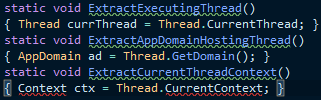
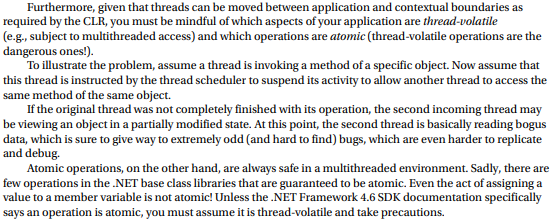
**Multithreaded, Parallel, and Async Programming** Nobody enjoys working with an application that is sluggish during its execution. Moreover, nobody enjoys starting a task in an application (perhaps initiated by clicking a toolbar item) that prevents other parts of the program from being as responsive as possible. Before the release of .NET, building applications that had the ability to perform multiple tasks typically required authoring complex C++ code that used the Windows threading APIs. Thankfully, the .NET platform provides a number of ways for you to build software that can perform complex operations on unique paths of execution, with far fewer pain points. This chapter begins by defining the overall nature of a “multithreaded application.” Next, you will revisit the .NET delegate type to investigate its intrinsic support for asynchronous method invocations. As you’ll see, this technique allows you to invoke a method on a secondary thread of execution without needing to manually create or configure the thread itself. Next, you’ll be introduced to the original threading namespace that has shipped since .NET 1.0, specifically System.Threading. Here you’ll examine numerous types (Thread, ThreadStart, etc.) that allow you to explicitly create additional threads of execution and synchronize your shared resources, which helps ensure that multiple threads can share data in a nonvolatile manner. The remaining parts of this chapter will examine three more recent techniques .NET developers can use to build multithreaded software, specifically the Task Parallel Library (TPL), Parallel LINQ (PLINQ), and the new intrinsic asynchronous keywords of C# (async and await). As you will see, these features can dramatically simplify how you can build responsive multithreaded software applications.

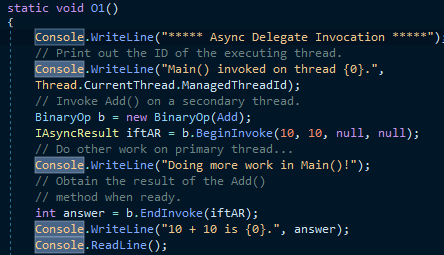
**The Process/AppDomain/Context/Thread Relationship** In Chapter 17, a thread was defined as a path of execution within an executable application. While many .NET applications can live happy and productive single-threaded lives, an assembly’s primary thread (spawned by the CLR when Main() executes) may create secondary threads of execution at any time to perform additional units of work. By creating additional threads, you can build more responsive (but not necessarily faster executing on single-core machines) applications

Under the .NET platform, there is not a direct one-to-one correspondence 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 Windows OS thread scheduler and the .NET CLR see fit. Although active threads can be moved between AppDomain boundaries, a given thread can execute within only 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 at once). When you want to programmatically gain access to the AppDomain that is hosting the current thread, call the static Thread.GetDomain() method, like so:

**The Problem of Concurrency** One of the many “joys” (read: painful aspects) of multithreaded programming is that you have little control over how the underlying operating system or the CLR uses 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.

**The Role of Thread Synchronization** At this point, it should be clear that multithreaded programs are in themselves quite volatile, as numerous threads can operate on the shared resources at (more or less) the same time. To protect an application’s resources from possible corruption, .NET developers must use any number of threading primitives (such as locks, monitors, and the [Synchronization] attribute or language keyword support) to control access among the executing threads.

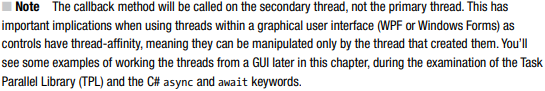
**The Asynchronous Nature of Delegates** If you are new to the topic of multithreading, you might wonder what exactly an asynchronous method invocation is all about. As you are no doubt fully aware, some programming operations take time. Although the previous Add() was purely illustrative in nature, imagine that you built a single-threaded application that is invoking a method on a remote web service operation, calling a method performing a long-running database query, downloading a large document, or writing 500 lines of text to an external file. While performing these operations, the application could appear to hang for some amount of time. Until the task at hand has been processed, all other aspects of this program (such as menu activation, toolbar clicking, or console output) are suspended (which can aggravate users).

**The BeginInvoke() and EndInvoke() Methods** When the C# compiler processes the delegate keyword, the dynamically generated class defines two methods named BeginInvoke() and EndInvoke().

**Invoking a Method Asynchronously** To instruct the BinaryOp delegate to invoke Add() asynchronously, you will modify the logic in the previous project (feel free to add code to the existing project; however, in your lab downloads, you will find a new Console Application project named AsyncDelegate).

**Synchronizing the Calling Thread** If you think carefully about the current implementation of Main(), you might realize that the timespan between calling BeginInvoke() and EndInvoke() is clearly less than five seconds. Therefore, once Doing more work in Main()! prints to the console, the calling thread is now blocked and waiting for the secondary thread to complete before being able to obtain the result of the Add() method. Therefore, you are effectively making yet another synchronous call

In addition to the IsCompleted property, the IAsyncResult interface provides the AsyncWaitHandle property for more flexible waiting logic. This property returns an instance of the WaitHandle type, which exposes a method named WaitOne(). The benefit of WaitHandle.WaitOne() is that you can specify the maximum wait time. If the specified amount of time is exceeded, WaitOne() returns false.

**The Role of the AsyncCallback Delegate** Rather than polling a delegate to determine whether an asynchronously invoked method has completed, it would be more efficient to have the secondary thread inform the calling thread when the task is finished. When you want to enable this behavior, you will need to supply an instance of the System.AsyncCallback delegate as a parameter to BeginInvoke(), which up until this point has been null. However, when you do supply an AsyncCallback object, the delegate will call the specified method automatically when the asynchronous call has completed