**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.

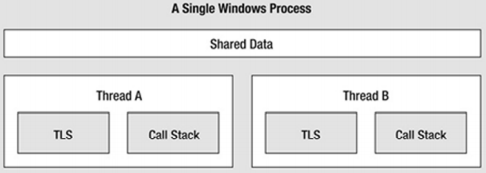
In a nutshell, application domains (or simply AppDomains) are logical subdivisions within a given process that host a set of related .NET assemblies. As you will see, an AppDomain is further subdivided into contextual boundaries, which are used to group like-minded .NET objects. Using the notion of context,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

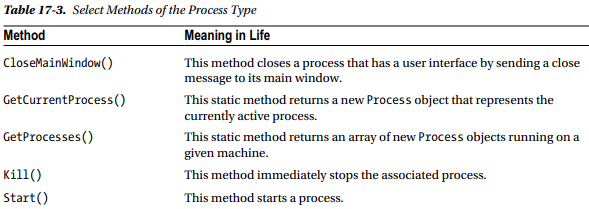
While it is true that many of your day-to-day programming tasks might not involve directly working with processes, AppDomains, or object contexts, understanding these topics is important when working with numerous .NET APIs, including Windows Communication Foundation (WCF), multithreading and parallel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

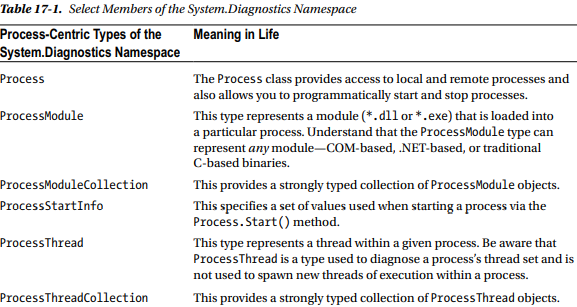
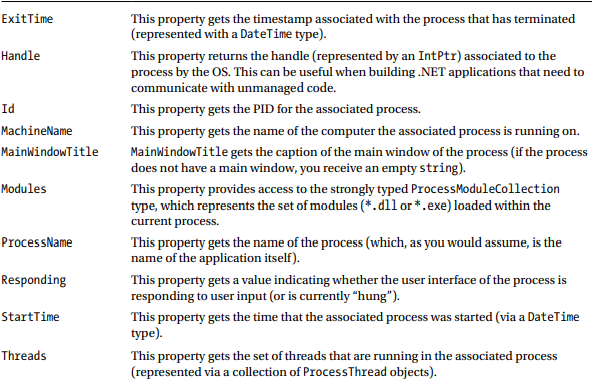
**The Role of a Windows Process The concept of a “process”**

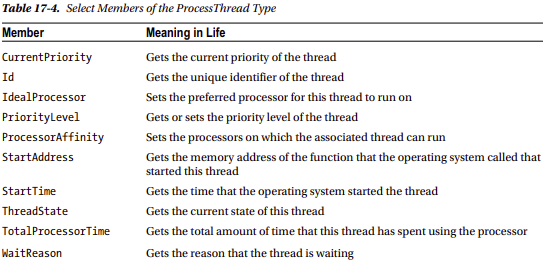
existed within Windows-based operating systems well before the release of the .NET platform. In simple terms, a process is a running program. However, formally speaking, a process is an operating system–level concept used to describe a set of resources (such as external code libraries and the primary 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.

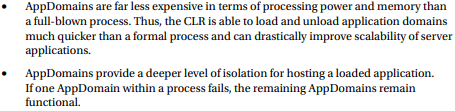
The Role of Threads Every Windows process contains an initial “thread” that functions as the entry point for the application. Chapter 19 examines the details of building multithreaded applications under the .NET platform; however, to facilitate the topics presented here, you need a few working definitions. 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.



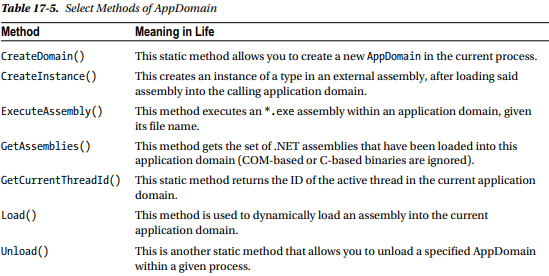
**Investigating a Specific Process** In addition to obtaining a complete list of all running processes on a given machine, the static Process.GetProcessById() method allows you to obtain a single Process object via the associated PID. I

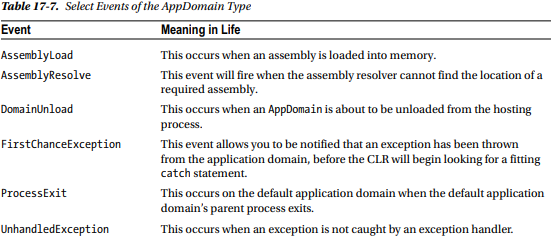


**Understanding .NET Application Domains** Under the .NET platform, executables are not hosted directly within a Windows process, as is the case in traditional unmanaged applications. Rather, a .NET executable is hosted by a logical partition within a process termed an application domain. As you will see, a single process may contain multiple application domains, each of which is hosting a .NET executable. This additional subdivision of a traditional Windows process offers several benefits, some of which are as follows:

As mentioned, a single process can host any number of AppDomains, each of which is fully and completely isolated from other AppDomains within this process (or any other process). Given this fact, be aware that an application running in one AppDomain is unable to obtain data of any kind (global variables or static fields) within another AppDomain, unless they use a distributed programming protocol (such as Windows Communication Foundation).

**The System.AppDomain Class** The .NET platform allows you to programmatically monitor AppDomains, create new AppDomains (or unload them) at runtime, load assemblies into AppDomains, and perform a whole slew of additional tasks, using the AppDomain class in the System namespace of mscorlib.dll.

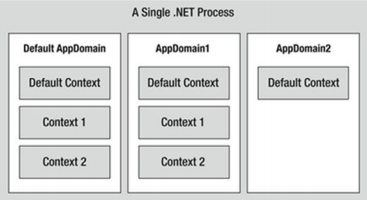




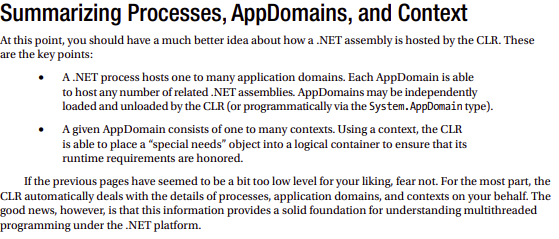
**Interacting with the Default Application Domain** Recall that when a .NET executable starts, the CLR will automatically place it into the default AppDomain of the hosting process. This is done automatically and transparently, and you never have to author any specific code to do so. However, it is possible for your application to gain access to this default application domain using the static AppDomain.CurrentDomain property. After you have this access point, you are able to hook into any events of interest or use the methods and properties of AppDomain to perform some runtime diagnostics.

**Receiving Assembly Load Notifications** If you want to be informed by the CLR when a new assembly has been loaded into a given application domain, you may handle the AssemblyLoad event. This event is typed against the AssemblyLoadEventHandler delegate, which can point to any method taking a System.Object as the first parameter and an AssemblyLoadEventArgs as the second.

**Loading Assemblies into Custom Application Domains** The CLR will always load assemblies into the default application domain when required. However, if you do ever manually create new AppDomains, you can load assemblies into said AppDomain using the AppDomain.Load() method. Also, be aware that the AppDomain.ExecuteAssembly() method can be called to load an \*.exe assembly and execute the Main() method.

**Programmatically Unloading AppDomains** It is important to point out that the CLR does not permit unloading individual .NET assemblies. However, using the AppDomain.Unload() method, you are able to selectively unload a given application domain from its hosting process. When you do so, the application domain will unload each assembly in turn.

**Understanding Object Context Boundaries** As you have just seen, AppDomains are logical partitions within a process used to host .NET assemblies. On a related note, a given application domain may be further subdivided into numerous context boundaries. In a nutshell, a .NET context provides a way for a single AppDomain to establish a “specific home” for a given object.

**Context-Agile and Context-Bound Types** .NET objects that do not demand any special contextual treatment are termed context-agile objects. These objects can be accessed from anywhere within the hosting AppDomain without interfering with the object’s runtime requirements.

**Summary** The point of this chapter was to examine exactly how a .NET-executable image is hosted by the .NET platform. As you have seen, the long-standing notion of a Windows process has been altered under the hood to accommodate the needs of the CLR. A single process (which can be programmatically manipulated via the System.Diagnostics.Process type) is now composed of one or more application domains, which represent isolated and independent boundaries within a process. As you have seen, a single process can host multiple application domains, each of which is capable of hosting and executing any number of related assemblies. Furthermore, a single application domain can contain any number of contextual boundaries. Using this additional level of type isolation, the CLR can ensure that special-need objects are handled correctly.