Forth Meets Smalltalk

A Presentation to SVFIG

October 23, 2010

by Douglas B. Hoffman

CONTENTS

WHY FMS (Forth Meets Smalltalk)?

- Neon-like Forth object extensions have a long history of successful use in several Forths.
- Friendly and familiar syntax for creating classes.
- But there have been complaints (ordering of message-object, [ ] syntax, lack of methodless ivar access, etc.)
- Claimed to have inefficient (slow) dynamic binding.
- FMS addresses the complaints, but keeps what is liked.

NEON HERITAGE

- FMS class building syntax resembles Neon, but there are important differences.
- Retains Neon’s ease of use in defining new classes, e.g., interface definitions are not needed.
- Fully supports objects-as-instance-variables.
- Upgrade path to Multiple Inheritance thanks to Michael Hore and PowerMops.

SMALLTALK HERITAGE

- Any message can be sent to any object (Duck Typing). No interfaces needed.
- Object-message syntax.
- Message names end with colon “:”

TERMINOLOGY

- Smalltalk terminology is used in this presentation.
- **Early Binding**, a message send is resolved to the correct method at compile time.
- **Late Binding**, a message send is resolved to the correct method at run time. Also called **Dynamic Binding**.

EXAMPLE FMS SYNTAX

Begin a class definition

The class name is foo

```
:class foo
   :m hi:
          ." Hello"
   ;m
;class
```

The message name is hi:

Begin a method definition

End the method definition

The method itself

End the class definition

```
foo myfoo
```

Instantiate an object in the dictionary of class foo. The object is named myfoo.

Send the hi: message to the object.

```
myfoo hi: Hello
```

Note that all messages sent to public objects (not objects as ivars) are late bound.

Alternatively, instantiate an object in the heap. Heap> is a runtime only word.

```
: make ( -- ^obj )
  heap> foo ;
make constant myfoo2
myfoo2 hi: Hello
myfoo2 <free
```

Free the heap memory when done with object myfoo2

Define a class whose objects will have an instance variable (ivar) of size one cell. We normally declare all ivars prior to defining the methods. Here the BYTES ivar defining primitive is used.

Executing the ivar's name will return its address.

```
:class var
  cell bytes data
  :m @: ( -- n )
     data @ ;m
  :m !: ( n -- )
     data ! ;m
;class
```

The scope of the ivar name, in this case “data”, is private to the class definition and any subsequent subclasses. Message names are always global in scope.

Define a subclass of var, named var2. Objects of class var2 inherit all data and methods from the superclass var. Objects of class var2 will also respond to the three new print messages.

```
:class var2 <super var
  :m print1:    data @ . ;m
  :m print2:   self @: . ;m
  :m print3: [self] @: . ;m
;class
var2 v2   25 v2 !:
v2 print1: 25
v2 print2: 25
v2 print3: 25
v2 IV data @ . 25
```

Note explicit declaration of superclass

Four different ways to print the ivar value in a var2 object.

What are the implications of each?

Method print1: will be the fastest message send because it uses in-class METHODLESS IVAR ACCESS.

```
:m print1:    data @ . ;m
```

Method print2: uses an early bound message to the pseudo ivar SELF.

```
:m print2:   self @: . ;m
```

Method print3: uses a late bound message to the pseudo ivar SELF. The late binding is invoked by using the bracketed [SELF]. The advantage of using a late bound message send is we can more easily change the behavior of subclass methods (see next slide). The disadvantage is a late bound message send is always somewhat slower than one that is early bound. This speed difference may or may not be important to the application. Late binding to SELF is the most flexible way to design a method and is considered by some to be the preferred technique in *all* cases. In FMS the programmer has a choice.

```
:m print3: [self] @: . ;m
```

Methodless ivar access outside of a class definition will always be fastest because no message is sent (anywhere). But the downside is this trick violates the fundamental idea behind object programming. What if we later change the internal structure of the class such as changing data to be a char-length integer?

```
v2 IV data @ . 25
```

Define a subclass of var2 named var3. The @: method inherited from the superclass var has been overridden with a different method. Note that the overriding is implicit, i.e., there is no need to declare something like “override” because the intent is (should be) obvious.

```
:class var3 <super var2
   :m @: ( -- n )
      .” fetch from var3 ”
      data @ ;m
;class
var3 v3   25 v3 !:
v3 print2: 25
```

Method print2: is unaffected because the @: in class var2 is early bound to SELF.

```
v3 print3: fetch from var3 25
```

The late bind of @: to [SELF] in the print3: method of class var2 will use the most recently defined @: in the class hierarchy chain. Objects of class var and var2 are *not affected* by the redefinition of @: in class var3. Late binding to [SELF] is sort of like a context-dependent deferred definition. The context is the class.

ACCESSING OVERRIDDEN METHODS

```
:class var4 <super var
  :m @: ( -- n )
     ." fetch from var4 "
     data @ ;m
  :m test: ( -- n )
     self  @: .          Call the @: method in this class.
   cr super @: 1+ . ;m   Call the @: method in class VAR.
;class
var4 v4   95 v4 !:
v4 test: fetch from var4 95
96
```

THE OBJECT - MESSAGE CONTRACT

- Classic OOP theory would mandate that the *only* way to access an object’s data or invoke its procedures is to *send a message* to the object.
- This a form of data protection. Data can *only* be changed via a message send.
- Makes it easier to avoid or track down bugs.
- Also allows the internal details of data formats and procedures to be changed without affecting the rest of the program. This is a form of *information hiding*.

THE OBJECT - MESSAGE CONTRACT

- In FMS the programmer is free to violate this contract. Why? This is Forth.
- Methodless ivar access is available.
- Ivar data and methods are available.
- Use with care, if used at all.

OBJECTS AS IVARS

We can declare as many ivars as we want. We can freely mix ivar objects and ivars created with the BYTES primitive.

```
:class bar
  var x
  var2 y
  foo aFoo
  :m !: ( x y --) y !:  x ! ;m
  :m p: x @: .  y print2:  aFoo hi: ;m
;class

bar b \ instantiate an object
5 10 b !: \ send a message to b
b p: 5 10 Hello \ send a message to b
```

Early vs Late Binding to SELF

- Defining all messages sent to SELF as late bound, (i.e., always use [SELF] instead of SELF) *might* make it easier to reuse method definitions in subclasses. This could be a benefit, but not in all cases.
- The downsides are: 1) Slower program execution due to many more (and unnecessary) late binds, and

  2) Must be careful to avoid infinite loops when a subclass method uses a superclass method which references back to the subclass via late binding.

Early vs Late Binding to SELF

- It has been the author's experience that when an opportunity for code reuse via late binding to SELF occurs, it is a simple matter to just redefine the binding involved by changing SELF to [SELF] in the superclass method.

WHY MESSAGE NAMES ARE SPECIAL

- Sending a message is a "special" event and it should be clear when reading source code where a message send occurs. This is an opinion.
- In FMS we require that all message names end in colon (<name>:).
- This naming convention has been used with success in other Forths for decades.
- Some seem fixated on wanting to hide the use of objects thus making object code indistinguishable from Forth code not using objects.
- This a mistake, in the author’s opinion. For example, if we define "@" to be a message then we can have code that looks like the following (next slide):

WHY MESSAGE NAMES ARE SPECIAL

`foo @` \ a normal fetch from the VARIABLE foo, or the address presented by the execution of foo.

`bar @` \ a message send to the OBJECT bar, or the object presented by the execution of bar.

MESSAGE NAME CONFLICTS

- FMS message names have global scope.
- A message name conflict with another globally scoped word is handled as you would with any name conflict in Forth.

RECORDS

- Instance variables can be declared as part of a record.
- Header information is then removed from an object’s data.
- Use the REC{ … }REC notation.
- Essentially a form of a structures package for objects.

REFERENCE IMPLEMENTATIONS

- Two example implementations of FMS are provided for reference.
- Not the only ways to implement FMS.
- “Production” FMS code would be optimized.
- Reference implementations are loosely based on Andrew McKewan’s work (Object Oriented programming in ANS Forth, Forth Dimensions, March 1997).
- One uses linked lists to resolve messages to methods. Faster than McKewan (1997) because the n-way thread is determined at compile time. This is actually pretty fast. N could be increased.
- One uses dispatch tables (very fast).
- A small library is supplied in order to help illustrate the use of FMS.

LINKED LISTS

```
 ^obj SelectorID

(compute mfa thread offset at compile time)

  : FINDM  ( SelID MFA -- ^xt )
    BEGIN @ DUP
    WHILE 2DUP CELL+ @ =
         IF 2 CELLS + NIP EXIT THEN
    REPEAT ;

   FINDM @ EXECUTE-METHOD

  : EXECUTE-METHOD ( ^obj xt -- )
    [SELF] >R SWAP TO [SELF] EXECUTE
    R> TO [SELF] ;
```

DISPATCH TABLES

```
^obj SelectorID

 OVER CELL - @ + @ EXECUTE-METHOD

 : EXECUTE-METHOD ( ^obj xt -- )
   [SELF] >R SWAP TO [SELF] EXECUTE
   R> TO [SELF] ;
```

SUMMARY

- FMS retains what we like from Neon.
- FMS improves upon any real or perceived weaknesses in Neon.