CHAPTER 10

Objects In-Depth

CHAPTER 5 ALREADY WENT into a great deal of depth on inheritance and object-oriented programming in Visual Basic .NET. The focus in that chapter was primarily on the concepts behind object-oriented programming and inheritance, as well as the syntax necessary to explain those concepts. In this chapter, we'll cover additional issues relating to classes and objects. Those issues covered in Chapter 5 will be given rather short shrift here for obvious reasons.

The Parts of a .NET Application

In Chapter 9, you learned about scoping within a function. Scoping at the class, module, assembly, and application domain-levels is considerably more complicated. Before you can understand scoping in .NET, it is necessary to look much more closely at the way .NET applications are put together.

You've read a little bit about assemblies in previous chapters. And you've probably read about assemblies and application domains in the .NET documentation. Chances are pretty good that you're still confused. My goal in the next few pages is not only to bring together all of the bits of information on the subject scattered throughout the documentation but to do so in a reasonable facsimile of plain English.

Application Domains

In the VB6 world (and programming before .NET in general), you dealt with two types of executables: EXE files and DLL files. They had the following characteristics:

An EXE file...

- runs in a separate process.
- is isolated from all other processes (no shared memory).
- is allowed to execute based on system security settings (NT/2000/XP only).
• has one main thread and may create others.

• may (optionally) expose objects via OLE (ActiveX EXE).

• represents a loading boundary (you can't load part of an EXE into memory).

• is debugged independently of any other process.

When we use the term “application” or “program,” we are usually referring to EXE files.

A DLL file...

• is loaded at runtime by a client process.

• shares memory space with other DLL's loaded by the process and the process itself.

• is allowed to load based on system security settings (NT/2000/XP only).

• shares the main thread of the process and may create new threads.¹

• may (optionally) expose objects via OLE (ActiveX DLL).

• represents a loading boundary (you can't load part of a DLL into memory).

• cannot be debugged independently of the calling process.

While .NET does represent a major change to developers, it is still based on Windows and uses EXE and DLL files, which work exactly as they always have. Why, then, introduce the confusion of assemblies and application domains? In other words, if EXE and DLL files continue to work as they always have and .NET applications continue to be made up of EXE and DLL files, why confuse the issue? Like many of the changes in .NET, this is a consequence of a feature that at first glance may seem completely unrelated and one you already know.

You know that the CLR manages memory very tightly. You know that pointers are not allowed with managed memory. The existence of pointers would make it impossible to accurately determine if an object or variable is in use (which is necessary for garbage collection to work). Pointers also allow errors to occur where code may corrupt application memory or even that of the .NET runtime. In fact, one of the key goals behind managed memory is to prevent memory leaks and memory corruption.

¹ VB6 DLL's cannot create their own threads without using a third-party component.