FMS-SI Object and Class Structure

July 31 2016 Douglas B. Hoffman

Creating an Object

An object in FMS is a contiguous section of memory residing either in the dictionary or the heap. The address of that memory is itself often called the object or ^obj (pointer to object) or reference to the object. Since the address is held in one cell the object can be passed around on the data stack, return stack, variables, locals, or anywhere the programmer wishes. Objects can be named dictionary objects or nameless objects located in either the dictionary or the heap. Objects are created, or instantiated, using the class name in three ways as follows:

1) Named dictionary object. Execute the class name followed by the name you wish assigned to the object. This is a compile time operation:

   ```
   classname s
   ```

   An object of class `classname` and name `s` will be created in the dictionary. Subsequently executing `s` will leave the address of `s` on the stack.

   ```
   s ( – obj )
   ```

2) Nameless dictionary object. Execute the word `dict>` followed by the class name. The object will be created in the dictionary and its address left on the stack. This can be done at compile time or run time:

   ```
   dict> classname ( – obj )
   ```

3) Nameless heap object. Execute the word `heap>` followed by the class name. The object will be created in the heap and its address left on the stack. This can be done at compile time or run time:

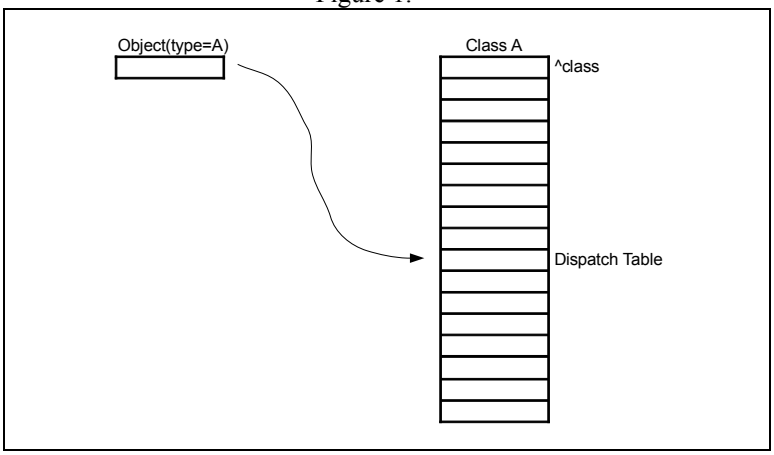
   ```
   heap> classname ( – obj )
   ```

There are some more details to be known about creating an object including the implicit initialization and creating indexed objects (objects with a built in array). These topics will be covered later.

The Structure of an Object

The simplest object has no data, (i.e., instance variables, also called ivars), and will consist of just one cell that contains the address of the dispatch table for the class representing the class of the object. Consider the following:

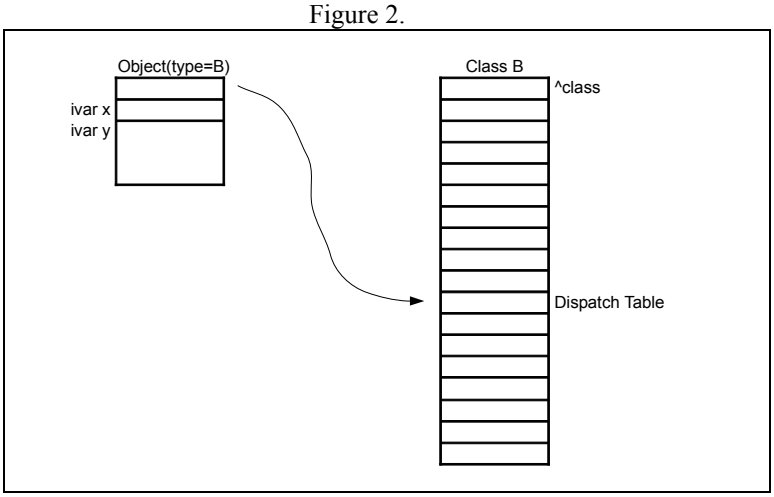
Figure 1.



Note that the number of cells shown pictorially for class A in Figure 1. do not represent the actual number of cells used for the class. The class structure is described in detail below. For now it is sufficient to know that the dispatch table is located inside the class definition.

If the object has instance variables, the more common situation, then they will be appended to the memory following the first cell of the object, as follows:

Figure 2.

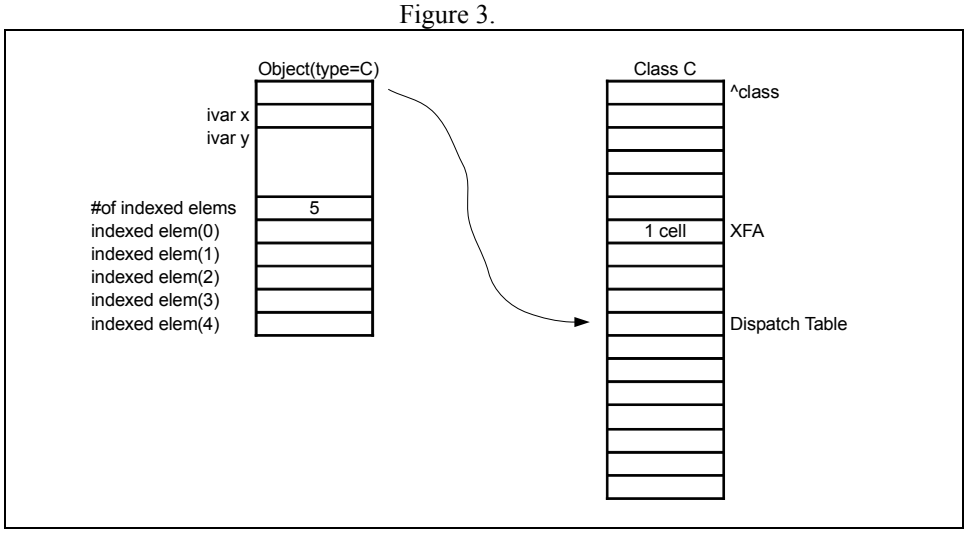


For illustration, consider when an object of class (or type) B has two ivars, x and y, and ivar x is one cell in size and ivar y is 3 cells. The overall memory required for this object is 5 cells. This does not include dictionary memory used for the name of the object, if there is a name, or a storage location, if any, to hold a reference to the object.

Objects With Indexed Data

Some classes of objects also contain an array of data, called the indexed area. The indexed area, which is just another form of instance variable, is always added to the end of the normal ivar area. An example of an indexed object is shown in Figure 3. where an object of class C is defined such that an array of cell-sized elements is appended to the normal ivar data. In this case the normal ivar data is the same as that for an object of class B.

Figure 3.

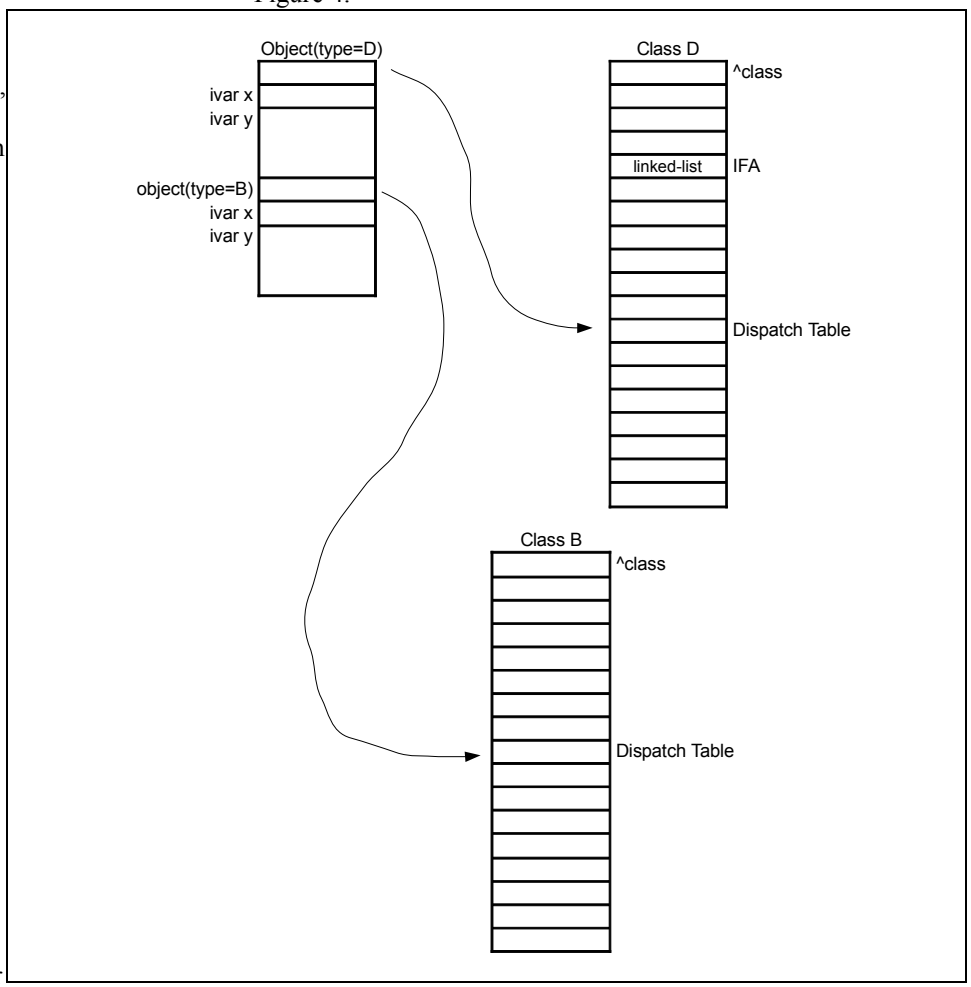


The width of each element in an indexed object is stored in the class of the object and is the same for all objects of that class. Class C maintains the width, in this case 1 cell, in the XFA field of the class. The #of indexed elements can vary for each object and so must be contained in the object (not the class). As can be seen in Figure 3. the #of indexed elements for this object is 5 and is contained in the first cell following the normal ivar data area. The #of indexed elements for an indexed object is declared at instantiation time for each object.

Embedded Objects As Instance Variables

The last form of instance variable is another object. The format of this ivar is identical to that of any object. Instance variable names belonging to the embedded object will not conflict with ivar names belonging to the owning object. But it must be understood that an embedded object is **not** the same as simply storing an object reference in a cell-sized instance variable (container object). There are important advantages to embedded objects vs container objects, although container objects have their own advantages and are fully supported in FMS.

Figure 4.



There is a field in every class, called IFA and shown in Figure 4., that provides the structure for all embedded objects-as-ivars used in a class of objects. The structure of each embedded object is contained in a node of a linked list.

Inherited Instance Variable Data

An object whose class is a subclass will inherit all instance variable data from its superclass. If additional ivars are defined for the subclass then they will be appended. Ivar names should not be re-used in a subclass. While there is no mechanism to prevent this and technically it could be done, doing so will "seal off" those ivars such that they cannot then be directly accessed (accessed by name) in subsequent methods and subclasses. An error check could be built in order to prevent this but it would cost more code so it is not.

A subclass of an indexed class will also be indexed and will have indexed elements of the same size. This happens automatically and cannot be changed. Each object can have only one indexed data area, with the exception that any object (with the proper class definition) can have any number of embedded or container indexed objects, with each having a unique element size and number of elements as desired. So in effect there can be any number of indexed areas in a single object. Of course there can also be dynamically size-able array objects as well.

Referring back to the first section Creating an Object, and figure 1. through 4., executing @ with the object on the stack will always return the address of the dispatch table for that object. Normally this is not done because the preferred use of objects is to send them messages.

Example Code for Defining Classes and Instance Variables

The following class definitions are provided to show exactly how they are created for each of the example classes shown so far. Note that no methods are defined. Object instantiation is shown as well.

```
:class Class-A   \ no instance variables
;class

Class-A x1  \ instantiate a Class-A object in the dictionary named x1


:class Class-B
  1 cells bytes x  \ declare an ivar of size 1 cell named x
  3 cells bytes y  \ declare an ivar of size 3 cells named y
;class

Class-B x2  \ instantiate a Class-B object in the dictionary named x2


:class Class-C   1 cells <indexed
  1 cells bytes x
  3 cells bytes y
;class

5 Class-C x3 \ object x3 will have 5 indexed elements of size 1 cell each


:class Class-D
  1 cells bytes x  \ declare an ivar of size 1 cell named x
  3 cells bytes y  \ declare an ivar of size 3 cells named y
  Class-B x2      \ Declare an embedded object of type Class-B named x2.
                  \ Note that the ivar name x2 does not conflict
                  \ with the public object name x2 created above.
;class

Class-D x4 \ instantiate a Class-D object in the  dictionary

0 value x5 \ a convenient place to store an object reference
heap> Class-D to x5  \ instantiate a Class-D object in the heap
( storing the object reference in a value is not a requirement )
x5 <free  \ FREE the memory allocated for the object
```

The Structure of a Class Definition in FMS-SI

While the size of a class definition can vary greatly depending on the specific settings used and the number and type of instance variable definitions and method definitions, the basic structure and fields of a generic class can be readily described. It should be understood that once defined, i.e., ended with `;class`, a class definition is sealed and cannot be changed by subsequent class definitions or anything else. A class definition resides entirely in the dictionary. So all class definitions will be saved when a dictionary image save is performed or a turnkey application is made.

Although there is an indirect way to add behaviors to an object after its class has been sealed. See the section “ADDING INSTANCE VARIABLES AND METHODS TO AN OBJECT AT RUN TIME” in the documentation file “About FMS-SI-f”.

Figure 5.

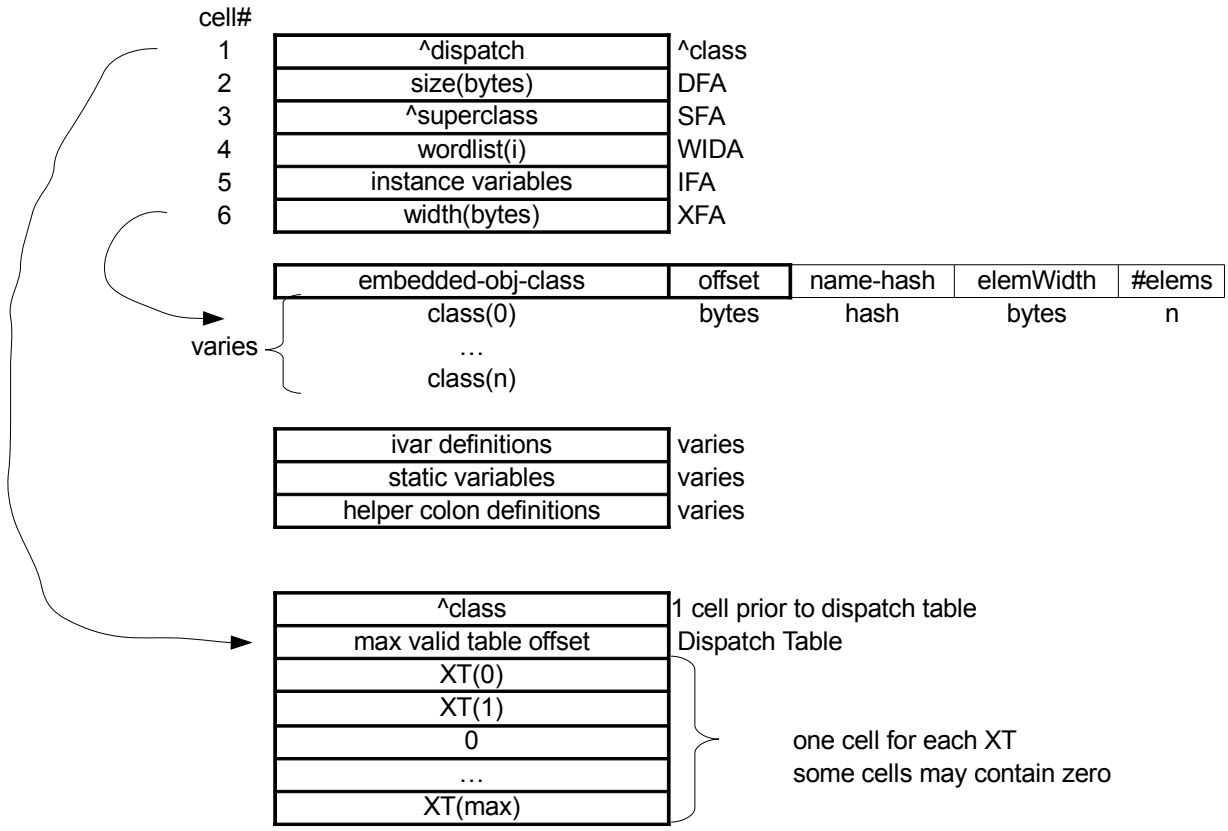


Figure 5. illustrates the generic layout of a class definition.

The first cell of a class definition, cell# 0 in the figure, can be located by the class pointer or ^class. The ^class can be obtained by ticking the class name and then using `>body` .

```
' <classname> >body => address = ^class
```

The cell at this address contains the address of the dispatch table (^dispatch).

The 2nd cell, called DFA, contains the total size in bytes of any object instantiated using the class. This size does not include the extra size required for indexed objects, if indexing applies, unless the indexed objects are embedded ivars.

The 3rd cell, called SFA, contains the ^superclass.

The 4th cell, called WIDA, contains the wordlist associated with this class.

The 5th cell, called IFA, contains the address of the information about the embedded object as instance variables for this class, if there are any.

The 6th cell, called XFA, contains the width in bytes of indexed elements. If this class or any of its superclasses, has not been declared as indexed then XFA will be 0.

After the XFA field come the various instance variable definitions which are defined using the wordlist contained in the WIDA field as the compilation wordlist. After the ivar definitions come the method definitions. The message name association to a method definition is performed simultaneously:

```
:m <message-name> … method definition … ;m
```

Finally, at the end of the class definition, after the message/method definitions, the method dispatch table is constructed. The dispatch table contains the XTs for methods corresponding to messages defined for the class. The first cell of the table contains the maximum valid table offset. This value is only used when `FMSCHECK?` is set to true and allows for error checking during development in case a message is sent to an object of class type where that message is not defined. The size of the table varies and will depend on several factors including the dispatch table structure of the superclass and the number of methods defined for that class. The size of each dispatch table is trimmed to the smallest size possible for that class. So if class foo has 100 methods (a dispatch table size of 100 cells) a subsequently defined class bar may have only 5 methods (a dispatch table size of 5 cells). It depends upon the number of methods/messages and the maximum table offset used for the messages in the class.

Messages require the address of the class dispatch table in order to resolve the message to its associated method. For this reason the first cell of every object, including embedded objects as instance variables, contains the address of the dispatch table.

The first cell prior to the dispatch table, in the class definition only, contains the `^class`. The object itself contains *only* the address of the dispatch table, from which the `^class` can be computed.

General Comments on Class Definition Size

The size also depends on the number of instance variables, the number of embedded objects as instance variables, the number of method definitions, the complexity of the method definitions, the number of helper colon definitions if any, and the number of class variables (also known as static variables) defined if any.

1) While some message name definitions (selectors) are created between `:CLASS` and `;CLASS` these definitions are global in scope and are not considered to belong to any particular class definition.
2) Any embedded objects as instance variable declarations will also create a new ivar name and additionally create a 3 or 4 cell node (includes the link address) added to the linked list at IFA. The #elems field at the node will only be created for indexed type objects.
3) Any instance variable definitions, static variable definitions, and helper colon definitions will belong to the class in which they are defined and will increase the size of the class definition.
4) The minimum dispatch table is defined in class `OBJECT` and consists of 4 cells: maximum valid table offset, XT for method `INIT:`, XT for method `FREE:`, and XT for method `HEAP:`. The number of messages defined will limit the maximum size of a dispatch table. But often the size of a dispatch table will be less than this maximum due to table trimming.
5) The dispatch tables are trimmed to just the maximum size required. So if there are 20 total selectors allowed, via the declarations mentioned above, but the maximum selectorID used in the class is the 10th selector then only 10+1 cells will be used for that table. This is called table trimming. The maximum of 20+1 cells would not be used in this case. It is still possible that some of those 10 XT cells could be empty (set to zero) depending on which of the first 10 messages are inherited or defined for that class. In general, less frequently used messages in a class should be declared after those that are used more frequently. This will have the effect of reducing dispatch table sizes.

Duck Typing

All methods in all versions of FMS use duck typing. Duck typing provides the necessary programming freedom required for best productivity and are a better fit for the Forth type-free way of doing things because there are no restrictions on method/message definitions and use.

The message resolution technique uses tables of XTs. The message simply provides an offset into the table. Each class has its own table. The code for method dispatch is as follows:

```
: ex-method ( obj xt – ) self >r swap to self execute r> to self ;
: sel-D create ... does> @ over @ + @ ex-method ;
```

A possible issue with a dispatch table type selector is that tables can become large when there are very many classes and many messages. But in practice the dictionary size penalty for empty table locations has proven to be small, even when there are many classes and messages defined.

Compiling with the `FMSCHECK?` constant set to true will result in "message not understood" error messages if no corresponding method has been defined for that class of object. There is no provision for a default action in the event of a not understood message (how could that be done?). The program will simply abort. With `FMSCHECK?` set to false (which is done after debugging is complete) the dispatch code will run slightly faster.

Instantiating Objects

A few things should be understood about instantiating FMS-SI objects.

1) Regardless of which memory is desired, the dictionary or the heap, the *same* class definition is used. The only difference is the way instantiation is performed. A named dictionary object is instantiated by executing a class name followed by the new object name.
2) A nameless object is instantiated, left on the stack, either in the dictionary or in the heap by executing `DICT>` or `HEAP>` respectively followed by the class name. `DICT>` and `HEAP>` may be used outside a definition.
3) The entire memory for an object is allotted or allocated with just *one* call to `ALLOT` or `ALLOCATE` (these calls to allot and allocate are performed implicitly by the object system). This holds true for embedded objects as well. Also, regardless of the number of embedded objects, only one call to `ALLOT` or `ALLOCATE` is (implicitly) made to reserve memory for the entire object.
4) When an object is instantiated the `INIT:` method is implicitly sent to the object by the FMS system. This is not true for embedded object-as-instance-variables which must have the `INIT:` message explicitly sent if it needs sending at all.