Chapter 4
Object-Oriented Development Environment (ODE)

Note: ODE was developed by Phil Burk to support HMSL, the Hierarchical Music Specification Language. ODE was then released as a part of JForth in an effort to promote object-oriented programming. This chapter, therefore, appears in both the JForth manual, and the HMSL manual. When information specific to JForth or HMSL appears, it will be noted as such.

Philosophy

Object-Oriented Programming (OOP) allows you to design programs in a way that more closely matches the real world. In the world, we are surrounded by objects. These objects can be thought of as belonging to different *classes*, for example pens and pencils. These can in turn be thought as belonging to larger, more general classes such as writing implements. In the same way, an object-oriented program involves software based *objects*. If you were writing a program for an airline you would have classes such as airplanes, airports, passengers, etc. Airplanes would have information associated with them such as flight numbers, fuel capacity, altitude, and so on. If you wanted a particular plane to climb to a new altitude, you could send it a message telling it to climb. The airplane object would then update its internal record of its altitude. For programs that are not tied so closely to the physical world, you might define classes of files, tables, arrays, or plots.

Quite often, the class of object that you need is already defined. You can, then, just use it without having to define new ones. If you need something very similar to an existing class but different, you can define a new class that *inherits* the desired qualities of the existing class. The fact that classes can often be used in more than one program allows you to build up a library of classes. This can save time when programming.

A class defines what an object is made of and what it can do. Once you have defined a class, you can create as many instances (objects) of that class as you want. Objects may be thought of as intelligent data structures. Each object knows how to manipulate its own internal data. All objects of a given class will have the same internal structure and will use the same methods for manipulating that data.

Existing Classes in ODE

A few classes have been defined as part of the official ODE package. The most commonly used ones are:

```
OB.BARRAY  = byte array
OB.ARRAY   = long word array
OB.ELMNTS  = list of N dimensional points, can also be thought of
as a 2D array.
OB.LIST    = list of single values
OB.OBJLIST = list of other objects.
```

Hidden Data

The organization of the data and the actual techniques used to manipulate it are *hidden* from users of the object. It effectively isolates the information needed to manipulate an object inside the object itself. This promotes a very modular structure. The code that uses an object doesn't have to know anything about how that object was implemented. This makes it very easy to modify how an object works without having to modify the code that uses it. This can be a great advantage when managing large software projects.

Generic Messages

When you want an object to perform some action, you send it a message. Examples of messages might be PRINT: or CLEAR:. Objects "know" how to respond to messages based on methods defined for their class.

An advantage of object-oriented programming is that you can send the same message to objects of different classes. As an example, you could send a PRINT: message to several different objects. Each one would know how to print its contents in a meaningful form. Some would print a table of values. Some would only print a single value. One advantage of this is that you don't have to memorize a differently named PRINT: function for each type of data structure. In a traditional procedural system you would be writing words like PRINT.ARRAY and PRINT.THING for different data structures.

Another advantage is that you can write generic code. If you have a picture containing different kinds of graphics objects, you don't have to know what they are or how to draw them. Just send each one a DRAW: message and they will each know how to draw themselves.

Tradeoffs

The use of object-oriented programming techniques can simplify software design, speed code development, reduce many types of common errors, and improve maintainability. When you use it for awhile, you will see why so many programmers are using the techniques in ODE and other OOP languages like C++ and Smalltalk.

I know what you're thinking now. There must be a catch. Well, one disadvantage with objects is that there is some extra memory and speed overhead. The object-oriented support code does use some dictionary space. It can also be slightly slower than traditional code. Standard Smalltalk is quite slow because everything is an object, including every variable, constant, etc. This approach is too restrictive for a Forth implementation. ODE was designed for real time music applications and was, therefore, optimized for speed. ODE only makes objects out of the more complex data structures. All ODE objects use common code that has been optimized for speed.

OOP techniques can also seem strange and confusing at first. After using it for awhile, however, you will feel a small "click" in your head and it will all become clear. You may even find, like I do, that it is hard to imagine writing some programs without using OOP.

All things considered, OOP techniques can be a real advantage when developing software. I think you will find it useful and enjoyable.

Origins of OOP

The inspiration for ODE came from Smalltalk, the original object-oriented programming language developed at the Xerox Palo Alto Research Center. In an effort to promote standardization, the syntax of ODE is similar to the syntax for both Smalltalk and NEON, an object-oriented dialect of Forth for the Macintosh.

For more information about object-oriented languages and their characteristics, see the August 1986 issue of *BYTE* magazine. There is also an excellent textbook available called *Smalltalk-80: The Language and Its Implementation* by Goldberg and Robson. Another good text is *Object Oriented Programming* by Brad J. Cox. These references and many others are in the bibliography at the end of this manual.

Terminology

Class - A description of the data contained within an object, and the methods used to manipulate that data.

Object - An instance of a class. Each object has its own data space and a pointer to its class.

Method - A function that is associated with a particular class. By convention, method names generally end in a colon, for example, PRINT: or ADD:.

Superclass - The class from which a new class is derived. All classes are derived ultimately from a root class named OBJECT .

Inheritance - Each new class automatically has available all of the properties of its superclass. This includes all instance variables and all methods. The new class then adds new methods and/or instance variables.

Instance Variable - A data item that is contained within each object of a given class. To access an instance variable from outside an object, you must use only that object's defined methods. Instance variables can be hidden from other code by not providing any methods for accessing them. Instance variables can themselves be objects.

Instantiate - To create an object from its class definition.

Turorial 1 - Creating and Using Objects

Including ODE

ODE is already compiled as part of **HMSL.** **JForth** users can load ODE by entering:

```
INCLUDE? OBJECT JO:LOAD_ODE
```

Creating an Object, *Instantiation*

Creating an object is similar to creating other Forth data structures, like VARIABLE. Enter the class name followed by the name of the object to be defined. Let's create an instance of the integer class OB.INT named MY-INT. Enter:

```
OB.INT MY-INT
```

Please note that this since this is a defining word, it should be used outside of colon definitions, just like VARIABLE and CONSTANT .

You have now defined an object that exists in the Forth dictionary. If you enter the name of an object, it will return its relative address. You can use this address if you want to pass the object as an item on the stack. Enter:

```
MY-INT  .
```

Sending Messages

To make an object do something, you must send it a message. For example, to put a value into this integer object, send a PUT: message to it. The integer will know what to do. In this case it will know to take the value at the top of the stack and store it internally.. You can then send a PRINT: message to verify that this worked. Here is an example:

```
456 PUT: MY-INT
PRINT: MY-INT ( 456 will get printed. )
```

The OB.INT class is not very useful except as an example. It was written as a simple exercise and is often used as a superclass for other classes. When I want an INTEGER I usually use a VARIABLE or a VALUE.

Using Arrays

A more useful class of objects is the array. To create an array of 32-bit words named MY-ARRAY, enter:

```
OB.ARRAY MY-ARRAY
```

The data for the array is stored in dynamically allocated memory that we request from the operating system. This allows small programs to access large amounts of memory whose size can vary as needed. To allocate memory, send a ?NEW: message to the array. The array object will request that enough space for that much data be allocated from a pool of free memory. To allocate room for 10 cells, enter:

```
10 ?NEW: MY-ARRAY .
```

Since each array cell is four bytes, 10*4=40 bytes were just allocated. If the allocation was successful, the address of the allocated memory is returned. If the allocation fails because of insufficient free memory, it will return a zero. You should always check to make sure that you got the memory you requested. No matter how much memory you have installed, you can always run out of free memory. If you are writing programs for others, remember that they may not have as much memory as you do.

Now let's see what is in our array.

```
PRINT: MY-ARRAY
```

When you printed that object, it was probably full of strange numbers. Those are the values that just happened to be in the memory you allocated. To clear it, enter:

```
CLEAR: MY-ARRAY
PRINT: MY-ARRAY
```

Notice that you do not have to use different messages for printing integers and arrays. This means that there will be fewer commands to memorize. You might try sending a CLEAR: message to your integer to see what happens.

To access individual items inside your array, you can use the messages AT: and TO: . For example:

```
98 4 TO: MY-ARRAY ( Store a 98 at item number 4)
101 3 TO: MY-ARRAY
0 AT: MY-ARRAY . ( Fetch and print value in cell 0, the 1st
cell )
PRINT: MY-ARRAY
```

Finding an item in an Array

So far this looks like a pretty standard array. (See ARRAY in the main glossary for an example of a standard array.) Because it is an object, however, it can be smarter than a standard array. Let's ask the array to find a value inside of it. Assuming you entered the examples above, let's ask it to find the first occurence of 101. Enter:

```
101 INDEXOF:  MY-ARRAY .S
```

Notice that it returned a 3 and a -1. The 3 is the index of the value 101 and the -1 is a TRUE value. If it can't find the value it just returns FALSE.

```
0SP   1969  INDEXOF: MY-ARRAY .S
```

Range Checking

A common error that can occur when programming is to use an array index that is too large for the array. Our array has 10 items in it. The indices run from 0 to 9. Let's try to read past the end of our array.

```
10 AT: MY-ARRAY
123 900 TO: MY-ARRAY
```

ODE can check for indices that are out of range and will abort if it detects one. Notice that it printed several things here. The index out of range is printed first. Then ODE dumps the object stack. (More about this later.) The current object is the bottom one listed. Then it prints the nature of the error. This is especially handy when debugging. Once you have an application thoroughly debugged, you can turn off this range checking. One way to do this is to enter:

```
FALSE DO.RANGE: MY-ARRAY
10 AT: MY-ARRAY  . \ you get a number but it is garbage
```

Warning, do **not** try to see if TO: will let you over-index. AT: is safe but over-indexing TO: can overwrite memory and cause you to crash.

Another way to turn off range checking is to enter RUN.FASTER and recompile your program. This will affect all objects compiled.

If you forget how many items you have, you can use LIMIT:. Enter:

```
LIMIT: MY-ARRAY .
```

Freeing Memory in Array Classes

Many classes use the ?NEW: method to allocate memory for their use, as we have seen. When one is finished using an object, one must DEallocate the memory. This is done with the FREE: method.

```
FREE: MY-ARRAY
```

I recommend writing a word that frees all the objects that need to be freed. You can then use IF.FORGOTTEN to make sure this word is called automatically if you forget the code that defines them. Once you FORGET an object, it is too late to FREE: it..

```
: CLEANUP
   FREE: MY-ARRAY
;
IF.FORGOTTEN CLEANUP
```

}STUFF: and FILL:

Let's now try to stuff some specific numbers into our array. One easy way to do this is using }STUFF:. Enter:

```
STUFF{  23  987  44  2001 }STUFF: MY-ARRAY
PRINT: MY-ARRAY
```

Notice that }STUFF: automatically allocates memory if needed. If you want you can call ?NEW: before }STUFF: to guarantee allocation.

To fill an entire array at once, enter:

```
345 FILL: MY-ARRAY
PRINT: MY-ARRAY
```

Now that we are done, enter:

```
FREE: MY-ARRAY
```

Tutorial 2 - Early versus Late Binding

To Whom It May Concern,

In the previous tutorial, we sent messages to a specific object, MY-ARRAY. This would be like sending a message to a specific person. We would put that person's name on the message, "Dear Larry, blah blah blah". Sometimes, however, we don't have any specific person in mind to receive the message. If a shopkeeper goes to lunch, he or she would put a "Gone to Lunch" message on their door. The message would then be received by anyone who happened to stand in front of that door. Similarly, we can send a message to *whatever object happens to have its address on the top of the stack*. When we write the message we may not know what object that is so we cannot use its name in the message.

The word `[]` (a left square bracket followed by a right square bracket pronounced "bracket bracket") can be used to send a message to an object on the stack. ODE programmers also pronounce this as "late-bind." The following two lines are essentially equivalent.

```
PRINT: MY-ARRAY
MY-ARRAY PRINT: []  ( MY-ARRAY leaves its address on stack)
```

The first technique is called *early binding* and the second is called *late binding*. When a word is compiled with early binding, the CFA (Code Field Address) of the method to use is determined at compile time and compiled into the dictionary. For late binding, this process doesn't happen until run time. Late binding is therefore slightly slower than early binding but is often required for its added functionality.

This may sound more complicated than it really is in actual use. Late-binding is simply a technique for sending a message to an object addressed “to whom it may concern.” The main difference is that in late-binding, the object’s address comes *before* the method, and the method is followed by the late-bind brackets.

Imagine that you wanted to define a word that would print then clear the contents of many different objects. This could easily be done with late binding. (You might want to put this next example in a file because we will change it later.)

```
: PRINT&CLEAR ( object-address -- , print then clear an object )
   DUP   ( duplicates the object address )
   PRINT: [] ( use late-binding )
   CLEAR: []
;
10 ?NEW: MY-ARRAY .
MY-INT PRINT&CLEAR ( pass the address of MY-INT )
MY-ARRAY PRINT&CLEAR
```

Local Variables and Late Binding

A useful technique with late-binding is to use local variables to hold the address of the object. (Please familiarize yourself with local variables before continuing. JForth users will find them in chapter 11. Macintosh users will find them described in the Macintosh supplement.) Locals can help eliminate confusing stack manipulations. As an example, you could slightly modify the above word to store the address of the object in a local variable:

```
: PRINT&CLEAR { obj -- , print then clear an object }
   OBJ PRINT: []
   OBJ CLEAR: []
;
```

Since this technique is used so often with local variables, ODE supports an alternative syntax. Local variables that contain an object address can be used as if they were an object. *The message is still late bound*, but it is easier to read. Here is another way to write the above word.

```
: PRINT&CLEAR { obj -- , print then clear an object }
   PRINT: OBJ
   CLEAR: OBJ
;
```

This example is rather trivial. The power of using these local variables will be more apparent when used in more complex words.

Now that we are done, don't forget to enter:

```
FREE: MY-ARRAY
```

Tutorial 3 - Using OB.ELMNTS

A very useful subclass of OB.ARRAY is OB.ELMNTS. Each *element* of this array can have multiple values. An example would be an array of X,Y points, or elements. Each element would have 2 *dimensions*, X and Y. In a 3 dimensional array, each element would have an X, a Y and a Z value. The space need not be geometric. Another 3 dimensional space could have the dimensions Time, Pitch, and Loudness. The elements in this space could be musical notes. Let's look at an example of an array of X,Y points. Let's make room for 100 points with 2 dimensions. Enter:

```
OB.ELMNTS  XYPS  \ x,y points
```

```
   100 2 ?NEW: XYPS .  \ if zero then not enough memory!
   PRINT: XYPS
```

When we printed the array, we saw that there were no points in the array yet. Let's add some. Enter:

```
   10 52 ADD: XYPS
   30 17 ADD: XYPS
   15 294 ADD: XYPS
   PRINT: XYPS
```

ADD: is based on the notion that elements have *no* value until a value has been given them. In reality, of course, every byte in a computer has a value from the moment it is turned on. In this model, however, even though data memory has been allocated, the object is considered to be initially empty. You can add elements one at a time by using ADD: and have it keep track of how many you have added. You can find out how many elements have been added by using MANY: .For example:

```
   MANY: XYPS . ( Prints '3' )
```

We can access the elements in the array using GET: and PUT:. These take an index and operate on the whole element. Let's change the value of the second point. Remember that elements are numbered starting with zero so the second point is number 1. The first point is number 0.

```
   50 99 1 PUT: XYPS
   PRINT: XYPS
   1 GET: XYPS .S
   CR SWAP . .
```

If you want to access an individual number, you can use ED.TO: and ED.AT:. The "ED" stands for **e**lement and **d**imension to help you remember how to pass the indices. To change the value in element 2, dimension 1, enter:

```
   75 2 1 ED.TO: XYPS
   PRINT: XYPS
   2 1 ED.AT: XYPS .
```

We can ADD: more elements to the end or we can insert elements anywhere in the middle or at the beginning. Enter:

```
   63 444 1 INSERT: XYPS \ before element 1
   PRINT: XYPS
```

We can remove an element as well. Enter:

```
   0 REMOVE: XYPS
   PRINT: XYPS
```

Another way to access the elements is sequentially. We can get the first element using: FIRST: then continue using NEXT:. Enter:

```
   FIRST: XYPS SWAP . .
   NEXT: XYPS SWAP . .
   MANYLEFT: XYPS .  \ just one left to process
   NEXT: XYPS SWAP . . \ that's the last one
```

If we want to keep going forever we could use NEXTWRAP:. It will wrap around back to the first element when it reaches the end.

The *read pointer* can be set to a specific point or reset back to the beginning.

```
   1 GOTO: XYPS  \ set to read #1
   NEXT: XYPS SWAP . .
   WHERE: XYPS .  \ where are we now?
   RESET: XYPS
   NEXT: XYPS SWAP . .
```

If you don't want to ADD: points, you can use SET.MANY: to make it seem as if there are many points. Enter:

```
   50 SET.MANY: XYPS
```

```
PRINT: XYPS
77 0 FILL.DIM: XYPS
PRINT: XYPS
```

When you want to get rid of those points, use EMPTY: which sets many to zero. (Don' t forget that you can save typing by using <UP-ARROW> to reenter PRINT: XYPS)

```
EMPTY: XYPS
PRINT: XYPS
50 SET.MANY: XYPS
PRINT: XYPS
```

If you want to actually clear the data use CLEAR: as follows:

```
CLEAR: XYPS
PRINT: XYPS
50 SET.MANY: XYPS
PRINT: XYPS
```

When you are done using the array, PLEASE free the memory. Enter:

```
FREE: XYPS
```

Predefined Classes

A number of predefined classes already exist in ODE. They can be used directly, or as the superclasses for newly defined classes.

OBJECT

The class OBJECT is the root class for all other classes. You will probably never use it directly, however, because all of the methods described here will work for all other classes. This is because all classes *inherit* this class' methods.

`ADDRESS: ( -- ivars-address , address of instance variables )`

`.CLASS: ( -- , print the class of an object )`

`DUMP: ( -- , dump object's instance variables in hex )`

`GET.NAME: ( -- $name , get printable name. )`

This name can be used for printing graphically with GR.TEXT or for writing to a file.

`NAME: ( -- , print the name of the object)`

This is usually used with late binding where the name is not known. NAME: SELF is often handy in error messages.

`PUT.NAME: ( $name -- , change the name of an object. )`

The default name is the dictionary name. This name is only used for error reporting and printing. *If you change the name, you must still send messages using the name in the dictionary.* This is very important to remember, especially when using dynamically instantiated objects (see the discussion later in this chapter).

`SPACE: ( -- nbytes , return size of instance variable space )`

This does not include specially allocated memory for arrays, etc. It does include the space required for the memory pointers, limits, etc. This method is not used very often.

OB.INT - subclass of OBJECT

This provides a simple integer object. Its superclass is OBJECT .

`CLEAR: ( -- , clears instance variables)`

`GET: ( -- value, returns value of integer)`

`PRINT: ( -- , print value )`

`PUT: ( value -- , sets value of integer)`

`+: ( n -- , add n to value of integer)`

OB.BARRAY - subclass of OBJECT

This is the basic array class. Its methods will also work for the other array classes. The data for the array is stored in memory allocated for it by the NEW: method. See "Using Arrays" above. The "elements" of the array are referred to as *items* to avoid confusion with the term *element* which has special meaning for OB.ELMNTS. The numbering of the items starts at zero. Thus an array with ten (10) items will have items numbered from zero (0) to nine (9).

`+TO: ( value index -- , add value to the indexed item )`

`?NEW: ( #items -- addr | 0, allocate memory for the array )`

This will automatically free any memory which has already been allocated, then allocate a new memory area. If memory cannot be allocated, a zero will be returned. If you run out of memory, quit from other programs or buy some more. See FREE:.

`AT: ( index -- value )`

Return the value of an indexed item.

`CLEAR: ( -- , sets every item to 0 )`

This executes a FILL: with zero.

`DATA.ADDR: ( -- data-address )`

Get address of allocated memory. This might be used on the Amiga version of HMSL, when using, for example, OB.BARRAY as an audio waveform.

`DO.RANGE: ( flag -- , enable or disable range checking )`

`EXTEND: ( #items -- )`

Extend the memory allocated. This allocates a new area of memory and copies the old data to that new area. The old area is then deallocated. This is useful if you run out of items in an array. It is slow, however, so don't use it too often.

`FILL: ( value -- )`

Set every item in the entire array to value.

`FREE: ( -- , free the memory allocated by the NEW: method )`

If you are finished using an array, use this method to deallocate its memory. If you do not deallocate memory for arrays in this way, the computer's memory will slowly get used up. It’s good programming practice to FREE: all your objects when you’re finished with them (like for example, when a piece is over).

`INDEXOF: ( value -- index true | false , search array for value )`

Returns the index and TRUE if found, otherwise just returns FALSE.

`LIMIT: ( -- #items , return the number of items allocated )`

This is used for setting DO … LOOP indices, checking for out-of-range conditions, etc.

`NEW: ( #items -- , allocate memory for the array )`

This calls ?NEW: and aborts if it returns zero.

RANGE: (index -- , range check index)

This will abort with an error message if the index exceeds the limit of the memory allocated. This is performed automatically but can be disabled using DO.RANGE: .

SET.WIDTH: (#bytes -- , Set width in bytes for array item)

This allows you to have 1-, 2-, or 4-byte wide items in the array. The value must be set before any calls to NEW: , so that the right amount of memory can be allocated.

}STUFF: (v0 v1 v2 ... vN -- , puts values in array)

Use with STUFF{ to load an array. If needed, NEW: will be called to make room for the values.

```
STUFF{  12  34  987  6  }STUFF: MY-ARRAY
```

TO: (value index -- , set the indexed item to the value)

USE.DICT: (flag --)

If flag is TRUE, then ?NEW: will allot space in the dictionary when called instead of allocating dynamic memory. This allows you to initialize an array with data at compile time and save it using SAVE-FORTH. Only store numeric values in such an array, not addresses since they will not be valid at a later time.

WIDTH: (-- #bytes , width of a single array item)

Example of Using Arrays

Put this part in a file.

```
OB.BARRAY BAR1
: RAMPUP ( -- , fill with increasing even values )
   32 NEW: BAR1 ( allocate memory )
   LIMIT: BAR1 0
   DO
      I 2* ( even number )
      I TO: BAR1 ( store 2*I at Ith item )
   LOOP
;
```

INCLUDE the file then test it by entering:

```
BAR1 RAMPUP ( put increasing values in array )
PRINT: BAR1
5 AT: BAR1 . ( will print 10 )
99 AT: BAR1 ( will report an "Index Out of Range" error )
FREE: BAR1
```

OB.ARRAY

This class is the same as the OB.BARRAY class except that each item is as wide as a standard stack item, which in JForth and HForth is *4 bytes*. Historically, this sensitivity to stack-width is important because HMSL has run on a wide variety of platforms and different versions of Forth (including 16- bit versions). The EXEC: method has been added to this class but the indexing is the same.

EXEC: (index -- , executes CFA stored at indexed item)

This assumes that you have stored some CFAs in the array to begin with. You may find it useful to fill such arrays with the CFA of an error handling routine. Then put in the specific CFAs you need. The following example shows how to get a properly handled error if you don't execute one of your specific routines:

Example of Execution Array

Put this part in a file.

```
: BAD.INDEX ( -- , report error )
   ." Invalid execution index." CR ABORT
;
: HI  . " Hello" CR ;
OB.ARRAY EVENTS
: INIT.EVENTS
   16 NEW: EVENTS ( allocate space for 16 CFAs )
   'C BAD.INDEX   FILL: EVENTS ( make safe )
   'C HI 3 TO: EVENTS
;
```

INCLUDE the file then test it by entering:

```
INIT.EVENTS
3 EXEC: EVENTS ( executes HI )
5 EXEC: EVENTS ( reports error )
FREE: EVENTS  ( frees memory now that we're done )
```

OB.ELMNTS

This class combines a two-dimensional array with the additional features of an ordered set of data. It has OB.ARRAY as a superclass. All of the methods that OB.ARRAY has, therefore, also apply to OB.ELMNTS. The rows of this array are called *elements.* The columns are called *dimensions.* This data structure can, therefore, be thought of as an ordered set of n-tuples. An example of this would be using OB.ELMNTS to represent X,Y,Z values. Each element would have 3 values, one for each physical dimension, X,Y and Z. Another example would be using an OB.ELMNTS to represent a melody. Then each point might have a Time and a Pitch value. The notes in this melody could be thought of as points in a 2 dimensional time/pitch space.

`?NEW: ( #elements #dimensions -- addr | 0 )`

Allocates memory for the data. The default width of each value is 4 bytes. If memory cannot be allocated a zero is returned.

`ADD: ( V1 V2 V3 … VN -- , adds an element to the end )`

This adds one row, or element, to the end. The first ADD: goes into element number 0. The next ADD: goes into element number 1 and so on. See the tutorial for an example. It is *veryimportant* to have the right number of values on the stack when using ADD: . If you do not have the same number of values as the object is dimensioned, then you will have stuff left over on the stack or get stack underflows.

`BACKWARD: ( -- )`

Advances the cursor used by NEXT: backward one position.

`CHOP: ( index count -- )`

Remove count elements starting at index.

`CURRENT: ( -- V1 V2 V3 … VN , element at current position )`

`DIMENSION: ( -- #dimensions )`

Return number of dimensions declared.

`DO:   ( function_cfa -- , pass each element to the function )`

This is useful when you want to do something to each of the elements. The function must “eat” as many values as there are dimensions. For example, let's calculate the sum of the products of a 2 dimensional elements array.

```
10 2 NEW: ELM1
2 3 ADD: ELM1
```

```
4 5 ADD: ELM1
VARIABLE SUM-PRODS
:  *+EACH   (  a b -- , add product of a and b to SUM-PRODS )
    *  ( multiply the two values )
   SUM-PRODS +!   ( add result to variable )
;
0 SUM-PRODS !
0 GET: ELM1 *+EACH   ( -- , test function outside DO: )
SUM-PRODS @ .  ( should be 6 )
0 SUM-PRODS !
'C *+EACH  DO: ELM1    ( pass each element to function )
SUM-PRODS @ .  ( should be 26 )
FREE: ELM1
```

DUMP.SOURCE: (--)

Prints ODE source code that could be used to regenerate the state of this object. Combining this with LOGTO allows you to save the contents of an OB.ELMNTS array as source code. This code can be reloaded using INCLUDE. Alternatively, you could use FWRITE and DATA.ADDR: to write a binary file containing the contents of an object. Be sure to put a header on the file that says how much to NEW: the object and how many data items and dimensions to use.

ED.AT: (element dimension -- value , return value at e,d)

Fetches value based on its row and column address.

ED.TO: (value element dimension -- , store value at e,d)

These two methods (ED.TO: and ED.AT:) are critical because all of the other access methods are written using them. If you need to define some special access methods for a new class based on OB.ELMNTS then you will probably need to use the methods ED.AT: and ED.TO: .

ED2I: (element dimension -- index , convert to linear index)

OB.ELMNTS are actually implemented using one dimensional arrays. This method returns the one dimensional array index for an element dimension pair.

EMPTY: (-- , set number of elements to zero)

This is faster then CLEAR: because CLEAR: actually sets the allocated memory to zero. This only sets the element counter to zero.

FILL.DIM: (value dim# -- , fill one dimension with a value)

FIRST: (-- V1 V2 V3 … VN , return first element, set pointer)

FOREWARD: (--)

Advances the cursor used by NEXT: foreward one position.

GET: (element -- V1 V2 V3 … VN , fetch a given element)

Using the example for the ADD: method:

```
1 GET: ELM1 .S ( will produce )
   777 888 999
```

GOTO: (element# --)

Sets the cursor used by NEXT: .

`I2ED:  ( index -- element dimension , opposite of ED2I: )`

`I2ADDR:  ( index -- address , determine address of an item )`

Calculates actual address of an item in the array. Use with ED2I: .

`INSERT: ( V1 V2 V3 … VN element -- , inserts new element before )`

The pointers are adjusted as in REMOVE: to maintain current position.

`LAST: ( -- V1 V2 V3 … VN , return last element added )`

Aborts if no data.

`MANY: ( -- N , return number of elements added )`

Sending a MANY: message to ELM1 (from the example for the ADD: method) returns a value of 2.

`MANYLEFT: ( -- N , number of elements left after current )`

This is handy for checking whether an array has been exhausted using NEXT: .

`MAX.ELEMENTS:  ( -- max , maximum number of elements allocated )`

This will be the same as the first value passed to NEW: .

`NEW: ( #elements #dimensions -- )`

Calls ?NEW: and aborts if zero is returned. ?NEW: is preferred for use in applications.

`NEXT: ( -- V1 V2 V3 … VN , element at next position )`

This also increments the internal pointer.   This is useful for sequential processing. An error will occur if you attempt to go past the end of the array. Use MANYLEFT: to see how may are left.

`NEXTWRAP: ( -- V1 V2 V3 … VN , element at next position )`

This also increments the internal pointer.   This is useful for sequential processing. When the end of the array is reached, NEXTWRAP: will automatically wrap around back to the first element. Use WHERE: to find out where you are if need be.

`PRINT: ( -- , prints out elements in a table )`

`PRINT.DIM: ( dim# -- , print the values of a single dimension )`

`PUT: ( V1 V2 V3 … VN element -- , stores a complete element )`

This stores a row without incrementing any internal pointers. It is usually used for editing.

`REMOVE: ( element -- , removes a given element )`

When pointing to an element above the one that was removed, the internal pointers will be decremented so they still point to the same data. This moves all the higher elements down to fill in the gap.

`RESET: ( -- , Resets the current element pointer to 0 )`

This often precedes successive calls to NEXT: .

`SET.MANY: ( N -- )`

Set the number of elements currently in object. If you want to make this object act as if you have ADDed a bunch of values, you can set the number of elements it thinks it has. You can use SET.MANY: and then use GET: and PUT: to access any element within the range you set. For example, instead of adding elements:

```
OB.ELMNTS ELM-1
20 3 NEW: ELM-1
15 SET.MANY: ELM-1
PRINT: ELM-1    ( notice 15 elements )
```

```
        FREE: ELM-1
```

Note that the value for SET.MANY: cannot be greater than that for which the object was NEWed.

SIZE: (-- N , return number of individual values added)

This is the product of MANY: and DIMENSION: .

SMEAR: (start count -- , smear element up)

Overwrites `count` elements with element at `start`.

SPLIT: (start count -- , move elements up and create a split)

This is used internally by INSERT: but can be used externally. Copies elements above and including `start` up by `count`. This is more difficult to explain than it is to see: try it.

STRETCH: (index count -- , dup element at index , count times)

This will "internally" duplicate a specific element in an object, `count` number of times.

WHERE: (-- element#)

Returns the cursor used by NEXT: .

OB.LIST

This is a one dimensional version of OB.ELMNTS. It is handy for keeping a list of things. It has one new method and one altered method.

NEW: (#elements -- , allocate memory for one dimension)

The number of dimensions is automatically one.

DELETE: (value -- , removes value from list)

Looks for the value in the list and removes it if found. For example:

```
        OB.LIST MY-LIST
        10 NEW:  MY-LIST    ( make room )
        1111 ADD: MY-LIST
        234 ADD: MY-LIST
        1988 ADD: MY-LIST
        PRINT: MY-LIST   ( see all values )
        234 DELETE: MY-LIST
        PRINT: MY-LIST   ( 234 now gone )
        FREE: MY-LIST
```

OB.OBJLIST

This class is used for storing the *addresses of other objects*. Its superclass is OB.LIST so all of those methods are inherited. OBJLIST's are extremely important in ODE, where you typically have lists of various objects that you want to be able to index into, rearrange, perform operations on, and so on.

?INSTANTIATE: (class_cfa #objects -- class_pfa | 0)

Dynamically instantiate the objects and store their addresses in the objlist. This will call ?NEW: first to make room for the object addresses. Returns zero if it could not allocate all of them. For more information on dynamic instantiation, see the advanced topic later in this chapter.

DEINSTANTIATE: (--)

Deinstantiates all objects created using ?INSTANTIATE. Then FREE: the objlist.

FREEALL: (--)

Send a FREE: message to all objects listed within. This can be very useful in FREE:ing all objects in a list, when you might not necessarily know exactly what those objects are! Note that FREEALL: does not actually FREE: the object list itself (just use FREE: for that).

Dynamic Instantiation using OB.OBJLIST

Here is an example that uses OB.OBJLIST to dynamically instantiate 25 arrays from memory. Put this code in a file so you can experiment with it.

```
OB.OBJLIST MUCHO-ARRAYS \ declare array to hold arrays!

:  MAKE.MUCHO-ARRAYS  ( -- error? )
\ make room for 25 arrays in the object list
   'C OB.ARRAY 25 ?INSTANTIATE: MUCHO-ARRAYS
   IF  \ yes we made them
      25 0
      DO
         I GET: MUCHO-ARRAYS  \ get Nth array
         10 OVER  ( -- array 10 array )
         NEW: []
      ( -- array , make room for 10 in each array )
         I SWAP   ( -- I array )
         FILL: [] ( -- , fill dynamic array with I )
      LOOP
      FALSE \ no error
   ELSE
      TRUE \ couldn't instantiate!!
   THEN
;

: CLEANUP.MUCHO-ARRAYS ( -- )
\ free memory in all arrays in object list
   FREEALL:  MUCHO-ARRAYS
   DEINSTANTIATE: MUCHO-ARRAYS
;
\ Automatically cleanup if file is forgotten.
IF.FORGOTTEN CLEANUP.MUCHO-ARRAYS
```

Now include the file, and enter:

```
MAKE.MUCHO-ARRAYS .
```

You will now have an object list, 25 long, which contains the addresses of the twenty-five arrays. If the instantiation failed, MAKE.MUCHO-ARRAYS would return a TRUE as an error flag.

Since these arrays have no names, only addresses in MUCHO-ARRAYS, you can only reference them by indexing into the list. This will generally involve the technique of late binding, since you will have to pass the address of the object to the method on the stack.

For example, if you wanted to PRINT: one of the dynamically instantiated arrays, here's how you would do it:

```
5 AT: MUCHO-ARRAYS  ( -- address-of-array#5 )
PRINT: []
```

When you are done using these arrays, you must free them and then deinstantiate them. Enter:

```
CLEANUP.MUCHO-ARRAYS
```

Defining New Classes and Methods

New classes can be defined if the existing ones do not provide the functionality that you need. You should first choose the closest available class to use as the *superclass*. Then decide what internal *instance variables* each object will need. Any methods that have not already been *declared* will need to be so that they will have a method index. Do this by entering:

```
    METHOD new_method:
```

before your class definition (where new_method: is the name of your new method). If you are just giving new functionality to an old method you will not need to declare a new name. For example, if you modify the PRINT: method, you won't have to *declare* PRINT: as a method since it already been declared for other classes.

It is generally good programming practice to use the same method name for functions that are more or less the same. For example PRINT: for OB.INT does different things than PRINT: for OB.ARRAY, but the user only has to remember a single method name.

If you *instantiate an object inside a class definition*, then it will become an *instance object* that will exist inside every instance of the new class. See the later example of Instance Objects.

Class Definition Glossary

The following words can be used in the definition of new classes. It is easiest to learn them from the examples.

:CLASS (<word> -- , declare a brand new class)

:M (<word> -- , start the definition of a method, like :)

;CLASS (-- , terminates a class' definition)

;M (-- , terminates a methods definition)

<SUPER (<word> -- , declare the superclass of a class)

The new class will *inherit* all of the *instance variables* and *methods* of the superclass. This word is required for every class definition. If there isn't any special existing class that you would like to inherit from then just use OBJECT .

IV.LONG (<name> --)

Creates a named 32 bit instance variable.

IV.SHORT (<name> --)

Creates a named 16 bit instance variable.

IV.BYTE (<name> --)

Creates a named 8 bit instance variable.

IV.BYTES (count <name> --)

Make room for "count" bytes in object. The other words, (IV.LONG, IV.SHORT, IV.BYTE, etc.) were created using BYTES. All instance variables must be declared *before* any method definitions.

METHOD (<word> -- , declare word to be a new method)

It is recommended that the method declared end in a colon to be consistent with the syntax of other object oriented languages. Methods only need to be declared once and can then be redefined for several different classes.

It is necessary to declare a method before the class is defined because the number of methods declared determines how large to make the classes method table. Each class has a table of CFAs, one for each declared method. Each method is assigned an index in that table. This technique allows for extremely fast binding between messages and method code at the expense of some memory. ODE requires more memory for class definitions than other similar systems, but executes late binding much faster. This is a big advantage in real time applications like games or music. Plus memory keeps getting cheaper so why not use it.

Instance Variables

The following words are used inside method definitions to access the instance variables of an object. When these words are executed they calculate the address of the instance variable data, then fetch that data. The fetch will automatically be of the proper width, for example, an IV.SHORT will use W@. In this way, different objects can have their own private copies of the data. To store data into these instance variables you must use one of the prefix operators described below. [Technical note: The address of the current object is kept on a special stack called the *object stack.* When a method starts to execute, it can be assumed that that object's address is at the top of the object stack. The instance variables are defined in terms of their offsets from the address of that object].

`IV=> ( value <instance_var> --, Store value in ivar )`

This word will look up the width of the instance variable and use the appropriate store function, C!, W! or ! . To fetch the value of an instance variable just give its name. As an example, if you want to set an instance named IV-DEPTH to 200:

```
200  IV=>  IV-DEPTH  ( only valid inside a method )
." Default depth = " IV-DEPTH  .  CR
```

`IV+> ( value <instance_var> -- , Add value to ivar )`

Note: This only works on long variables, ie. those declared with IV.LONG.

`IV&> ( <instance_var> -- addr , Address of ivar )`

This is useful if you need an indexed instance variable or if you need to pass an instance variable's address. Generally, if you are doing a lot of indexing of instance variables, it is probably better to use *instance objects*.

Using SELF in Method Definitions

Sometimes, to define a new method, you will need to use other methods already defined for that class. This creates a problem: what object do you pass the message to inside the method definition? The usual technique of METHOD: OBJECT can't work because the objects themselves haven't been defined yet, so there is no address of any object to use for the method. Smalltalk and ODE solve this problem with two special words, SELF and SUPER . SELF allows you to refer to whatever object is currently being defined inside a method definition for that object. For example, when you see, CLEAR: SELF inside a method definition you know that the object currently being called will be cleared. SELF and SUPER can only be used inside a method definition.

Warning: methods referenced using method_name: self must already be defined. Otherwise any previous definition of that method for that class will be used. This is consistent with the Forth convention of not allowing forward referencing.

For example, let's define a method called DIM.SUM: for a class which is a subclass of OB.ELMNTS. DIM.SUM: will sum the values for a given dimension. (Note that we use local variables in the method. The '{' denotes local variables instead of a normal stack diagram. See the JForth or HForth manual for more information on local variables.) Enter this in a file using a text editor.

```
\ Declare method to be defined.
METHOD DIM.SUM:

\ Declare new class as a subclass of OB.ELMNTS
:CLASS  OB.NEW.ELMNTS  <SUPER OB.ELMNTS

:M DIM.SUM: {  dim | cursum -- sum-of-all-values }
```

```
      0 -> CURSUM
      MANY: SELF 0
      DO
         I DIM ED.AT: SELF
         +-> CURSUM
      LOOP
      CURSUM
   ;M
   ;CLASS
   \ Now let's instantiate one and test it.
   OB.NEW.ELMNTS  NELM-1
   : TEST.NELM
      10 2 NEW: NELM-1
      5 20 ADD: NELM-1
      5 7 ADD: NELM-1
      ." Sum of dimension 1 = " 1 DIM.SUM: NELM-1 . CR
      FREE: NELM-1
   ;
```

To test it, INCLUDE the file and enter:

```
TEST.NELM
```

Using SUPER and SUPER-DOOPER in Method Definitions

SELF is used more often than SUPER. SUPER is only needed when the new class redefines a method but you still want to access the old one. For example, if you want a method that prints an object as previously defined but also prints out some dashes, and an item-count:

```
:M PRINT: ( -- , extended print )
   PRINT: SUPER
   CR ." --------------" CR
   MANY: SELF . ." items" CR
;M
```

In this example, using PRINT: SELF would have resulted in a fatal recursion — you can't use a method that is currently in the midst of being defined!.

Occasionally you won't like the way your superclass performs a method, but you like the way that superclass's superclass does. In this situation you can use SUPER-DOOPER instead of SUPER.

```
:CLASS OB.STRANGE <SUPER OB.ELMNTS
:M PRINT:   ( -- )
   ." Double Print!!!" CR
   PRINT: SUPER-DOOPER ( print like an ob.array )
   ." -----------------------" CR
   PRINT: SUPER ( print like an ob.elmnts )
;M
;CLASS
```

SUPER-DOOPER is not really standard object oriented programming, it's something we added to ODE when we needed it in the implementation of HMSL, and it has proved useful ever since.

Special Methods: INIT:

The method INIT: is automatically executed when an object is *instantiated*. If you want to set *default values for the instance variables*, or perform any *special initialization*, just define an INIT: method. You will probably want to include an INIT: SUPER to invoke any initialization that the superclass was doing. Generally it's best to avoid doing things inside INIT: that affect the state of the operating system. This includes allocating memory (NEW:), opening files, etc. Otherwise you will not be able to save the INITed object in a precompiled form, for its pointers to memory and files will be invalid the next time it is run.

Example Class Definition

To clarify this, let's define a class of object that is a city. We want to keep track of population and area. Let's use the OB.INT class as the superclass since it already does some of what we want the new class to do. We can use the existing instance variable in OB.INT for the population. We will need to add a new instance variable for the area.

```
METHOD PUT.AREA: ( declare new methods )
METHOD GET.AREA:
METHOD CALC.DENSITY:

:CLASS OB.CITY <SUPER OB.INT  ( inherit properties of OB.INT )
   IV.SHORT IV-AREA ( declare 2 bytes for the area )

:M INIT: ( -- , Set defaults )
   INIT: SUPER ( perform initialization of superclass,
important )
   1 IV=> IV-AREA ( avoid 0 divide in CALC.DENSITY: )
;M

:M PUT.AREA: ( area -- , set area of city )
   IV=> IV-AREA ( store using prefix operator )
;M

:M GET.AREA: ( -- area , return city's area )
   IV-AREA ( leaves value automatically )
;M

:M CALC.DENSITY: ( -- density, return people per acre )
   GET: SELF GET.AREA: SELF /
;M

:M PRINT: ( -- , redefine print to perform new function )
   ." The city of " NAME: SELF CR
   ." has a population of " GET: SELF . CR
   ." and an area of " GET.AREA: SELF . CR
   ." The density is " CALC.DENSITY: SELF . CR
;M
;CLASS ( finish class definition )

OB.CITY ARMPIT ( declare two instances )
OB.CITY BIGHOLE

10034  PUT: ARMPIT ( set populations )
795  PUT: BIGHOLE

45  PUT.AREA: ARMPIT ( set areas )
29  PUT.AREA: BIGHOLE

PRINT: ARMPIT ( print city reports as defined )
PRINT: BIGHOLE
```

Example of Creating a Class with Instance Objects

Let's create a subclass of OB.ELMNTS that keeps information about each of its dimensions. It will have an internal array with one item per dimension. It will be sent a NEW: and FREE: message as part of the class's NEW: and FREE: method.

```
( declare methods for accessing Instance Object)
METHOD PUT.DIM.DATA:
METHOD GET.DIM.DATA:
:CLASS FOO  <SUPER OB.ELMNTS
    IV.LONG IV-FOO-FLAVOR   ( normal instance variable )
    OB.ARRAY IV-FOO-ARRAY  ( Declare Instance Object )

:M NEW: ( #elmnts #dimensions -- )
    TUCK NEW: SUPER    ( calls FREE: )
    NEW: IV-FOO-ARRAY  ( so NEW: this after SUPER )
( If you reverse the order above, IV-FOO-ARRAY will get FREE:d )
( after NEW: SUPER because NEW: SUPER calls  SELF FREE: [] )
;M

:M FREE: ( -- , called by NEW: )
    FREE: SUPER
    FREE: IV-FOO-ARRAY
;M

:M PUT.DIM.DATA: ( data dim -- )
   \ set associated data for dimension
    TO: IV-FOO-ARRAY
;M
:M GET.DIM.DATA: ( dim -- data , get associated dim data)
    AT: IV-FOO-ARRAY
;M

:M PRINT:  ( -- print values and dim data )
   PRINT: SUPER
   ." Dimension values ---" CR
   PRINT: IV-FOO-ARRAY
;M
;CLASS

\ Instantiate and use a FOO
FOO MY-FOO
20 3 NEW: MY-FOO
11 22 33 ADD: MY-FOO
234 345 456 ADD: MY-FOO
77 1 PUT.DIM.DATA: MY-FOO    ( set dimension value )
PRINT: MY-FOO
FREE: MY-FOO
```

Remember, instance objects, like instance variables, can *only* be referenced from inside a method or a Forth word called from a method for that object!

Advanced Topics

ODE Functions

`CURRENT.OBJECT     ( -- object )`

Object currently being processed. This word is handy if a method calls a function and the function needs to know which object called it. The function can then send late bound messages to the object. A lot of HMSL objects, like jobs and interpreters automatically pass their own addresses to many of their methods, so CURRENT.OBJECT is only necessary in unusual cases.

`RUN.FASTER  ( -- )`

Turn off some error checking for speed. Turns off late bound class checking which ensures that you are sending messages to a real object and not some random piece of memory. Also turns off range checking for subsequently compiled array based classes. Only call this when you have finished debugging your program. Recompile your aplication after calling this.

`RUN.SAFER  ( -- )`

Turns on error checking turned off by RUN.FASTER   Default mode for ODE.

Getting Information About Classes

There are several words that you can use to find out information about a particular class. These are useful if you want to get the most use from a given class.

`METHODS.OF  ( <class>  --  )`

Show the methods supported by a class. For example, to see what methods are supported by the OB.ELMNTS class, enter:

```
METHODS.OF OB.ELMNTS
```

`INHERITANCE.OF     ( <class>  --  )`

Show the superclasses of a class   To see what classes OB.ELMNTS inherited its methods and instance variables from, enter:

```
INHERITANCE.OF OB.ELMNTS
```

`ALL.METHODS ( -- )`

Show all methods declared in dictionary.

Dynamically Allocated Objects

Sometimes you may want to create, or *instantiate,* an object while a program is executing. Otherwise all objects would have to be defined at compile time.   There are two words used in dynamic instantiation:

`?INSTANTIATE ( <class> -- object-address | 0 )`

This creates an object of the requested class in free memory. The object is added to a list of dynamic objects where it can be found using 'O . It will be given the name DYNn, where nnn goes from 000 to however many you have. You can't reference these by name since there will be no NFA (name field address) to use for the name. You can change the name using PUT.NAME: but this is only used for printing purposes, and cannot be used for messaging. All messages to this object must be sent using late binding, and for this you would typically retrieve the address of the object from some user created object list or array (see the example with OB.OBJLIST).

`DEINSTANIATE ( object-address -- )`

Deallocate dynamically instantiated object.

Here is an example of a dynamically allocated array object. It uses a local variable to store the object address. Binding to a local variable is always late binding. Notice that in this example we check to make that INSTANTIATE and ?NEW: returned address indicating success before proceding.

```
: DYNARRAY { newobj -- , Demo dynamic array. }
   ?INSTANTIATE OB.ARRAY \ Create dynamic object
   DUP -> NEWOBJ  \ save in local
   IF
\ Allocate data space, note use of late-binding
      12 ?NEW: NEWOBJ
      IF
         789 FILL: NEWOBJ
         PRINT: NEWOBJ
         FREE: NEWOBJ  \ Free allocated memory
      THEN
      NEWOBJ DEINSTANTIATE \ Deallocate object.
   THEN
;
Now test this by entering:
DYNARRAY
```

'O (<dynamic-object> -- object-address)

Search the list of dynamically instantiated objects for the object with the matching name.

If you need to use an array inside of another object, you could instantiate one and store its address in an instance variable. However, you can also declare instance objects (described previously in this chapter).

Examining Instance Variables

When debugging object-oriented code, it is useful to be able to examine instance variables even if there is no method for accessing them. To do this you can use the DUMP: method. ODE supplies an alternative that is “technically” illegal in an object-oriented system and allows the user to reference an instance variable using techniques normally used only for *C* structures. In the previous example, if you did not have a method for GET.AREA: , you could enter:

```
ARMPIT REL->USE ..@ IV-AREA . ( Get the area of Armpit )
```

Don’t try this at home, kids.

Error Reporting

There is a special facility for reporting errors inside objects. It dumps the object stack which gives a traceback of which objects were calling which.

OB.REPORT.ERROR ($method-name $message severity --)

The severity is either ER_WARNING , ER_RETURN , or ER_FATAL . For fatal errors, the object stack is printed, cleared, and execution aborted. Here is an example of detecting an out of range error in a method called DOIT:

```
" DOIT:"  " Input out of range"
ER_FATAL  OB.REPORT.ERROR
```

Inheritance

A class definition contains several things used when one class inherits instance variables and methods from another class. The number of bytes of instance variable space is kept. When the subclass has instance variables defined, their offset begins after the superclass's area.

A table of CFAs for the methods of a given class is kept at the end of a class definition. Each declared method has a *method index* that is the same for all classes. When a new class is defined, a table large enough to hold all of the declared methods is alloted in the dictionary. The word <SUPER copies the CFAs from the superclass into the CFA table of the subclass. A message sent to an object will use these CFAs unless a new method has been defined that overwrites that entry in the table. When SUPER is used, the CFA from the superclass's table is compiled, regardless of the value in the current class CFA table.

Memory Placement for Amiga

If you create an array object that will be used by the Amiga coprocessors, then the data must be allocated in chip memory. This can be controlled by setting a variable called MM-TYPE to MEMF_CHIP before calling NEW: .

```
OB.BARRAY  IMAGE-DATA
MM-TYPE @ ( Save old value )
MEMF_CHIP MM-TYPE !
32 NEW: IMAGE-DATA ( Allocate space in CHIP RAM )
MM-TYPE !
```

This is done automatically for HMSL classes that need it, i.e. OB.WAVEFORM, OB.SAMPLE and OB.ENVELOPE.

Cloning ODE Programs using JForth

If you are going to Clone a JForth program that uses ODE, you MUST do the following:

1) Compile REDEFs that are needed by Clone. These are loaded by default if you use the file LOAD_ODE. If not you MUST:

```
INCLUDE JO:CLONE_SUPPORT
```

2) Do all memory allocation at Run Time. This means you **cannot** call NEW: at compile time. You must call it from a word in your program if you need it. If you can save a program using SAVE-FORTH then you are OK as far as this requirement is concerned.

3) Initialize the Object Stack. The pointer uses absolute addresses for speed and must be converted before running ODE. At the beginning of your program, therefore, you must call **OS.SP!** or you will definitely crash.

4) We recommend that you compile Clone before compiling ODE but it is not required.

5) Since name fields are not in a Cloned image, if you are going to use NAME: or PRINT: then you must give the object a name explicitly using PUT.NAME:.

Here is an example of an ODE program that will work with Clone.

```
INCLUDE? TASK-CLONE_SUPPORT  JO:CLONE_SUPPORT
OB.ARRAY  MY-AR1
: GOOD.ODE  ( -- simple clonable program )
    OS.SP!   ( REQUIRED!!!!! )
    10 NEW: MY-AR1   ( only call NEW: at run time )
    " TestArray" PUT.NAME: MY-AR1    ( since NFAs will be gone )
    761 FILL: MY-AR1
    PRINT: MY-AR1
    FREE: MY-AR1
;
( now clone it )
CLONE GOOD.ODE
SAVE-IMAGE GOOD.ODE  RAM:GOOD.ODE
( now in the CLI, enter )
RAM:GOOD.ODE
```

Explanation of ODE Structures Diagram

This is a *very* technical discussion of how ODE is implemented and is probably more detailed than most people need to know.

A class structure contains information about how to create an object and how to implement its methods. Please refer to the accompanying diagram when reading this section. The first field in the class structure is the Size of an Object. When an object is instantiated, this much room is alloted or allocated. The second field is the Number of Methods that this class supports. This determines how large the jump table containing method addresses is. The next field, the Validation Code, is used to distinguish valid classes from random memory and is used for error checking when binding. The next field is the Superclass pointer.

When a new class is defined the following things occur. The new class first inherits the superclass's initial object size. As new instance variables are declared, this size is increased. A jump table is alloted for the class that has enough entries for all of the methods declared using METHOD. The superclass' method pointers are copied from the superclass jump table so that the new class can inherit those methods. When new methods are defined, the entry in the jump table is overwritten. Each declared method has a specific index which determines its offset in any class's jump table.

When an object is instantiated, space is alloted based on the Size of Object field in the class. The first cell of the object is set with a pointer to the method jump table for the class. Then the first method in that jump table, INIT: , is called, which initializes the object.

When a message is sent to an object, the object's address is first pushed onto the "current object stack". Then the appropriate method is looked up in the object's classes jump table and executed. When finished, the object is popped from the object stack. This allows nesting of object method calls.

When a method is declared using METHOD, a structure is created that has an index equal to the current value of MI-NEXT. MI-NEXT is then incremented. This is the index for a specific method in each class's jump table. The methods are linked using the Previous field so that the METHODS.OF word can scan a method list for a class.