**Processes, AppDomains, and Object Contexts**

In Chapters 14 and 15, you examined the steps taken by the CLR to resolve the location of a referenced external assembly, as well as the role of .NET metadata. In this chapter, you’ll drill deeper into the details of how an assembly is hosted by the CLR and come to understand the relationship between processes, application domains, and object contexts.

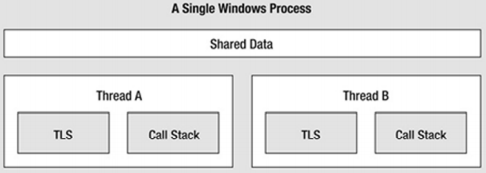
In a nutshell, application domains (or simply AppDomains) are logical subdivisions within a given process that host a set of related .NET assemblies. As you will see, an AppDomain is further subdivided into contextual boundaries, which are used to group like-minded .NET objects. Using the notion of context, the CLR is able to ensure that objects with special runtime requirements are handled appropriately.

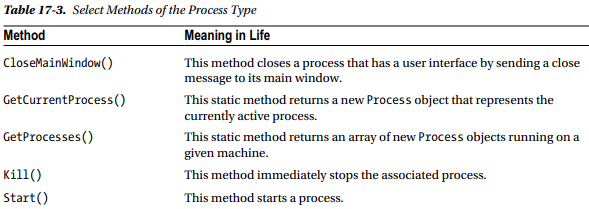
While it is true that many of your day-to-day programming tasks might not involve directly working with processes, AppDomains, or object contexts, understanding these topics is important when working with numerous .NET APIs, including Windows Communication Foundation (WCF), multithreading and parallel processing, and object serialization.

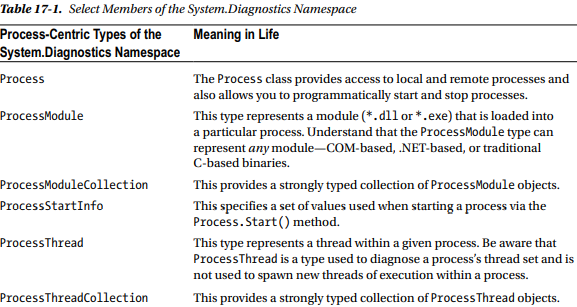
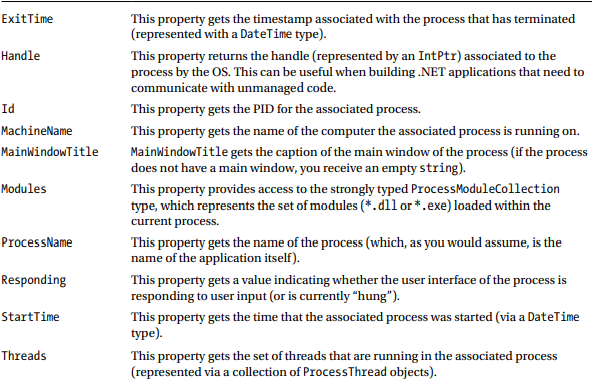
**The Role of a Windows Process The concept of a “process”**

existed within Windows-based operating systems well before the release of the .NET platform. In simple terms, a process is a running program. However, formally speaking, a process is an operating system–level concept used to describe a set of resources (such as external code libraries and the primary thread) and the necessary memory allocations used by a running application. For each \*.exe loaded into memory, the OS creates a separate and isolated process for use during its lifetime. Using this approach to application isolation, the result is a much more robust and stable runtime environment, given that the failure of one process does not affect the functioning of another. Furthermore, data in one process cannot be directly accessed by another process, unless you make use of a distributed computing programming API such as Windows Communication Foundation. Given these points, you can regard the process as a fixed, safe boundary for a running application.

The Role of Threads Every Windows process contains an initial “thread” that functions as the entry point for the application. Chapter 19 examines the details of building multithreaded applications under the .NET platform; however, to facilitate the topics presented here, you need a few working definitions. First, a thread is a path of execution within a process. Formally speaking, the first thread created by a process’s entry point is termed the primary thread. Any .NET executable program (Console Application, Windows service, WPF application, etc.) marks its entry point with the Main() method. When this method is invoked, the primary thread is created automatically.

**Interacting with Processes** **Under the .NET Platform** Although processes and threads are nothing new, the manner in which you interact with these primitives under the .NET platform has changed quite a bit (for the better). To pave the way to understanding the world of building multithreaded assemblies (see Chapter 19), let’s begin by checking out how to interact with processes using the .NET base class libraries.



**Investigating a Specific Process** In addition to obtaining a complete list of all running processes on a given machine, the static Process.GetProcessById() method allows you to obtain a single Process object via the associated PID. I

