C# and the .NET Platform, Second Edition

ANDREW TROELSEN

Apress™

C# and the .NET Platform, Second Edition
Copyright ©2003 by Andrew Troelsen

ISBN : 1-59059-055-4

Printed and bound in the United States of America 12345678910

Technical Reviewers: Gregory A. Beamer, Gary Cornell, Eric Gunnerson, Joe Nalewabau, Kent Sharkey, Nick Symmonds, Pradeep Tapadiya

Editorial Directors: Dan Appleman, Gary Cornell, Simon Hayes, Martin Streicher, Karen Watterson, John Zukowski

Assistant Publisher: Grace Wong

Copy Editors: Anne Friedman and Ami Knox

Proofreader: Liz Berry

Production Goddess: Susan Glinert Stevens

Indexer: Ron Strauss

Artist and Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

Contents at a Glance

Introduction

I REMEMBER A TIME years ago when I proposed a book to Apress regarding a forthcoming software SDK named Next Generation Windows Services (NGWS). As you may already know, NGWS eventually became what we now know as the .NET platform. My research of the C# programming language and the .NET platform took place in parallel with the authoring of the text. It was a fantastic project; however, I must confess that it was more than a bit nerve-wracking writing about a technology that was undergoing drastic changes over the course of its development. It pains me to recall how many chapters had to be completely destroyed and rewritten during that time. Thankfully, after many sleepless nights, the first edition of *C# and the .NET Platform* was published in conjunction with the release of .NET Beta 2, circa the summer of 2001.

Since that point, I have been extremely happy and grateful to see that the first edition of this text was very well received by the press and, most importantly, the readers. Over the years, it was nominated as a Jolt award finalist (I lost . . . crap!) as well as the 2003 Referenceware programming book of the year (I won . . . cool!). Although the first edition of this book has enjoyed a good run, it became clear that a second edition was in order—not only to account for the changes brought about with the minor release of the .NET platform, but to expand upon and improve the existing content. As I write this front-matter, version 1.1 of the .NET platform is just about official, and I am happy to say that *C# and the .NET Platform, Second Edition* is being released in tandem.

As in the first edition, this second edition presents the C# programming language and .NET base class libraries using a friendly and approachable tone. I have never understood the need some technical authors have to spit out prose that reads more like a GRE vocabulary study guide than a readable discourse. As well, this new edition remains focused on providing you with the information you need to build software solutions today, rather than spending too much time focusing on esoteric details that few individuals will ever actually care about. To this end, when I do dive under the hood and check out some more low-level functionality of the CLR (or blocks of CIL code), I promise it will prove enlightening (rather than simple eye candy).

We're a Team, You and I

Technology authors write for a demanding group of people (I should know, I'm one of them). You know that building software solutions using any platform is extremely detailed and is very specific to your department, company, client base, and subject matter. Perhaps you work in the electronic publishing industry, develop systems for the state or local government, work at NASA or a branch of the military. Speaking for myself, I have developed children's educational software, various n-tier systems, as well as numerous projects within the medical and financial world. The chances are almost 100 percent that the code you write at your place of employment has little to do with the code I write at mine (unless we happened to work together previously!).

Therefore, in this book, I have deliberately chosen to avoid creating examples that tie the example code to a specific industry or vein of programming. Rather, I choose to explain C#, OOP, the CLR, and the .NET base class libraries using industry-agnostic examples. Rather than having every blessed example fill a grid with data, calculate payroll, or whatnot, I'll stick to subject matter we can all relate to: automobiles (with some geometric structures and employees thrown in for good measure). And that's where you come in.

My job is to explain the C# programming language and the core aspects of the .NET platform the best I possibly can. As well, I will do everything I can to equip you with the tools and strategies you need to continue your studies at this book's conclusion. Your job is to take this information and apply it to your specific programming assignments.

I obviously understand that your projects most likely don't revolve around automobiles with pet names; however, that's what applied knowledge is all about! Rest assured, once you understand the concepts presented within this text, you will be in a perfect position to build .NET solutions that map to your own unique programming environment.

An Overview of the Second Edition

C# and the .NET Platform, Second Edition is logically divided into five distinct sections, each of which contains some number of chapters that somehow "belong together." If you read the first edition of this text, you will notice some similarities in chapter names; however, be aware that just about every page has been updated with new content. You will also notice that some topics in the first edition (such as .NET delegates) have been moved into an entire chapter of their very own. Of course, as you would hope, the second edition contains several brand new chapters (such as an exploration of .NET Remoting, and a much deeper examination of ASP.NET).

On the flip side, I did choose to *remove* some topics from the second edition to make room for new content. The most notable omission is the topic of COM and .NET interoperability, which in no way, shape, or form reflects the importance of this topic. In fact, I felt this topic was so important, that I wrote an entire book on the subject. If you require a detailed examination, check out *COM and .NET Interoperability* (Apress, 2002).

These things being said, here is a chapter-by-chapter breakdown of the text.

Part One: Introducing C# and the .NET Platform

Chapter 1: The Philosophy of .NET

This first chapter functions as the backbone for the remainder of this text. We begin by examining the world of traditional Windows development and uncover the shortcomings with the previous state of affairs. The primary goal of this chapter, however, is to acquaint you with a number of .NET-centric building blocks such as the common language runtime (CLR), Common Type System (CTS), Common Language Specification (CLS), and the base class libraries. You also take an initial look at the C# programming language, the role of the .NET assembly, and various development utilities that ship with the .NET SDK.

Chapter 2: Building C# Applications

The goal of this chapter is to introduce you to the process of compiling and debugging C# source code files using various approaches. First, you learn to make use of the command-line compiler (csc.exe) and examine each of the corresponding command-line flags. Over the remainder of the chapter, you learn how to make use of the Visual Studio .NET IDE, navigate the official .NET help system (MSDN), and understand the role of XML-based source code comments.

Part Two: The C# Programming Language

Chapter 3: C# Language Fundamentals

This chapter examines the core constructs of the C# programming language. Here you come to understand basic class construction techniques, the distinction between value types and reference types, iteration and decision constructs, boxing and unboxing, and the role of everybody's favorite base class, System.Object. Also, Chapter 3 illustrates how the .NET platform places a spin on various commonplace programming constructs such as enumerations, arrays, and string processing.

Chapter 4: Object-Oriented Programming with C#

The role of Chapter 4 is to examine the details of how C# accounts for each "pillar" of object-oriented programming: encapsulation, inheritance, and polymorphism. In addition to examining the syntax used to build class hierarchies, you are exposed to various tools within Visual Studio .NET which may be used to decrease your typing time.

Chapter 5: Exceptions and Object Lifetime

Here you learn how to handle runtime anomalies using the official error handling mechanism of the .NET platform: structured exception handling. As you will see, exceptions are class types that contain information regarding the error at hand and can be manipulated using the "try", "catch", "throw", and "finally" keywords of C#. The latter half of this chapter examines how the CLR manages the memory consumed by allocated objects using an associated garbage collector. This discussion also examines the role of the IDisposable interface, which is a perfect lead-in to the next chapter.

Chapter 6: Interfaces and Collections

This material builds upon your understanding of object-based development by checking out the topic of interface-based programming. Here you learn how to define types that support multiple behaviors, how to discover these behaviors at runtime, and how to selectively hide select behaviors using *explicit interface implementation*. To showcase the usefulness of interface types, the remainder of this chapter examines the System.Collections namespace. As you will see, this region of the base class libraries contains numerous types that may be used out of the box, or serve as a foundation for the development of strongly typed collections.

Chapter 7: Callback Interfaces, Delegates, and Events

This chapter begins by examining how interface-based programming techniques can be used to build an event-based system. This will function as a point of contrast to the meat of Chapter 7: the delegate type. Simply put, a .NET delegate is an object that “points” to other methods in your application. Using this pattern, you are able to build systems that allow multiple objects to engage in a two-way conversation. After you examine the use of .NET delegates, you are then introduced to the C# “event” keyword, which is used to simplify the manipulation of raw delegate programming.

Chapter 8: Advanced C# Type Construction Techniques

The final chapter of this section completes your study of the C# programming language by introducing you to a number of advanced programming techniques. For example, here you learn how to overload operators and create custom conversion routines (both implicit and explicit), as well how to manipulate C-style pointers within a *.cs code file. This chapter also takes the time to explain how these C#-centric programming constructs can be accessed by other .NET programming languages (such as Visual Basic .NET), which is a natural lead-in to the topic of *.NET assemblies*.

Part Three: Programming with .NET Assemblies

Chapter 9: Understanding .NET Assemblies

From a very high level, an assembly can be considered the term used to describe a managed *.dll or *.exe file. However, the true story of .NET assemblies is far richer than that. Here you learn the distinction between single-file and multifile assemblies and how to build and deploy each entity. Next, this chapter examines how private and shared assemblies may be configured using XML-based *.config files and publisher policy *.dlls. Along the way, you investigate the internal structure of the Global Assembly Cache (GAC) and learn how to force Visual Studio .NET to display your custom assemblies within the Add Reference dialog box (trust me, this is one of the most common questions I am asked).

Chapter 10: Processes, AppDomains, Contexts, and Threads

Now that you have a solid understanding of assemblies, this chapter dives much deeper into the composition of a loaded .NET executable. The goal of Chapter 10 is to define several terms and illustrate the relationship between processes, application domains, contextual boundaries, and threads. Once these terms have been qualified, the remainder of this chapter is devoted to the topic of building multithread applications using the types of the System.Threading namespace. Be aware that the information presented here provides a solid foundation for understanding the .NET Remoting layer (examined in Chapter 12).

Chapter 11: Type Reflection, Late Binding, and Attribute-Based Programming

Chapter 11 concludes our examination of .NET assemblies by checking out the process of runtime type discovery via the System.Reflection namespace. Using these types, you are able to build applications that can read an assembly's metadata on the fly (think object browsers). Next, you learn how to dynamically activate and manipulate types at runtime using *late binding*. The final topic of this chapter explores the role of .NET attributes (both standard and custom). To illustrate the usefulness of each of these topics, the chapter concludes with the construction of an extendable Windows Forms application.

Part Four: Leveraging the .NET Libraries

Chapter 12: Object Serialization and the .NET Remoting Layer

Contrary to popular belief, XML Web services are not the only way to build distributed applications under the .NET platform. Here you learn about the managed equivalent of the (now legacy) DCOM architecture: .NET Remoting. Unlike DCOM, .NET supports the ability to *easily* pass objects between application and machine boundaries using marshal-by-value (MBV) and marshal-by-reference (MBR) semantics. Also, the runtime behavior of a distributed .NET application can be altered without the need to recompile the client and server code bases using XML configuration files.

Chapter 13: Building a Better Window (Introducing Windows Forms)

Despite the term *.NET,* the base class libraries provide numerous namespaces used to build traditional GUI-based desktop applications. Here you begin your examination of the System.Windows.Forms namespace and learn the details of building main windows (as well as MDI applications) that support menu systems, toolbars, and status bars. As you would hope, various aspects of the Visual Studio .NET IDE are examined over the flow of this material.

Chapter 14: A Better Painting Framework (GDI+)

This chapter examines how to dynamically render graphical data in the Windows Forms environment. In addition to learning how to manipulate fonts, colors, geometric images, and image files, you also examine *hit testing* and GUI-based drag-and-drop techniques. You learn about the new .NET resource format, which, as you may suspect by this point in the text, is based on XML data representation. By way of a friendly heads up, don't pass over this chapter if you are primarily concerned with ASP.NET. As you will see later in Chapter 18, GDI+ can be used to dynamically generate graphical data on the Web server.

Chapter 15: Programming with Windows Forms Controls

This final Windows-centric chapter examines numerous GUI widgets that ship with the .NET Framework. Not only do you learn how to program against the core Windows Forms controls, but you also learn about the related topics of dialog box development and Form inheritance, and how to build *custom* Windows Forms controls. If you have a background in ActiveX control development, you will be pleased to find that the process of building a custom GUI widget has been greatly simplified (especially with regard to design time support).

Chapter 16: The System.IO Namespace

As you can gather from its name, the System.IO namespace allows you to interact with a machine's file and directory structure. Over the course of this chapter, you learn how to programmatically create (and destroy) a directory system as well as move data into and out of various streams (file based, string based, memory based, and so forth). In addition, this chapter illustrates some more exotic uses of System.IO, such as monitoring a set of files for modification using the FileSystemWatcher type. We wrap up by building a complete Windows Forms application that illustrates the relationship between object serialization (described in Chapter 12) and file I/O operations.

Chapter 17: Data Access with ADO.NET

ADO.NET is an entirely new data access API that has practically nothing to do with classic (COM-based) ADO. Here you learn about the fundamental shift away from Universal Data Access (UDA) to a namespace-based data access mentality. As you will see, you are able to interact with the types of ADO.NET using a "connected" and "disconnected" layer. Over the course of this chapter, you have the chance to work with both modes of ADO.NET, and come to understand the role of data readers, DataSets, and DataAdapters. The chapter concludes with coverage of various data-centric wizards of Visual Studio .NET.

Part Five: Web Applications and XML Web Services

Chapter 18: ASP.NET Web Pages and Web Controls

This chapter begins your study of Web technologies supported under the .NET platform. ASP.NET is a completely new approach for building Web applications and has absolutely nothing to do with classic (COM-based) ASP. For example, server-side scripting code has been replaced with "real" object-oriented languages (such as C#, VB.NET, managed C++ and the like). This chapter introduces you to key ASP.NET topics such as working with (or without) code behind files, the role of ASP.NET Web controls (including the mighty DataGrid), validation controls, and interacting with the base class libraries from *.aspx files.

Chapter 19: ASP.NET Web Applications

This chapter extends your current understanding of ASP.NET by examining various ways to handle state management under .NET. Like classic ASP, ASP.NET allows you to easily create cookies, as well as application-level and session-level variables. However, ASP.NET also introduces a new state management technique: the application cache. Once you examine the numerous ways to handle state with ASP.NET, you then learn the role of the System.HttpApplication base class (lurking within the Global.asax file) and how to dynamically alter the runtime behavior of your Web application using the web.config file.

Chapter 20: XML Web Services

In this final chapter of this book, you examine the role of .NET XML Web services. Simply put, a *Web service* is an assembly that is activated using standard HTTP requests. The beauty of this approach is the fact that HTTP is the one wire protocol that is almost universal in its acceptance and is, therefore, an excellent choice for building platform- and language-neutral distributed systems. You also check out numerous surrounding technologies (WSDL, SOAP, and UDDI) which enable a Web service and external client to communicate in harmony.

Obtaining This Book's Source Code

All of the code examples contained within this book (minus small code snippets here and there) are available for free and immediate download from the Apress Web site. Simply navigate to `http://www.apress.com` and look up this title by name. Once you are on the homepage for *C# and the .NET Platform, Second Edition,* you may download a self-extracting .zip file. After you unzip the contents, you will find that the code has been logically divided by chapter. Do be aware that the following icon:

SOURCE CODE

is your cue that the example under discussion may be loaded into Visual Studio .NET for further examination and modification. To do so, simply open the *.sln file found in the correct subdirectory.

NOTE All of the source code for this book as been compiled using Visual Studio .NET 2003. Sadly, *.sln files created with VS .NET 2003 cannot be open using VS .NET 2002. If you are still currently running Visual Studio .NET 2002, my advice is to simply create the appropriate project workspace, delete the auto-generated C# files, and copy the supplied *.cs files into the project using the Project | Add Existing Item menu selection.

Obtaining Updates for This Book

As you read over this text, you may find an occasional grammatical or code error (although I sure hope not). If this is the case, my apologies. Being human, I am sure that a glitch or two may be present, despite my best efforts. If this is the case, you can obtain the current errata list from the Apress Web site (located once again on the "homepage" for this book) as well as information on how to notify me of any errors you might find.

Contacting Me

If you have any questions regarding this book's source code, are in need of clarification for a given example, or simply wish to offer your thoughts regarding the .NET platform, feel free to drop me a line at the following e-mail address (to ensure your messages don't end up in my junk mail folder, please include "C# SE" in the title somewhere!): `atroelsen@intertech-inc.com`.

Please understand that I will do my best to get back to you in a timely fashion; however, like yourself, I get busy from time to time. If I don't respond within a week or two, do know I am not trying to be a jerk or don't care to talk to you. I'm just busy (or if I'm lucky, on vacation somewhere).

So then! Thanks for buying this text (or at least looking at it in the bookstore, trying to decide if you will buy it). I hope you enjoy reading this book and put your newfound knowledge to good use.

Take care,

Andrew Troelsen
Minneapolis, MN

CHAPTER 10

Processes, AppDomains, Contexts, and Threads

IN THE PREVIOUS CHAPTER, you examined the steps taken by the CLR to resolve the location of an externally referenced assembly. Here, you drill deeper into the constitution of a .NET executable host and come to understand the relationship between Win32 processes, application domains, contexts, and threads. In a nutshell, *application domains* (or simply, AppDomains) are logical subdivisions within a given process, which host a set of related .NET assemblies. As you will see, an application domain is further subdivided into contextual boundaries, which are used to group together like-minded .NET objects. Using the notion of context, the CLR is able to ensure that objects with special needs are handled appropriately.

Once you have come to understand the relationship between processes, application domains, and contexts, the remainder of this chapter examines how the .NET platform allows you to manually spawn multiple threads of execution for use by your program within its application domain. Using the types within the System.Threading namespace, the task of creating additional threads of execution has become extremely simple (if not downright trivial). Of course, the complexity of multithreaded development is not in the creation of threads, but in ensuring that your code base is well equipped to handle concurrent access to shared resources. Given this, the chapter closes by examining various synchronization primitives that the .NET Framework provides (which you will see is somewhat richer than raw Win32 threading primitives).

Reviewing Processes and Threads Under Traditional Win32

The concept of processes and threads has existed within Windows-based operating systems well before the release of the .NET platform. Simply put, *process* is the term used to describe the set of resources (such as external code libraries and the primary thread) as well as the necessary memory allocations used by a running application. For each *.exe loaded into memory, the operating system creates a separate and isolated memory partition (aka 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 effect the functioning of another.

Now, every Win32 process is assigned a unique process identifier (PID), and may be independently loaded and unloaded by the operating system as necessary (as well as programmatically using Win32 API calls). As you may be aware, the Processes tab of the

Task Manager utility (activated via the Ctrl+Shift+Esc keystroke combination) allows you to view statistics regarding the set of processes running on a given machine, including its PID and image name (Figure 10-1). (If you do not see a PID column, select the View | Select Columns menu and check the PID box.)

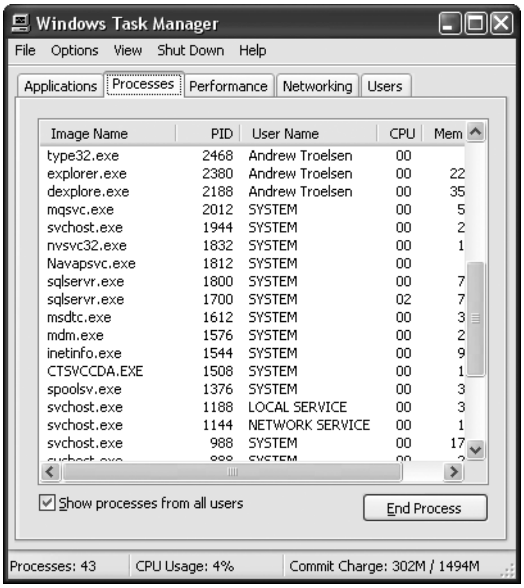

Windows Task Manager
File Options View Shut Down Help
Applications Processes Performance Networking Users
Show processes from all users
End Process

Image Name	PID	User Name	CPU	Mem
type32.exe	2468	Andrew Troelsen	00	
explorer.exe	2380	Andrew Troelsen	00	22
dexplore.exe	2188	Andrew Troelsen	00	35
mqsvc.exe	2012	SYSTEM	00	5
svchost.exe	1944	SYSTEM	00	2
nvsvc32.exe	1832	SYSTEM	00	1
Navapsvc.exe	1812	SYSTEM	00	
sqlservr.exe	1800	SYSTEM	00	7
sqlservr.exe	1700	SYSTEM	02	7
msdtc.exe	1612	SYSTEM	00	3
mdm.exe	1576	SYSTEM	00	2
inetinfo.exe	1544	SYSTEM	00	9
CTSVCCDA.EXE	1508	SYSTEM	00	1
spoolsv.exe	1376	SYSTEM	00	3
svchost.exe	1188	LOCAL SERVICE	00	3
svchost.exe	1144	NETWORK SERVICE	00	1
svchost.exe	988	SYSTEM	00	17

Processes: 43 | CPU Usage: 4% | Commit Charge: 302M / 1494M

Figure 10-1. The Windows Task Manager

Every Win32 process has at least one main "thread" that functions as the entry point for the application. Formally speaking, the first thread created by a process' entry point is termed the *primary thread*. Simply put, a thread is a specific path of execution within a Win32 process. Traditional Windows applications define the WinMain() method as the application's entry point. On the other hand, console application provides the main() method for the same purpose.

Processes that contain a single primary thread of execution are intrinsically "thread-safe," given the fact that there is only one thread that can access the data in the application at a given time. However, a single-threaded process (especially one that is GUI-based) will often appear a bit unresponsive to the user if this single thread is performing a complex operation (such as printing out a lengthy text file, performing an exotic calculation, or attempting to connect to a remote server thousands of miles away).

Given this potential drawback of single-threaded applications, the Win32 API makes it is possible for the primary thread to spawn additional secondary threads (also termed *worker threads*) in the background, using a handful of Win32 API functions such as CreateThread(). Each thread (primary or secondary) becomes a unique path of execution in the process and has concurrent access to all shared points of data.

As you may have guessed, developers typically create additional threads to help improve the program's overall responsiveness. Multithreaded processes provide the illusion that numerous activities are happening at more or less the same time.

For example, an application may spawn a worker thread to perform a labor-intensive unit of work (again, such as printing a large text file). As this secondary thread is churning away, the main thread is still responsive to user input, which gives the entire process the potential of delivering greater performance. However, this may not actually be the case: using too many threads in a single process can actually *degrade* performance, as the CPU must switch between the active threads in the process (which takes time).

In reality, it is always worth keeping in mind that multithreading is most commonly an illusion provided by the operating system. Machines that host a single CPU do not have the ability to literally handle multiple threads at the same exact time. Rather, a single CPU will execute one thread for a unit of time (called a *time-slice*) based on the thread's priority level. When a thread's time-slice is up, the existing thread is suspended to allow another thread to perform its business. For a thread to remember what was happening before it was kicked out of the way, each thread is given the ability to write to Thread Local Storage (TLS) and is provided with a separate call stack, as illustrated in Figure 10-2.

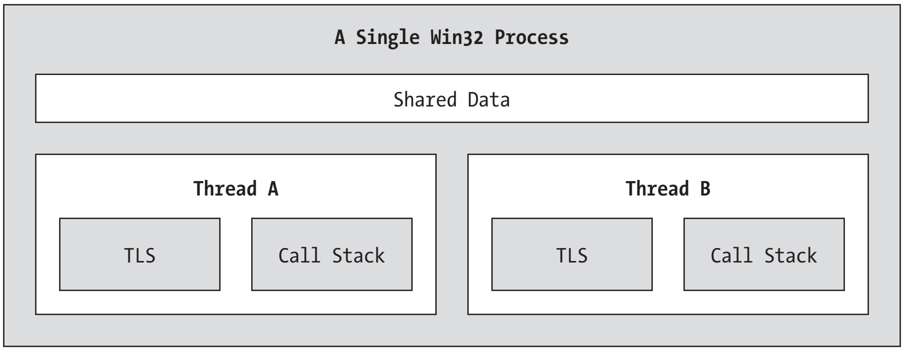


Figure 10-2. The Win32 process / thread relationship

NOTE The newest Intel CPUs have an ability called hyperthreading that allows a single CPU to handle multiple threads simultaneously under certain circumstances. See `http://www.intel.com/info/hyperthreading` for more details.

Interacting with Processes Under the .NET Platform

Although processes and threads are nothing new, the manner in which we 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, let's begin by checking out how processes have been altered to accommodate the needs of the CLR.

The System.Diagnostics namespace defines a number of types that allow you to programmatically interact with processes and various diagnostic-related types such as the system event log and performance counters. For the purposes of this chapter, we are only concerned with the process-centric types, defined in Table 10-1.

Table 10-1. Select Members of the System.Diagnostics Namespace

Process-Centric Types of the System.Diagnostics Namespace	**Meaning in Life**
Process	The Process class provides access to local and remote processes and also allows you to programmatically start and stop processes.
ProcessModule	This type represents a module (*.dll or *.exe) that is loaded into a particular process. Understand that the ProcessModule type can represent *any* module, COM-based, .NET-based, or traditional C-based binaries.
ProcessModuleCollection	Provides a strongly typed collection of ProcessModule objects.
ProcessStartInfo	Specifies a set of values used when starting a process via the Process.Start() method.
ProcessThread	Represents a thread within a given process. ProcessThread is a type used to diagnose a process' thread set, and is not used to spawn new threads of execution within a process. As you will see later in this chapter, duties of this sort are the role of the types within the System.Threading namespace.
ProcessThreadCollection	Provides a strongly typed collection of ProcessThread objects.

The System.Diagnostics.Process type allows you to identify the running processes on a given machine (local or remote). The Process class also provides members that allow you to programmatically start and terminate processes, establish a process' priority level, and obtain a list of active threads and/or loaded modules within a given process. Table 10-2 illustrates some (but not all) of the key members of System.Diagnostics.Process.

Table 10-2. Select Members of the Process Type

Member of System.Diagnostic.Process	Meaning in Life
ExitCode	This property gets the value that the associated process specified when it terminated. Do note that you will be required to handle the Exited event (for asynchronous notification) or call the WaitForExit() method (for synchronous notification) to obtain this value.
ExitTime	This property gets the time stamp associated with the process that has terminated (represented with a DateTime type).
Handle	Returns the handle associated to the process by the OS.
HandleCount	Returns the number of handles opened by the process.
Id	This property gets the process ID (PID) for the associated process.
MachineName	This property gets the name of the computer the associated process is running on.
MainModule	Gets the ProcessModule type that represents the main module for a given process.
MainWindowTitle MainWindowHandle	MainWindowTitle gets the caption of the main window of the process (if the process does not have a main window, you receive an empty string). MainWindowHandle gets the underlying handle (represented via a System.IntPtr type) of the associated window. If the process does not have a main window, the IntPtr type is assigned the value System.IntPtr.Zero.
Modules	Provides access to the strongly typed ProcessModuleCollection type, which represents the set of modules (*.dll or *.exe) loaded within the current process.
PriorityBoostEnabled	Determines if the OS should temporarily boost the process if the main window has the focus.
PriorityClass	Allows you to read or change the overall priority for the associated process.
ProcessName	This property gets the name of the process (which as you would assume is the name of the application itself).
Responding	This property gets a value indicating whether the user interface of the process is responding (or not).
StartTime	This property gets the time that the associated process was started (via a DateTime type).

Table 10-2. Select Members of the Process Type (Continued)

Member of System.Diagnostic.Process	**Meaning in Life**
Threads	This property gets the set of threads that are running in the associated process (represented via an array of ProcessThread types).
CloseMainWindow()	Closes a process that has a user interface by sending a close message to its main window.
GetCurrentProcess()	This static method returns a new Process type that represents the currently active process.
GetProcesses()	This static method returns an array of new Process components running on a given machine.
Kill()	Immediately stops the associated process.
Start()	Starts a process.

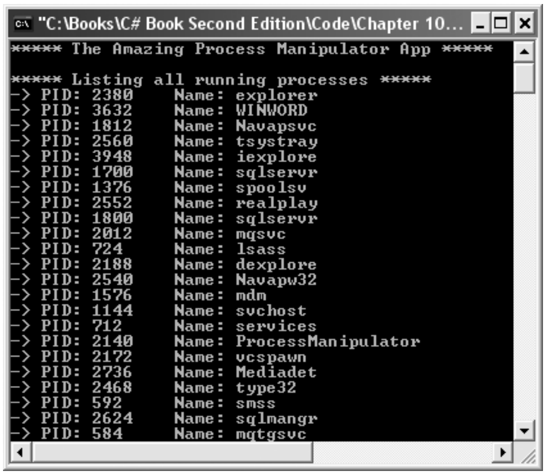
Enumerating Running Processes

To illustrate the process of manipulating Process types (pardon the redundancy), assume you have a C# console application named ProcessManipulator, which defines the following static helper method:

```
public static void ListAllRunningProcesses()
{
     // Get all the processes on the local machine.
     Process[] runningProcs = Process.GetProcesses(".");
     // Print out PID and name of each proc.
     foreach(Process p in runningProcs)
     {
          string info = string.Format("-> PID: {0}\tName: {1}",
               p.Id, p.ProcessName);
          Console.WriteLine(info);
     }
     Console.WriteLine("************************************\n");
}
```

Notice how the static Process.GetProcesses() method returns an array of Process types that represent the running processes on the target machine (the dot notation seen here represents the local computer).

Once you have obtained the array of Process types, you are able to trigger any of the members seen in Table 10-2. Here, simply dump the process identifier (PID) and the name of each process. Assuming the Main() method has been updated to call this helper function, you will see something like the output in Figure 10-3.



```
"C:\Books\C# Book Second Edition\Code\Chapter 10...
***** The Amazing Process Manipulator App *****

***** Listing all running processes *****
-> PID: 2380    Name: explorer
-> PID: 3632    Name: WINWORD
-> PID: 1812    Name: Navapsvc
-> PID: 2560    Name: tsystray
-> PID: 3948    Name: iexplore
-> PID: 1700    Name: sqlservr
-> PID: 1376    Name: spoolsv
-> PID: 2552    Name: realplay
-> PID: 1800    Name: sqlservr
-> PID: 2012    Name: mqsvc
-> PID: 724     Name: lsass
-> PID: 2188    Name: dexplore
-> PID: 2540    Name: Navapw32
-> PID: 1576    Name: mdm
-> PID: 1144    Name: svchost
-> PID: 712     Name: services
-> PID: 2140    Name: ProcessManipulator
-> PID: 2172    Name: vcspawn
-> PID: 2736    Name: Mediadet
-> PID: 2468    Name: type32
-> PID: 592     Name: smss
-> PID: 2624    Name: sqlmangr
-> PID: 584     Name: mqtgsvc
```

Figure 10-3. Enumerating running processes

Investigating a Specific Process

In addition to obtaining a full and complete list of all running processes on a given machine, the static Process.GetProcessById() method allows you to obtain a single Process type via the associated PID. As you would hope, if you request access to a nonexistent process ID, an ArgumentException exception is thrown:

```
// If there is no process with the PID of 987, a
// runtime exception will be thrown.
int pID = 987;
Process theProc;
try
{ theProc = Process.GetProcessById(pID); }
catch  // Generic catch for simplicitiy
{ Console.WriteLine("-> Sorry...bad PID!"); }
```

Investigating a Process' Thread Set

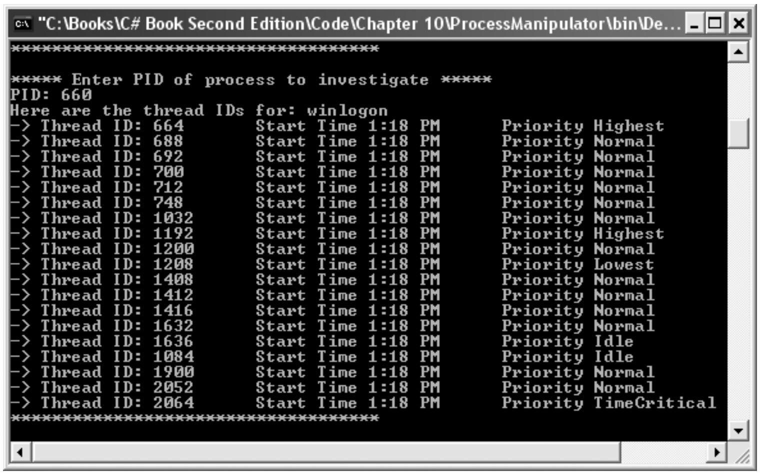
Now that you understand how to gain access to a Process type, you are able to programmatically investigate the set of all threads currently alive in the process at hand. This set of threads is represented by the strongly typed ProcessThreadCollection collection, which contains any number of individual ProcessThread types. To illustrate, assume the following additional static helper function has been added to your current application:

```
public static void EnumThreadsForPid(int pID)
{
     Process theProc;
     try
     { theProc = Process.GetProcessById(pID); }
     catch
     {
          Console.WriteLine("-> Sorry...bad PID!");
          Console.WriteLine("************************************\n");
          return;
     }
     // List out stats for each thread in the specified process.
     Console.WriteLine("Here are the thread IDs for: {0}",
          theProc.ProcessName);
     ProcessThreadCollection theThreads = theProc.Threads;
     foreach(ProcessThread pt in theThreads)
     {
          string info =
              string.Format("-> Thread ID: {0}\tStart Time {1}\tPriority {2}",
               pt.Id , pt.StartTime.ToShortTimeString(), pt.PriorityLevel);
          Console.WriteLine(info);
     }
     Console.WriteLine("************************************\n");
}
```

As you can see, the Threads property of the System.Diagnostics.Process type provides access to the ProcessThreadCollection class. Here, we are printing out the assigned thread ID, start time, and priority level of each thread in the process specified by the client. Thus, if you update your program's Main() method to prompt the user for a PID to investigate:

```
Console.WriteLine("***** Enter PID of process to investigate *****");
Console.Write("PID: ");
string pID = Console.ReadLine();
int theProcID = int.Parse(pID);
EnumThreadsForPid(theProcID);
```

you would find output along the lines of the Figure 10-4.


"C:\Books\C# Book Second Edition\Code\Chapter 10\ProcessManipulator\bin\De...

```
************************************

***** Enter PID of process to investigate *****
PID: 660
Here are the thread IDs for: winlogon
-> Thread ID: 664       Start Time 1:18 PM      Priority Highest
-> Thread ID: 688       Start Time 1:18 PM      Priority Normal
-> Thread ID: 692       Start Time 1:18 PM      Priority Normal
-> Thread ID: 700       Start Time 1:18 PM      Priority Normal
-> Thread ID: 712       Start Time 1:18 PM      Priority Normal
-> Thread ID: 748       Start Time 1:18 PM      Priority Normal
-> Thread ID: 1032      Start Time 1:18 PM      Priority Normal
-> Thread ID: 1192      Start Time 1:18 PM      Priority Highest
-> Thread ID: 1200      Start Time 1:18 PM      Priority Normal
-> Thread ID: 1208      Start Time 1:18 PM      Priority Lowest
-> Thread ID: 1408      Start Time 1:18 PM      Priority Normal
-> Thread ID: 1412      Start Time 1:18 PM      Priority Normal
-> Thread ID: 1416      Start Time 1:18 PM      Priority Normal
-> Thread ID: 1632      Start Time 1:18 PM      Priority Normal
-> Thread ID: 1636      Start Time 1:18 PM      Priority Idle
-> Thread ID: 1084      Start Time 1:18 PM      Priority Idle
-> Thread ID: 1900      Start Time 1:18 PM      Priority Normal
-> Thread ID: 2052      Start Time 1:18 PM      Priority Normal
-> Thread ID: 2064      Start Time 1:18 PM      Priority TimeCritical
************************************
```

Figure 10-4. Enumerating the threads within a running process

The ProcessThread type has additional members of interest beyond Id, StartTime, and PriorityLevel. Table 10-3 documents some members of interest.

Table 10-3. Select Members of the ProcessThread Type

Member of System.Diagnostics.ProcessThread	**Meaning in Life**
BasePriority	Gets the base priority of the thread
CurrentPriority	Gets the current priority of the thread
Id	Gets the unique identifier of the thread
IdealProcessor	Sets the preferred processor for this thread to run on
PriorityLevel	Gets or sets the priority level of the thread
ProcessorAffinity	Sets the processors on which the associated thread can run
StartAddress	Gets the memory address of the function that the operating system called that started this thread
StartTime	Gets the time that the operating system started the thread
ThreadState	Gets the current state of this thread
TotalProcessorTime	Gets the total amount of time that this thread has spent using the processor
WaitReason	Gets the reason that the thread is waiting

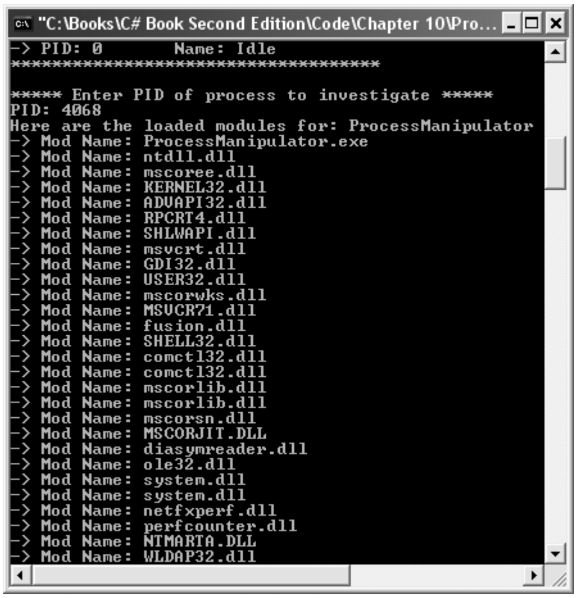
Now before reading any further, be very aware that the ProcessThread type is *not* the entity used to create, suspend, or kill threads under the .NET platform. Rather, ProcessThread is a vehicle used to obtain diagnostic information for the active threads within a running process.

Investigating a Process' Module Set

Next up, let's check out how to iterate over the number of loaded modules that are hosted within a given process. Recall that a *module* is a generic name used to describe a given *.dll (or the *.exe itself) which is hosted by a specific process. When you access the ProcessModuleCollection via the Process.Module property, you are able to enumerate over *all modules* in a process; .NET-based, COM-based, or traditional C-based binaries. Ponder the following helper function:

```
public static void EnumModsForPid(int pID)
{
     Process theProc;
     try
     { theProc = Process.GetProcessById(pID); }
     catch
     {
          Console.WriteLine("-> Sorry...bad PID!");
          Console.WriteLine("************************************\n");
          return;
     }
     Console.WriteLine("Here are the loaded modules for: {0}",
          theProc.ProcessName);
     try
     {
          ProcessModuleCollection theMods = theProc.Modules;
          foreach(ProcessModule pm in theMods)
          {
               string info = string.Format("-> Mod Name: {0}", pm.ModuleName);
               Console.WriteLine(info);
          }
     Console.WriteLine("************************************\n");
     }
     catch{Console.WriteLine("No mods!");}
}
```

To illustrate one possible invocation of this function, let's check out the loaded modules for the process hosting your current console application (ProcessManipulator). To do so, run the application, identify the PID assigned to ProcessManipulator.exe, and pass this value to the EnumModsForPid() method (be sure to update your Main() method accordingly). Once you do, you may be surprised to see the list of *.dlls used for a simple console application (atl.dll, mfc42u.dll, oleaut32.dll and so forth.) Figure 10-5 shows a test run.



```
"C:\Books\C# Book Second Edition\Code\Chapter 10\Pro...
-> PID: 0       Name: Idle
************************************

***** Enter PID of process to investigate *****
PID: 4068
Here are the loaded modules for: ProcessManipulator
-> Mod Name: ProcessManipulator.exe
-> Mod Name: ntdll.dll
-> Mod Name: mscoree.dll
-> Mod Name: KERNEL32.dll
-> Mod Name: ADVAPI32.dll
-> Mod Name: RPCRT4.dll
-> Mod Name: SHLWAPI.dll
-> Mod Name: msvcrt.dll
-> Mod Name: GDI32.dll
-> Mod Name: USER32.dll
-> Mod Name: mscorwks.dll
-> Mod Name: MSVCR71.dll
-> Mod Name: fusion.dll
-> Mod Name: SHELL32.dll
-> Mod Name: comctl32.dll
-> Mod Name: comctl32.dll
-> Mod Name: mscorlib.dll
-> Mod Name: mscorlib.dll
-> Mod Name: mscorsn.dll
-> Mod Name: MSCORJIT.DLL
-> Mod Name: diasymreader.dll
-> Mod Name: ole32.dll
-> Mod Name: system.dll
-> Mod Name: system.dll
-> Mod Name: netfxperf.dll
-> Mod Name: perfcounter.dll
-> Mod Name: NTMARTA.DLL
-> Mod Name: WLDAP32.dll
```

Figure 10-5. Enumerating the loaded modules within a running process

Starting and Killing Processes Programmatically

The final aspects of the System.Diagnostics.Process type examined here are the Start() and Kill() methods. As you can gather by their names, these members provide a way to programmatically launch and terminate a process. For example:

```
public static void StartAndKillProcess()
{
     // Launch Internet Explorer.
     Process ieProc = Process.Start("IExplore.exe",
          "www.intertech-inc.com");
     Console.Write("--> Hit enter to kill {0}...", ieProc.ProcessName);
     Console.ReadLine();
     // Kill the iexplorer.exe process.
     try { ieProc.Kill(); }
     catch{}   // In case user already killed it...
}
```

The static Process.Start() method has been overloaded a few times, however. At minimum you will need to specify the friendly name of the process you wish to launch (such as MS Internet Explorer). This example makes use of a variation of the Start() method that allows you to specify any additional arguments to pass into the program's entry point (i.e., the Main() method).

The Start() method also allows you to pass in a System.Diagnostics.ProcessStartInfo type to specify additional bits of information regarding how a given process should come into life. Here is the formal definition of ProcessStartInfo (see online Help for full details of this type):

```
public sealed class System.Diagnostics.ProcessStartInfo :
    object
{
    public ProcessStartInfo();
    public ProcessStartInfo(string fileName);
    public ProcessStartInfo(string fileName, string arguments);
    public string Arguments { get; set; }
    public bool CreateNoWindow { get; set; }
    public StringDictionary EnvironmentVariables { get; }
    public bool ErrorDialog { get; set; }
    public IntPtr ErrorDialogParentHandle { get; set; }
    public string FileName { get; set; }
    public bool RedirectStandardError { get; set; }
    public bool RedirectStandardInput { get; set; }
    public bool RedirectStandardOutput { get; set; }
    public bool UseShellExecute { get; set; }
    public string Verb { get; set; }
    public string[] Verbs { get; }
    public ProcessWindowStyle WindowStyle { get; set; }
    public string WorkingDirectory { get; set; }
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
}
```

Regardless of which version of the Process.Start() method you invoke, do note that you are returned a reference to the newly activated process. When you wish to terminate the process, simply call the instance level Kill() method.

SOURCE CODE The ProcessManipulator application is included under the Chapter 10 subdirectory.

Understanding the System.AppDomain Type

Now that you understand how to interact with a Win32 process from managed code, we need to examine more closely the new (but related) concept of a *.NET application domain*. As I mentioned briefly in the introduction, unlike a traditional (non-.NET) Win32 *.exe application, .NET assemblies are hosted in a logical partition within a process termed an application domain (aka AppDomain) and many application domains can be hosted inside a single OS process. This additional subdivision of a traditional Win32 process offers several benefits, some of which are:

- AppDomains are a key aspect of the OS-neutral nature of the .NET platform, given that this logical division abstracts away the differences in how an underlying operating system represents a loaded executable.
- AppDomains are far less expensive in terms of processing power and memory than a full blown process (for example, the CLR is able to load and unload application domains much quicker than a formal process).
- AppDomains provide a deeper level of isolation for hosting a loaded application. If one AppDomain within a process fails, the remaining AppDomains remain functional.

As suggested in the previous hit-list, 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 factoid, be very aware that applications that run in unique AppDomains are unable to share any information of any kind (global variables or static fields) unless they make use of the .NET Remoting protocol (examined in Chapter 12) to marshal the data.

NOTE In some respects, .NET application domains are reminiscent of the "apartment" architecture of classic COM. Of course, .NET AppDomains are managed types whereas the COM apartment architecture is built on an unmanaged (and hideously complex) structure.

Understand that while a single process *may* host multiple AppDomains, this is not always the case. At the very least an OS process will host what is termed the default application domain. This specific application domain is automatically created by the CLR at the time the process launches. After this point, the CLR creates additional application domains on an as-needed basis. If the need should arise (which it most likely *will not* for a majority of your .NET endeavors), you are also able to programmatically create application domains at runtime within a given process using static methods of the System.AppDomain class. This class is also useful for low-level control of application domains. Key members of this class are shown in Table 10-4.

Table 10-4. Select Members of AppDomain

AppDomain Member	Meaning in Life
CreateDomain()	This static method creates a new AppDomain in the current process. Understand that the CLR will create new application domains as necessary, and thus the chance of you absolutely needing to call this member is slim to none (unless you happen to be building a custom CLR host).
GetCurrentThreadId()	This static method returns the ID of the active thread in the current application domain.
Unload()	Another static method that allows you to unload a specified AppDomain within a given process.
BaseDirectory	This property returns the base directory that the assembly resolver used to probe for dependent assemblies.
CreateInstance()	Creates an instance of a specified type defined in a specified assembly file.
ExecuteAssembly()	Executes an assembly within an application domain, given its file name.
GetAssemblies()	Gets the set of .NET assemblies that have been loaded into this application domain. Unlike the Process type, the GetAssemblies() method will only return the list of true-blue .NET binaries. COM-based or C-based binaries are ignored.
Load()	Used to dynamically load an assembly into the current application domain.

In addition, the AppDomain type also defines a small set of events that correspond to various aspects of an application domain's life-cycle (Table 10-5).

Table 10-5. Events of the AppDomain Type

Events of System.AppDomain	Meaning in Life
AssemblyLoad	Occurs when an assembly is loaded
AssemblyResolve	Occurs when the resolution of an assembly fails
DomainUnload	Occurs when an AppDomain is about to be unloaded
ProcessExit	Occurs on the default application domain when the default application domain's parent process exits
ResourceResolve	Occurs when the resolution of a resource fails
TypeResolve	Occurs when the resolution of a type fails
UnhandledException	Occurs when an exception is not caught by an event handler

Fun with AppDomains

To illustrate how to interact with .NET application domains programmatically, assume you have a new C# console application named AppDomainManipulator. The static PrintAllAssembliesInAppDomain() helper method makes use of AppDomain.GetAssemblies() to obtain a list of all .NET binaries hosted within the application domain in question.

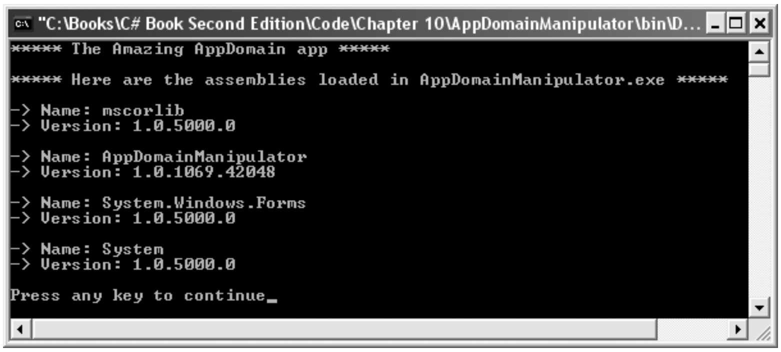
This list is represented by an array of System.Reflection.Assembly types, and thus we are required to use the System.Reflection namespace (full details of this namespace and the Assembly type are seen in Chapter 11). Once we obtain the list of loaded assemblies, we iterate over the array and print out the friendly name and version of each module:

```
using System.Reflection;  // For the Assembly type.
...
public static void PrintAllAssembliesInAppDomain(AppDomain ad)
{
     Assembly[] loadedAssemblies = ad.GetAssemblies();
     Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
          ad.FriendlyName);
     foreach(Assembly a in loadedAssemblies)
     {
          Console.WriteLine("-> Name: {0}", a.GetName().Name);
          Console.WriteLine("-> Version: {0}\n", a.GetName().Version);
     }
}
```

Now assume you have updated the Main() method to obtain a reference to the current application domain before invoking PrintAllAssembliesInAppDomain(), using the AppDomain.CurrentDomain property. To make things a bit more interesting, notice that the Main() method launches a message box to force the assembly resolver to load the System.Windows.Forms.dll and System.dll assemblies (so be sure to set a reference to these assemblies and update your "using" statements appropriately):

```
public static int Main(string[] args)
{
     Console.WriteLine("***** The Amazing AppDomain app *****\n");
     // Get info for current AppDomain.
     AppDomain defaultAD= AppDomain.CurrentDomain;
     MessageBox.Show("This call loaded System.Windows.Forms.dll and System.dll");
     PrintAllAssembliesInAppDomain(defaultAD);
     return 0;
}
```

Figure 10-6 shows the output.



```
"C:\Books\C# Book Second Edition\Code\Chapter 10\AppDomainManipulator\bin\D...
***** The Amazing AppDomain app *****

***** Here are the assemblies loaded in AppDomainManipulator.exe *****

-> Name: mscorlib
-> Version: 1.0.5000.0

-> Name: AppDomainManipulator
-> Version: 1.0.1069.42048

-> Name: System.Windows.Forms
-> Version: 1.0.5000.0

-> Name: System
-> Version: 1.0.5000.0

Press any key to continue_
```

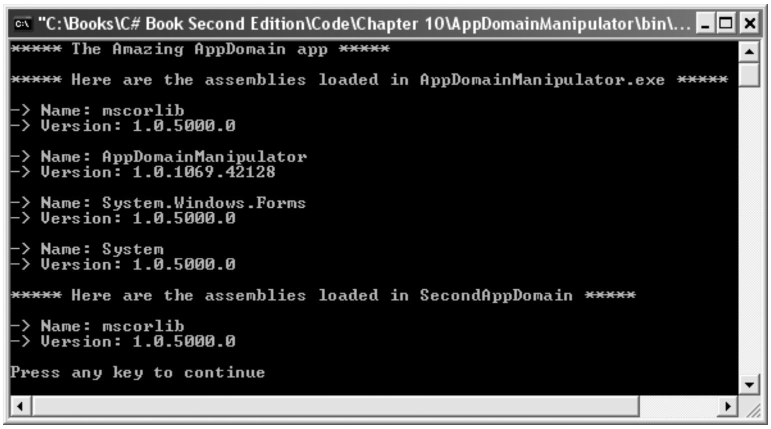
Figure 10-6. Enumerating assemblies within a given app domain (within a given process)

Programmatically Creating New AppDomains

Recall that a single process is capable of hosting multiple AppDomains. While it is true that you will seldom (if ever) need to manually create AppDomains directly (unless you happen to be creating a custom host for the CLR), you are able to do so via the static CreateDomain() method. As you would guess, this method has been overloaded a number of times. At minimum you will simply specify the friendly name of the new application domain as seen here:

```
public static int Main(string[] args)
{
...
     // Make a new AppDomain in the current process.
     AppDomain anotherAD = AppDomain.CreateDomain("SecondAppDomain");
     PrintAllAssembliesInAppDomain(anotherAD);
     return 0;
}
```

Now, if you run the application again (Figure 10-7), notice that the System.Windows.Forms.dll and System.dll assemblies are only loaded within the default application domain! This may seem counterintuitive if you have a background in traditional Win32 (as you might suspect that both application domains have access to the same assembly set). Recall, however, that an assembly loads into an *application domain,* not directly into the process itself.



```
"C:\Books\C# Book Second Edition\Code\Chapter 10\AppDomainManipulator\bin\...
***** The Amazing AppDomain app *****

***** Here are the assemblies loaded in AppDomainManipulator.exe *****

-> Name: mscorlib
-> Version: 1.0.5000.0

-> Name: AppDomainManipulator
-> Version: 1.0.1069.42128

-> Name: System.Windows.Forms
-> Version: 1.0.5000.0

-> Name: System
-> Version: 1.0.5000.0

***** Here are the assemblies loaded in SecondAppDomain *****

-> Name: mscorlib
-> Version: 1.0.5000.0

Press any key to continue
```

Figure 10-7. A single process with two application domains

Next, notice how the SecondAppDomain application domain automatically contains its own copy of mscorlib.dll, as this key assembly is automatically loaded by the CLR for each and every application domain. This begs the question, "How can I programmatically load an assembly into an application domain?" Answer? The AppDomain.Load() method (or alternatively, AppDomain.ExecuteAssembly()). I'll hold off on this topic until I discuss the process of dynamically loading assemblies in Chapter 11 during our examination of .NET reflection services.

To solidify the relationship between processes, application domains, and assemblies, Figure 10-8 diagrams the internal composition of the AppDomainManipulator.exe process you have just constructed.

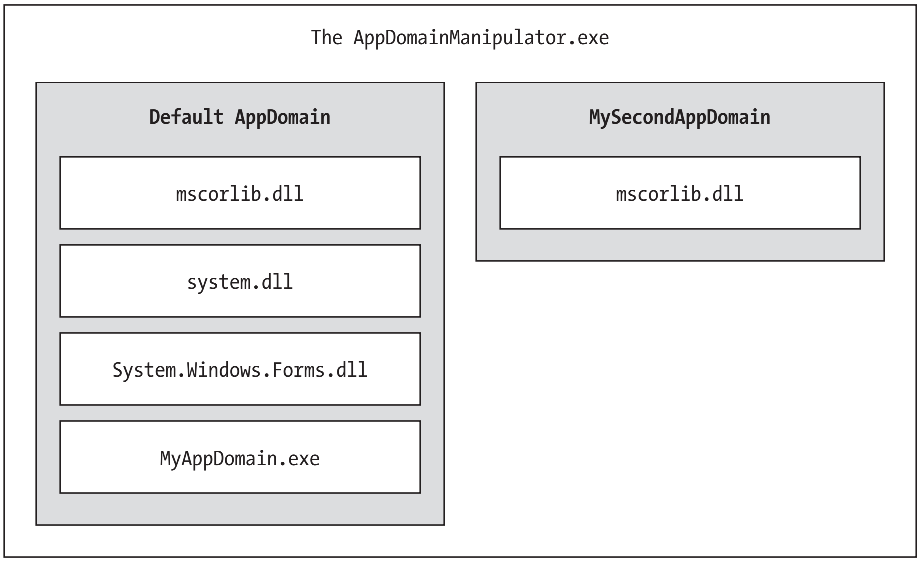


Figure 10-8. The AppDomainManipulator.exe process under the hood

Programmatically Unloading AppDomains (and Hooking into Events)

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.

Recall that the AppDomain type defines a small set of events, one of which is DomainUnload. This is fired when a (non-default) AppDomain is unloaded from the containing process. Another event of interest is the ProcessExit event, which is fired when the default application domain is unloaded from the process (which obviously entails the termination of the process itself). Thus, if you wish to programmatically unload anotherAD from the AppDomainManipulator.exe process, and be notified when the application domain is torn down, you are able to write the following event logic:

```
public static void anotherAD_DomainUnload(object sender, EventArgs e)
{ Console.WriteLine("***** Unloaded anotherAD! *****\n"); }
...
// Hook into DomainUnload event.
anotherAD.DomainUnload +=
     new EventHandler(anotherAD_DomainUnload);
// Now unload anotherAD.
AppDomain.Unload(anotherAD);
```

If you wish to be notified when the default AppDomain is unloaded, modify your application to support the following event logic:

```
private static void defaultAD_ProcessExit(object sender, EventArgs e)
{ Console.WriteLine("***** Unloaded defaultAD! *****\n");  }
...
defaultAD.ProcessExit +=new EventHandler(defaultAD_ProcessExit);
```

SOURCE CODE The AppDomainManipulator project is included under the Chapter 10 subdirectory.

Understanding Context (or How Low Can You Go?)

As you have just seen, AppDomains are logical partitions within a process used to host .NET assemblies. 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 partition .NET objects that have similar execution requirements. Using context, the CLR is able to ensure that objects that have special runtime requirements are handled appropriately and in a consistent manner by intercepting method invocations into and out of a given context. This layer of interception allows CLR to adjust the current method invocation to conform to the contextual settings of a given type.

Just as a process defines a default AppDomain, every application domain has a default context. This default context (sometimes referred to as *context 0*, given that it is always the first context created within an application domain) is used to group together .NET objects that have no specific or unique contextual needs. As you may expect, a vast majority of your .NET class types will be loaded into context 0. If the CLR determines a newly created object has special needs, a new context boundary is created within the hosting application domain. Figure 10-9 illustrates the process, AppDomain, context relationship.

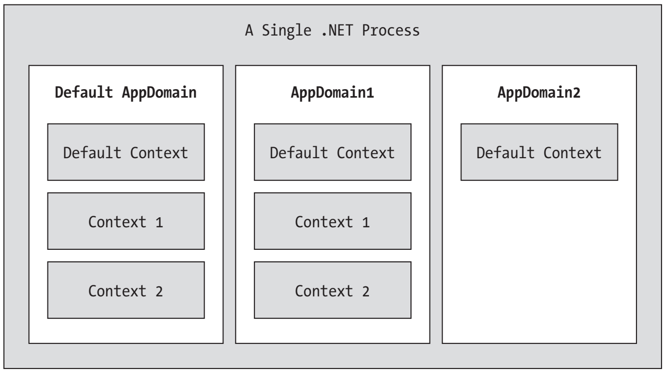


Figure 10-9. Processes, application domains, and context boundaries

Context-Agile and Context-Bound Types

.NET types 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. Building context-agile objects is a no-brainer, given that you simply do nothing (specifically, you do not adorn the type with any contextual attributes and do not derive from the System.ContextBoundObject base class):

```
// A context-agile object is loaded into context 0.
public class IAmAContextAgileClassType{}
```

On the other hand, objects that do demand contextual allocation are termed *context-bound* objects, and *must* derive from the System.ContextBoundObject base class. This base class solidifies the fact that the object in question can only function appropriately within the context in which it was created.

In addition to deriving from System.ContextBoundObject, a context-sensitive type will also be adorned with a special category of .NET attributes termed (not surprisingly) *context attributes*. All context attributes derive from the System.Runtime.Remoting.Contexts.ContextAttribute base class, which is defined as follows (note this class type implements two context-centric interfaces, IContextAttribute and IContextProperty):

```
public class System.Runtime.Remoting.Contexts.ContextAttribute :
    Attribute,
    System.Runtime.Remoting.Contexts.IContextAttribute,
    System.Runtime.Remoting.Contexts.IContextProperty
{
    public ContextAttribute(string name);
    public string Name { virtual get; }
    public object TypeId { virtual get; }
    public virtual bool Equals(object o);
    public virtual void Freeze(System.Runtime.Remoting.Contexts.Context newContext);
    public virtual int GetHashCode();
    public virtual void GetPropertiesForNewContext(
         System.Runtime.Remoting.Activation.IConstructionCallMessage ctorMsg);
    public Type GetType();
    public virtual bool IsContextOK(
      System.Runtime.Remoting.Contexts.Context ctx,
      System.Runtime.Remoting.Activation.IConstructionCallMessage ctorMsg);
    public virtual bool IsDefaultAttribute();
    public virtual bool IsNewContextOK(
          System.Runtime.Remoting.Contexts.Context newCtx);
    public virtual bool Match(object obj);
    public virtual string ToString();
}
```

The .NET base class libraries define numerous context attributes that describe specific runtime requirements (such as thread synchronization and URL activation). Given the role of .NET context, it should stand to reason that if a context-bound object were to somehow end up in an incompatible context, bad things are guaranteed to occur at the most inopportune times.

Creating a Context-Bound Object

So, what sort of bad things might occur if a context-bound object is placed into an incompatible context? The answer depends on the object's advertised contextual settings. Assume for example that you wish to define a .NET type that is automatically thread-safe in nature, even though you have not hard-coded thread-safe-centric logic within the method implementations. To do so, you may apply the System.Runtime.Remoting.Contexts.SynchronizationAttribute attribute as follows:

```
using System.Runtime.Remoting.Contexts;
...
// This context-bound type will only be loaded into a
// synchronized (and hence, thread safe) context.
[Synchronization]
public class MyThreadSafeObject : ContextBoundObject
{}
```

As you will see in greater detail later in this chapter, classes that are attributed with the [Synchronization] attribute are loaded into a thread-safe context. Given the special contextual needs of the MyThreadSafeObject class type, imagine the problems that would occur if an allocated object were moved from a synchronized context into a non-synchronized context. The object is suddenly no longer thread-safe and thus becomes a candidate for massive data corruption, as numerous threads are attempting to interact with the (now thread-volatile) reference object. This is obviously a huge problem, given that the code base has not specifically wrapped thread-sensitive resources with hard-coded synchronization logic.

Placing Context in Context

Now, the good news is that you do *not* have to concern yourself with the act of ensuring that your context-bound objects are loaded into the correct contextual setting. The .NET runtime will read the assembly metadata when constructing the type, and build a new context within the current application domain when necessary.

To be honest, the notion of .NET context is an extremely low-level facility of the CLR. So much so, that a key context-centric namespace, System.Runtime.Remoting.Context, only formally lists the SynchronizationAttribute class type within online Help (as of .NET version 1.1). The remaining members are considered usable only by the CLR, and are intended to be ignored (which you should do, given that many members of this namespace are subject to change in future releases of .NET).

Nevertheless, it is possible to make use of these hands-off types, simply as an academic endeavor. Thus, by way of a friendly heads-up, understand that the following example is purely illustrative in nature. As a .NET programmer, you can safely ignore these low-level primitives in 99.99 percent of your applications. If you wish to see the formal definition of the any of the following context-centric types, make use of the wincv.exe utility.

NOTE COM+ developers are already aware of the notion of context. Using COM IDL attributes and the Component Services utility, developers are able to establish contextual settings for a given COM+ type. Although .NET and COM+ both make use of contextual boundaries, understand that the underlying implementations of both systems are not the same and cannot be treated identically.

Fun with Context

To begin, assume you have a new console application named ContextManipulator. This application defines two context-agile types and a single context-bound type:

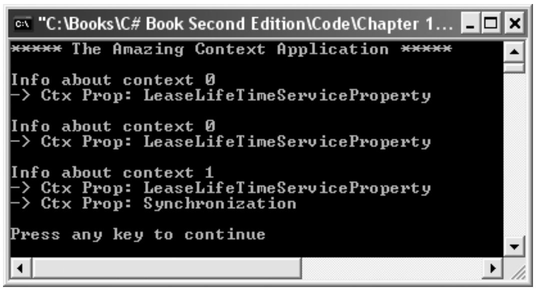
```
using System.Runtime.Remoting.Contexts;  // For Context type.
using System.Threading;  // For Thread type.
...
// These classes have no special contextual
// needs and will be loaded into the
// default context of the app domain.
public class NoSpecialContextClass
{
     public NoSpecialContextClass()
     {
          // Get context information and print out context ID.
          Context ctx = Thread.CurrentContext;
          Console.WriteLine("Info about context {0}", ctx.ContextID);
          foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
               Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
     }
}
public class NoSpecialContextClass2
{
     public NoSpecialContextClass2()
     {
          // Get context information and print out context ID.
          Context ctx = Thread.CurrentContext;
          Console.WriteLine("Info about context {0}", ctx.ContextID);
          foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
               Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
     }
}
// This class demands to be loaded in
// a synchronization context.
// RECALL! All context-bound types must derive from
// System.ContextBoundObject.
[Synchronization]
public class SynchContextClass : ContextBoundObject
{
     public SynchContextClass()
     {
          // Get context information and print out context ID.
          Context ctx = Thread.CurrentContext;
          Console.WriteLine("Info about context {0}", ctx.ContextID);
          foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
               Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
     }
}
```

Notice that each of the class constructors obtains a System.Runtime.Remoting.Contexts.Context type from the current thread of execution, via the static Thread.CurrentContext property. Using this type, you are able to print out statistics about the contextual boundary, such as its assigned ID, as well as a set of descriptors obtained via the Context.ContextProperties property.

This instance-level property returns a (very) low-level interface named IContextProperty, which exposes each descriptor through the Name property.

Now, assume Main() has been updated to allocate an instance of each class type. As the objects come to life, the class constructors will dump out various bits of context-centric information (Figure 10-10):

```
static void Main(string[] args)
{
     Console.WriteLine("***** The Amazing Context Application *****\n");
     // Make each class type and print contextual info.
     NoSpecialContextClass noBigDealObj = new NoSpecialContextClass();
     Console.WriteLine();
     NoSpecialContextClass2 noBigDealObj2 = new NoSpecialContextClass2();
     Console.WriteLine();
     SynchContextClass synchObj = new SynchContextClass();
     Console.WriteLine();
}
```



```
***** The Amazing Context Application *****

Info about context 0
-> Ctx Prop: LeaseLifeTimeServiceProperty

Info about context 0
-> Ctx Prop: LeaseLifeTimeServiceProperty

Info about context 1
-> Ctx Prop: LeaseLifeTimeServiceProperty
-> Ctx Prop: Synchronization

Press any key to continue
```

Figure 10-10. Investigating an object's context

Given that the NoSpecialContextClass and NoSpecialContextClass2 types have not been qualified with a specific context attribute, the CLR will have both loaded into context 0 (i.e., the default context). However, the SynchContextClass type is loaded into a unique contextual boundary within the current application domain, given the fact that this context-bound type was adorned with the [Synchronization] attribute.

SOURCE CODE The ContextManipulator project is included under the Chapter 10 subdirectory.

Summarizing Processes, AppDomains, and Context

So then, at this point in the game you have seen how the .NET runtime has altered the composition of a traditional Win32 process to function within the common language runtime (CLR). To summarize the key points, remember the following:

- A .NET process hosts one to many application domains. Each AppDomain is able to host any number of related .NET assemblies, and may be independently loaded and unloaded by the CLR (or programmatically via the System.AppDomain type).
- A given AppDomain consists of one to many contexts. Using a context, the CLR is able to group special needs objects into a logical container, to ensure that their runtime requirements are honored.

If the previous pages have seemed to be a bit too low-level for your liking, fear not. For the most part, the .NET runtime automatically deals with the details of processes, application domains, and contexts on your behalf. However, this background discussion has provided a solid foundation regarding how the CLR creates, processes, and destroys specific threads of execution (as well as increased your understanding of some underlying CLR concepts).

The Process/AppDomain/Context/Thread Relationship

As mentioned at the opening of this chapter, a thread is a path of execution within a loaded application. While many .NET applications can live happy and productive single-threaded lives, the primary thread may spawn secondary threads of execution to perform additional units of work. In just a moment, you will be introduced to the System.Threading namespace, which defines numerous types used to create multithreaded .NET applications. First however, we need to check out exactly "where" a thread lives within a .NET process.

The first thing you must understand is that under the .NET platform, there is *not* a direct one-to-one correlation between application domains and threads. In fact, a given AppDomain can have numerous threads executing within it at any given time. Furthermore, a particular thread is not confined to a single application domain during its lifetime. Threads are free to cross application domain boundaries as the thread scheduler and the CLR see fit.

Although active threads can be moved between application boundaries, a given thread can only execute within a single application domain at any point in time (in other words, it is impossible for a single thread to be doing work in more than one AppDomain). If you wish to gain access to the AppDomain that is currently hosting the Thread type, call the static Thread.GetDomain method:

```
// Thread.GetDomain() returns an AppDomain type.
Console.WriteLine(Thread.GetDomain().FriendlyName);
```

A single thread may also be moved into a particular context at any given time, and may be relocated within a new context at the whim of the CLR. If you wish to programmatically discover the current context a thread happens to be executing in, make use of the static Thread.CurrentContext property:

```
// CurrentContext returns a Context type.
Console.WriteLine(Thread.CurrentContext.ContextID);
```

As you would guess, the CLR is the entity that is in charge of moving threads into (and out of) application domains and contexts. As a .NET developer, you are able to remain blissfully unaware where a given thread ends up (or exactly when it is placed into its new boundary). Nevertheless, you should be aware of the underlying model.

The Problem of Concurrency and Thread Synchronization

One of the many joys (read: painful aspects) of multithreaded programming is that you have little control over how the underlying operating system makes use of its threads. For example, if you craft a block of code that creates a new thread of execution, you cannot guarantee that the thread executes immediately. Rather, such code only instructs the OS to execute the thread as soon as possible (which is typically when the thread scheduler gets around to it).

Furthermore, given that threads can be moved between application and contextual boundaries as required by the CLR, you must be mindful of which aspects of your application are *thread-volatile* and which operations are *atomic.* (Thread-volatile operations are the dangerous ones!) For example, assume a given thread is accessing a shared point of data (or type member), and begins to modify its contents. Now assume that this thread is instructed to suspend its activity (by the thread scheduler) to allow another thread to access the same point of data.

If the original thread was not completely finished with its current modification of the type, the second incoming thread may be viewing a partially modified object. At this point, the second thread is basically reading bogus values, which is sure to give way to extremely odd (and very hard to find) bugs (which are even harder to replicate and thus debug).

Atomic operations, on the other hand, are always safe in a multithreaded environment, and yet there are very few operations in .NET that are guaranteed to be atomic. Even a simple assignment statement to a double is not atomic! Unless the .NET Framework documentation specifically says an operation is atomic, you must assume it is thread-volatile and take precautions.

Given this, it should be clear that multithreaded application domains are in themselves quite volatile, as numerous threads can operate on the shared functionality at (more or less) the same time. To protect an application's resources from possible corruption, the .NET developer must make use of any number of threading primitives such as locks, monitors, and the [synchronization] attribute, to control access among the executing threads.

Although the .NET platform cannot make the difficulties of building robust multithreaded applications completely disappear, the process has been simplified considerably. Using types defined within the System.Threading namespace, you are able to spawn additional threads with minimal fuss and bother. Likewise, when it is time to lock down shared points of data, you will find additional types that provide the same functionality as the raw Win32 threading primitives (using a much cleaner object model).

Multithreaded Programming via Delegates

Before checking out the details of programming with threads under the .NET platform, it is worth reiterating that the whole point of creating additional threads is to increase the overall functionality of a given application to the *user*. Recall, however, that many common programming tasks that traditionally required manual creation of threads (e.g., remote method invocations, manipulating IO streams, and so forth) are automated using asynchronous delegates (first examined in Chapter 7).

As you have seen throughout various points in this text (and will see in future chapters), when you invoke a delegate asynchronously, the CLR automatically creates a worker thread to handle the task at hand. However, if you have an application-specific task to account for (such as printing a lengthy document or working with a GUI display), you will be required to manually manipulate threads if you wish to keep your primary thread responsive. This disclaimer aside, allow me to formally introduce the System.Threading namespace.

The System.Threading Namespace

Under the .NET platform, the System.Threading namespace provides a number of types that enable multithreaded programming. In addition to providing types that represent a specific CLR thread, this namespace also defines types that can manage a collection of threads (ThreadPool), a simple (non-GUI based) Timer class, and various types used to provide synchronized access to shared resources. Table 10-6 lists some (but not all) of the core members of this namespace.

Table 10-6. Select Types of the System.Threading Namespace

System.Threading Type	**Meaning in Life**
Interlocked	Provides atomic operations for objects that are shared by multiple threads.
Monitor	Provides the synchronization of threading objects using locks and wait/signals.
Mutex	Synchronization primitive that can be used for interprocess synchronization.
Thread	Represents a thread that executes within the CLR. Using this type, you are able to spawn additional threads in the originating AppDomain.
ThreadPool	This type manages related threads in a given process.

Table 10-6. Select Types of the System.Threading Namespace (Continued)

System.Threading Type	**Meaning in Life**
Timer	Provides a mechanism for executing a method at specified intervals.
ThreadStart	Delegate that specifies the method to call for a given Thread.
ThreadState	This enum specifies the valid states a thread may take (Running, Aborted, etc.).
TimerCallback	Delegate type used in conjunction with Timer types.
ThreadPriority	This enum specifies the valid levels of thread priority.

Examining the Thread Class

The most primitive of all types in the System.Threading namespace is Thread. This class represents an object-oriented wrapper around a given path of execution within a particular AppDomain. This type also defines a number of methods (both static and shared) that allow you to create new threads from the scope of the current thread, as well as suspend, stop, and destroy a particular thread. Consider the list of core static members given in Table 10-7.

Table 10-7. Key Static Members of the Thread Type

Thread Static Member	**Meaning in Life**
CurrentContext	This (read-only) property returns the context the thread is currently running.
CurrentThread	This (read-only) property returns a reference to the currently running thread.

GetDomain()
GetDomainID()
	Returns a reference to the current AppDomain (or the ID of this domain) in which the current thread is running.

Sleep()	Suspends the current thread for a specified time.

Thread also supports the object level members shown in Table 10-8.

Table 10-8. Select Instance Level Members of the Thread Type

Thread Instance Level Member	**Meaning in Life**
Abort()	This method instructs the CLR to terminate the thread ASAP.
IsAlive	This property returns a Boolean that indicates if this thread has been started.
IsBackground	Gets or sets a value indicating whether or not this thread is a background thread.
Name	This property allows you to establish a friendly textual name of the thread.
Priority	Gets or Sets the priority of a thread, which may be assigned a value from the ThreadPriority enumeration.
ThreadState	Gets the state of this thread, which may be assigned a value from the ThreadState enumeration.
Interrupt()	Interrupts the current thread.
Join()	Instructs a thread to wait for another thread to complete.
Resume()	Resumes a thread that has been previously suspended.
Start()	Instructs the CLR to execute the thread ASAP.
Suspend()	Suspends the thread. If the thread is already suspended, a call to Suspend() has no effect.

Gathering Basic Thread Statistics

Recall that the entry point of an executable assembly (i.e., the Main() method) runs on the primary thread of execution. To illustrate the basic use of the Thread type, assume you have a new console application named ThreadStats. The static Thread.CurrentThread property retrieves a Thread type that represents the currently executing thread. Thus, if you trigger this member within the scope of Main(), you are able to print out various statistics about the primary thread:

```
using System.Threading;
...
static void Main(string[] args)
{
     // Get some info about the current thread.
     Thread primaryThread = Thread.CurrentThread;
     // Get name of current AppDomain and context ID.
     Console.WriteLine("***** Primary Thread stats *****");
     Console.WriteLine("Name of current AppDomain: {0}",
          Thread.GetDomain().FriendlyName);
     Console.WriteLine("ID of current Context: {0}",
          Thread.CurrentContext.ContextID);
     Console.WriteLine("Thread Name: {0}", primaryThread.Name);
     Console.WriteLine("Apt state: {0}", primaryThread.ApartmentState);
     Console.WriteLine("Alive: {0}", primaryThread.IsAlive);
     Console.WriteLine("Priority Level: {0}", primaryThread.Priority);
     Console.WriteLine("Thread State: {0}", primaryThread.ThreadState);
     Console.WriteLine();
}
```

Naming Threads

When you run this application, notice how the name of the default thread is currently an empty string. Under .NET, it is possible to assign a human-readable string to a thread using the Name property. Thus, if you wish to be able to programmatically identify the primary thread via the moniker "ThePrimaryThread," you could write the following:

```
// Name the thread.
primaryThread.Name = "ThePrimaryThread";
Console.WriteLine("This thread is called: {0}", primaryThread.Name);
```

.NET Threads and Legacy COM Apartments

Also notice that every Thread type has a property named ApartmentState. Those of you who come from a background in classic COM may already be aware of apartment boundaries. In a nutshell, COM *apartments* were a unit of isolation used to group COM objects with similar threading needs. Under the .NET platform however, apartments are no longer used by managed objects. If you happen to be making use of COM objects from managed code (via the interoperability layer), you are able to establish the apartment settings that should be simulated to handle the coclass in question. To do so, the .NET base class libraries provide two attributes to mimic the single-threaded apartment (STAThreadAttribute, which is added by default to your application's Main() method when using the VS .NET IDE) and multithreaded-apartment (MTAThreadAttribute) of classic COM. In fact, when you create a new *.exe .NET application type, the primary thread is automatically established to function as an STA:

```
// This attribute controls how the primary thread should
// handle COM types.
[STAThread]
static void Main(string[] args)
{
     // COM objects will be placed into an STA.
}
```

If you wish to specify support for the MTA, simply adjust the attribute:

```
[MTAThread]
static void Main(string[] args)
{
     // COM objects will be placed into the MTA.
}
```

Of course, if you don't know (or care) about classic COM objects, you can simply leave the [STAThread] attribute on your Main() method. Doing so will keep any COM types thread-safe without further work on your part. If you don't make use of COM types within the Main() method, the [STAThread] attribute does nothing.

Setting a Thread's Priority Level

As mentioned, we programmers have little control over when the thread scheduler switches between threads. We can, however, mark a given thread with a priority level to offer a hint to the CLR regarding the importance of the thread's activity. By default, all threads have a priority level of normal. However this can be changed at any point in the thread's lifetime using the ThreadPriority property and the related ThreadPriority enumeration:

```
public enum System.Threading.ThreadPriority
{
    AboveNormal, BelowNormal,
    Highest, Lowest,
    Normal,  // Default value.
}
```

Always keep in mind that a thread with the value of ThreadPriority.Highest is not necessarily guaranteed to given the highest precedence. Again, if the thread scheduler is preoccupied with a given task (e.g., synchronizing an object, switching threads, moving threads, or whatnot) the priority level will most likely be altered accordingly. However, all things being equal, the CLR will read these values and instruct the thread scheduler how to best allocate time slices. All things still being equal, threads with an identical thread priority should each receive the same amount of time to perform their work.

NOTE Again, you will seldom (if ever) need to directly alter a thread's priority level. In theory, it is possible to jack up the priority level on a set of threads, thereby preventing lower priority threads from executing at their required levels (so use caution).

Spawning Secondary Threads

When you wish to create additional threads to carry on some unit of work, you need to interact with the Thread class as well as a special threading-related delegate named ThreadStart. The general process is quite simple. First and foremost, you need to create a function (static or instance-level) to perform the additional work. For example, assume the current SimpleThreadApp project defines the following additional static method, which mimics the work seen in Main():

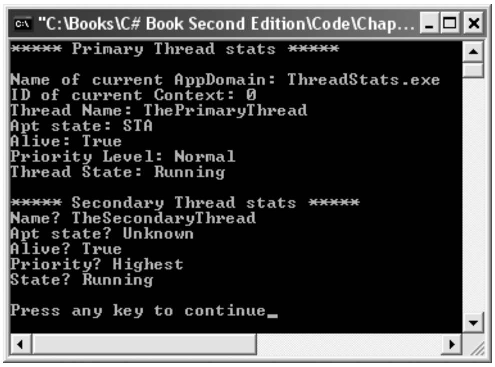
```
static void MyThreadProc()
{
     Console.WriteLine("***** Secondary Thread stats *****");
     Thread.CurrentThread.Name = "TheSecondaryThread";
     Thread secondaryThread = Thread.CurrentThread;
     Console.WriteLine("Name? {0}", secondaryThread.Name);
     Console.WriteLine("Apt state? {0}", secondaryThread.ApartmentState);
     Console.WriteLine("Alive? {0}", secondaryThread.IsAlive);
     Console.WriteLine("Priority? {0}", secondaryThread.Priority);
     Console.WriteLine("State? {0}", secondaryThread.ThreadState);
     Console.WriteLine();
}
```

NOTE The target for the ThreadStart delegate cannot take any arguments and must return void.

Now, within Main(), create a new Thread class and specify a new ThreadStart delegate as a constructor parameter (note the lack of parentheses in the constructor when you give the method name). To inform the CLR that this new thread is ready to run, call the Start() method (but always remember that Start() doesn't actually start the thread). Starting a thread is a nondeterministic operation under the total control of the CLR—you can't do *anything* to force the CLR to execute your thread. It will do so on its own time and on its own terms:

```
[STAThread]
static void Main(string[] args)
{
...
     // Start a secondary thread.
     Thread secondaryThread = new Thread(new ThreadStart(MyThreadProc));
     secondaryThread.Start();
}
```

The output is seen in Figure 10-11.



```
"C:\Books\C# Book Second Edition\Code\Chap...
***** Primary Thread stats *****

Name of current AppDomain: ThreadStats.exe
ID of current Context: 0
Thread Name: ThePrimaryThread
Apt state: STA
Alive: True
Priority Level: Normal
Thread State: Running

***** Secondary Thread stats *****
Name? TheSecondaryThread
Apt state? Unknown
Alive? True
Priority? Highest
State? Running

Press any key to continue_
```

Figure 10-11. Your first multithreaded application

One question that may be on your mind is exactly when a thread terminates. By default, a thread terminates as soon as the function used to create it in the ThreadStart delegate has exited.

Foreground Threads and Background Threads

The CLR assigns a given thread to one of two broad categories:

- *Foreground threads:* Foreground threads have the ability to prevent the current application from terminating. The CLR will not shut down an application (which is to say, unload the hosting AppDomain) until all foreground threads have ended.
- *Background threads:* Background threads (sometimes called *daemon* threads) are viewed by the CLR as expendable paths of execution, which can be ignored at any point in time (even if it is currently laboring over some unit of work). Thus, if all foreground threads have terminated, any and all background threads are automatically killed.

It is important to note that foreground and background threads are *not* synonymous with primary and worker threads. By default, every thread you create via the Thread.Start() method is automatically a *foreground* thread. Again, this means that the AppDomain will not unload until all threads of execution have completed their units of work. In most cases, this is exactly the behavior you require.

For the sake of argument, however, assume that you wish to spawn a secondary thread that should behave as a background thread. Again, this means that the method pointed to by the Thread type (via the ThreadStart delegate) should be able to halt safely as soon as all foreground threads are done with their work. Configuring such a thread is as simple as setting the IsBackground property to true:

```
// Start a new background thread.
Thread secondaryThread = new Thread(new ThreadStart(MyThreadProc));
secondaryThread.Priority = ThreadPriority.Highest;
secondaryThread.IsBackground = true;
```

Now, to illustrate the distinction, assume that the MyThreadProc() method has been updated to print out 1000 lines to the console, pausing for 5 milliseconds between iterations using the Thread.Sleep() method (more on the Thread.Sleep() method later in this chapter):

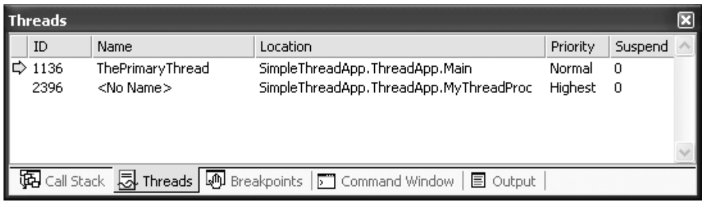
```
static void MyThreadProc()
{
     ...
     for(int i = 0; i < 1000; i ++)
     {
          Console.WriteLine("Value of i is: {0}", i);
          Thread.Sleep(5);
     }
}
```

If you run the application again, you will find that the for loop is only able to print out a tiny fraction of the values, given that the secondary Thread object has been configured as a background thread. Given that the Main() method has spawned a primary foreground thread, as soon as the secondary thread has been started, it is ready for termination.

Now, you are most likely to simply allow all threads used by a given application to remain configured as foreground threads. If this is the case, all threads must finish their work before the AppDomain is unloaded from the hosting process. Nevertheless, marking a thread as a background type can be helpful when the worker-thread in question is performing noncritical tasks or helper tasks that are no longer needed when the main task of the program is over.

The VS .NET Threads Window

To wrap up our initial investigation of threads, it is worth pointing out that the Visual Studio .NET IDE provides a Threads window, which can be accessed from the Debug | Windows menu item during a debugging session. As you can see from Figure 10-12, this window allows you to view the set of currently executing threads in your .NET assembly.



ID	Name	Location	Priority	Suspend
1136	ThePrimaryThread	SimpleThreadApp.ThreadApp.Main	Normal	0
2396	<No Name>	SimpleThreadApp.ThreadApp.MyThreadProc	Highest	0

Figure 10-12. The VS .NET Threads window

SOURCE CODE The ThreadStats project is included under the Chapter 10 subdirectory.

A More Elaborate Threading Example

Now that you have seen the basic process of creating a new thread of execution, we can turn to a more illustrative example. Create a new console application named SimpleMultiThreadApp. Next, define a helper class that supports a public method named DoSomeWork():

```
internal class WorkerClass
{
     public void DoSomeWork()
     {
          // Get hash code for this worker thread.
          Console.WriteLine("ID of worker thread is: {0} ",
               Thread.CurrentThread.GetHashCode());
          // Do the work.
          Console.Write("Worker says: ");
          for(int i = 0; i < 10; i++)
          {
               Console.WriteLine(i + ", ");
          }
          Console.WriteLine();
     }
}
```

Now assume the Main() method creates a new instance of WorkerClass. For the primary thread to continue processing its workflow, create and start a new Thread that is configured to execute the DoSomeWork() method of the WorkerClass type:

```
public class MainClass
{
     public static int Main(string[] args)
     {
          // Get hash code of the current thread.
          Console.WriteLine("ID of primary thread is: {0} ",
               Thread.CurrentThread.GetHashCode());
          // Make worker class.
          WorkerClass w = new WorkerClass();
          // Now make (and start) the worker thread.
          Thread workerThread =
                new Thread(new ThreadStart(w.DoSomeWork));
          workerThread.Start();
          return 0;
     }
}
```

If you run the application you would find each thread has a unique hash code (which is a good thing, as you should have two separate threads at this point).

Clogging Up the Primary Thread

Currently, our application creates a secondary thread to perform a unit of work (in this case, printing 10 numbers). The problem is the fact that printing 10 numbers takes no time at all, and therefore we are not really able to appreciate the fact that the primary thread is free to continue processing. Let's update the application to illustrate this very fact. First, let's tweak the WorkerClass to print out 30,000 numbers, to account for a more labor-intensive process:

```
internal class WorkerClass
{
     public void DoSomeWork()
     {
          ...
          // Do a lot of work.
          Console.Write("Worker says: ");
          for(int i = 0; i < 30000; i++)
          {  Console.WriteLine(i + ", "); }
          Console.WriteLine();
     }
}
```

Next, update the MainClass such that it launches a Windows Forms message box directly after it creates the worker thread (don't forget to set a reference to System.Windows.Forms.dll):

```
public class MainClass
{
     public static int Main(string[] args)
     {
          // Create worker thread as before.
...
          // Now while worker thread is busy,
          // do some additional work on primary thread.
          MessageBox.Show("I'm buzy");
          return 0;
     }
}
```

If you were to now run the application, you would see that the message box is displayed and can be moved around the desktop while the worker thread is busy pumping numbers to the console (Figure 10-13).

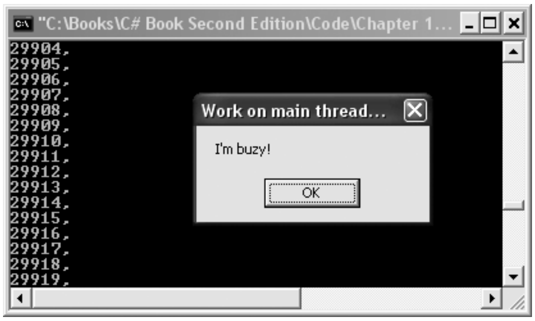

```
29904,
29905,
29906,
29907,
29908,
29909,
29910,
29911,
29912,
29913,
29914,
29915,
29916,
29917,
29918,
29919,
```

Figure 10-13. Two threads each performing a unit of work

Now, contrast this behavior with what you might find if you had a single-threaded application. Assume the Main() method has been updated with logic that allows the user to enter the number of threads used within the current AppDomain (one or two):

```
public static int Main(string[] args)
{
     Console.Write("Do you want [1] or [2] threads? ");
     string threadCount = Console.ReadLine();
...
     // Make worker class.
     WorkerClass w = new WorkerClass();
     // Only make a new thread if the user said so.
     if(threadCount == "2")
     {
          // Now make the thread.
          Thread workerThread =
               new Thread(new ThreadStart(w.DoSomeWork));
          workerThread.Start();
     }
     else  // Execute this method on the single thread.
          w.DoSomeWork();
     // Do some additional work.
     MessageBox.Show("I'm buzy");
     return 0;
}
```

As you can guess, if the user enters the value “1” he or she must wait for all 30,000 numbers to be printed before seeing the message box appear, given that there is only a single thread in the AppDomain. However, if the user enters “2” he or she is able to interact with the message box while the secondary thread spins right along.

Putting a Thread to Sleep

The static Thread.Sleep() method can be used to currently suspend the current thread for a specified amount of time (specified in milliseconds). In particular, you can use this to pause a program. To illustrate, let's update the WorkerClass again. This time around, the DoSomeWork() method does not print out 30,000 lines to the console, but 10 lines. The trick is, between each call to Console.WriteLine(), this worker thread is put to sleep for approximately 2 seconds.

```
internal class WorkerClass
{
     public void DoSomeWork()
     {
          // Get some information about the worker thread.
          Console.WriteLine("ID of worker thread is: { 0} ",
               Thread.CurrentThread.GetHashCode());
          // Do the work (and take a nap).
          Console.Write("Worker says: ");
          for(int i = 0; i < 10; i++)
          {
               Console.WriteLine(i + ", ");
               Thread.Sleep(2000);
          }
          Console.WriteLine();
     }
}
```

Now run your application a few times and specify both threading options. You will find radically different behaviors based on your choice of thread number.

SOURCE CODE The SimpleMultiThreadApp project is included under the Chapter 10 subdirectory.

Concurrency Revisited

Given this previous example, you might be thinking that threads are the magic bullet you have been looking for. Simply create threads for each part of your application and the result will be increased application performance to the user. You already know this is a loaded question, as the previous statement is not necessarily true. If not used carefully and thoughtfully, multithreaded programs are slower than single threaded programs.

Even more important is the fact that each and every thread in a given AppDomain has direct access to the shared data of the application. In the current example, this is not a problem. However, imagine what might happen if the primary and secondary threads were both modifying a shared point of data. As you know, the thread scheduler will force threads to suspend their work at random. Since this is the case, what if thread A is kicked out of the way before it has fully completed its work? Again, thread B is now reading unstable data.

To illustrate, let's build another C# console application named MultiThreadSharedData. This application also has a class named WorkerClass, which maintains a private

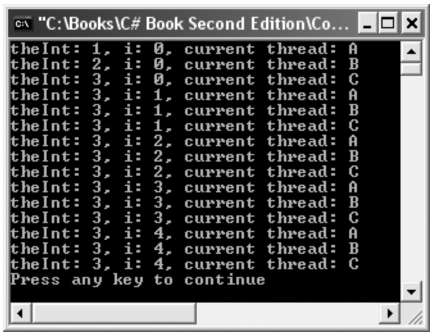
System.Int32 that is manipulated by the DoSomeWork() helper function. Also notice that this helper function also leverages a for loop to printout the value of this private integer, the iterator's value as well as the name of the current thread. Finally, to simulate additional work, each iteration of this logic places the current thread to sleep for approximately one second. Here is the type in question:

```
internal class WorkerClass
{
     private int theInt;
     public void DoSomeWork()
     {
          theInt++;
          for(int i = 0; i < 5; i++)
          {
               Console.WriteLine("theInt: {0}, i: {1}, current thread: {2}",
                    theInt, i, Thread.CurrentThread.Name);
               Thread.Sleep(1000);
          }
     }
}
```

The Main() method is responsible for creating three uniquely named secondary threads of execution, each of which is making calls to the same instance of the WorkerClass type:

```
public class MainClass
{
     public static int Main(string[] args)
     {
          // Make the single worker object.
          WorkerClass w = new WorkerClass();
          // Create and name three secondary threads,
          // each of which makes calls to the same shared object.
          Thread workerThreadA =
                 new Thread(new ThreadStart(w.DoSomeWork));
          workerThreadA.Name = "A";
          Thread workerThreadB =
                 new Thread(new ThreadStart(w.DoSomeWork));
          workerThreadB.Name = "B";
          Thread workerThreadC =
                 new Thread(new ThreadStart(w.DoSomeWork));
          workerThreadC.Name = "C";
          // Now start each one.
          workerThreadA.Start();
          workerThreadB.Start();
          workerThreadC.Start();
          return 0;
     }
}
```

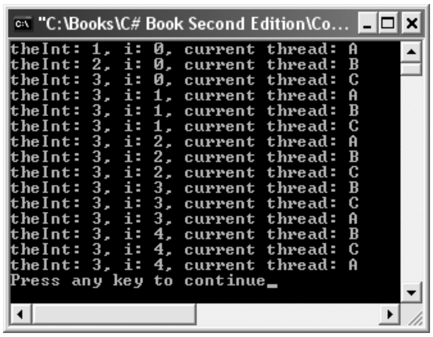
Now before you see some test runs, let's recap the problem. The primary thread within this AppDomain begins life by spawning three secondary worker threads. Each worker thread is told to make calls on the DoSomeWork() method of a single WorkerClass instance. Given that we have taken no precautions to lock down the object's shared resources, there is a good chance that a given thread will be kicked out of the way before the WorkerClass is able to print out the results for the previous thread. Because we don't know exactly when (or if) this might happen, we are bound to get unpredictable results. For example, you might find the output shown in Figure 10-14.



```
"C:\Books\C# Book Second Edition\Co...
theInt: 1, i: 0, current thread: A
theInt: 2, i: 0, current thread: B
theInt: 3, i: 0, current thread: C
theInt: 3, i: 1, current thread: A
theInt: 3, i: 1, current thread: B
theInt: 3, i: 1, current thread: C
theInt: 3, i: 2, current thread: A
theInt: 3, i: 2, current thread: B
theInt: 3, i: 2, current thread: C
theInt: 3, i: 3, current thread: A
theInt: 3, i: 3, current thread: B
theInt: 3, i: 3, current thread: C
theInt: 3, i: 4, current thread: A
theInt: 3, i: 4, current thread: B
theInt: 3, i: 4, current thread: C
Press any key to continue
```

Figure 10-14. Possible output of the MultiThreadSharedData application

Now run the application a few more times. Figure 10-15 shows another possibility (note the ordering among thread names).



```
"C:\Books\C# Book Second Edition\Co...
theInt: 1, i: 0, current thread: A
theInt: 2, i: 0, current thread: B
theInt: 3, i: 0, current thread: C
theInt: 3, i: 1, current thread: A
theInt: 3, i: 1, current thread: B
theInt: 3, i: 1, current thread: C
theInt: 3, i: 2, current thread: A
theInt: 3, i: 2, current thread: B
theInt: 3, i: 2, current thread: C
theInt: 3, i: 3, current thread: B
theInt: 3, i: 3, current thread: C
theInt: 3, i: 3, current thread: A
theInt: 3, i: 4, current thread: B
theInt: 3, i: 4, current thread: C
theInt: 3, i: 4, current thread: A
Press any key to continue_
```

Figure 10-15. Another possible output of the MultiThreadSharedData application

Humm. There are clearly some problems here. As each thread is telling the WorkerClass to "do some work," the thread scheduler is happily swapping threads in the background. The result is inconsistent output. What we need is a way to programmatically enforce synchronized access to the shared resources.

As you would guess, the System.Threading namespace provides a number of synchronization-centric types. The C# programming language also provides a particular keyword for the very task of synchronizing shared data in multithreaded applications.

Synchronization Using the C# "lock" Keyword

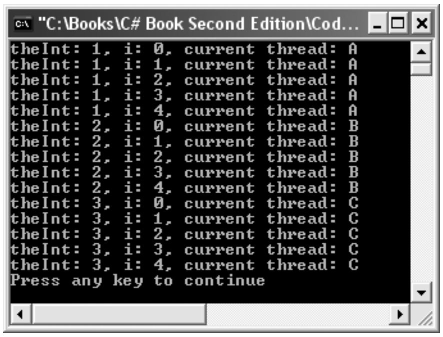
The first approach to providing synchronized access to our DoSomeWork() method is to make use of the C# "lock" keyword. This intrinsic keyword allows you to lock down a block of code so that incoming threads must wait in line for the current thread to finish up its work completely. The "lock" keyword requires you to pass in a token (an object reference) that must be acquired by a thread to enter within the scope of the lock statement. When you are attempting to lock down an instance level method, you can simply pass in a reference to the current type:

```
internal class WorkerClass
{
     private int theInt;
     public void DoSomeWork()
     {
          lock(this)
          {
               theInt++;
               for(int i = 0; i < 5; i++)
               {
                    Console.WriteLine("theInt: {0}, i: {1}, current thread: {2}",
                         theInt, i, Thread.CurrentThread.Name);
                    Thread.Sleep(1000);
               }
          }  // Lock token released here!
     }
}
```

Now, once a thread enters into a locked block of code, the token (in this case, a reference to the current object) is inaccessible by other threads until the lock is released. Thus, if threadA has obtained the lock token, and threadB or threadC are attempting to enter, they must wait until threadA relinquishes the lock.

NOTE If you are attempting to lock down code in a static method, you obviously cannot use the "this" keyword. If this is the case, you can simply pass in the System.Type of the current class using the C# "typeof" operator (although any object reference will work).

If you now rerun the application, you can see that the threads are instructed to politely wait in line for the current thread to finish its business (Figure 10-16).



```
theInt: 1, i: 0, current thread: A
theInt: 1, i: 1, current thread: A
theInt: 1, i: 2, current thread: A
theInt: 1, i: 3, current thread: A
theInt: 1, i: 4, current thread: A
theInt: 2, i: 0, current thread: B
theInt: 2, i: 1, current thread: B
theInt: 2, i: 2, current thread: B
theInt: 2, i: 3, current thread: B
theInt: 2, i: 4, current thread: B
theInt: 3, i: 0, current thread: C
theInt: 3, i: 1, current thread: C
theInt: 3, i: 2, current thread: C
theInt: 3, i: 3, current thread: C
theInt: 3, i: 4, current thread: C
Press any key to continue
```

Figure 10-16. Consistent output of the MultiThreadSharedData application

SOURCE CODE The MultiThreadSharedData application is included under the Chapter 10 subdirectory.

Synchronization Using the System.Threading.Monitor Type

The C# lock statement is really just a shorthand notation for working with the System.Threading.Monitor class type. Under the hood, the previous locking logic (via the C# "lock" keyword) actually resolves to the following (which can be verified using ildasm.exe):

```
internal class WorkerClass
{
     private int theInt;
     public void DoSomeWork()
     {
          // Enter the monitor with token.
          Monitor.Enter(this);
          try
          {
               theInt++;
               for(int i = 0; i < 5; i++)
               {
                    Console.WriteLine("theInt: {0}, i: {1}, current thread: {2}",
                         theInt, i, Thread.CurrentThread.Name);
                    Thread.Sleep(1000);
               }
          }
```

```
            finally
            {
                // Error or not, you must exit the monitor
                // and release the token.
                Monitor.Exit(this);
            }
        }
    }
```

If you run the modified application, you will see no changes in the output (which is good). Here, you make use of the static Enter() and Exit() members of the Monitor type, to enter (and leave) a locked block of code. Now, given that the "lock" keyword seems to require less code than making explicit use of the System.Threading.Monitor type, you may wonder about the benefits. The short answer is control.

If you make use of the Monitor type, you are able to instruct the active thread to wait for some duration of time (via the Wait() method), inform waiting threads when the current thread is completed (via the Pulse() and PulseAll() methods), and so on. As you would expect, in a great number of cases, the C# "lock" keyword will fit the bill. If you are interested in checking out additional members of the Monitor class, consult online Help.

Synchronization Using the System.Threading.Interlocked Type

Although it always is hard to believe until you look at the underlying CLR code, assignments and simple arithmetic operations are *not atomic*. For this reason, the System.Threading namespace also provides a type that allows you to operate on a single point of data atomically. The Interlocked class type defines the static members shown in Table 10-9.

Table 10-9. Members of the Interlocked Type

Member of the System.Threading.Interlocked Type	**Meaning in Life**
Increment()	Safely increments a value by one
Decrement()	Safely decrements a value by one
Exchange()	Safely swaps two values
CompareExchange()	Safely tests two values for equality, and if so, changes one of the values with a third

Although it might not seem like it from the onset, the process of atomically altering a single value is quite common in a multithreaded environment. Thus, rather than writing synchronization code such as the following:

```
int i = 9;
lock(this)
{ i++; }
```

you can simply write:

```
// Pass by reference the value you wish to alter.
int i = 9;
Interlocked.Increment(ref i);
```

Likewise, if you wish to assign the value of a previously assigned System.Int32 to the value 83, you can avoid the need to an explicit lock statement (or Monitor logic) and make use of the Interlocked.Exchange() method:

```
int i = 9;
Interlocked.Exchange(ref i, 83);
```

Finally, if you wish to test two values for equality to change the point of comparison in a thread-safe manner, you would be able to leverage the Interlocked.CompareExchange() method as follows:

```
// If the value of i is currently 83, change i to 99.
Interlocked.CompareExchange(ref i, 99, 83);
```

Synchronization Using the [Synchronization] Attribute

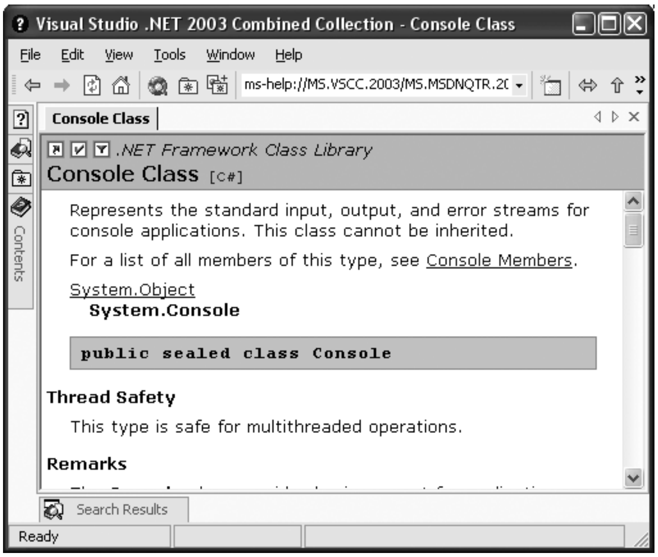
The final synchronization primitive examined here is the [Synchronized] attribute, which, as you recall, is a contextual attribute that can be applied to context-bound objects. When you apply this attribute on a .NET class type, you are effectively locking down *all* members of the object for thread safety:

```
using System.Runtime.Remoting.Contexts;
...
// This context-bound type will only be loaded into a
// synchronized (and hence, thread-safe) context.
[Synchronization]
public class MyThreadSafeObject : ContextBoundObject
{ /* all methods on class are now thread safe */}
```

In some ways, this approach can be seen as the lazy approach to writing thread-safe code, given that we are not required to dive into the details about which aspects of the type are truly manipulating thread-sensitive data. The major downfall of this approach, however, is that even if a given method is not making use of thread-sensitive data, the CLR will *still* lock invocations to the method. Obviously, this could degrade the overall functionality of the type, so use this technique with care.

Thread Safety and the .NET Base Class Libraries

Although this chapter has illustrated how you can build custom thread-safe types, you should also be aware that many of the types of the base class libraries have been preprogrammed to be thread-safe. In fact, when you look up a given type using online Help (such as System.Console) you will find information regarding its level of thread safety (Figure 10-17).


Visual Studio .NET 2003 Combined Collection - Console Class
File
Edit
View
Tools
Window
Help
ms-help://MS.VSCC.2003/MS.MSDNQTR.2(
Console Class
Contents
Search Results
Ready

.NET Framework Class Library

Console Class [C#]

Represents the standard input, output, and error streams for console applications. This class cannot be inherited.

For a list of all members of this type, see Console Members.

System.Object
  System.Console

```
public sealed class Console
```

Thread Safety

This type is safe for multithreaded operations.

Remarks

Figure 10-17. Many (but not all) .NET types are already thread-safe .

Sadly, many .NET types in the base class libraries are *not* thread-safe, and therefore, you will have to make use of the various locking techniques you have examined to ensure the object is able to survive multiple requests from the thread base.

Programming with Timer Callbacks

At this point you have seen a number of ways in which you are able to provide synchronized access to shared blocks of data. To be sure, there are additional types under the System.Threading namespace, which I will allow you to explore at your leisure. However, to wrap up our examination of thread programming, allow me to introduce two additional types, TimerCallback and Timer.

Many applications have the need to call a specific method during regular intervals of time. For example, you may have an application that needs to display the current time on a status bar via a given helper function. As another example, you may wish to have your application call a helper function every so often to perform noncritical background tasks such as checking for new e-mail messages. For situations such as these, the System.Threading.Timer type can be used in conjunction with a related delegate named TimerCallback.

To illustrate, assume you have a console application that will print the current time every second until the user hits a key to terminate the application. The first obvious step is to write the method that will be called by the Timer type:

```
class TimePrinter
{
     static void PrintTime(object state)
     {
          Console.WriteLine("Time is: {0}",
               DateTime.Now.ToLongTimeString());
     }
...
}
```

Notice how this method has a single parameter of type System.Object and returns void. This is not optional, given that the TimerCallback delegate can only call methods that match this signature. The value passed into the target of your TimerCallback delegate can be any bit of information whatsoever (in the case of the e-mail example, this parameter might represent the name of the MS Exchange server to interact with during the process). Also note that given that this parameter is indeed a System.Object, you are able to pass in multiple arguments using a System.Array type.

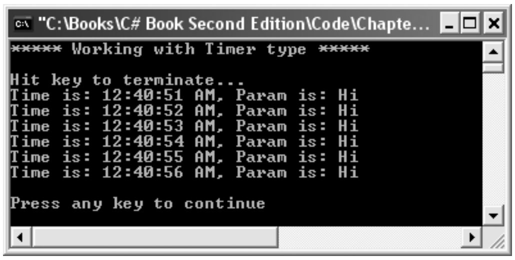
The next step would be to configure an instance of the TimerCallback type and pass it into the Timer object. In addition to a TimerCallback delegate, the Timer constructor also allows you to specify the optional parameter information to pass into the delegate target, the interval to poll the method, as well as the amount of time to wait before making the first call. For example:

```
static void Main(string[] args)
{
     Console.WriteLine("***** Working with Timer type *****\n");
     // Create the delegate for the Timer type.
     TimerCallback timeCB = new TimerCallback(PrintTime);
     // Establish timer settings.
     Timer t = new Timer(
         timeCB,     // The TimerCallback delegate type.
         null,       // Any info to pass into the called method (null for no info).
         0,          // Amount of time to wait before starting.
         1000);      // Interval of time between calls.
     Console.WriteLine("Hit key to terminate...");
     Console.ReadLine();
}
```

In this case, the PrintTime() method will be called roughly every second, and will pass in no additional information to said method. If you did wish to send in some information for use by the delegate target, simply substitute the null value of the second constructor parameter with the appropriate information. For example, ponder the following updates:

```
static void PrintTime(object state)
{
     Console.WriteLine("Time is: {0}, Param is: {1}",
          DateTime.Now.ToLongTimeString(), state.ToString());
}
...
Timer t = new Timer(timeCB, "Hi", 0, 1000);
```

Figure 10-18 shows the output.



```
***** Working with Timer type *****

Hit key to terminate...
Time is: 12:40:51 AM, Param is: Hi
Time is: 12:40:52 AM, Param is: Hi
Time is: 12:40:53 AM, Param is: Hi
Time is: 12:40:54 AM, Param is: Hi
Time is: 12:40:55 AM, Param is: Hi
Time is: 12:40:56 AM, Param is: Hi

Press any key to continue
```

Figure 10-18. The (very useful) console-based clock application

SOURCE CODE The TimerApp application is included under the Chapter 10 subdirectory.

That wraps up our examination of multithreaded programming under .NET. To be sure, the System.Threading namespace defines numerous types beyond what I had the space to examine in this chapter. Nevertheless, at this point you should have a solid foundation to build on.

Summary

The point of this chapter was to expose the internal composition of a .NET executable image. As you have seen, the long-standing notion of a Win32 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 on multiple application domains, which represent isolated and independent boundaries within a process. As you recall, 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.

The remainder of this chapter examined the role of the System.Threading namespace. As you have seen, when an application creates additional threads of execution, the result is that the program in question is able to carry out numerous tasks at (what appears to be) the same time. Finally, the chapter examined various manners in which you can mark thread-sensitive blocks of code to ensure that shared resources do not become unusable units of bogus data.