XLISP 2.0 OBJECTS PRIMER

by

Tim I Mikkelsen

February 3, 1990

One of the features in the design of XLISP is object-oriented programming. This primer is intended to serve as a very brief introduction to the object facilities of the XLISP 2.0 dialect of LISP. Note that the object features of XLISP are not based on other existing object definitions in other LISP dialects. If you find problems in the primer, I'd appreciate hearing.

Tim Mikkelsen
4316 Picadilly Drive
Fort Collins, Colorado 80526

PROGRAMMING STYLES

There are many programming paradigms (models). Some of the paradigms are procedural, functional, rule-based, declarative and object-oriented. A language can have aspects of one or many of these programming models.

Procedure-Oriented

The programming paradigm most people are familiar with is the procedural style. The primitives in procedural programming are: subroutines and data structures. Through these primitives, programmers have some limited abilities to share programs and program fragments. C and Pascal are examples of procedural languages. Some procedural languages (such as Modula and ADA) have extensions that provide for better sharing of code.

Object-Oriented Programming

Object-oriented programming is based on the primitives of objects, classes and messages. Objects are defined in terms of classes. Actions occur by sending a message to an object. An object's definition can be inherited from more general classes. Objective-C and C++ both are object-oriented dialects of the C language. Many dialects of LISP have some object oriented extension (Flavors, Common LOOPS, CLOS and others). There currently is standards work proceeding to add object-oriented programming to Common LISP.

Object Oriented Programming

So, the object-oriented programming model is based around the concepts of objects, classes and messages. An object is essentially a black box that contains internal state information. You send an object a message which causes the object to perform some operation. Objects are defined and described through classes.

One aspect of an object is that you do not have to know what is inside - or how it works - to be able to use it. From a programming point of view, this is very handy. You can develop a series of objects for someone to use. If you need to change what goes on inside, the users of the objects should be unaware.

Another aspect of objects is that of inheritance. You can build up new classes from existing classes by inheriting the existing class's functionality and then extending the new definition. For example, you can define a tool class (with various attributes) and then go about creating object instances tool-1, tool-2, and so on. You can also create new sub-classes of the tool class like power-tool. This is also very handy because you don't have to re-implement something if you can

build it up from existing code.

XLISP OBJECT TERMINOLOGY

There are, as previously mentioned, many different languages with object-oriented extensions and facilities. The terminology, operations and styles of these are very different. Some of the main definitions for XLISP's object-oriented extensions are:

Object data type — The OBJECT DATA TYPE is a built-in data type of XLISP. Members of the object data type are object instances and classes.

Object instances — An OBJECT INSTANCE is a composite structure that contains internal state information, methods (the code which respond to messages), a pointer to the object instance's defining class and a pointer to the object's super-class. XLISP contains no built-in object instances.

Class objects — A CLASS OBJECT is, essentially, the template for defining the derived object instances. A class object, although used differently from a simple object instance, is structurally a member of the object data type. It is also contains the linking mechanism that allows you to build class hierarchies (sub-classes and super-classes). XLISP contains two built-in class objects: OBJECT and CLASS.

Message selector — The MESSAGE SELECTOR is the symbol that is used to select a particular action (Method) from the object.

Message — The MESSAGE is the combination of the message selector and the data (if any) to be sent to the object.

Method — The METHOD is the actual code that gets executed when the object receives the Message.

SENDING MESSAGES

The mechanism for sending messages to XLISP objects is via the SEND function. It takes an object, a message selector and various optional arguments (depending on the message selector).

The way that a user creates a new object is to send a :NEW message to a previously defined class. The result of this SEND will return an object, so this is normally preceded by a SETQ. The values shown in the examples that follow may not match what you see if you try this on your version of XLISP - this is not an error. The screens that are used in the various examples are similar to what you should see on your computer screen. The ">" is the normal XLISP prompt (the characters that follow the prompt is what you should type in to try these examples).

```
> (setq my-object (send object :new))
#<Object: #2e100>
```

The object created here is of limited value. Most often, you create a class object and then you create instances of that class. So in the following example, a class called MY-CLASS is created that inherits its definition from the a built-in CLASS definition. Then two instances are created of the new class.

```
> (setq my-class (send class :new '()))
#<Object: #27756>

> (setq my-instance (send my-class :new))
#<Object: #27652>

> (setq another-instance (send my-class :new))
#<Object: #275da>
```

CLASSES

Previously, a :NEW message was used to create an object. The message used to see what is in an object is the :SHOW message.

```
> (send my-class :show)
Object is #<Object: #27756>, Class is #<Object: #23fe2>
  MESSAGES = NIL
  IVARS = NIL
  CVARS = NIL
  CVALS = NIL
```

```
|       SUPERCLASS = #<Object: #23fd8>
|       IVARCNT = 0
|       IVARTOTAL = 0
|     #<Object: #27756>
|_____________________________________________________________________
```

From the display of the MY-CLASS object you can see there are a variety of components. The components of a class are:

Class Pointer — This pointer shows to what class the object (instance or class) belongs. For a class, this always points to the built-in object CLASS. This is also true of the CLASS object, its class pointer points to itself.

Superclass Pointer — This pointer shows what the next class up the class hierarchy is. If the user does not specify what class is the superclass, it will point to the built-in class OBJECT.

Messages — This component shows what messages are allowed for the class, and the description of the method that will be used. If the method is system-defined, it will show up in the form of '#<Subr-: #18b98>'. Remember that the class hierarchy (through the Superclass Pointer) is searched if the requested message is not found in the class.

Instance Variables — This component lists what instance variables will be created when an object instance is created. If no instances of the class exist, there are no Instance Variables. If there are 5 instances of a class, there are 5 complete and different groups of the Instance Variables.

Class Variables and Values — The CLASS VARIABLES (CVAR) component lists what class variables exist within the class. The Class Values (CVAL) component shows what the current values of the variables are. Class Variables are used to hold state information about a class. There will be |Bone of each|A of the Class Variables, independent of the number of instances of the class created.

A BETTER EXAMPLE

The example previously shown does work, but the class and instances created don't really do anything of interest. The following example sets up a tool class and creates some tool instances.

```
> (setq my-tools (send class :new '(power moveable operation)))
#<Object: #277a6>

> (send my-tools :answer :isnew '(pow mov op)
                         '((setq power pow)
                           (setq moveable mov)
                           (setq operation op)))
#<Object: #277a6>

> (setq drill (send my-tools :new 'AC t 'holes))
#<Object: #2ddbc>

> (setq hand-saw (send my-tools :new 'none t 'cuts))
#<Object: #2dc40>

> (setq table-saw (send my-tools :new 'AC nil 'cuts))
#<Object: #2db00>
```

So, a class of objects called MY-TOOLS was created. Note that the class object MY-TOOLS was created by sending the :NEW message to the built-in CLASS object. Within the MY-TOOL class, there are three instances called DRILL, HAND-SAW and TABLE-SAW. These were created by sending the :NEW message to the MY-TOOLS class object. Notice that the parameters followed the message selector.

INSTANCES

The following is a display of the contents of some of the previously created instances:

```
> (send drill :show)
Object is #<Object: #2ddbc>, Class is #<Object: #277a6>
  POWER = AC
  MOVEABLE = T
  OPERATION = HOLES
#<Object: #2ddbc>

> (send hand-saw :show)
Object is #<Object: #2dc40>, Class is #<Object: #277a6>
  POWER = NONE
  MOVEABLE = T
  OPERATION = CUTS
#<Object: #2dc40>
```

From the display of these instances you can see there are some components and values. The components of an instance are:

Class Pointer — This pointer shows to which class the current object instance belongs. It is through this link that the system finds the methods to execute for the received messages.

Instance Variables and Values — The Instance Variables (IVAR) component lists what variables exist within the instance. The Instance Values component holds what the current values of the variables are. Instance Variables are used to hold state information for each instance. There will be a group of Instance Variables for each instance.

METHODS

There have been a few of the messages and methods in XLISP shown to this point (:NEW and :SHOW). The following are the methods built into XLISP:

:ANSWER — The :ANSWER method allows you to define or change methods within a class.

:CLASS — The :CLASS method returns the class of an object.

:ISNEW — The :ISNEW method causes an instance to run its initialization code. When the :ISNEW method is run on a class, it resets the class state. This allows you to re-define instance variables, class variables, etc.

:NEW — The :NEW method allows you to create an instance when the :NEW message is sent to a user-defined class. The :NEW method allows you to create a new class (when the :NEW message is sent to the built-in CLASS).

:SHOW — The :SHOW method displays the instance or class.

SENDING MESSAGES TO A SUPERCLASS

In addition to the SEND function, there is another function called SEND-SUPER. The SEND-SUPER function causes the specified message to be performed by the superclass method. This is a mechanism to allow chaining of methods in a class hierarchy. This chaining behavior can be

achieved by creating a method for a class with the :ANSWER message. Within the body of the method, you include a SEND-SUPER form. This function is allowed only inside the execution of a method of an object.

OBJECT AND CLASS

The definition of the built-in class OBJECT is:

```
> (send object :show)
Object is #<Object: #23fd8>, Class is #<Object: #23fe2>
  MESSAGES = ((:SHOW . #<Subr-: #23db2>)
              (:CLASS . #<Subr-: #23dee>)
              (:ISNEW . #<Subr-: #23e2a>))
  IVARS = NIL
  CVARS = NIL
  CVALS = NIL
  SUPERCLASS = NIL
  IVARCNT = 0
  IVARTOTAL = 0
#<Object: #23fd8>
```

Note that OBJECT is a class - as opposed to an "instance-style" object. OBJECT has no superclass, it is the top or root of the class hierarchy. OBJECT's class is CLASS.

```
> (send class :show)
Object is #<Object: #23fe2>, Class is #<Object: #23fe2>
  MESSAGES = ((:ANSWER . #<Subr-: #23e48>)
              (:ISNEW . #<Subr-: #23e84>)
              (:NEW . #<Subr-: #23ea2>))
  IVARS = (MESSAGES IVARS CVARS CVALS SUPERCLASS
         IVARCNT IVARTOTAL)
  CVARS = NIL
  CVALS = NIL
  SUPERCLASS = #<Object: #23fd8>
  IVARCNT = 7
  IVARTOTAL = 7
#<Object: #23fe2>
```

CLASS has a superclass of OBJECT. It's class is itself - CLASS.

A MORE REALISTIC EXAMPLE

The following is an example, using the idea of tools again. It contains a hierarchy of tool classes. The top of the class hierarchy is TOOLS. HAND-TOOLS and SHOP-TOOLS are sub-classes of TOOLS. The example creates instances of these sub-classes. It is possible to extend this example in various ways. One obvious extension would be to create a third tier of classes under HAND-TOOLS that could contain classes like drills, screwdrivers, pliers and so on.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;     File name: tools.lsp
;     Author:          Tim Mikkelsen
;     Description:     Object-oriented example program
;     Language:  XLISP 2.0
;
;     Date Created:    10-Jan-1988
;     Date Updated:    2-Apr-1989
;
;     (c) Copyright 1988, by Tim Mikkelsen, all rights reserved.
;         Permission is granted for unrestricted non-commercial use.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;     Define the superclasses and classes
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
; make TOOLS superclass
;     with a different :ISNEW method
;     added methods are :BORROW and :RETURN
;     class variables are    NUMBER           contains # of tool instances
;                       ACTIVE-LIST contains list of current objects
;     instance variables are       POWER            list - (AC BATTERY HAND)
;                       MOVEABLE    CAN-CARRY or CAN-ROLL or FIXED
;                       OPERATIONS  list
;                       MATERIAL    list - (WOOD METAL PLASTIC ...)
;                       PIECES           list
;                       LOCATION    HOME or person's name
;

(setq tools (send class :new '(power
                        moveable
                        operations
```

```
                        material
                        pieces
                        location)
                       '(number active-list)))
(send tools :answer :isnew '()
                       '((if (null number) (setq number 1)
                                           (setq number (1+ number)))
                        (setq active-list (cons self active-list))
                        (setq location 'home)))
(send tools :answer :borrow '(by-who)
                        '((if (eq location 'home) (setq location by-who)
                                                  (print "you can't"))))
(send tools :answer :return '()
                        '((if (eq location 'home) (print "got it already")
                                                  (setq location 'home))))

;
; make HAND-TOOLS class
;     with a different :ISNEW method
;     new instance variable  WEIGHT           <number> of pounds
;     the rest is inherited from TOOLS
;

(setq hand-tools (send class :new '(weight) '() tools))
(send hand-tools :answer :isnew '(pow op mat parts w-in)
                         '((setq power pow)
                           (setq moveable 'can-carry)
                           (setq operations op)
                           (setq material mat)
                           (setq pieces parts)
                           (setq weight w-in)
                           (send-super :isnew)))

;
; make SHOP-TOOLS class
;     with a different :ISNEW method
;     no new instance variables
;     the rest is inherited from TOOLS
;

(setq shop-tools (send class :new '() '() tools))
(send shop-tools :answer :isnew '(pow mov op mat parts)
                         '((setq power pow)
                           (setq moveable mov)
                           (setq operations op)
                           (setq material mat)
                           (setq pieces parts)
                           (send-super :isnew)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;     Create instances of various tool classes
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(setq hand-drill (send hand-tools :new        ; make an instance - HAND-DRILL
                       '(ac)
                       '(drill polish grind screw)
                       '(wood metal plastic)
                       '(drill drill-bits screw-bits buffer)
                       '2.5))

(setq table-saw (send shop-tools :new         ; make an instance - TABLE-SAW
                       '(ac)
                       'fixed
                       '(rip cross-cut)
                       '(wood plastic)
                       '(saw blades fence)))


(setq radial-arm (send shop-tools :new        ; make an instance = RADIAL-ARM
                       '(ac)
                       'can-roll
                       '(rip cross-cut)
                       '(wood plastic)
                       '(saw blades dust-bag)))
```

The following session shows how to use the tool definitions from this better example. The example starts at the OS shell and brings up xlisp running the file 'tools.lsp'.

```
cmd? xlisp tools
; loading "init.lsp"
; loading "tools.lsp"
> (send hand-drill :borrow 'fred)
FRED

> (send table-saw :return)
"got it already"
"got it already"

> (send hand-drill :borrow 'joe)
"you can't"
"you can't"

> (send hand-drill :return)
HOME
```

So, Fred was able to borrow the HAND-DRILL. When an attempt was made to return the TABLE-SAW, it was already at home. A second attempt to borrow the HAND-DRILL indicated that "you can't" because it was already lent out. Lastly, the HAND-DRILL was returned successfully. (Note that the "got it already" and "you can't" strings show up twice in the display because the methods both print and return the string.)

The following session shows the structure of the TOOLS object:

```
> (send tools :show)
Object is #<Object: #276fc>, Class is #<Object: #23fe2>
  MESSAGES = ((:RETURN . #<Closure-:RETURN: #2dbd0>)
              (:BORROW . #<Closure-:BORROW: #2ddba>)
              (:ISNEW . #<Closure-:ISNEW: #274a4>))
  IVARS = (POWER MOVEABLE OPERATIONS MATERIAL PIECES LOCATION)
  CVARS = (NUMBER ACTIVE-LIST)
  CVALS = #(3 (#<Object: #2cadc>
               #<Object: #2cda2>
          #<Object: #2d0e0>))
  SUPERCLASS = #<Object: #23fd8>
  IVARCNT = 6
  IVARTOTAL = 6
#<Object: #276fc>
```

The two TOOLS sub-classes HAND-TOOLS and SHOP-TOOLS structure looks like:

```
> (send hand-tools :show)
Object is #<Object: #2dab8>, Class is #<Object: #23fe2>
  MESSAGES = ((:ISNEW . #<Closure-:ISNEW: #2d7a2>))
  IVARS = (WEIGHT)
  CVARS = NIL
  CVALS = NIL
  SUPERCLASS = #<Object: #276fc>
  IVARCNT = 1
  IVARTOTAL = 7
#<Object: #2dab8>

> (send shop-tools :show)
Object is #<Object: #2d680>, Class is #<Object: #23fe2>
  MESSAGES = ((:ISNEW . #<Closure-:ISNEW: #2d450>))
  IVARS = NIL
  CVARS = NIL
  CVALS = NIL
  SUPERCLASS = #<Object: #276fc>
  IVARCNT = 0
  IVARTOTAL = 6
#<Object: #2d680>
```

The class HAND-TOOLS has an instance HAND-DRILL which looks like:

```
> (send hand-drill :show)
Object is #<Object: #2d0e0>, Class is #<Object: #2dab8>
  WEIGHT = 2.5
  POWER = (AC)
  MOVEABLE = CAN-CARRY
  OPERATIONS = (DRILL POLISH GRIND SCREW)
  MATERIAL = (WOOD METAL PLASTIC)
  PIECES = (DRILL DRILL-BITS SCREW-BITS BUFFER)
  LOCATION = HOME
#<Object: #2d0e0>
```