ForthOS Object Oriented Extensions

Introduction

Classic Forth organizes data and functions in a manner similar to C; for a given data structure, uniquely named functions ("words" in Forth) are invoked to operate on a given data structure. A class of programming languages known as *object oriented* instead define a smaller set of generic function names; the correct code for a specific type of data structure is called when the generic function is invoked on the specific data structure. ForthOS offers facilities for programming in this object oriented fashion.

Overview

An true object oriented programming generally must offer both polymorphism and inheritance. Polymorphism is the ability for more than one type of data structure to respond similarly to the same generic invocation (as described in the Introduction above). Inheritance is the ability to define a new data structure as an extension of an existing one, with the ability to *inherit* functionality from the parent data structure, overriding and augmenting as needed for the newly defined, derivative (inheriting) version of the data structure.

Object oriented systems often offer dynamic collection of unused memory ("garbage collection"); ForthOS, like C++, but unlike Smalltalk and Python, does not offer garbage collection.

Accessing an Existing Class

The ForthOS objected oriented system (hereafter, "OO") is all located in the "extensions" vocabulary, so the first step is to enable access to these words:

```
only   also extensions
```

A "class" is the name for the shape of a particular data structure, along with the functions it provides for manipulating that data structure. For the predefined classes supplied with ForthOS, these classes each have a name, which is registered in the "extensions" vocabulary. In this section, we will illustruate the OO facilities using a Set, which is a data structure which can hold an arbitrary collection of numbers; if a number is added more than once, it exists in the Set only once. There are methods for adding and removing members, as well as for testing for the presence of members.

First we will create a new Set for us to play with:

```
Set -> new   constant mySet
```

Now we can see what's in our new Set:

```
mySet -> .
```

As you can see, a new Set is empty. Let us add some elements, and then print its contents (actually, ask the data structure to list its own contents):

```
1 mySet -> add   123 mySet -> add   1 mySet -> add
```

Arguments are passed on the Forth operand stack just like they would be for any other word. The difference is that instead of "add" having a single unique set of actions, "-> add" causes the pointer on the top of the stack (which points to our Set instance--we access it via "mySet") to be used to look up its type (Set), and then find the code for "add"'ing to a Set, and then jumping into that Set-specific code.

Now we verify that the Set holds both 1 and 123 (but 1 only once, even though it was add'ed twice):

```
mySet -> .
```

As you might expect, it's easy enough to remove something from a Set:

```
1 mySet -> remove   mySet -> .
```

Adding Our Own Methods

Even though Set is provided with ForthOS, you can easily add methods to the Set class. Imagine we want "addRange", which takes numeric arguments like a do..loop, and puts all numbers from that range into the Set.

```
Set -> :method addRange ( high low self -- )
-rot do   i over -> add   loop   drop method;
```

First, notice that the word "Set" in the "extensions" dictionary is actually just a Forth constant which pushes a pointer to an object onto the stack. What object? The object which describes the Set class! So then when this object is told "-> :method", it tells the ForthOS OO system to define a new method named "addRange" for the Set data structure.

Next, notice that the actual code defined receives, in addition to the user provided arguments, a pointer to the Set instance on which "addRange" was invoked (our stack comment, in the tradition of Smalltalk, calls this "self"). If we invoke this new addRange on our Set instance mySet, "self" would be the same value as if we simply executed "mySet .".

The actual code body of a method shows that OO compilation is a seamless blend of traditional Forth and any needed OO invocations. We use "-rot do" to start a loop across the requested values, "i over" to get the current iteration value, along with a copy of the pointer for our Set instance. Then we use "-> add" to add the value of "i" to the contents of this Set.

Definitions are finished with "method;" rather than just ";". There is some unique cleanup needed when building an OO definition. If you forget, the compiler will flag the ";" as an error.

Creating a New Class

Imagine that we're completely happy with our Set class, except that we have some sets which hold an enormous number of elements, and we keep using "-> ." and being buried in the output. We still want *most* of our Set's to print out the list of members, but not these big ones--so we don't want to change Set's "." method.

One way would be to create a BigSet class. It's just like a Set, but it elides the actual list of members. First we define a new class, inheriting from Set:

```
Set -> subclass: BigSet
```

Now we redefine just the printing method:

```
BigSet -> :method . ( self -- )   drop   ." a BigSet{...}"
method;
```

You can use a BigSet just like a Set, because it uses the exact same data structure design as a Set, and offers all the same methods:

```
BigSet -> new   constant bset1
123 bset1 -> add   456 bset1 -> add   789 bset1 -> add
456 bset1 -> remove   bset1 -> .
```

The Fine Points of Inheritance

Following our BigSet example, we can now refine based on a deeper understanding of how the ForthOS system puts together its data structures. The "-> ." method is not actually defined in Set; it's defined in an outer, generic class (specifically, Collection). This definition prints the name of the class of the specific object, and then invokes "-> .elems" to ask the specific code for a given data structure to list its contents. So rather than define a "-> ." method for BigSet, we could much more elegantly define:

```
BigSet -> :method .elems ( self -- )
-> size 0> if   ." ..."   then   method;
```

This avoids a subtle bug in our simpler definition for "."; if we further subclassed BigSet as, say, BiggerSet, the "." method would continue to incorrectly describe the data structure as a BigSet. The generic method leveraged from Collection gets the correct name.

This definition also avoids printing ellipses when the BigSet is, in fact, empty. We could even have BigSet list its members until the size gets beyond 20 members:

```
20 constant ellideSize
BigSet -> :method .elems ( self -- )
-> size ellideSize < if   super-> .elems   else   ." ..."
then   method;
```

This version uses a new construct, "super->". Rather than invoke the method defined for this particular object, it invokes

the method which would have been used if our current object was actually an instance of our parent class. So in this case, the parent of BigSet is Set, and "super-> .elems" will execute the ".elems" code for a Set.

For BigSet's which have not grown large, we still see their contents (empty BitSet's are also displayed correctly by Set). Once the size reaches ellideSize, we instead display ellipses.

Some Notes on Debugging

ForthOS provides some amenities in marrying the OO methods with the regular Forth world. For the method "." of the class "BigSet", the Forth name "BigSet:." is inserted in the "oo" dictionary. ("oo" itself located within "extensions"). This means you can set breakpoints with "debug oo.BigSet:.", and then stack backtraces will give you a symbolic interpretation of program counter values within OO code.