Object Oriented Extensions to Forth

Dick Pountain

8 Rousden Street

London NW1, England

Abstract

This paper describes an extension to a standard Forth-79 or -83 system which enables programming with objects organised into classes (with sub-class inheritance). Objects are activated by sending messages to them, following the model of Smalltalk-80.

Object Orientation

Object oriented computer languages are those which allow the programs to be created by combining modules which can only interact in prescribed ways. Such modules contain templates for data structures and definitions of the procedures which operate on them.

Objects are copies made from the template; they are structured variables whose internal detail is hidden from the programmer, and which can only be manipulated by the prescribed procedures. The set of data structures and procedures which the module's implementer permits to be visible is called its *interface* and all other internal detail is inaccessible.

Existing languages which support some degree of object-orientation include Simula-67, ("classes"), Modula-2 ("modules"), Ada ("packages"), CLU ("clusters") [GHEZ82], and Smalltalk-80 ("classes") [GOL83].

These languages exhibit two somewhat different approaches to object orientation. Simula, Modula, CLU and Ada are compiled languages of the Algol/Pascal family. In these languages object orientation can be seen as an extension of structured programming techniques, and in particular of user defined data types. The definition of a type of object is a module containing some variable declarations (which define the physical representation of an object of this type) and some procedure declarations (which define the operations that may be performed on an object of the type).

The interface is usually specified explicitly by means of "export" and "import" lists which state which of the module's variables and procedures are to be visible outside it, and what external ones are to be visible within it. This mechanism is sometimes called Abstract Data Typing rather than object orientation. For example an abstract data type called `STACK` might be defined, containing a stack structure (which might be an array or a linked list) and some procedures to perform all the generic behaviour of a stack (`PUSH`, `POP`, test for empty etc.). The type would be used by creating new instances of STACK, just like declaring variables. The actual implementation of the stack, as array or linked list, is hidden from the user, and could be changed without affecting programs which use a stack.

An important characteristic of this kind of object orientation is that it involves strong type checking, also called early binding. This means that the type of an object must be known when the program is compiled, and the legality or otherwise of an operation on the object is checked at compile time.

Smalltalk-80 exhibits another, more radical style of object orientation. Everything in a Smalltalk program is an object and programs are executed by sending messages to objects. Messages invoke operations called "methods", which correspond to the procedures in an abstract data type declaration. A crucial difference is that Smalltalk is, like Forth, a partially interpreted language, and it does not need to know the type of an object to which a message is to be sent until the program is run. This is called late binding; a message is not bound to its method until runtime. This is less efficient than early binding, in which most of the work is performed at compile time, but it allows for great flexibility. The same program might work on many types of object, which have different but related behaviours.

Benefits of Object Orientation

The benefit of object orientation is felt particularly in the production of large programs, where several programmers have to share the tasks. The independence of modules allows each to be tested individually. When they are combined into a program, there is a guarantee that the modules cannot interact in unexpected ways. This is not so with languages that permit global access to data, where accidental name clashes can cause hard to detect errors by inadvertently modifying data structures.

An additional benefit is improved maintainability. Module independence isolates the rest of the program from detail changes made to a module, provided that its interface is unchanged.

Forth programming already has a "flavour" of object orientation about it. Unlike block structured languages (eg. C, Pascal), its subprogram units, words, are independently executable and testable. Vocabularies also provide a degree of overloading (ie. using the same name for different operations) and information hiding.

However the independence of modules cannot be guaranteed as all Forth words, including data structures, are either global or "masked" by redefinition. Forth provides for modularity of code but not of data.

This project arose from a desire to see whether full object-orientation could be introduced into Forth at an acceptable cost in memory and processing time.

Classes, Inheritance and Message Passing

The most radical model for object-orientation so far developed is found in Smalltalk-80 [GOL83]. This language represents everything that it can handle, right down to numbers and characters, as objects with behaviours which are activated by sending messages to them. The internal structure of an object is not visible, and sending an appropriate message is the only way in which an object can be affected. The set of messages an object understands is therefore its interface.

Objects are organised into *classes*, defined by the messages that they understand. Classes are the modules of Smalltalk programming. Every object is an *instance* of a class; for example the literal `2` is an instance of the class `SmallInteger`. Sub-classes of a class may be defined, and their instances *inherit* all the behaviours of the parent class, in addition to any new ones. These behaviours are known as *methods*. Inheritance encourages a programming style which progresses from the general to the more specific.

The inheritance mechanism fits well with the "nested" Forth approach to programming, while Forth's streamed input is very congenial to a message passing syntax.

The author decided to adopt these concepts of class, inheritance and message passing, but to embed them in a conventional Forth system. There has been no attempt to create Smalltalk style control structures, as the author finds them unattractive compared to those in Forth. Examples of the kind of source code which results can be studied in Listing 2.

The resulting system is perhaps more nearly equivalent to Simula-67 or Apple's "Clascal", in which classes and inheritance are made available inside a conventional language, than it is to full Smalltalk.

Neon

Since implementing this system, the author has become aware of the commercially available Neon system, a Forth/Smalltalk hybrid written for the Apple Macintosh. Neon is well described in [DUF84], and that paper also provides a deeper description of some of the features of Smalltalk style object orientation than has been attempted here.

The principal design difference between Neon and the author's system is that the former uses early binding of messages to methods for better runtime efficiency (late or "deferred" binding may be explicitly requested as an option). The author's system uses late binding exclusively. Another minor difference is that in the author's system, as in Smalltalk, classes are themselves objects, while in Neon they are conventional Forth structures.

However it must be stressed that the system described here is neither as complete nor efficient as Neon. It is more in the nature of experimental work in progress, the emphasis being mainly on small code size and portability, to permit users of Standard Forth systems on small computers to play with the ideas of object oriented programming.

The Design Tasks

The definition of a typical class, called `%point`, can be found in the first screen of Listing 2 (the `%` is a suggested naming convention for class names; it's not required by the syntax). This class is meant to represent points on a cursor addressable VDU screen, and its definition contains two *instance variables* called `X` and `Y` which hold the coordinates of a point. An instance of this class is therefore just a structured variable with two fields called `X` and `Y`. In fact an instance of `%point` also has another field which contains a pointer to its class, so points are six byte objects :-

class ptr	X	Y

The class `%point` also has a number of methods, including `show`, `hide`, `up`, `down` and several more. These are the operations which a point can perform; any point can be told to move itself, show itself, and hide itself. These operations are executed by sending a message, which is just the name of a method, to an instance of `%point`. If `fred` is an instance of `%point`, messages are sent like this :-

```
    10 15 fred \ put    fred \ show
```

The code for a method may contain references to any words in the Forth dictionary, to the instance variables of its class and to any class methods defined prior to it.

In the third screen of Listing 2 there is defined a class called `%square`, whose instance variables are four points, and understand all the messages of `%point`. This class has methods of its own. In the fifth screen of Listing 2 there is defined a sub-class of `%square` called `%Gsquare`. This class inherits all the instance variables from `%square` and also all its methods, to which it adds a new one of its own called `grow` (which causes an instance to be enlarged by one unit in all directions).

The following observations define the design requirements.

a) The internal details of classes `%point`, `%square` and `%Gsquare` are not visible to a Forth dictionary search. For example, only the word `%point` itself appears in the dictionary. Unless the source code is available, `%point` is a "black box".

b) The instance variables X and Y are not directly accessible outside the class definition and Forth considers them to be undefined. They may not be directly addressed in any instance of `%point`; an instance is also a "black box". They may only be referred to in the methods, and their values can only be modified by sending messages to invoke such methods.

c) The code for the methods exists only in one place, in the class definition. However there may be hundreds of instances of a class, each with its own private data.

d) A subclass inherits all the class and instance variables of its superclass, and all of its methods. No declaration of these need be made in the definition of a sub-class, but they can be used in its methods. Any variables or methods defined in a subclass are *in addition* to those it inherited. A new method which has the same name as a superclass method replaces that method.

Hence in order to introduce classes into Forth, the following mechanisms are required :-

a) A means for encapsulating code and data, and concealing it from Forth. Encapsulation implies that nothing inside the "capsule" is visible to an ordinary Forth dictionary search. Encapsulation in turn involves :-
a1) A means of creating encapsulated classes which contain the code.
a2) A means of creating encapsulated instances of a class, which contain the data.
b) A means of passing messages to an object, which causes the code in its class to be executed but to act on the data in the instance.
c) A means of chaining from the code in a sub-class to that of its parent class when an inherited message is sent.
It turns out that almost all these mechanisms are either already present in standard Forth, or are described in well known extensions.

Encapsulation

The means for encapsulating class code was found in the "modules" described by D.V.Schorre [SCHOR80] in *Forth Dimensions*. It works simply by redirecting link field pointers at compile time so that a group of definitions are "skipped" in subsequent run time dictionary searches; however they are available to each other at compile time.

compilation →

$$\begin{tikzcd} {} & \text{word A} \arrow[l] & \text{word B} \arrow[l] & \text{word C} \arrow[l] & \text{word D} \arrow[l] & \text{word E} \arrow[llll, bend left] \end{tikzcd}$$

This technique was adapted to encapsulate classes, so that only the class name appears in the dictionary. A rather more complex variant was developed, which allows modules to be locked and unlocked (see `CLASS:-` and `ENDCLASS;` in blocks 1003 and 1005 of Listing 1 and "*Inheritance*" below).

Instance Creation

Forth already includes a concept similar to class, in its powerful `CREATE...DOES>` mechanism. Defining words such as `VARIABLE` and `CONSTANT` produced by this mechanism are in effect classes, whose instances are named variables and constants.

It falls short of what is required for full object orientation because instances may only have one behaviour, the code which follows `DOES>`. However if this code takes the form of a `CASE` statement, then "messages" left on the stack can be used to trigger different code sections, and thus provide methods.

A working first attempt was produced by employing immediate second-order compiling words ie. words which compile code which compiles code, giving the equivalent of an indirect `CREATE...DOES>`. A word called `CLASS:` defined named class words, which themselves defined named instances. A modified `CASE` statement, using strings instead of numbers as case selectors, permitted message passing.

Though compact and fairly elegant this solution was rejected because it drew the boundary between Forth words and "true objects" too close to Forth. Classes became ordinary Forth words and thus could not receive messages, nor have their own *class* variables and methods.

As well as this semantic objection, the author's experience was that such second order compiling words were very difficult to debug, and the source code became quite impossible to understand. A sample fragment of the code was :-

```
. . . . .
LIT-CFA , COMPILE BRANCH COMPILE ,
LIT-CFA , , COMPILE ALLOT
. . . . .
```

Instant brain damage. This was one of those traps which Forth's immense flexibility occasionally lays for the programmer; using a powerful technique for its own sake without first examining weaker alternatives.

A fresh attempt using conventional data structures, created in the dictionary by `ALLOT`, allowed classes to become objects with full rights. Instances became headerless storage areas in the dictionary which contain pointers to their class code. The Forth representation of an object is a pointer to the object left on the stack. This pointer can be followed into the class body, inside which a `CASE` statement permits a method to be selected at runtime.

Instance Variables

The most important discovery was the mechanism for addressing instance variables.

Each instance of a class contains an identical set of instance variables. The name of the variable (which lives in the *class*) must somehow be associated with different actual addresses depending upon which instance is being addressed.

Using `CREATE...DOES>`, instance variable names can be made "active". At compile time they store into themselves an offset relative to the start of the abstract object. At runtime they add this offset to the start address of the message receiving instance (obtained from the object stack) and leave their effective address on the parameter stack just like regular Forth variables. Inside a method, they can therefore be manipulated by the usual words `@` and `!` for wordsize, `C@` and `C!` for bytesize and so on.

Exactly the same considerations apply to *class variables*, local variables which can be used in class or instance methods. Class variables are visible to the instance methods but not *vice versa*. Class variables are shared by all the instances of a class, unlike instance variables which are private to each instance; ie. each instance has its own set.

This code can be inspected in the definitions of `CLASSVAR` and `INSTVAR` in block 1011 of Listing 1.

Message Passing

Much effort was devoted to experimenting with message passing mechanisms. Three different schemes were tried, each of which had implications for the syntax, before settling on the final version. These were :-

a) Make objects "active" and messages "passive". In other words, objects are named Forth words which gobble up messages from the input stream.

Rejected because this makes objects messy to assign, to pass as parameters or to return as results from a method (they need to be "ticked" if their address is required, and `>BODY`'d as well in Forth-83).

b) Make objects passive and messages active. Messages become Forth words which gobble up an object address from the stack.

Rejected because messages now have to be defined as globals in the visible Forth dictionary. If they are to be sent in their own class's methods (which is often desirable), then they have to be defined before the class has been defined. This is a horribly unattractive constraint in an interactive language.

c) Make both objects and messages passive. Objects merely place their address on the stack, while messages are simply strings in the input stream. This requires either a modified Forth outer interpreter which can parse the message passing syntax, or more simply, a word which passes a message to an object whose address is on the stack. This latter course was chosen because it leaves the outer interpreter intact, permitting ordinary Forth programs to executed alongside object oriented ones.

The word \ (block 1012 in Listing 1) takes a message from the input stream using `WORD`, and hashes it into a numeric code. The object's address on the stack is used to fetch a pointer to its method code, which can then be `EXECUTEd`, passing the message (as its hash code) on the stack.

The resulting syntax is `<object> \ <message>`. Although this departs from Smalltalk syntax, it has the advantage of making clear in the source code that a message is being passed. Together with suitable naming conventions, this helps to avoid confusion between ordinary Forth operations and message passing to objects, without incurring the cost of parsing names.

True hashing is not used, the hash value being merely used as the argument in a `CASE` look-up. Hence collision resolution is not possible; if two codes collide, the first matching method will always be selected. A relatively crude hashing algorithm has been adopted here for illustration purposes, and it would pay to incorporate a more powerful one to reduce the likelihood of collisions. Hashing is performed at compile time so this would only reduce efficiency when interpreting from the keyboard.

A third stack called `OBSTACK` is deployed to hold object addresses, acting as a return stack for nested message passing (which occurs when a message invokes a method in which another message is sent). An arbitrary limit of 32 levels of such nesting is implemented in Listing 1.

The top of `OBSTACK` provides the equivalent of Smalltalk's "self", a reference to the current receiver of a message. (The author has written Z80 code primitives for the object stack and message pathway, but prefers to present a high-level Forth version here for readability and portability).

Inside the method code, the well known (and loved) Eaker `CASE` statement [EAK80] is used to select a method, using the hashed value as a selector. For cosmetic reasons, `OF` and `ENDOF` are redefined as `::` and `;;`, which are meant to suggest the analogy between methods and colon definitions.

This message passing pathway incurs most of the runtime overhead, and therefore needs to be as short and fast as possible. Wherever possible work has been pushed back to compile time by using `[COMPILE] LITERAL`. Note however that using a `CASE` statement prohibits any modifications to provide early binding. The code for an individual method has no header and hence it does not have a code field address that could be directly compiled.

Inheritance

The inheritance of methods is trivially simple to do (see `>super` in block 1013 of Listing 1). Into the default option of the `CASE` statement is compiled a pointer to the methods of the parent class ("superclass"). If a message selection fails to find a match, it continues to try in each of a possible chain of superclasses until it reaches `CLASS %OBJECT`, the superclass of all classes. If a match is not found here then an error message is issued; the object does not understand that message.

Inheritance of class or instance variables is more tricky, especially as such variables are themselves allowed to be objects.

Class or instance variables may either be scalars ie. plain storage areas any number of bytes long, or they may be objects of a previously defined class. The reason for allowing scalars is purely one of efficiency; in low level classes it is not desirable to incur the overheads associated with objecthood merely to store a 16-bit integer. It also allows scalar arrays (sic!!) to be indexed in plain Forth for maximum speed, for example when handling text strings. The Smalltalk concept of a variably sized *indexed variable* is not implemented.

The disadvantage of this choice is that the programmer needs to keep a clear distinction between these two types of instance variable; this problem is however inherent in any hybrid system and can only be surmounted by a rigorously homogeneous object orientation such as is found in full Smalltalk-80.

Scalar variables are represented merely as offsets into the object body, and can be inherited very simply by bumping up the allocation of variable space in a new class.

However when class or instance variables are themselves objects, they have *structure* in the shape of a pointer to their parent class. The mechanism devised for inheriting such objects involves creating a *symbol table* from the variable declarations in a class definition (see `=>` and `make.symtab` in block 1007).

The format of a symbol table is :-

```
count offset pointer offset pointer offset pointer ...............
```

Symbol table entries are only created for variables which are objects, scalars being inherited by the simpler means described above.

When a new instance of a class is created, this symbol table is used to guide a run through the new object's data space, inserting the various class pointers in the correct places (`embed` block 1006). Recursion, implemented by `MYSELF`, is employed when objects composed of objects composed of objects etc. are encountered.

Having in this way inherited all the *storage space* of the variables from superclasses, it remains to inherit the variable identifiers.

The naive solution of actually copying the original `CLASSVAR` and `INSTVAR` code into each new subclass was rejected as too wasteful of memory. Another possible solution, used in Neon, is to implement separate dictionaries for methods and instance variables, and to chain these directly to those of their superclass. This would involve a rather more complex encapsulation scheme than that used here, in which the search order was directed; in the author's scheme, the context and current vocabularies are in force until the final act of sealing occurs. Therefore a solution was devised whereby the superclasses of a new class are *unlocked* at compile time so that all their internal declaration details are in the current search order. References to them can be compiled in the normal way. When compilation ends and the new class is locked, all the superclasses are locked up again. The words `ZIP` and `UNZIP` in block 1008 do this job.

Classtable

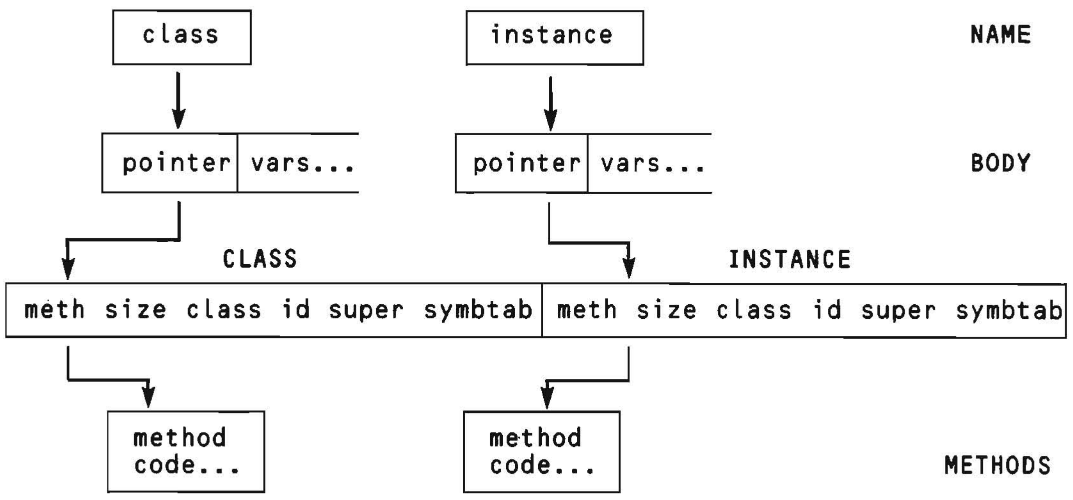
To introduce symmetry between classes and instances, it was necessary to build two levels of indirection into the message passing pathway. An object name points to the object body, which begins with a pointer to a table, which in turn points to the method code.

This table, the classtable, has fields which contain all the information needed to traverse the system, including the methods address, storage size, address of superclass, address of the print name (id) and address of symbol table.

It also includes a circular reference to the class itself, which allows the class of any object to be determined by sending it a message. This in turn allows a class *instance* to execute *class* methods such as `new`, so that objects can spawn copies of themselves at runtime.

Classes are objects with almost full rights but they are not instances of any class (in Smalltalk they would be – see *Comparison* section below). Hence the classtable is split into two symmetrical halves, one for the class's own use and the other for the use of its instances. At the cost of some minimal duplication of data, such a split table permits switching from class to instance data by adding a constant offset, and thus simplifies earlier stages of the message pathway.

The chain of indirection looks like this :-



Given an object's address on the stack, a method can therefore be executed by `@ @ EXECUTE`.

Class %OBJECT

To terminate the chain of inheritance it's necessary to define a primitive class in the system called `CLASS %OBJECT`.

This class is the superclass of all other classes. It has the useful property that its class and instance methods are inherited by all user defined classes. Among its instance methods one can put any desired default behaviours, including on-line help facilities.

Only a few default class methods are shown here, namely :-

`new`, the method which creates new instances.

`size` returns the size of a class's storage in bytes.

`name` prints the name of the class.

`gen` prints the "genealogy" of a class, ie. the list of all its superclasses.

The default instance methods are :-

`class`, which returns a pointer to the instance's class.

`size` and `gen`, as above, but for an instance.

Syntax

The screens in Listing 2 show four sample class definitions to illustrate the definition syntax.

Methods can be written in plain Forth, or in a mixture of plain and message passing Forth. Typically low level classes will be defined in pure Forth, and the message passing syntax will be introduced progressively into their sub-class definitions.

It's permissible to introduce ordinary Forth variables, constants and ordinary colon definitions into the declarations section of a class definition. Forth variables so used have the valuable property that they are not reproduced in the individual instances but are accessible from all instances (they behave like class variables). The important difference is that they cannot be inherited by subclasses, and of course they cannot be objects of a previously defined class. Like class variables, they can be used as semaphores, maps or counters or as communication channels between instances, and they permit nested scoping of variables; the only reason to prefer them to class variables is that they are slightly more efficient. Colon definitions may be introduced like "macros" to replace repeated sections of method code.

Instances of a class (say class `%point` from the examples), are made by sending the message `new` to the class. This creates an anonymous instance of `%point` at `HERE`, and leaves its address on the stack. This address might be stored into a Forth variable :-

```
VARIABLE fred
%point \ new fred !
```

but this results in a rather nasty syntax for message passing :-

```
fred @ \ show
```

For this reason block 1016 supplies definitions for a version of the well-known `TO` variables [BAR79], called `OBJ`, which behave like Forth constants (ie.return contents rather than address) and offer a much prettier syntax. :-

```
OBJ fred
%point \ new -> fred
...
fred \ show
```

A more efficient way of achieving this effect would be to use the multiple code-field words described in [LYO80] and [SCHL83] which do the work at compile time.

Alternatively one could create an instance as a named Forth word, by :-

```
CREATE fred  %point \ new  DROP
```

A variable is to be preferred when the object is being returned as the value of an expression. However be aware that subsequently assigning to such a variable can leave the original object inaccessible as its pointer is lost.

From now on this object can be used in any Forth program, but its contents and behaviour can only be accessed or modified by sending the appropriate message to it :-

```
: streak  20 0 DO I I fred \ put delay
                      fred \ hide
                 LOOP ;
```

Messages may also be sent to the class, to alter class variables. We could change the origin of the coordinate system by :-

```
20 20 %point \ setorg
```

and now whenever the message `home` is sent to an instance of `%point`, the point will return to `20 20`. Note that there is no initialisation in class `%point` so a `setorg` ought to be sent before using any instances.

Because the outer interpreter is unaltered, the basic syntax of Forth is unaffected. In particular, arithmetic and expression syntax is still Reverse Polish :-

```
fred \ where 2 +  fred \ put
```

As mentioned above, there is a serious penalty for adopting this hybrid approach which must be clearly understood. Because there exist entities which are not objects (viz. numbers and Forth words), the possibility exists of sending a message to such an entity. This is a fatal error, which cannot be trapped in any easy way by the system since a number cannot be distinguished from an object address. Trapping such errors would involve building some kind of Lisp-style tagged memory system. The choice of a good naming convention may help to discourage such mistakes.

Comparison with Smalltalk

Despite the small size of these extensions (they compile to 2883 bytes of code on the author's system), they mimic quite a lot of the semantics of the Smalltalk-80 kernel.

Methods may be redefined ("overidden") in sub-classes, and the associated messages will always trigger the appropriate version. For example, the same message + could be used to add numbers, dates, complex numbers and to concatenate strings and lists.

Both *self* and *super* are implemented, to send a message to the receiver itself, or to its superclass (to access a previous version of a locally overidden method).

The author conjectures that the system is sufficiently expressive to bootstrap a fully object-oriented system, comparable to the Smalltalk environment, by compiling the extensions onto a minimal Forth nucleus and writing the high-level system (editor, file system, graphics etc.) in the message passing syntax. However he has no interest in doing so!

The inheritance mechanism is not precisely equivalent to that in Smalltalk-80. Here, classes are not instances of any other class, and are therefore not quite full objects. This is a consequence of a profound philosophical problem concerning class membership (and Godel's solution is not altogether helpful here). Smalltalk-80 was forced to resolve it by an ugly and confusing artifice, namely the introduction of Metaclasses (of which classes are instances) and the class METACLASS of which Metaclasses are instances. Metaclasses are used to provide default values so that their member classes can create *initialised* instances [see GOL83 chapter 5].

In the interest of a simple implementation the author has avoided Metaclasses. Initialised instances can be created using class variables to hold the default values, and an instance method `init` which reads them into an instance. For example in the class `%point` of Listing 2, we might add an instance method `init` like this :

```
   ~  init  ::  Xorg @ X !  Yorg @ Y !  self  ;;
```

Newly created points will have to be explicitly initialised by sending them the message `init` :-

```
   %point \ new \ init -> fred
```

It should be possible to modify the code for `new` in class `%OBJECT` so that it automatically sends `init`, if present.

In addition to class variables, Smalltalk provides other kinds of shared variables called *pool variables*, which are shared by all the instances of several classes. These are not implemented here. Of course global variables can be provided by ordinary Forth variables, declared outside any class definition.

A major difference from Smalltalk lies in memory management. Smalltalk has dynamic memory management, so that objects can be created and disposed of "on the fly", and also virtual memory, so that objects can be kept on disk and read into main memory as needed. The requirements of this memory system are very severe and are the prime reason for the difficulty and inefficiency of current Smalltalk implementations. In particular the "garbage collection" of no longer needed objects imposes a huge computational burden.

Because of Forth's singly linked dictionary structure, it is not easily possible to do garbage collection; this limits the capacity to dynamically create and destroy objects at runtime. Objects will continue to occupy dictionary space until removed (rather unselectively) by `FORGET`.

A full solution would involve allocating objects on a heap, and would require memory compaction with pointer adjustment (since objects are not fixed length). This goes far beyond the original intention of the project but could be added to the system. [DRES85] describes a heap manager in Forth which would fit admirably. Another possible avenue for experiment is to use "caged" allocation in the dictionary (ie. keeping objects of the same class, and hence size, in the same cage) which would allow garbage collection without relocation or compaction.

It is possible to create virtual memory objects which are called in from disk. This involves rewriting the `new` method to create objects in a disk buffer instead of the dictionary, and making such objects return a virtual address ie. block and offset. Class definitions would remain RAM resident.

An irritating deficiency is the lack of a method which makes an object list all the messages that it understands. The first version had such a message called `all`; this was possible because string comparison was used for method selection. The current implementation uses hashing for efficiency, but the price paid is that the message selector strings are not stored at all in the compiled code. A not too costly remedy would be to add a field to the class table which points to a list of message selector strings, copied from the source at compile time, and which could reside in RAM or on disk.

Implementation Details

These extensions have been successfully implemented on both Forth-79 and Forth-83 standard systems (xForth v 2.0 and LMI PC-Forth respectively). The program is not a standard program, because it interferes directly with dictionary headers and vocabularies; nevertheless it uses very few non-standard words.

Listings 1 and 2 are in Forth-83. The code can be very easily converted back to Forth-79, by removing the redefinition of `NFA` from block 1001, increasing the argument to PICK by one wherever it occurs, and replacing `' >BODY` by `[COMPILE] '` in block 1008, line 6.

The well known Eaker case statement is used in screen 1014 to define the method delimiters. There is not room here to supply Eaker's code, which is in the public domain [EAK80]. Eaker's `ENDCASE` always compiles a `DROP` which is not required in `ENDMETHODS;`. Hence I define a new version of `ENDCASE`, called `XENDCASE` for use in `ENDMETHODS;`, which doesn't compile this `DROP`. In blocks 1018 and 1019 (the definition of class `%OBJECT`), the ordinary `ENDCASE` is used.

Some further comment on the definitions of `METHODS:-` and `ENDMETHODS;` is in order (blocks 1013 and 1014). The code for the methods in a class body is headerless, and the word `METHODS:-` has to explicitly construct a code field which simulates a colon defined CASE statement.

This is clearly very implementation dependent; you must discover what the code field of a colon definition contains in your system (ie. the address of `DOCOLON`) and substitute it here. The code in block 1013 assumes two byte code fields; if your system is direct threaded they may be three bytes (CALL xx xx).

Similarly, the word `ENDMETHODS;` finishes the compilation by directly compiling the address of `SEMI` (C9D hex in my case). This is because built-in stack checking defeated higher level solutions such as `[COMPILE] ;` which may work on your system.

The words `LOCK` and `UNLOCK` (screen 1003) which actually operate the encapsulation mechanism, work by swapping link field contents on the assumption that the dictionary is a simple singly linked list. They may need to be modified in systems with more complex dictionary linking (eg. polyFORTH); only experimentation with your system will tell.

`MYSELF` is a word which permits recursion; it invokes the word in whose definition it appears. It can be defined :-

```
: MYSELF  CONTEXT @ @ NAME> , ;  IMMEDIATE
```

`TRUE` and `FALSE` are just constants of value `-1` and `0` (`1` and `0` in Forth-79).

In block 1018 the word `ID.`, which is not a standard word, prints the name of a definition given its name field address. It, or a synonym such as `.NAME`, is supplied in most Forth systems.

`ENDIF` is merely a synonym for the loathsome `THEN` (`THEN` what?), an unashamed returned to FIG!. Replace it by `THEN` if you must.

Finally some comments on style. This code makes much heavier use of variables than is normal in Forth. This was done deliberately for readability; had optimal use been made of the stack then a definition like SUPERCLASS:- would probably never have got debugged! Compilation could be speeded up somewhat by rewriting for better stack usage.

In potentially explosive new control structures such as these it is desirable to incorporate error checking. This has been omitted here for the sake of clarity. CLASS:- ENDCLASS; and the other pairs would be checked for parity using PAIRS, and also for illegal use inside a colon definition.

Performance

Benchmarking such a set of language extensions is not an easy task, as one ought to test every possible kind of program. Nevertheless some idea of the performance penalties of using objects can be gleaned from some relatively simple tests.

A class %benchmark was defined as follows :-

```
CLASS:- %benchmark
SUPERCLASS:- %OBJECT
ENDCLASSVARS; CLASS METHODS:- ENDMETHODS; - no class vars or methods

WRD  INSTVAR test   ENDINSTVARS;

INSTANCE METHODS:-
~ go1    :: 9 test ! ;;
~ go10   :: 9 test ! 9 test ! 9 test ! 9 test ! 9 test !
            9 test ! 9 test ! 9 test ! 9 test ! 9 test !  ;;
~ go20   :: 9 test ! 9 test ! 9 test ! 9 test ! 9 test !
            9 test ! 9 test ! 9 test ! 9 test ! 9 test !
            9 test ! 9 test ! 9 test ! 9 test ! 9 test !
            9 test ! 9 test ! 9 test ! 9 test ! 9 test !  ;;
~ go50   :: 9 test ! etc ..........
ENDMETHODS;
ENDCLASS;
```

The following benchmark programs were then run. Each word performs the same number of variable stores, namely 10000 :-

```
VARIABLE test
: bmk1  10000 0 DO  9 test ! LOOP ;

OBJ  bmk   %benchmark \ new -> bmk
: bmk2  10000 0 DO  bmk \ go1  LOOP ;
: bmk3  1000  0 DO  bmk \ go10 LOOP ;
: bmk4  500   0 DO  bmk \ go20 LOOP ;
: bmk5  200   0 DO  bmk \ go50 LOOP ;
```

The following timings, in seconds, were obtained on a 4MHz Z80 system.

bmk1	bmk2	bmk3	bmk4	bmk5
1.2	6.3	2.3	2.1	2.0

The conclusion which can be drawn from these is that, as expected, the penalty is largely a fixed overhead caused by the message passing code which has to executed. The contribution of this overhead to the total time diminishes as the amount of work performed in a method increases.

The overhead is not quite fixed, because there is a smaller overhead associated with the action of instance variables, which have to perform an addition to produce the absolute address of their target. Hence the timings do not converge to the bmk1 time (for pure Forth). bmk2 which is dominated by the fixed overhead is 5.25 times slower than pure Forth; rather better than the case for Neon with its late binding option engaged [see DUF84 page 24]. The convergence time appears to be 2 seconds (1.7 times slower than pure Forth), which represents the instance variable overhead, the fixed message passing overhead having been swamped by this stage.

It's difficult to conclude from these timings just how fast real programs will run; it clearly depends upon the amount of work done in the methods, and the degree of nesting of objects (nested message sends incur the fixed overhead at each level). They suggest that late binding imposes penalties which are significant but not catastrophic. They also suggest that optimisation work (ie. rewriting in code) should be concentrated on the message passing pathway (ie. \), the object stack and instance variable addressing (INSTVAR), and that nesting of objects should not be indulged in without thought.

Conclusion

The source code in Listing.1 provides a set of extensions to Forth which introduce classes with sub-class inheritance and message passing. These new features may be freely combined with conventional Forth programs.

The extensions permit information hiding of both code and data, and thereby a secure way of factorising large programs into sealed modules which have a well defined programmer interface.

The intended use of these extensions is to simplify the management of complex data objects (especially ones of which multiple copies are needed eg. animated graphic images, database records, resource managers). It is not really worthwhile to use them on simple objects for which Forth already supplies an efficient representation. The author intends to experiment with Forth objects as a representation for frames and rules in knowledge based systems.

Acknowledgments

The author wishes to thank the editor and his referees for their constructive comments which have helped to improve this paper considerably.

References

[GHEZ82] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, Wiley, New York, NY, 1982. ISBN 0-471-08755-6

[GOL83] A. Goldberg and D. Robson, *Smalltalk-80: the Language and Its Implementation*, Addison-Wesley, New York, NY, 1983. ISBN 0-201-11371-6

[SCHOR80] D.V. Schorre, *Forth Dimensions*, Vol.2 No.5:132, Forth Interest Group, San Jose, CA, 1980.

[EAK80] Charles E. Eaker, *Forth Dimensions*, Vol.2, No.3:37, Forth Interest Group, San Jose, CA, 1980.

[DUF84] C. Duff and N.D. Iverson,"Forth meets Smalltalk", *Journal of Forth Application and Research*, Vol. 2, No. 3, Institute for Applied Forth Research, Inc., Rochester, NY, 1984.

[SCHL83] K. Schliesiek,"Multiple Code Field Data Types and Prefix Operators", *Journal of Forth Applications and Research*, Vol.1, No.2, Institute for Applied Forth Research, Inc., Rochester, NY, 1983.

[BAR79] P. Bartholdi,"The TO solution", *Forth Dimensions*, Vol.1, No.4, Forth Interest Group, San Jose, CA, 1979.

[LYO80] G.B. Lyons, "Note on the TO solution", *Forth Dimensions*, Vol.1, No.5, Forth Interest Group, San Jose, CA, 1980.

[DRES85] W.B. Dress, "A Forth Implementation of the Heap Data Structure", *Journal of Forth Application and Research*, Vol. 3, No. 3, Institute for Applied Forth Research, Rochester, NY, 1985.

Manuscript received June 1985.

Dick Pountain graduated in Chemistry and Biochemistry from Imperial College, London University. He is a technical author and journalist living in London, and is at present UK Contributing Editor to Byte magazine (USA) and Contributing Editor to Personal Computer World (UK). Has been a keen Forth programmer for 5 years. Programs in several other languages including Pascal, occam, Prolog, Lisp and Hope and is interested in applying their better features for the enhancement of Forth.

Listing 1

```
Screen # 1001
( Object oriented Forth - v 8.0        Dick Pountain March 1985)

-- these variables are for the compiler's use; their values are
-- not meaningful at runtime, and must NOT be used in methods.

VARIABLE class.table         -- holds start address of class table
VARIABLE class.id            -- holds class name's PFA.
VARIABLE class.adr           -- holds start address of class body.
VARIABLE alloc               -- accumulates storage size allocated
VARIABLE symbols             -- accumulates count of symtab entrys
VARIABLE cl/inst             -- flag for class or instance.

: NFA   BODY> >NAME ;        -- back to Forth-79 !!

: 1+!  1 SWAP +! ;           -- for convenience.
                                                              -->
```

```
Screen # 1002
( Object oriented Forth - v 8.0        Dick Pountain March 1985)

-- these words are used as field identifiers in the class-table

: .size    2+  ;              -- to storage size field
: .class   4 + ;              -- to class body pointer field
: .id      6 + ;              -- to class name pointer field
: .super   8 + ;              -- to superclass body pointer field
: .symbol 10 + ;              -- to symbol table pointer field
: .inst   12 + ;              -- to instance half of class table

0 .inst .inst  CONSTANT class.table.size

: .store  2+ ;            -- offset to storage part of object body

  -->
```

```
Screen # 1003
( Object oriented Forth - v 8.0        Dick Pountain March 1985)

-- Start the named module which hides class code

: CLASS:-    CREATE HERE class.id !                -- Store its PFA
             6 ALLOT                    -- Make space for 3 addresses
             0 alloc !  0 symbols !               -- Initialisations
             DOES> @ ;                                     ( ---  )

-- point end-of-module link field to classname's NFA
: LOCK       DUP  2+ @  SWAP NFA SWAP ! ;  ( classnamePFA ---  )

-- point endmod link field back to last definition inside module
: UNLOCK     DUP 4 + @  SWAP 2+ @  ! ;      ( classnamePFA ---  )

                                                              -->
```

```
Screen # 1004
( Object oriented Forth - v 8.0       Dick Pountain March 1985)

-- unlock all the parent classes of a subclass to permit the
-- inheritance of identifiers

: UNZIP   class.table @ BEGIN  .super @  ?DUP           ( --- )
                        WHILE  @ DUP  .id @  UNLOCK
                        REPEAT ;

-- re-lock all the parent classes of a subclass

: ZIP     class.table @ BEGIN  .super @  ?DUP           ( --- )
                        WHILE  @ DUP  .id @  LOCK
                        REPEAT ;

 -->

Screen # 1005
( Object oriented Forth - v 8.0       Dick Pountain March 1985)

-- seal the class module; creates a 'phantom' definition which
-- has only a null name field and a link field.

: ENDCLASS;    HERE                          -- this is endmod's NFA
               0 [COMPILE] C,                -- make null name field
               0 [COMPILE] ,              -- and an empty link field
               class.id @                       -- get classname PFA
               DUP class.adr @ SWAP !       -- store class body addr
               OVER 1+  OVER 2+ !              -- then endmod LFA
               CONTEXT @ @  OVER 4 + !           -- then last NFA
               LOCK   CONTEXT @ !          -- lock module and relink
               ZIP ;                         -- re-lock superclasses
                                                         ( --- )
  -->

Screen # 1006
( Object oriented Forth - v 8.0       Dick Pountain March 1985)

-- insert class pointers into objects used as class or instance
-- variables of new objects

: embed     DUP  2+ SWAP @       -- fetch count of symtab entries
            ?DUP                 -- if zero, nothing to do
            IF
              0 DO DUP I 4 * +               -- addr of tab entry
                   DUP @ SWAP 2+ @ SWAP      -- fetch offs and ptr
                   3 PICK .store +  2DUP !   -- insert ptr in obj
                   SWAP .symbol @ DUP @     -- is varobj compound?
                   IF MYSELF ELSE 2DROP ENDIF    -- if so recurse
                LOOP
            ENDIF  2DROP ;                ( newobj symtab ---   )
  -->
```

```
Screen # 1007
  Ø ( Object oriented Forth - v 8.Ø       Dick Pountain March 1985)
  1
  2 -- declare class of object used as class or instance variable
  3 : =>        @ .inst  alloc @  OVER .size @ .store  symbols 1+! ;
  4                                     ( obj --- ptr offset size)
  5
  6 -- make a symbol table from data left on stack by =>
  7 : make.symtab   symbols @  DUP , ?DUP IF Ø DO , , LOOP ENDIF ;
  8                                                ( .... n2 n1  ---  )
  9
 10 -- inherit all variable-objects from superclasses      ( --- )
 11 : inherit.vars    class.adr @ DUP             -- get class body addr
 12                   BEGIN @ DUP .symbol @       -- get its symtab addr
 13                        2 PICK SWAP embed      -- insert pointers
 14                        .super @ ?DUP          -- then to superclass
 15                        WHILE REPEAT DROP ;                        -->

Screen # 1008
  Ø ( Object oriented Forth - v 8.Ø       Dick Pountain March 1985)
  1
  2 -- set up classtable and fill in those fields which are known
  3
  4 : SUPERCLASS:-    HERE class.table !       -- start of class table
  5                   class.table.size ALLOT      -- allot table space
  6                   ' >BODY  @ DUP             -- get superclass ptr
  7                   @ .size @ alloc +!    -- inherit classvars size
  8                   DUP class.table @ .super !        -- write super
  9                   class.table @ .inst .super !        -- fields &
 10                   class.id @  DUP class.table @ .id !  -- id.....
 11                   class.table @ .inst .id !              -- fields
 12                   UNZIP ;                    -- unlock superclasses
 13
 14
 15  -->

Screen # 1009
  Ø ( Object oriented Forth - v 8.Ø       Dick Pountain March 1985)
  1
  2 -- complete filling in classtable; construct class symboltable
  3
  4 : ENDCLASSVARS;   HERE class.adr !            -- start of class body
  5                   class.table @ ,       -- store pointer to table
  6                   alloc @  DUP ALLOT     -- allot classvars space
  7                   class.table @ .size !       -- & write to table
  8                   class.adr @ DUP class.table @ .class !
  9                   class.table @ .inst .class !
 10                   HERE class.table @ .symbol !  -- start of symtab
 11                   make.symtab   Ø symbols !  -- build symbol table
 12                   inherit.vars                -- inherit class vars
 13                   class.table @ .super @ @  -- and size of...
 14                   .inst .size @ alloc !  ;  -- inst vars.
 15  -->
```

```
Screen # 1010
( Object oriented Forth - v 8.0        Dick Pountain March 1985)

-- Create a third stack to hold addresses of nested objects.
-- Stack pointer is at OBSTACK, grows to low addresses.

CREATE  OBSTACK HERE 64 + ,  64 ALLOT

-- push object address to OBSTACK, fetch method CFA and execute.
: send    DUP  -2 OBSTACK +!  OBSTACK @ !  @ @  EXECUTE ;
                                             ( hash obj ---  )
-- pop object stack
: pop     2 OBSTACK +! ;                                ( ---  )

-- copy object stack top to parameter stack
: self    OBSTACK @ @ ;                               ( --- obj)
: self+   OBSTACK @ @ + ;                                   -->

Screen # 1011
( Object oriented Forth - v 8.0        Dick Pountain March 1985)

-- Declare variables ; preceded by a size or 'class' =>

: CLASSVAR        CREATE alloc @ ,  alloc +!   IMMEDIATE
                  DOES> @ class.adr @ .store + [COMPILE] LITERAL ;
                                                    ( size ---   )
: INSTVAR         CREATE alloc @ ,  alloc +!   IMMEDIATE
                  DOES> @ .store [COMPILE] LITERAL COMPILE self+ ;
                                                    ( size ---   )

-- Create instance variables symbol table
: ENDINSTVARS;    HERE class.table @ .inst .symbol !      ( ---  )
                  make.symtab
                  alloc @  class.table @ .inst .size ! ;
                                                             -->

Screen # 1012
( Object oriented Forth - v 8.0        Dick Pountain March 1985)

-- compute hash from string                           ( s --- n )
: hash      32 MIN 0 SWAP 0
            DO  OVER I + C@ I 1+ * +  LOOP  SWAP DROP ;

-- send message to object; syntax is :-  object \ message

: \          BL WORD COUNT hash        -- get selector & hash it
             STATE @ IF [COMPILE] LITERAL    -- if compiling...
                        COMPILE SWAP        -- compile message-
                        COMPILE send         -- passing code...
                        COMPILE pop
                     ELSE SWAP send  pop      -- else do it now.
                     ENDIF ;                          IMMEDIATE
-->
```

```
Screen # 1013
( Object oriented Forth - v 8.0       Dick Pountain March 1985)

: CLASS  0 cl/inst ! ;     : INSTANCE  0 .inst cl/inst ! ;  HEX

: METHODS:- HERE                                   -- methods CFA
            class.table @  cl/inst @ + !        -- write to table
            A4 [COMPILE] C,         -- methods are headerless, so
            19 [COMPILE] C,             -- must build a code field
            [COMPILE] ]  [COMPILE] CASE ;       -- Start compiling

-- chain to superclass's methods if selector not matched
: >super    class.table @ .super @ @                  (  ---  )
            cl/inst @ +  @
            [COMPILE] LITERAL  COMPILE EXECUTE ;     IMMEDIATE

 -->

Screen # 1014
( Object oriented Forth - v 8.0       Dick Pountain March 1985)

-- delimiters for method source code (uses Eaker CASE construct)

: ::          [COMPILE] OF ;                          IMMEDIATE

: ;;          [COMPILE] ENDOF ;                       IMMEDIATE

: XENDCASE    4 ?PAIRS BEGIN SP@ CSP @ = 0=
                       WHILE 2 [COMPILE] THEN REPEAT CSP ! ;
                                                      IMMEDIATE
: ENDMETHODS; [COMPILE] >super
              [COMPILE] XENDCASE
              [COMPILE] [  C9D , ;  -- dirty end to compilation
                                                       IMMEDIATE
 DECIMAL                                                     -->

Screen # 1015
( Object oriented Forth - v 8.0       Dick Pountain March 1985)

-- Hash a method selector and compile into method code.

: ~        BL WORD COUNT   hash   [COMPILE] LITERAL ;   IMMEDIATE

-- Pseudo-object super; returns pointer to superclass

: super   self @ .super @ ;                         (  --- obj)



-- some possible prefixes for CLASSVAR and INSTVAR
1 CONSTANT BYTE    2 CONSTANT WRD    4 CONSTANT DBL

                                                            -->
```

```
Screen # 1016
( Object oriented Forth - v 8.0       Dick Pountain March 1985)

-- Special 'TO' variable to hold object. When executed they
-- return their contents rather than addresses, like a constant.

  VARIABLE storing?    FALSE storing?   !

-- declare a new object variable eg. OBJ X.
: OBJ   CREATE 2 ALLOT
        DOES>  storing? @  IF  FALSE storing? !   !
                           ELSE  @  ENDIF ;

-- Input to variable; eg. 23 -> X
: ->     TRUE storing? !  ;


                                                              -->

Screen # 1017
( Object oriented Forth - v 8.0       Dick Pountain March 1985)

-- Class %OBJECT is the parent class of all classes; it contains
-- the default behaviour of all objects. As it is the primitive
-- class we have to use some kludges when building it.
HEX
CLASS:- %OBJECT
HERE class.table !             -- build its classtable 'by hand'
class.table.size  ALLOT
HERE class.table  @ .symbol ! 0 ,
HERE DUP  class.adr !        class.table @ ,
DUP   class.table @ .class ! class.table @  .inst .class !
0 DUP class.table @ .super ! class.table @  .inst .super !
0 DUP class.table @ .size !  class.table @  .inst .size  !
class.id @ DUP class.table @ .id ! class.table @ .inst .id !
                                                              -->

Screen # 1018
( Object oriented Forth - v 8.0       Dick Pountain March 1985)
CLASS METHODS:-
~ new    ::    HERE
               self @ .inst DUP ,  .size @ ALLOT
               self BEGIN @ DUP .inst .symbol @
                          2 PICK SWAP embed
                          .super @ ?DUP
                    WHILE REPEAT                              ;;
~ size   ::    self @ .size @                                 ;;
~ name   ::    self @ .id @ NFA ID.                           ;;
~ gen    ::    2 SPACES self BEGIN  DUP \ name
                                    @ .super @ ?DUP
                             WHILE  ." <- " REPEAT            ;;
2 SPACES  ." That's not a "  self \ name
          ." message!" CR
ENDCASE [ C9D ,                                               -->
```

```
Screen # 1019
( Object oriented Forth - v 8.0       Dick Pountain March 1985)

HERE class.table @ .inst .symbol !  0 ,

INSTANCE  METHODS:-
~ class  ::  self @  .class                             ;;
~ size   ::  self @  .size @                            ;;
~ gen    ::  self \ class \ gen                         ;;

2 SPACES  ." That's not a "  self \ class \ name
          ." message!"   CR
ENDCASE [ C9D ,
ENDCLASS;

DECIMAL

```

Listing 2

```
Screen # 1001
( Sample class definitions to show syntax)

CLASS:-  %point
SUPERCLASS:- %OBJECT

WRD CLASSVAR Xorg  WRD CLASSVAR Yorg  -- 16-bit scalar variables
ENDCLASSVARS;

CLASS METHODS:-
~ setorg :: Yorg ! Xorg ! ;;            -- set origin ( n n ---   )
~ getorg :: Xorg @ Yorg @ ;;            -- get origin (   --- n n )
ENDMETHODS;

WRD INSTVAR X    WRD INSTVAR Y  ENDINSTVARS;


                                                                -->

Screen # 1002
( Sample class definitions to show syntax)

INSTANCE METHODS:-
~ show  :: X @ Y @ GOTOXY ." *" ;;  -- print a * at X,Y
~ hide  :: X @ Y @ GOTOXY ."  " ;;  -- blank it out again
~ put   :: Y ! X !  self \ show ;;  -- load new X and Y ( n n ---)
~ left  :: self \ hide -1 X +!  self \ show ;;
~ right :: self \ hide  1 X +!  self \ show ;;
~ up    :: self \ hide -1 Y +!  self \ show ;;
~ down  :: self \ hide  1 Y +!  self \ show ;;
~ where :: X @ Y @  ;;
~ home  :: self \ hide   self \ class \ getorg  self \ put ;;
ENDMETHODS;
ENDCLASS;

  -->

Screen # 1003
( Sample class definitions to show syntax)

CLASS:- %square
SUPERCLASS:- %OBJECT

ENDCLASSVARS; CLASS METHODS:- ENDMETHODS;
-- NB there are no class variables or methods but
-- these null declarations are compulsory to make
-- a correct class table

%point => INSTVAR A     %point => INSTVAR B
%point => INSTVAR C     %point => INSTVAR D
-- instance variables are objects of class %point

VARIABLE size  -- shared by all instances of %square
10 size !                                                   -->
```

```
Screen # 1004
( Sample class definitions to show syntax)

INSTANCE METHODS:-
~ size :: size !  ;;                       ( n ---  )
~ put  :: 2DUP A \ put                     ( n n ---    )
          2DUP size @ +  D \ put
          2DUP SWAP size @ +  C \ put
          SWAP size @ +  SWAP B \ put ;;
ENDMETHODS;
ENDCLASS;





 -->

Screen # 1005
( Sample class definitions to show syntax)

CLASS:- %Gsquare
SUPERCLASS:- %square

ENDCLASSVARS; CLASS METHODS:- ENDMETHODS; ENDINSTVARS;

INSTANCE METHODS:-
~ grow :: A DUP \ left  \ up
          B DUP \ right \ up
          C DUP \ right \ down
          D DUP \ left  \ down  ;;
ENDMETHODS;
ENDCLASS;

  -->

Screen # 1006
( Sample class definitions to show syntax)

CLASS:- %GSsquare
SUPERCLASS:- %Gsquare

ENDCLASSVARS; CLASS METHODS:- ENDMETHODS; ENDINSTVARS;

INSTANCE METHODS:-
~ shrink :: C DUP \ left  \ up
            D DUP \ right \ up
            A DUP \ right \ down
            B DUP \ left  \ down  ;;
ENDMETHODS;
ENDCLASS;
```