debit**. It is the same as **Credit** except that it subtracts an amount from the balance. Overdrafts are discussed below. Your method should look like Figure 6.6.

**DEdit of function GenericAccount.Debit**

```
(Method
 ((GenericAccount Debit)
  self DebitAmount)  (* edited:
                     "17-Jan-87 15:42")

      (* * Adds (DebitAmount , date) to
      DebitHistory and subtracts DebitAmount
      from Balance.)

 (←@
  DebitHistory
  (CONS (CONS DebitAmount (DATE)
              (@ DebitHistory))))
 (←@
  Balance
  (DIFFERENCE (@ Balance)
              DebitAmount))
 (@ Balance))
```

**Figure 6.6.** The method **Debit** for **GenericAccount**

### 6.3.3 A Simple Test of GenericAccount

Now, before defining the rest of the classes, test an instance of **GenericAccount** to ensure that the methods **Credit** and **Debit** are working properly.

Create an instance of **GenericAccount**, with the name **MyGeneric**, by sending the message **New**:

```
(← ($ GenericAccount) New 'MyGeneric)
```

Now send a **Credit** message to **MyGeneric**. Credit the account with 1000 by typing:

```
(← ($ MyGeneric) Credit 1000)
```

The new balance should be returned because we put **(@ Balance)** at the end of the method. To verify that the method updated the instance variables correctly, inspect **MyGeneric** by calling the function **(INSPECT ($ MyGeneric))**. It should appear as shown below in Figure 6.7.


**Figure 6.7.** Result of sending the message **Credit**

Now test **Debit** in the same way. Try withdrawing 500 from your generic account by typing **(← ($ MyGeneric) Debit 500)**. Remember to use **Refetch** in the inspector's title bar menu to update the values shown there. It should now look like Figure 6.8.

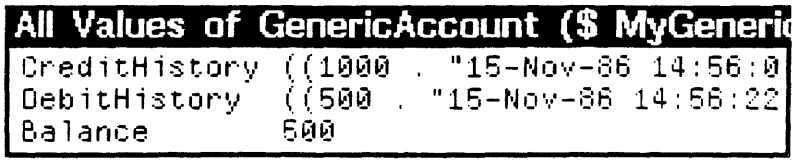



**Figure 6.8.** Result of sending the message **Debit**

## 6.4 Editing Savings

When **GenericAccount** works properly, you can define **Savings**. Because **Savings** inherited all of **GenericAccount's** methods and variables, you only need to add one additional instance variable which contains the interest rate, and a method to compute the interest.

### 6.4.1 Adding Variables, Values, and Documentation

Add the variable, **InterestRate**, with a value of **.05**, to your **Savings** class. When you are done, your class should look like Figure 6.9.

```
DEdit of CLASSES #.($ Savings)
((MetaClass Class doc          (* simulates a
                                standard savings
                                account)
        Edited:                (* edited:
                                "15-Nov-86 15:06"))
 (Supers GenericAccount)
 (ClassVariables)
 (InstanceVariables (InterestRate .05 doc
                                (* default value of
                                interest rate is 5
                                percent)))
 (MethodFns))
```

**Figure 6.9.** Inserting instance variable InterestRate in **Savings**

## 6.4.2 Defining a ComputeInterest Method

Now you must define a method that computes the interest earned, based on the current balance, and then adds the interest to the balance. For simplicity, we ignore the length of time various amounts have been in the account.

Add the method **ComputeInterest** to **Savings** and give it the following body:

```
(←@  Balance
   (PLUS
          (@  Balance)
          (TIMES
                 (@  InterestRate)
                 (@  Balance))))
(@ Balance)
```

Your method should look like Figure 6.10.

```
DEdit of function Savings.ComputeInterest
(Method
 ((Savings ComputeInterest)
  self)                (* edited:
                       "15-Nov-86 15:15")
                       (* credits account with
                       interest based on the interest
                       rate and the current
                       balance)
 (←@
  Balance
  (PLUS (@ Balance)
        (TIMES (@ InterestRate)
               (@ Balance))))
 (@ Balance))
```

**Figure 6.10.** The method **ComputeInterest**

### 6.4.3 Simple Test of Savings

Now you can test **Savings** in the same way that you tested **GenericAccount**. Create an instance of **Savings** called **MySavings**. Credit it with 1000 and debit it by 500. (Remember, it inherits these methods from **GenericAccount**.) Now send **MySavings** the message **ComputeInterest**. Finally, inspect **MySavings** to insure that everything worked properly (Figure 6.11).

```
All Values of Savings ($ MySavings).
CreditHistory ((1000 . "15-Nov-86 15:23:5
DebitHistory  ((500 . "15-Nov-86 15:24:05
Balance       525.0
InterestRate  .05
```

Figure 6.11. Results of **Credit**, **Debit**, and **ComputeInterest** on **MySavings**

## 6.5 Defining Checking

Next, define the class, **Checking**. Like your previously defined class, **Savings**, **Checking** has inherited all of **GenericAccount's** methods and variables. You only need to add one new instance variable and a method to write checks.

### 6.5.1 Add Variables, Values, and Documentation

The instance variable will contain the checking history. It will be a list of triples. Each triple will contain the check's number, its amount and its date. When you are finished, your class should look like Figure 6.12.

```
DEdit of CLASSES #.($ Checking)
((MetaClass Class doc      (* simulates a
                            standard checking
                            account)
        Edited:            (* edited;
                            "15-Nov-86 13:51"))
 (Supers GenericAccount)
 (ClassVariables)
 (InstanceVariables (CheckingHistory
                     NIL doc
                          (* keeps list of
                          (CheckNumber
                          Amount Date) triples))
         )
 (MethodFns))
```

Figure 6.12. The class **Checking**

## 6.5.2 Defining a WriteCheck Method

You need a method to store check numbers, amounts, and dates in CheckingHistory, and to update the Balance.

Add the method **WriteCheck** to **Checking** with the following body:

```
(←@ CheckingHistory
   (CONS
         (LIST CheckNumber Amount (DATE))
         (@ CheckingHistory)))
(← self Debit Amount)
```

Your defined method looks like Figure 6.13. Notice the last line of **WriteCheck**. You have already created a method to debit an account so **WriteCheck** can use this method. **Checking** inherits the method, **Debit**, from **Generic Account**. The debit message can be sent to *self*; that is, to the instance of **Checking** which received the **WriteCheck** message. This example shows a way that methods can be built out of simpler methods by having them send messages to *self*, just as functions can be built out of calls to simpler functions. Note: when sending a message, *self* cannot be omitted as it can in the **@** and **←@** expressions.

**DEdit of function Checking.WriteCheck**

```
(Method ((Checking WriteCheck)
         self CheckNumber Amount)
                        (* edited:
                        "15-Nov-86 15:45")
                        (* adds a (CheckNumber
                        Amount Date) triple to
                        CheckingHistory and debits
                        the account)
      (←@
       CheckingHistory
       (CONS (LIST CheckNumber Amount
                   (DATE))
             (@ CheckingHistory)))
      (← self Debit Amount))
```

**Figure 6.13.** The method **WriteCheck** for **Checking**

## 6.5.3 Simple test of Checking

Now test **Checking**. Create an instance called **MyChecking** and credit it with 100. Write a check by sending the message **WriteCheck** with the arguments 100 and 25.00. Then inspect your instance. The result should look like the Inspector window shown below in Figure 6.14.

**All Values of Checking ($ MyChecking).**

```
CreditHistory    ((100 . "15-Nov-86 15:50:
DebitHistory     ((25.0 . "15-Nov-86 15:50
Balance          75.0
CheckingHistory  ((100 25.0 "15-Nov-86 15:
```

**Figure 6.14.** Inspection of **MyChecking**

## 6.6 Testing NOWAccount

Because a NOW account is a combination of checking and savings, the class **NOWAccount** inherits everything it needs except documentation. All you need to do is create an instance. Name the instance **MyNOW** and test it by crediting it with 500, debiting it by 100, writing a check for 55.55 and asking for the interest. Then inspect it. Your results should look like Figure 6.15.

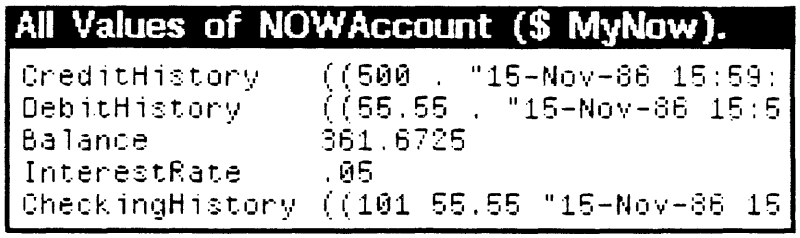



**Figure 6.15.** Inspection of **MyNOW**

This example is expanded in later chapters. If you are not continuing through the primer at this time, you should save your LOOPS program so that you can load it in again later. See Section 5.1 for instructions on how to do this.

# 7. STRATEGIES FOR ORGANIZING OBJECTS

Designing the class lattice for a LOOPS program is central to the effective use of LOOPS. A carefully designed lattice can result in a simpler and more effective problem solution. This chapter presents three typical strategies for organizing objects: *elision through inheritance*, *incremental customization*, and *factoring functionality*.

These strategies are a starting point. Often an application will require a combination of two or even all three strategies. With experience you will discover strategies of your own.

## 7.1 Elision Through Inheritance

Elision through inheritance is the most basic strategy used to organize a lattice of objects. The word "elide" means to eliminate or to leave out. When creating classes, it is not necessary to specify each class completely. Instead, common characteristics can be grouped in a super object. To use elision through inheritance, determine which characteristics are common to all objects that must be organized. The top-most class in the lattice has variables and methods to implement those characteristics. Each successive specialization adds only those characteristics which make it different from its supers. Thus, parts of the description of a given class can be elided, making the construction of a set of classes much easier.

Elision through inheritance is useful for defining complex taxonomically related networks of objects. The Animal lattice used to introduce classes in Chapter 1 is an example. Such a network is often called an "is-a" hierarchy, e.g. a woman is-a person, a person is-a animal, etc.

An example of organizing objects with elision through inheritance is shown in Figure 7.1.

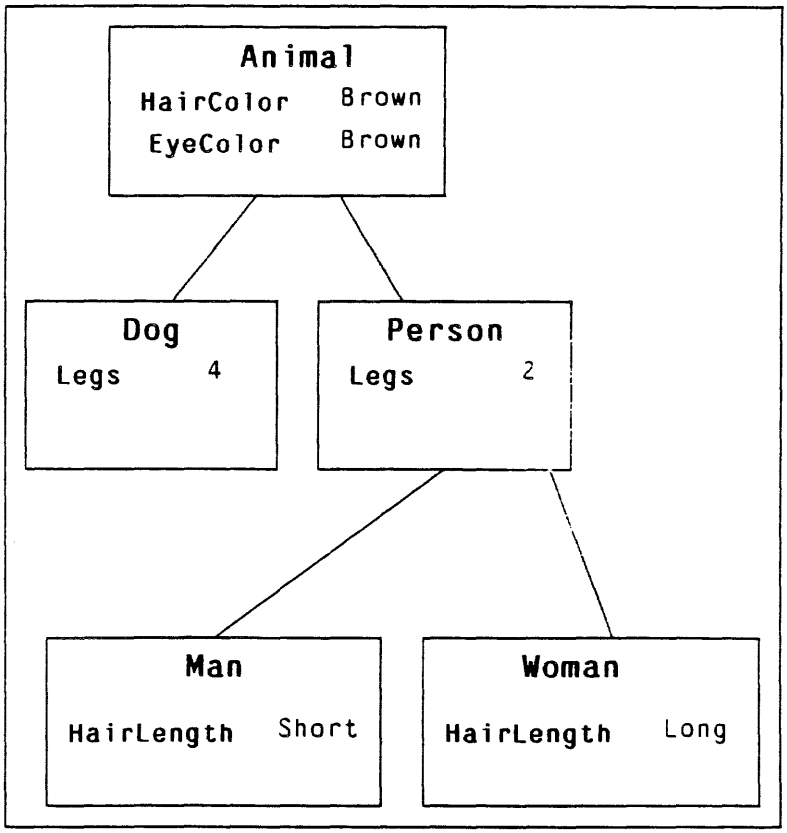



**Figure 7.1.** Organization of objects for elision through inheritance

In this example, the lattice is used to describe classes. The description of each class is simplified by class inheritance. The class, **Man**, inherits the instance variables, HairColor and EyeColor, along with their default values from **Animal**. Man also inherits the instance variable Legs along with its default value from **Person**. Due to inheritance, **Man** has four instance variable/default value pairs. Only one of these pairs is actually defined in the class **Man**: the other three can be elided because of LOOPS inheritance.

When objects are organized using elision through inheritance, usually only the objects lower in the inheritance lattice are used to create instances. Although the objects higher in the lattice do represent real or existing things, they are primarily used for their taxonomic or classifying function.

## 7.2 Incremental Customization

Incremental customization is another way to simplify the specification of classes by using inheritance. In incremental customization, certain more general classes are not designed to have instances; they are meant to be combined with other classes to create new classes that do have instances. An example of this strategy is shown in Figure 7.2.

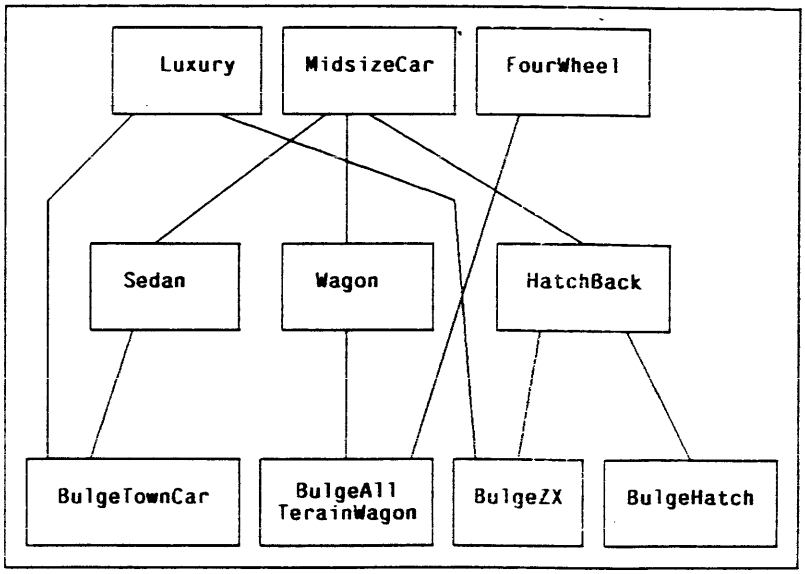



**Figure 7.2.** Lattice showing example of incremental customization

Figure 7.2 shows a series of automobile classes. **MidsizeCar** has three specializations, **Sedan**, **Wagon** and **HatchBack**. Each of the classes in the bottom row represents a specific model of car. These bottom classes inherit from one of the specializations of **MidsizeCar**. Most also inherit from either **Luxury** or **FourWheel** in the top row.

The classes **Luxury**, **MidsizeCar**, and **FourWheel** are not designed to be used alone. They are not complete enough to be instantiated. Rather, they are designed to be used together with the classes in the middle row. Each provides a package of features that can be combined to create a description of specific automobile models. For instance, **Luxury** can be mixed together with any one of the three middle classes to produce a specific model.

The key to using incremental customization is recognizing a generic set of prototypes in your problem domain. It must be possible to describe most problem situations in terms of unique combinations of the generic prototypes.

For example, consider an expert system to diagnose assembly line faults based on specific error reports from a set of standard tests. With incremental customization each test is represented as a high level class. A particular failed product is represented by an instance of a class that inherits from each class representing each test the product failed.

## 7.3 Factoring Functionality

Organizing objects to factor functionality is done by grouping related variables and methods for an object into a set of multiple supers. When objects are defined in this way, only the class lowest in the inheritance lattice is used to create instances.

Factoring functionality is a strategy useful for developing programs with several distinct major components. Super classes are created to represent those major components. This allows for a modular partitioning of distinct system components. An example of factoring functionality is shown in Figure 7.3.

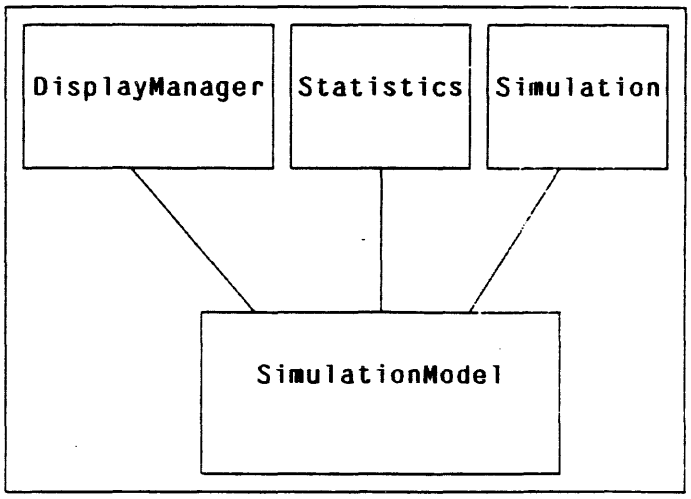



**Figure 7.3.** Lattice showing example of factoring functionality

Instances of **SimulationModel** will simulate some process, collect statistics, and produce an animation on the screen. Each instance includes all three capabilities because all three are inherited by **SimulationModel**. To modify the statistics capabilities it is only necessary to edit the **Statistics** class.

The three super classes, **DisplayManager**, **Statistics**, and **Simulation**, each contribute their definitions to **SimulationModel**. Instances of the super classes alone are not instantiated. If some aspect of the functionality of **SimulationModel** needs to be changed, only one of its supers needs to be edited.

# 8. SPECIALIZING METHODS

Specializing methods can be more complex then specializing variables. Class and instance variables are inherited from a class to its specializations. Methods are also inherited in this manner. With methods, however, there are additional concerns.

When specializing a class, you can specify variables in the new specialization in three ways:

- Inherit a variable and its defaults as specified in the super. In LOOPS, this is the default way of specifying variables in specializations.
- Inherit a variable but change the default from what is specified by the super.
- Specify variables in the specialization that are not inherited from the super.

With methods, specialization is more complex. You might, for example, wish a specialized method to set a few instance variables, run the super's method, then reset the instance variables. Such finer grained control of method inheritance is discussed in this chapter.

## 8.1 ←Super and ←SuperFringe

LOOPS provides two special versions of ← (the **Send** operation) which facilitate the incremental specialization of methods. They are **←Super**, pronounced "send super" and **←SuperFringe**, pronounced "send super fringe". They allow you to make changes to a method contained in a class that is higher in the class hierarchy, without changing the original method.

When **←Super** is placed in one of a specialization's method definitions, a super's version of the same method is run. Execution then returns to the method of the object instance which originally received the message. In other words, **←Super** forwards a message up the class hierarchy and causes the next more general version of the method to be invoked. **←SuperFringe** is similar. If the receiving object has multiple supers, it will forward the message to all of them, possibly causing several versions of the method to be invoked. **←Super** stops once it finds one version of the appropriate method.

The syntax is the same for both:

(**←Super** *object selector arg1 arg2 ...*)

(**←SuperFringe** *object selector arg1 arg2 ...*)

*Object* should be **self**. *Selector* is not evaluated. It is often the case that a specialized method has exactly the same arguments as the more general method. In this case, the following shorthand may be used:

**(←Super)**

Before attempting the following examples, be sure the Bank Account example from Chapter 6 is loaded.

As a first example of specializing methods, new methods with the selector **Status** can be created for the **Checking** and **Savings** classes. Then **NOWAccount** can be given two different specializations of **Status** to demonstrate how **←Super** and **←SuperFringe** behave. All of the **Status** methods will simply print out a message telling something about the account.

For **Checking, Status** will print the checking history of the account. For **Savings, Status** will print out the current interest rate. Create these methods now. The body of **Checking.Status** should be:

```
(PRIN1 "THE CHECKING HISTORY OF YOUR ACCOUNT IS:  "
PROMPTWINDOW)
(PRINT (@ CheckingHistory) PROMPTWINDOW)
```

and the body of **Savings.Status** should be:

```
(PRIN1 "THE INTEREST RATE FOR YOUR ACCOUNT IS:  ")
(PRINT (@ InterestRate))
```

The class **NOWAccount** inherits from both **Checking** and **Savings**. Each one now has a method called **Status**. Does **NOWAccount** inherit both of them? If not, which one does it inherit? Create an instance of **NOWAccount** named **MyNow** (if **MyNow** does not still exist). Try sending the message **Status** to **MyNow** and see what happens. The result should be something like Figure 8.1.

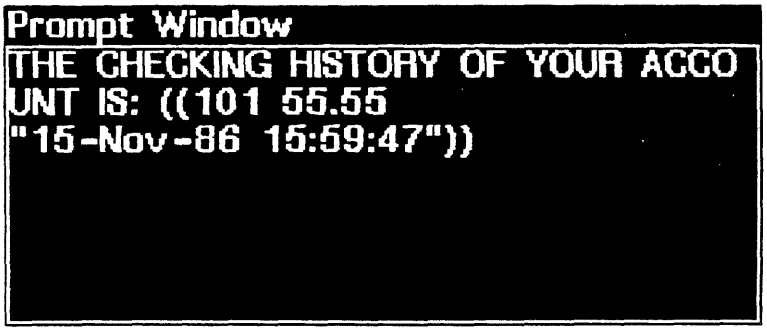



**Figure 8.1.** Sending **Status** to **MyNow**

Only one of the two methods was invoked. Looking at the class definition for **NOWAccount**, notice that **Checking** is first on the list of supers. When a class has multiple supers, they are tried in left to right order to find any inherited parts. Thus, classes can not inherit conflicting characteristics.

Now two different specializations of **Status** will be created, one using **←Super** and then one using **←SuperFringe**. First, select **SpecializeMethod** from the submenu of **Add(AddMethod)** in the editing menu on **NOWAccount**. Then select **Status** from the method menu that pops up. When the

editing template appears, note that ←**Super** is already present as in Figure 8.2.

```
DEdit of function NOWAccount.Status
(Method ((NOWAccount Status)
          self)          (* edited;
                            "18-Nov-86 17:49")
                         (* Method to print status of
                            an account.)
        (←Super
         self Status))
```

**Figure 8.2.** The method **Status** for **NOWAccount**

Ordinarily, the next step would be to add some code before and/or after the call to ←**Super** to produce a more specialized method. For now, simply add some documentation and exit.

Try sending the message **Status** to **MyNow** again. The result should be exactly the same as before (see Figure 8.1). The version of the method that ←**Super** finds is the same one originally inherited.

Now edit the method, **NOWAccount.Status**, replacing ←**Super** with ←**SuperFringe** as in Figure 8.3.

```
DEdit of function NOWAccount.Status
(Method ((NOWAccount Status)
          self)          (* edited;
                            "18-Nov-86 17:58")
                         (* Method to print status of
                            an account.)
        (←SuperFringe
         self Status))
```

**Figure 8.3.** The method **Status** for **NOWAccount**

Send the message **Status** to **MyNow** again. Notice that the **Status** methods for both **Checking** and **Savings** are invoked. This is because both are supers of **NOWAccount**.

←**SuperFringe** is rarely used. This is because it is unusual to find two methods with the same selector that are truly complementary. Methods with the same selector often duplicate and/or conflict with each other's actions. In most cases, ←**Super** is used to add functionality to the methods of the first super on a specialization's supers list.

## 8.2 Specializing a Method in the Bank Account

Neither of the method specializations you have created so far have added any functionality to the method being specialized. This section illustrates how to augment inherited methods.

First, add a specialization of the **Savings** class, named **MinimumBalance** as shown in Figure 8.4.

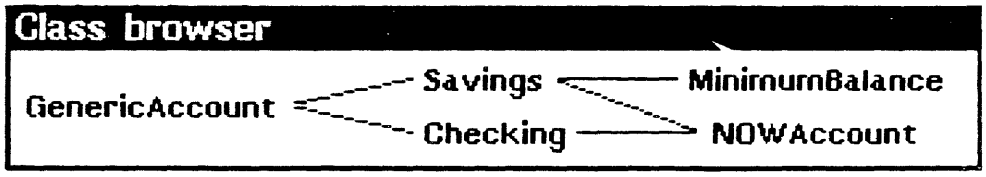



**Figure 8.4.** Class inheritance lattice for Bank Account example

A minimum balance account is a savings account bearing a higher interest rate than a regular savings account. A minimum balance account carries a penalty if the balance goes below the minimum allowed. The penalty is deducted after any debit leaves the account's balance below the minimum balance.

The **MinimumBalance** class needs new variables to specify the minimum balance and the penalty. Since these are the same for all individual accounts, they should be class variables. Add the class variables, **Minimum** and **Penalty** with values of 1000 and 100, respectively.

The **MinimumBalance.Debit** method needs to be specialized. It should first execute the **GenericAccount.Debit** method to actually debit the account. Then, if the balance is below Minimum, **Penalty** should be deducted from **Balance**.

To do this, select **SpecializeMethod** from the submenu of **Add(AddMethod)** and then select **Debit** from the pop-up menu.

Add the following Lisp code after the call to **←Super** in the body:

```
(if       (LESSP (@ Balance) (@ self ::Minimum))
  then    (←Super self Debit (@self  ::Penalty)))
```

The **MinimumBalance.Debit** method will look like Figure 8.5.

**DEdit of function MinimumBalance.Debit**

```
(Method ((MinimumBalance Debit)
         self DebitAmount)
                    (* edited:
                    "18-Nov-86 18:44")

      (* * Adds (CreditAmount , date) to
      CreditHistory and subtracts CreditAmount
      from Balance. Then checks if Penalty
      should be debited.)

        (←Super
         self Debit DebitAmount)
        (if (LESSP (@ Balance)
                   (@ ::Minimum))
            then (←Super
                  self Debit (@ ::Penalty)
                  )))
```

**Figure 8.5.** Method **Debit** for class **MinimumBalance**

Note that this **Debit** method uses the inherited **Debit** method twice whenever the balance is low.

To try out this new **Debit** method, create an instance of **MinimumBalance** named **MyMinimum**. Credit it with 5000 and debit it by 4500. The balance returned should be 400 rather than 500.

# 9. ACTIVE VALUES AND ACCESS-ORIENTED PROGRAMMING

*Access-oriented programming* is a programming paradigm in which fetching or storing data activates computations. In LOOPS, access-oriented programming is implemented using objects called active values. When an active value is read or set, a desirable side effect happens automatically.

The following sections describe how to define LOOPS active values and how to use them to monitor program states, to guard variables values, and to propagate values among objects.

This chapter continues to work with the Bank Account example. Load that example now if it is not currently loaded.

## 9.1 Defining Active Values

To make the value of a variable active, the value is replaced by an active value object. When an attempt is made to access the value, the active value object does some computation, using its methods. The actual value may be stored inside the active value object or it may be computed by that object.

The process for defining and using active values can be divided into four basic steps:

(1) **Choose an Active Value Class.** LOOPS provides a set of classes designed to create various kinds of active values. To see these classes, browse the class **ActiveValue**. The portion of the lattice initially concentrated on is shown in Figure 9.1.

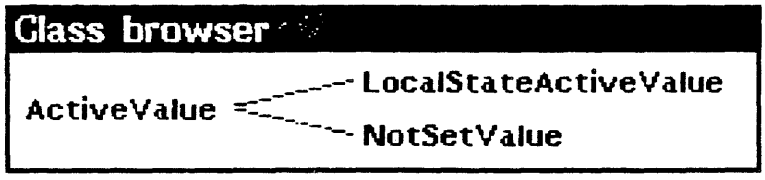



**Figure 9.1.** A portion of the **ActiveValue** lattice

**ActiveValue** is an *abstract class*. It contains all variables and methods common to active values but it is not complete enough to function on its own. **ActiveValue** is never instantiated directly. **LocalStateActiveValue** and **NotSetValue** are specializations of **ActiveValue**.

(2) **Specialize the ActiveValue.** Some of the active value classes provided by LOOPS can be used without change. Others must be specialized to yield the desired effects.

(3) **Create Instances.** As with other classes, the actual work is done by instances of active value classes. Each active value is a separate instance of the appropriate active value class.

(4) **Install the Active Value.** Active values all inherit the ability to install themselves. This is done with the message **AddActiveValue**. The syntax of this message is:

(← *self* **AddActiveValue** *containingObj varName*)

*self* is the active value which is installed, *containingObj* is the object in which the active value is installed, and *varName* is the variable on which the active value is installed. There are optional arguments and we will not use them in these examples.

To illustrate how active values are typically used, we begin by discussing how **NotSetValue** works. **LocalStateActiveValue** and its specializations are used in the examples presented later in this chapter.

When an instance is created, the instance variables are each bound to an instance of **NotSetValue**. Each instance of **NotSetValue** is an object with a method for finding and setting the default value of the variable to which it has been bound. If an instance variable is directly bound to a local value, that value replaces the instance of **NotSetValue** (which was bound to the value of the variable instantiation). When an attempt is made to access the value of an instance variable which has no local value, **NotSetValue** finds the default value in one of the supers and copies it into the instance. Since default values are often replaced with local values, this approach avoids fetching default values unnecessarily.

We continue by demonstrating how to apply the four basic steps to an example.

## 9.2 Using Active Values to Monitor State

Generally there are certain variables whose values are critical to the running of a program. Monitoring such variables during execution can be helpful in understanding and controlling the program's behavior. Active values provide an easy way to do this monitoring: to each critical variable, an active value is attached. Each active value prints its variable's value or updates a display each time the value is changed.

LOOPS provides **Gauges**, a large selection of display classes which facilitate this technique. **Gauges** are explained in Chapter 10. In this section you learn how to do monitoring by using active values directly. The value of the **Balance** variable in the **GenericAccount** class is made active so that it prints itself in the prompt window whenever it is changed.

*Step one* chose an active value class. A class is needed to keep track of the actual value and take some action whenever it is changed. For this purpose, LOOPS provides the

**LocalStateActiveValue** class. This is the most versatile of the active values classes. **LocalStateActiveValue** contains an instance variable called **localState**. The variable's actual value resides in **LocalState**. **LocalStateActiveValue** also has methods for reading and setting this value. They are called **GetWrappedValue** and **PutWrappedValue**.

*Step two* specialize the active value class. As is, **LocalStateActiveValue** has no real effect; it simply sets or returns its **localState**. In this example, **LocalStateActiveValue** is specialized to create an active value that prints the value whenever it is set.

Create a specialization of **LocalStateActiveValue** called **PrintValueAV**.

Whenever an attempt is made to set a value that is active, the message **PutWrappedValue** is automatically sent to that active value instance. The message is called **PutWrappedValue** because the active value can be viewed as being wrapped around the real value.

Now specialize **PutWrappedValue**. Bring up **PrintValueAV's** editing menu and select **SpecializeMethod** from the submenu of **Add(AddMethod)**. Then select **PutWrappedValue** from the menu that pops up.

Code to print the value needs to be added before the call to **←Super**. Unlike the methods specialized in Chapter 8, **PutWrappedValue** is part of the LOOPS system. Nevertheless, methods provided by the LOOPS system can be specialized just as those created by the user. As long as you know what a method does, you can specialize it.

Add a print statement like the one shown in Figure 9.2 before the **←Super** and exit the editor.

```
DEdit of function PrintValueAV.PutWrappe
(Method ((PrintValueAV PutWrappedValue)
         self containingObj varName
         newValue propName type)
                  (* edited:
                  "16-Nov-86 17:19")

    (* * Print newValue and Replace the
    value wrapped in the active value)

      (PRINT (CONCAT "Your new "
                     varName " is: "
                     newValue)
             PROMPTWINDOW)
      (←Super
       self PutWrappedValue
       containingObj varName newValue
       propName type))
```

**Figure 9.2.** The method **PrintValueAV.PutWrappedValue**

*Step three* create an instance of the active value class. Create an instance of **PrintValueAV**. Name it **PrintValueAV1**. If you were going to use **PrintValueAV** in a real application, you would probably need a number of different instances in order to make different values active at the same time. The 1 on the name of the instance anticipates this.

*Step four* install the active value. If the **GenericAccount** instance **MyGeneric** is not currently in your environment, you should load it now or create it from the class, **GenericAccount**. Put **PrintValueAV1** into **MyGeneric** by typing:

```
(← ($ PrintValueAV1) AddActiveValue ($ MyGeneric) 'Balance)
```

This sends **PrintValueAV1** the message to **AddActiveValue** to the **Balance** instance variable of **MyGeneric**. That is, **PrintValueAV1** installs itself as a wrapper on the value of **Balance**.

To see if everything is working, send **MyGeneric** some **Credit** and **Debit** messages. The resulting balance should be printed in the prompt window.

Before going on, inspect **MyGeneric**. Instead of simply a number for the value of **Balance**, you should see something like:

```
#.($AV PrintValueAV (PrintValueAV1 &) (localState 800))
```

The **$AV** indicates that this is an active value. **PrintValueAV** indicates which active value class is being used and **(PrintValueAV1 &)** indicates the particular instance. The & is used by the Interlisp-D inspector to indicate additional embedded list structure. Here the & represents the **PrintValueAV1** instance. Note that **localState** is actually embedded inside the **PrintValueAV1** instance. It appears in the Browser for convenience so you do not have to inspect **PrintValueAV1** to see the value.

## 9.3 Using Active Values to Guard Variables

Often it is useful to restrict the values of variables. An active value can be used to restrict the value of a variable by taking some action whenever an attempt is made to set the variable to an improper value. The action might be to cause an error break, refuse to set the variable, or simply print a warning. For simplicity, we demonstrate the latter strategy.

The goal here is to create an active value that prints a warning if the balance in a NOW account goes below 100. **PrintValueAV1** is an active value class with a specialized **PutWrappedValue** method that prints the balance. This is used as a starting point to create a specialization of **PrintValueAV** called **WarnValueAV**. Once again the **PutWrappedValue**

method is specialized. This time the specialized version previously created is further specialized. As before, **PutWrappedValue** prints out the balance. In addition, it monitors the balance and prints a warning when necessary.

Create a specialization of **PrintValueAV** and name it **WarnValueAV**. Bring up **WarnValueAV's** editing menu and select **SpecializeMethod** from the submenu of **Add(AddMethod)**. Then select **PutWrappedValue** from the menu that pops up. Add:

```
(if (LESSP newValue 100) then (CLRPROMPT) (PRINT
(CONCAT "WARNING: " varName " is less then 100")
PROMPTWINDOW))
```

before the ←**Super** call as shown in in Figure 9.3.

**DEdit of function WarnValueAV.PutWrapp**

```
(Method
 ((WarnValueAV PutWrappedValue)
  self containingObj varName newValue
  propName type)     (* edited:
                     "17-Nov-86 19:45")

     (* * Print newValue and Replace the
     value wrapped in the active value)

 (if (ILESSP newValue 100)
     then (CLRPROMPT)
          (PRINT (CONCAT "WARNING: "
                         varName
                   " is less than 100"
                         )
                 PROMPTWINDOW))
 (←Super
  self PutWrappedValue containingObj
  varName newValue propName type))
```

**Figure 9.3.** The method **WarnValueAV.PutWrappedValue**

Create an instance of **WarnValueAV** and add it to **MyNow** by typing:

```
(← (← ($ WarnValueAv) New) AddActiveValue ($
MyNow) 'Balance)
```

This is a quick way to create and install a new instance of an active value. An instance of the **WarnValueAV** active value is created and the message **AddActiveValue** is sent to it in one expression. Note that the instance was not given a name. Unlike classes, instances are not required to have names. When an instance is immediately put into some other structure the name can be left out.

Try sending some **Credit** and **Debit** messages to **MyNow**. You should see a warning in the prompt window whenever the balance is below 100.

Look at Figure 9.3. To recap how the messages are passed consider what happens when **MyNow** is sent the message **Debit**. First an attempt is made to set **Balance**. This causes the message **PutWrappedValue** to be sent to the instance of **WarnValueAV** that is wrapped around the instance variable **Balance**. If the balance is low, **WarnValueAV.PutWrappedValue** prints a warning. Using **←Super**, the **WarnValueAV.PutWrappedValue** method then forwards the message to **PrintValueAV**. **PrintValueAV.PutWrappedValue** prints the balance. Finally, the **PutWrappedValue** message is again forwarded with a **←Super** and **LocalStateActiveValue.PutWrappedValue** actually sets **localState**.

## 9.4 Using Active Values to Propagate Values

Sometimes the value of a variable depends upon the values of other variables. Such a variable is referred to in mathematics as a dependent variable and those upon which it depends are called independent variables. The value of a dependent variable should change whenever any of the independent variables changes. In LOOPS, this relationship is implemented with an active value stored in the dependent variable. Any attempt to get the dependent variable's value results in the active value checking the independent variable(s) upon which the dependent variable is based. Of course, any attempt to directly set a dependent variable should be prohibited.

To illustrate this technique, a class that computes **Balance** on demand from **CreditHistory** and **DebitHistory** is created. An active value in **Balance** adds up all the credits and subtracts all the debits whenever **Balance** is read. In addition, **Balance** is protected from being set directly. Clearly, this is not a particularly efficient way to keep track of an account balance. However, this is a viable way to handle a balance that is needed only infrequently.

Use the **NoUpdatePermittedAV** class, one of the specializations of **LocalStateActiveValue** provided by the LOOPS system. **NoUpdatePermittedAV** has a specialization of the **PutWrappedValue** method that prevents **localState** from being changed. The method invoked when an attempt is made to read the value of **localState** is **GetWrappedValue**. A specialization of **GetWrappedValue** is needed to compute the value of **balance** instead of simply accessing it.

To begin, specialize **GenericAccount** and name the specialization **CompBalAccount**. This new class inherits **Credit** and **Debit** methods from **GenericAccount**. New versions of these methods must be created. These new versions will be very similar to the old ones, but they will not update **balance**. To save some work, you can copy both methods from **GenericAccount** to **CompBalAccount**. First, box

**CompBalAccount**. Next select **Copy(CopyMethodTo)** from **GenericAccount**'s edit menu. When the menu of methods pops up, select **Credit** and **Debit**. Now edit both methods to remove the parts that update **Balance** and change the documentation appropriately. **CompBalAccount**'s **Credit** and **Debit** method should look like Figure 9.4 and Figure 9.5.

**DEdit of function CompBalAccount.Credit**

```
(Method
 ((CompBalAccount Credit)
  self CreditAmount) (* edited:
                          "19-Jan-87 15:32")

      (* * Adds (CreditAmount , date) to
      CreditHistory)

 (←@
  CreditHistory
  (CONS (CONS CreditAmount (DATE)
              (@ CreditHistory))))
 (@ Balance))
```

**Figure 9.4.** Version of **Credit** that does not update **Balance**

**DEdit of function CompBalAccount.Debit**

```
(Method
 ((CompBalAccount Debit)
  self DebitAmount)  (* edited:
                          "19-Jan-87 15:33")

      (* * Adds (DebitAmount , date) to
      DebitHistory)

 (←@
  DebitHistory
  (CONS (CONS DebitAmount (DATE)
              (@ DebitHistory))))
 (@ Balance))
```

**Figure 9.5.** Version of **Debit** that does not update **Balance**

Both methods still access **Balance** in order to return the new account balance but neither one updates it.

Now, create a specialization of the active value class, **NoUpdatePermitted**. Call it **TotalBalAV**. The desired

behavior is to recompute **Balance** whenever it is read. To achieve this, the **GetWrappedValue** method is specialized. **GetWrappedValue** does not actually read a value in this case, so the **←Super** will be replaced by the body of the method. This is shown in Figure 9.6. Note that the *object* argument to **@** can not be left out. This is because *self* refers to the instance of the active value while **containingObj** refers to the instance which contains the active value.

**DEdit of function TotalBalAV.GetWrappedV**

```
(Method ((TotalBalAV GetWrappedValue)
         self containingObj varName
         propName type)
                    (* edited:
                    "19-Jan-87 15:47")

    (* * Compute the balance from credit and
    debit histories.)

      (DIFFERENCE
       (for CreditItem
          in (@ containingObj
                CreditHistory)
          sum (CAR CreditItem))
       (for DebitItem
          in (@ containingObj
                DebitHistory)
          sum (CAR DebitItem))))
```

**Figure 9.6.** The method **TotalBalAV.GetWrappedValue**

Now create an instance of **CompBalAccount** named **MyCompBal**. Use **AddActiveValue** to install an instance of the **TotalBalAV** active value:

```
(← (← ($ TotalBalAV) New) AddActiveValue ($ MyCompBal) 'Balance)
```

Test this example by sending both **Credit** and **Debit** messages to see that the balance is returned correctly. Also attempt to directly set the value of **Balance** by typing something like the following:

```
(←@ ($ MyCompBal) Balance 40000000)
```

**NoUpdatePermittedAV.PutWrappedValue** responds to this with an error message. If you wanted something else to happen -- send a message to the police for instance -- you would specialize the **NoUpdatePermittedAV.PutWrappedValue** method.

## 9.5 Nesting Active Values

At times, more than one action needs to be associated with an active value. Instead of creating a new active value class that combines the functions of several existing active values, the active values can be nested. This technique is only briefly introduced here. See the chapter *ANNOTATED AND ACTIVE VALUES* in *The LOOPS Reference Manual* for more information.

Active values are nested by installing them one after the other on the same value. The order in which they are nested is controlled by a property of active values called *wrapping precedence*. When **AddActiveValue** tries to install a new active value where an active value is already present, it sends the message **WrappingPrecedence** to the active value that is being installed. In the simplest case, the message is sent to *self* (the active value) and either **T** or **NIL** is returned. **T** means wrap the new active value around the outermost active value(s) and **NIL** means put the new active value inside the innermost one.

It is also possible to exert finer control by using numerical precedences. The method **WrappingPrecedence** returns 100 by default. In order to control the order of nesting, **WrappingPrecedence** must be specialized to return **T**, **NIL** (as in simple case described in the preceding paragraph) or an appropriate number.

As an example, the value of **Balance** is once again guarded. This time **Balance** is wrapped with a **PrintValueAV** and with a new version of the warning active value. The active value that prints the warning will be a specialization of **LocalStateActiveValue**. Create it now using the name **NestWarnValueAV**.

Two methods, **PutWrappedValue** and **WrappingPrecedence**, need to be specialized. **PutWrappedValue** checks the value and prints a warning if needed. This is exactly what **WarnValueAV.PutWrappedvalue** does, so just copy it. To do this, first box **NestWarnValueAV**. Next bring up **WarnValueAV**'s editor menu and select **Copy(CopyMethodTo)**. Finally, select **PutWrappedValue** from the menu that pops up.

It would be best to have the warning message printed before the balance is printed. In order for this to happen, the warning active value should be the outermost one. This order can be guaranteed by giving it a wrapping precedence of **T**. Specialize **NestWarnValueAV**'s **WrappingPrecedence** method so that it returns **T** and does nothing else. In other words, *replace* the **←Super** call with T. If this were not done, **NestWarnValueAV** would run the **WrappingPrecedence** method it inherits from **LocalStateActiveValue**.
**LocalStateActiveValue.WrappingPrecedence** returns the default precedence of 100.

Now recall that **MyGeneric** already has an instance of **PrintValueAV** installed in it. An instance of **NestWarnValueAV** needs to be installed:

```
(← (← ($ NestWarnValueAV) New) AddActiveValue ($ MyGeneric) 'Balance)
```

Test the results as you have before by sending some **credit** and **debit** messages. Push the balance below 100 and note the order in which the messages are printed out. The outer active value is triggered first and prints its warning. It then passes the new value to the inner active value which prints the value.

To see the effect on **($ MyGeneric)**, inspect it. It appears that **NestWarnValueAV** is the sole value of **Balance**. Inspect this value by highlighting it with the left button. Then hold down the middle button and select **Inspect**. **PrintValueAV**, the inner active value of **MyGeneric**'s **Balance** variable, appears.

## 9.6 A Final Note On Active Values

Active values are quite powerful, but they should be used very judiciously in LOOPS programs.

The use of active values makes programs more difficult to follow and debug. This is because they tend to point all over the program and their pointers remain hidden from the outside. Active values should only be used if there is no other way to accomplish the same thing or if they greatly simplify the program.

**Xerox Artificial Intelligence Systems**
**250 North Halstead Street**
**P.O. Box 7018**
**Pasadena, California 91109-7018**