October, 1995

ASK ADVISOR

Visual FoxPro 3.0

Q: An object's properties are inherited from its parent class until the property value of the instantiated object is changed. How can the property be reverted to a default so that the value will once again be inherited?

–Michael Thomas (via CompuServe)

A: One of the great features of object-oriented programming is that once you've specified a property value or defined a method for a class, every subclass you base on that class inherits the property value or method. Similarly, every instance of the class you create has the parent class' property values and method code.

In a subclass or instance, you can *override* the default value by specifying your own. For example, you might create a form based on a form class you've designed for a particular application. For this one form, you decide the background should be purple, so you set backcolor to RGB(64,0,128)

When you're working with Visual FoxPro's design tools, though, sometimes, you change a property value, then later realize you didn't really want to change it. For example, after more thought, you decide maybe a purple background isn't such a great idea for that form and want to restore it to use the same backcolor as all the other forms in the app, the default backcolor for the form class.

Removing your overriding definition doesn't solve the problem. (If you just highlight the "64,0,128" in BackColor and delete it, it gets replaced with "0,0,0", which gives you black.) You have to explicitly tell Visual FoxPro to restore the default value. This is another case where right-click comes to the rescue. Click on the property in question in the Property Sheet, then right-click. The menu that appears has "Reset to Default" as the first option. Choose it and the job is done.

Resetting methods to the class' default is simple, too. Just open up the code editor for the specified method, highlight all the code and delete it. When a method of an object contains no code, the code from the class is used. Similarly, when a subclass contains no code for a particular method, objects of that subclass use the parentclass' method code.

The techniques above work at design-time, when you're specifying a class or object. At run-time, the problem changes. There is no way in Visual FoxPro to restore a property value to the default at run-time. You can set it to the default value, but it's still considered overridden and doesn't inherit from its defining class. Generally, this isn't much of a problem since the value is only inherited at the time the object is created. Once you start changing a property value at runtime, that initial value is lost, anyway.

Methods, however, have a built-in mechanism for accessing the defining class' code. You can call the class' method explicitly by putting <classname>::<method name> in the method in question.

In fact, you're not restricted to simply calling an overridden method. You can call any method of any visible class this way. You can also call the same method more than once.

This structure gives you incredible flexibility. Say you have some custom code you need to put in a method, but you still want to execute the class' code for that method. You can choose whether your code goes before or after the class' code. For example, if you're in the Click method of a sub-class of the class MyButton, you could do this:

```
* here's some custom code you want executed
* more custom code

* now call the MyButton code
MyButton::Click
```

or you could do it this way:

```
* first call the MyButton code
MyButton::Click

* now here's the custom code
* more custom code
```

If you want, you can even call the class' code more than once, like this:

```
* call the MyButton code
MyButton::Click

* now some custom code
* and more custom code

* now call the MyButton code again
MyButton::Click
```

You can make such calls more flexible by referring to the class indirectly rather than by name. Here's an approach that works no matter what method of what object you're in:

```
myParentMethod = This.ParentClass) + "::" + ;
                 SUBSTR(PROGRAM(),RAT(".",PROGRAM())+1)
&myParentMethod
```

–Tamar

FoxPro 2.x and Visual FoxPro 3.0

Q: How can I store a command in a memo field in a table and call it up later. What I am trying to do is create a table of the reports needed in my project and store the indexes and filters in a memo field. I am having trouble creating the syntax to retrieve the stored commands. Can you help?

–Joe Leale (via CompuServe)

A: It's not clear from your question whether you're storing just the index and filter expressions in the memo field or the entire command. You can do it either way, but storing the entire command has uses beyond reporting, so we'll look at that approach.

The key to executing a stored command is the macro operator, &. When FoxPro finds an &, it takes the variable that follows, evaluates it, then executes the variable's contents. You can use a macro to replace an entire command or just a part of one. (For more on macros and where you need them, see Robert Gryphon's article in the August issue.)

There's one important rule about macros. They only work on variables - you can't use them with fields.

Say you've stored the report command you need in a field called mRepCmd. To run the report takes two steps. First, we copy the memo field to a variable, then we execute the variable contents. The code looks like:

```
cRepCmd=mRepCmd
&cRepCmd
```

This approach does the trick if all you need to store is a single command. If you need to store a block of code, another method is called for. We'll store the whole program in the memo field. When we need to execute it, we can copy it to a file, then DO the file.

This technique works if the user has a copy of FoxPro. But, a distributed application running with the FoxPro libraries (whether a stand-alone or compact executable) can't compile programs. In this case, though, you can store FXPs (compiled PRGs) in the memo field and then apply this technique.

This time, assume you've stored the program to execute in a field called mProgram. Here's the code to copy the program out to a file, execute the program, then clean up:

```
* get a new file name - make sure it's unique
cTempFile = SYS(3)
DO WHILE FILE(cTempFile+".PRG")
   cTempFile = SYS(3)
ENDDO

* assume we're pointing to the right record
COPY MEMO mProgram TO (cTempFile+".PRG")

* now execute it
DO (cTempFile)

* now clean up
ERASE (cTempFile+".PRG")
```

This technique is handy when you need to generate some code at run-time, then execute it.

–Tamar