FoxRockX

Speed in Object Creation and Destruction

Does the approach you choose for creating and destroying objects have an impact on performance?

Tamar E. Granor, Ph.D.

March 2016
Number 49

A client asked me to speed up part of an application. In doing so, I was forced to re-evaluate one of my personal best practices: using NewObject() rather than CreateObject() to instantiate objects.

One of my goals is to let each line of code I write stand alone as much as possible. So, for example, I always include the IN clause to specify the work area for any command that accepts it. Similarly, I prefer to use NewObject() to create objects rather than CreateObject(), because the latter assumes that the right class library has already been identified via SET CLASSLIB or SET PROCEDURE.

But a few months ago, a client asked me to see if I could speed up a key process in their application. Data from this application is stored as XML and, at the user's request, read into tables, and then converted into a complex object hierarchy. For a small dataset, the hierarchy might contain 1,500 objects; for a large one, perhaps 30,000.

Users were finding the process of converting from XML to DBFs to objects painfully slow. In the process of speeding it up, I compared the performance of CreateObject() and NewObject(). I found that CreateObject() was faster by about a third, so changed the application to use CreateObject for almost all object instantiation.

When I presented these results at Southwest Fox 2015 as part of a session on optimization, some interesting questions were raised about the speed of both instantiating and destroying objects. So I decided to dig deeper.

Quick review

Both CreateObject() and NewObject() can be used to instantiate classes from either VCX-based class libraries or PRG-based code classes. The primary difference between them is that NewObject() accepts a parameter to specify the class library (either VCX or PRG). CreateObject() expects to find the specified class in either a VCX or a PRG that has already been specified via SET CLASSLIB or SET PROCEDURE. Listing 1 shows a simple example of each.

Listing 1. With CreateObject(), the class definition must already be visible, while NewObject() lets you specify the class library on the fly.

```
LOCAL o

* With CREATEOBJECT(), make sure to
* SET CLASSLIB first
SET CLASSLIB TO MyClasses
O = CREATEOBJECT("cusMyClass")

* With NEWOBJECT(), you can specify the
* class library directly.
O = NEWOBJECT("cusMyClass", "MyClasses")
```

Given the additional work it must do to find the right class library, it's not surprising that CreateObject() is somewhat faster than NewObject(). What I wanted to know was how much faster and under what circumstances. Because the same application seemed very slow when closing, I was also interested in the speed of destroying objects, and whether some ways of doing so might be faster than others.

Test methodology

As I mentioned in my last article, doing timing tests in Windows is tricky. For these tests, I did my best to make sure not to have side effects that would interfere with accuracy.

I ran each test on two different machines, closing any applications I could that might be doing things in the background. On my notebook, I ran one set of tests across my network (that is, using class libraries stored on a different machine) and one set locally. For the local tests on the notebook, I disconnected from the network.

Most importantly for these tests, after each individual test, I shut down VFP and restarted it to avoid caching effects.

My test code is included in this month's downloads as ObjectCDTests.PRG. (The classes used for testing are in ClassDefs.VCX, also included in this month's downloads.) The program accepts three parameters: a code for which test to run, a one-character code to identify the machine it's running on, and a flag to indicate whether it's running locally or across the network. Listing 2 shows a sample call.

Listing 2. This line calls my program for testing object creation and destruction speed. This call runs test 3c on the laptop across the network.

```
DO ObjectCDTests.PRG WITH "3c", "L", .T.
```

The program is a giant CASE statement with a separate case for each test. Tests of the same thing use the same number, and are then lettered to distinguish them. For example, tests 1a through 1f test different ways of instantiating an object, while tests 2a through 2e test whether different ways of storing object references have an impact on instantiation speed. All told, there are nine sets of tests. Listing 3 shows one case. All the cases use the same basic structure; in particular, they all set up nPasses, nStart and nEnd and use a DO WHILE loop to manage the timing test.

Listing 3. Each test is one case in a large CASE statement.

```
CASE m.cTest = "1a"
  * CreateObject() assuming SET CLASSLIB
  SET CLASSLIB TO ClassDefs.VCX
  nPasses = 0
  nStart = SECONDS()
  nEnd = CalcEndTime(m.nStart, TESTTIME)

  DO WHILE m.nEnd >= SECONDS()
    nPasses = nPasses + 1
    oObject = CREATEOBJECT("cusMinimal")
  ENDDO

  SET CLASSLIB TO
```

CreateObject() vs. NewObject

The first set of tests compare different ways of instantiating an object. Altogether, I tested six approaches to instantiating multiple copies of a single class:

1a) SET CLASSLIB once, then instantiate multiple instances with CreateObject() (Listing 3);

1b) Issue SET CLASSLIB right before each instantiation with CreateObject() (Listing 4);

1c) SET CLASSLIB once, then instantiate multiple instances with NewObject();

1d) Do not SET CLASSLIB, instantiate multiple instances with NewObject();

1e) SET CLASSLIB once, then check whether classlib is already set before each instantiation with CreateObject() (Listing 5);

1f) Do not SET CLASSLIB, then check whether classlib is already set before each instantiation with CreateObject().

Listing 4. This case tests CreateObject() with an explicit SET CLASSLIB for each instantiation.

```
CASE m.cTest = "1b"
  * CreateObject() with explicit SET CLASSLIB
  nPasses = 0
  nStart = SECONDS()
  nEnd = CalcEndTime(m.nStart, TESTTIME)

  DO WHILE m.nEnd >= SECONDS()
    nPasses = nPasses + 1
    SET CLASSLIB TO ClassDefs.VCX
    oObject = CREATEOBJECT("cusMinimal")
    SET CLASSLIB TO
  ENDDO
```

Listing 5. This code checks whether the appropriate class library has been set and SETs it, if not. In this test, it's already set.

```
CASE m.cTest = "1e"
  * CreateObject() checking
  * SET CLASSLIB first
  SET CLASSLIB TO ClassDefs.VCX
  nPasses = 0
  nStart = SECONDS()
  nEnd = CalcEndTime(m.nStart, TESTTIME)
```

```
DO WHILE m.nEnd >= SECONDS()
  nPasses = nPasses + 1
  IF NOT ("CLASSDEFS." $ SET("Classlib"))
    SET CLASSLIB TO ClassDefs.VCX
  ENDIF
  oObject = CREATEOBJECT("cusMinimal")
ENDDO

SET CLASSLIB TO
```

In these tests, the instantiated object is stored to the same variable on each pass, meaning that we don't hold on to the object between passes (which, of course, means that each object except the last is destroyed inside the test loop).

The results of this group of tests make it obvious that the slow part is opening the VCX. The first version (1a), with a single SET CLASSLIB, is far faster than any of the others. The only one even close is 1e, and it still completes only about half as many passes in the same time. Version 1b, that issues SET CLASSLIB before each call to CREATEOBJECT(), is two orders of magnitude slower in the local case and three orders of magnitude slower in the networked case. With my computers, Test 1a averages over 300,000 instantiations per second with everything local, and around 250,000 instantiations per second across the network. Test 1b averages over 7,000 instantiations per second in the local case, and 330 instantiations per second in the networked case.

The two versions that use NewObject(), 1c and 1d, get results very similar to case 1b. As with 1b, running them locally is about an order of magnitude faster than running them across the network.

The cause of the slowness in cases 1b, 1c, and 1d became apparent when I looked at the results for 1e and 1f, the two cases that first check SET("CLASSLIB") to see whether the VCX is already available. In 1e, where it is, my machines were able to instantiate over 170,000 objects per second. 1f, where the VCX is not already available and thus SET CLASSLIB is needed, completed about half as many instantiations as 1b, 1c and 1d in the local case; in the networked case, surprisingly, it was the same order of magnitude as 1b, 1c, and 1d, but it actually managed about 25% more passes.

In other words, what makes NewObject() slower is finding and opening the VCX. When the file has to be found and opened before each instantiation, things slow down considerably.

The one surprise in this group is 1c, where the specified class library is already available. I'd expect VFP to notice that the class library is already open and not bother to open it. To understand why this case is also slow, I tried a very different test. I created a class with code in Init and Destroy to show the current value of SET("CLASSLIB"), as in Listing 6.

Listing 6. This code in Init and Destroy helps us see how VFP handles access to class libraries with NewObject().

```
DEBUGOUT PROGRAM(), "Before DODEFAULT()", ;
         SET("classlib")
DODEFAULT()
DEBUGOUT PROGRAM(), "After DODEFAULT()", ;
         SET("classlib")
```

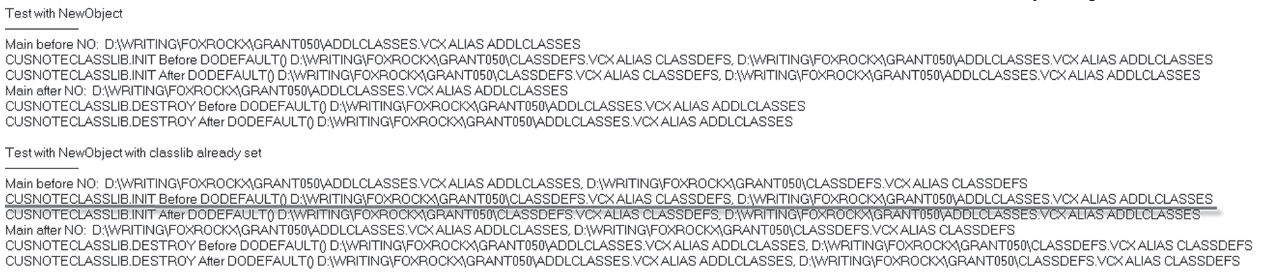
I wrote a program (shown in Listing 7 and included in this month's downloads as TestSetClassLib.PRG) to instantiate an object using NewObject(), first with the appropriate class library not included in SET CLASSLIB, and then with it included.

Listing 7. This code lets us see how NewObject() modifies SET("CLASSLIB").

```
LOCAL o

DEBUGOUT "Test with NewObject"
DEBUGOUT "-------------------"

SET CLASSLIB TO AddlClasses
DEBUGOUT "Main before NO: ", SET("Classlib")
o=NEWOBJECT("cusnoteclasslib", "ClassDefs")
DEBUGOUT "Main after NO: ", SET("Classlib")
o=.null.
SET CLASSLIB TO
DEBUGOUT " "

DEBUGOUT "Test with NewObject with " + ;
         "classlib already set"
DEBUGOUT "-------------------"

SET CLASSLIB TO AddlClasses, ClassDefs
DEBUGOUT "Main before NO: ", SET("Classlib")
o=NEWOBJECT("cusnoteclasslib", "ClassDefs")
DEBUGOUT "Main after NO: ", SET("Classlib")
o=.null.
SET CLASSLIB TO
```

Figure 1 shows the output from the program and explains why case 1c is slow. In order to ensure that the right class is instantiated, VFP rearranges the order of the class libraries (which, presumably, requires reopening one or more). After the object is instantiated, the original order of the list is restored.

The lesson here is that if you can SET CLASSLIB and have some assurance that it won't change, CREATEOBJECT() is the way to go. Next best is to



```
Test with NewObject
───────────
Main before NO:  D:\WRITING\FOXROCKX\GRANT050\ADDLCLASSES.VCX ALIAS ADDLCLASSES
CUSNOTECLASSLIB.INIT Before DODEFAULT() D:\WRITING\FOXROCKX\GRANT050\CLASSDEFS.VCX ALIAS CLASSDEFS, D:\WRITING\FOXROCKX\GRANT050\ADDLCLASSES.VCX ALIAS ADDLCLASSES
CUSNOTECLASSLIB.INIT After DODEFAULT() D:\WRITING\FOXROCKX\GRANT050\CLASSDEFS.VCX ALIAS CLASSDEFS, D:\WRITING\FOXROCKX\GRANT050\ADDLCLASSES.VCX ALIAS ADDLCLASSES
Main after NO:  D:\WRITING\FOXROCKX\GRANT050\ADDLCLASSES.VCX ALIAS ADDLCLASSES
CUSNOTECLASSLIB.DESTROY Before DODEFAULT() D:\WRITING\FOXROCKX\GRANT050\ADDLCLASSES.VCX ALIAS ADDLCLASSES
CUSNOTECLASSLIB.DESTROY After DODEFAULT() D:\WRITING\FOXROCKX\GRANT050\ADDLCLASSES.VCX ALIAS ADDLCLASSES

Test with NewObject with classlib already set
───────────
Main before NO:  D:\WRITING\FOXROCKX\GRANT050\ADDLCLASSES.VCX ALIAS ADDLCLASSES, D:\WRITING\FOXROCKX\GRANT050\CLASSDEFS.VCX ALIAS CLASSDEFS
CUSNOTECLASSLIB.INIT Before DODEFAULT() D:\WRITING\FOXROCKX\GRANT050\CLASSDEFS.VCX ALIAS CLASSDEFS, D:\WRITING\FOXROCKX\GRANT050\ADDLCLASSES.VCX ALIAS ADDLCLASSES
CUSNOTECLASSLIB.INIT After DODEFAULT() D:\WRITING\FOXROCKX\GRANT050\CLASSDEFS.VCX ALIAS CLASSDEFS, D:\WRITING\FOXROCKX\GRANT050\ADDLCLASSES.VCX ALIAS ADDLCLASSES
Main after NO:  D:\WRITING\FOXROCKX\GRANT050\ADDLCLASSES.VCX ALIAS ADDLCLASSES, D:\WRITING\FOXROCKX\GRANT050\CLASSDEFS.VCX ALIAS CLASSDEFS
CUSNOTECLASSLIB.DESTROY Before DODEFAULT() D:\WRITING\FOXROCKX\GRANT050\ADDLCLASSES.VCX ALIAS ADDLCLASSES, D:\WRITING\FOXROCKX\GRANT050\CLASSDEFS.VCX ALIAS CLASSDEFS
CUSNOTECLASSLIB.DESTROY After DODEFAULT() D:\WRITING\FOXROCKX\GRANT050\ADDLCLASSES.VCX ALIAS ADDLCLASSES, D:\WRITING\FOXROCKX\GRANT050\CLASSDEFS.VCX ALIAS CLASSDEFS
```

Figure 1. The underlined line, showing SET("CLASSLIB") during instantiation, shows that the necessary class library has been moved to be first in the list of class libraries.

check for the library and SET CLASSLIB, if necessary. In real applications, it's likely that you'll only have to SET CLASSLIB the first time you use a particular library, and after that, it'll be available.

All that said, it's worth noting that even the slowest cases here were capable of instantiating my simple object hundreds of times per second. So, in many cases, it doesn't matter whether you use CreateObject() or NewObject(). My client's application, which involves instantiating thousands of objects while the user waits, is an exception.

Where to store objects

The next question I considered was whether it mattered how you store the object references, once you've instantiated the objects. I tested storing to a single variable (thus releasing each object when the next was instantiated), storing each reference to a separate variable, storing all references to an array, storing all references to a collection that already existed, and storing them to a new collection. I tested these five cases with both CreateObject() (tests 2a-2e) and NewObject() (tests 3a-3e). Listing 8 shows three CreateObject() cases: storing to multiple variables, storing to an array, and storing to an existing collection.

Listing 8. Testing whether it matters where you put the variable references.

```
CASE m.cTest = "2b"
  * CreateObject to multiple vars
  SET CLASSLIB TO ClassDefs.VCX
  nPasses = 0

  * Declare the necessary variables
  FOR nCount = 1 TO ObjectCount
    cVarName = "oObject" + ;
               ALLTRIM(TRANSFORM(m.nCount))
    LOCAL (cVarName)
  ENDFOR

  nStart = SECONDS()
  nEnd = CalcEndTime(m.nStart, TESTTIME)

  DO WHILE m.nEnd >= SECONDS()
    nPasses = nPasses + 1
    FOR nCount = 1 TO OBJECTCOUNT
      STORE CREATEOBJECT("cusMinimal") TO ;
       ("oObject" + ;
        ALLTRIM(TRANSFORM(m.nCount)))
    ENDFOR
  ENDDO

  SET CLASSLIB TO

CASE m.cTest = "2c"
  * CreateObject to an array
  SET CLASSLIB TO ClassDefs.VCX
  DIMENSION aObjects[OBJECTCOUNT]
  nPasses = 0
  nStart = SECONDS()
  nEnd = CalcEndTime(m.nStart, TESTTIME)

  DO WHILE m.nEnd >= SECONDS()
    nPasses = nPasses + 1
    FOR nCount = 1 TO OBJECTCOUNT
      aObjects[m.nCount] = ;
        CREATEOBJECT("cusMinimal")
    ENDFOR
  ENDDO

  SET CLASSLIB TO

CASE m.cTest = "2d"
  * CreateObject to an existing collection
  SET CLASSLIB TO ClassDefs.VCX
  oObjectCollection = ;
    CREATEOBJECT("Collection")

  nPasses = 0
  nStart = SECONDS()
  nEnd = CalcEndTime(m.nStart, TESTTIME)

  DO WHILE m.nEnd >= SECONDS()
    nPasses = nPasses + 1
    FOR nCount = 1 TO OBJECTCOUNT
      oObjectCollection.Add( ;
        CREATEOBJECT("cusMinimal"))
    ENDFOR

    oObjectCollection.Remove(-1)
  ENDDO

  SET CLASSLIB TO
```

These tests didn't show much new information. The CreateObject() code was faster than the NewObject() code operating locally by an order of magnitude; similarly, the NewObject() version run locally was an order of magnitude faster than NewObject() across the network.

With CreateObject(), the variation from fastest to slowest was less than an order of magnitude. Not surprisingly, storing to a single variable was fastest. (One weakness in this code is the need to use a name expression in the multiple variables case rather than hard-coding the appropriate variable name; that may slow down this test.) The slowest CreateObject() version was 2e, which creates a collection inside the loop and then adds each object to the collection.

The network penalty with CreateObject() wasn't too big, varying from about 6% up to about 30%.

With NewObject(), the single variable version was again fastest. But the difference between cases was small. The real differences here were local vs. network and between the two machines. As with the first group of tests, when using NewObject() and a locally stored VCX, my desktop machine completed an order of magnitude more passes than my notebook. (That's actually surprising, given that the notebook has a solid-state drive, and the desktop does not.)

Once again, tests using a local VCX were an order of magnitude faster than those using a VCX stored elsewhere on the network.

The bottom line here is that where you store object references doesn't matter very much. Use whichever works best for you.

Instantiation from multiple classes

My next set of tests was designed to see whether instantiating objects from more than one class changed the timing equation. The previous tests worked with cusMinimal, a subclass of Custom with no added PEMs and no changes to the default. I added a second class to ClassDefs.VCX; cusSmall is also subclassed from Custom, but has four custom properties added.

I then repeated the first three groups of tests (except for 1e and 1f), with each pass instantiating one object from cusMinimal and one from cusSmall. For example, Listing 9 shows test 4d, which measures performance of NewObject() without a corresponding SET CLASSLIB.

Listing 9. Test groups 4, 5 and 6 look at performance when instantiating from multiple classes in the same class library.

```
CASE m.cTest = "4d"
  * NewObject() without SET CLASSLIB
  nPasses = 0
  nStart = SECONDS()
  nEnd = CalcEndTime(m.nStart, TESTTIME)

  DO WHILE m.nEnd >= SECONDS()
    nPasses = nPasses + 1
    oObject = NEWOBJECT("cusMinimal", ;
              "ClassDefs.VCX")
    oObjectT2 = NEWOBJECT("cusSmall", ;
               "ClassDefs.VCX")
  ENDDO
```

On the whole, these tests give similar results to the earlier tests. Because two objects are being instantiated on each pass rather than one, they complete about half as many passes in the same time as the earlier tests, though some actually do better than that.

Object destruction

I also compared various ways of destroying objects I'd instantiated. Many of the earlier tests included implicit destruction of objects, but I wanted to test in a more controlled way.

For each group of destruction tests, I held the method of instantiation and the storage method constant, and varied the ways of destroying the objects. The first group (tests 7a to 7c) involved objects stored in distinct individual variables. I tested destroying them with RELEASE ALL (actually, RELEASE ALL LIKE since I couldn't release all variables used for the test), with individual RELEASE commands and by setting each to .null. Listing 10 shows the first of these tests, using RELEASE ALL LIKE.

Listing 10. The first group of object destruction tests looks at objects stored in individual variables. Here, they're all destroyed by releasing all the variables.

```
CASE m.cTest = "7a"
  * Destroy via RELEASE ALL. Can't test unless
  * we use RELEASE ALL LIKE.
  SET CLASSLIB TO ClassDefs.VCX
  nPasses = 0

  * Declare the necessary variables
  nStart = SECONDS()
  nEnd = CalcEndTime(m.nStart, TESTTIME)

  DO WHILE m.nEnd >= SECONDS()
    nPasses = nPasses + 1

    FOR nCount = 1 TO ObjectCount
      cVarName = "oObject" + ;
                 ALLTRIM(TRANSFORM(m.nCount))
      LOCAL (cVarName)
    ENDFOR

    FOR nCount = 1 TO OBJECTCOUNT
      STORE CREATEOBJECT("cusMinimal") TO ;
        ("oObject" + ;
         ALLTRIM(TRANSFORM(m.nCount)))
    ENDFOR

    * Now release
    RELEASE ALL LIKE oObject*
  ENDDO

  SET CLASSLIB TO
```

I found no significant difference between the three approaches. The differences between the two machines and between different runs on the same machine were as large as the differences between the various approaches.

The next set of tests (8a through 8c) looked at objects stored in a collection. Again, I tried three ways of destroying them: by releasing the collection variable; by calling the collection's Remove method, passing -1 as a parameter, so all were removed at once; and by setting the collection variable to .null. Note that in each of these cases, you end up with something slightly different. In the first case, not only are the objects gone, but so is the collection. In the second case, you're left with an empty collection. In the third case, the variable that held the collection still exists, but no longer holds a collection. Listing 11 shows test 8b, a call to the collection's Remove method.

Listing 11. Here, we remove all objects from the collection, causing them to be destroyed.

```
CASE m.cTest = "8b"
  * Objects in collection, REMOVE
  SET CLASSLIB TO ClassDefs.VCX

  nPasses = 0
  nStart = SECONDS()
  nEnd = CalcEndTime(m.nStart, TESTTIME)

  DO WHILE m.nEnd >= SECONDS()
    LOCAL oObjectCollection
    oObjectCollection = ;
      CREATEOBJECT("Collection")
    nPasses = nPasses + 1
    FOR nCount = 1 TO OBJECTCOUNT
      oObjectCollection.Add( ;
        CREATEOBJECT("cusMinimal"))
    ENDFOR

    oObjectCollection.Remove(-1)
  ENDDO

  SET CLASSLIB TO
```

As with the prior group, the differences between the two machines, between the local and the networked cases, and between different runs of the same test were larger than any differences between the different approaches.

The final group of tests compared two approaches to destroying members of a collection one by one. Both tests use the collection's Remove method. Test 9a loops backward, removing the specified item, while test 9b loops forward, always removing the first item. **Listing 12** shows test 9a, looping backward. (Removing a specified item requires looping backward because the size of the collection changes with each item removed.)

Listing 12. In removing collection items, there's no significant performance difference between looping backward removing the nth item on each pass, and looping forward, always removing the first item.

```
CASE m.cTest = "9a"
  * Objects in collection, RELEASE
  SET CLASSLIB TO ClassDefs.VCX

  nPasses = 0
  nStart = SECONDS()
  nEnd = CalcEndTime(m.nStart, TESTTIME)

  DO WHILE m.nEnd >= SECONDS()
    LOCAL oObjectCollection
    oObjectCollection = ;
      CREATEOBJECT("Collection")
    nPasses = nPasses + 1
    FOR nCount = 1 TO OBJECTCOUNT
      oObjectCollection.Add( ;
        CREATEOBJECT("cusMinimal"))
    ENDFOR

    FOR nCount = OBJECTCOUNT TO 1 STEP -1
      oObjectCollection.Remove(m.nCount)
    ENDFOR
  ENDDO

  SET CLASSLIB TO
```

Again, there was no significant difference between the approaches.

What about PRGs?

I wondered whether the same rules would apply to classes defined with code in PRG files, so I created corresponding PRG-based classes and modified the tests to use those with SET PROCEDURE. The class definitions are included in this month's downloads as ClassDefs.PRG, while the tests are in ObjectCDTestsPrg.PRG

I found that the same basic rules applied, in terms of which approaches were faster. What was a surprise to me was that in most cases, using VCX-based classes was faster. Specifically, in about 10% of my test runs (where a test run is the combination of a particular case, a particular machine and a particular setting of across the network or not), the PRG-based tests completed more passes. In those cases, the difference was mostly within 10%. A few tests were within 1%, close enough to be considered equivalent.

In the remaining roughly 87% of the tests, the VCX-based version completed more passes. These tests averaged about 85% faster, but ranged from completing a little more than 1% more passes up to completing about four times (400%) as many passes.

All of the tests where PRG was faster than VCX were run on my laptop and all but one were conducted across the network. That suggests to me that the real difference here might have to do with the fact that the PRG is 140 bytes, while the combination of the VCX and VCT is more than 4 KB.

The bottom line

The most important lesson here, I think, is that object creation and destruction is fast. Except when looking for class libraries across a network, VFP can create or destroy thousands, tens of thousands, even hundreds of thousands of objects per second.

The second big takeaway is that SETting CLASSLIB once and using CreateObject() is an order of magnitude faster than using NewObject(). Finding a class library is actually the slow part; when doing so across a network, you lose another order of magnitude.

That instantiating a class from a VCX is faster than doing so from a PRG is worth more investigation. One case to test is the difference between having the class defined in the same PRG that uses it vs. having it in a separate PRG.

Another open question is how all this works when the class has lots of properties and custom code. These tests looked only at small classes.

Finally, the slowness on closing my client's application turned out to have nothing to do with object destruction. About the same time I was performing these tests, I found that the code to save the data was doing a lot of extra work; a couple of minor tweaks to ensure that only changed data is resaved sped things up considerably in most cases.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. Tamar is author or co-author of a dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with Visual FoxPro *and* Taming Visual FoxPro's SQL. *Her latest collaboration is* VFPX: Open Source Treasure for the VFP Developer, *available at www.foxrockx.com. Her other books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar was a Microsoft Support Most Valuable Professional from the program's inception in 1993 until 2011. She is one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.*