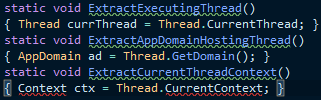
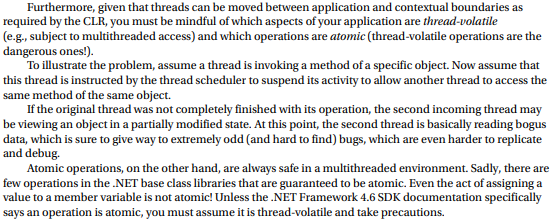
**Multithreaded, Parallel, and Async Programming** Nobody enjoys working with an application that is sluggish during its execution. Moreover, nobody enjoys starting a task in an application (perhaps initiated by clicking a toolbar item) that prevents other parts of the program from being as responsive 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.

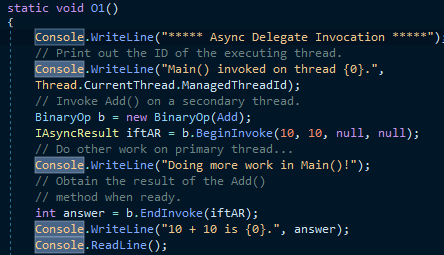
**The Process/AppDomain/Context/Thread Relationship** In Chapter 17, a thread was defined as a path of execution within an executable application. While many .NET applications can live happy and productive single-threaded lives, an assembly’s primary thread (spawned by the CLR when Main() executes) may 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.

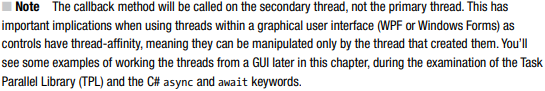
**The Asynchronous Nature of Delegates** If you are new to the topic of multithreading, you might wonder what exactly an asynchronous method invocation is all about. As you are no doubt fully aware, some programming operations take time. Although the previous Add() was purely illustrative in nature, 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).

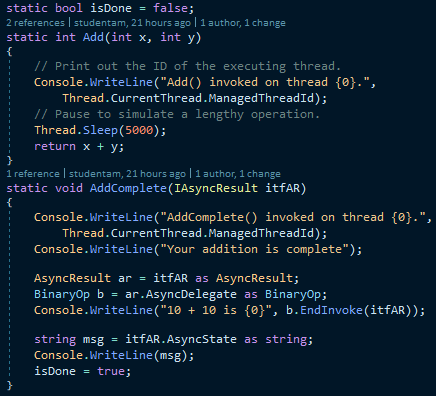
**The BeginInvoke() and EndInvoke() Methods** When the C# compiler processes the delegate keyword, the dynamically generated class defines two methods named BeginInvoke() and EndInvoke().

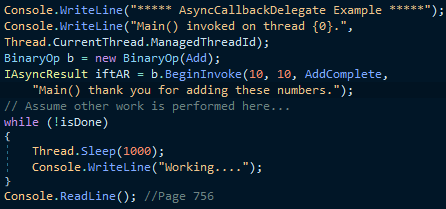
**Invoking a Method Asynchronously** To instruct the BinaryOp delegate to invoke Add() asynchronously, you will modify the logic in the previous project (feel free to add code to the existing project; however, in your lab downloads, you will find a new Console Application project named AsyncDelegate).

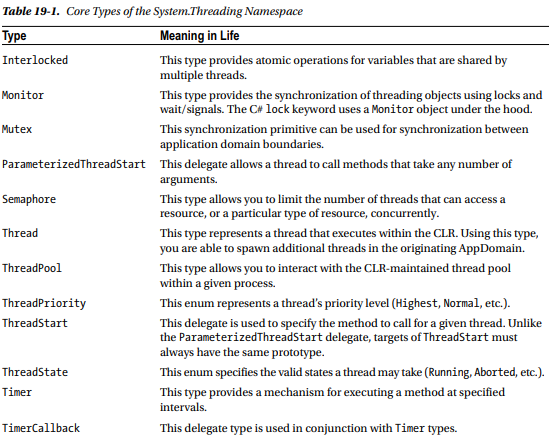
**Synchronizing the Calling Thread** If you think carefully about the current implementation of Main(), you might realize that the timespan between calling BeginInvoke() and EndInvoke() is clearly less than five seconds. Therefore, once Doing more work in Main()! prints to the console, the calling thread 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

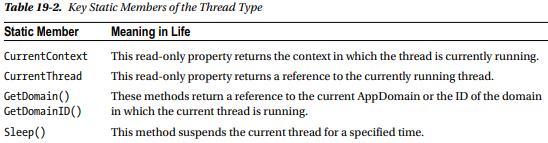
In addition to the IsCompleted property, the IAsyncResult interface provides the AsyncWaitHandle property for more flexible waiting logic. This property returns an instance of the WaitHandle type, which exposes a method named WaitOne(). The benefit of WaitHandle.WaitOne() is that you can specify the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

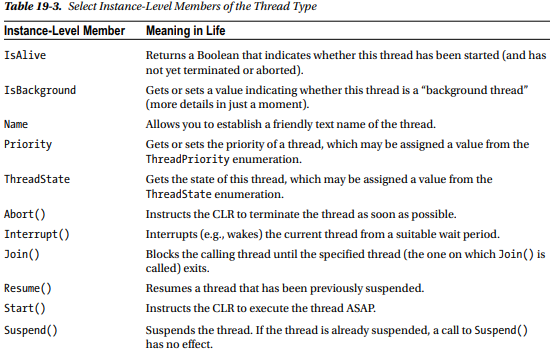
**The Role of the AsyncCallback Delegate** Rather than polling a delegate to determine whether an asynchronously invoked method has completed, it would be more efficient to have the secondary thread inform the calling thread when the task is finished. When you want to enable this behavior, you will need 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

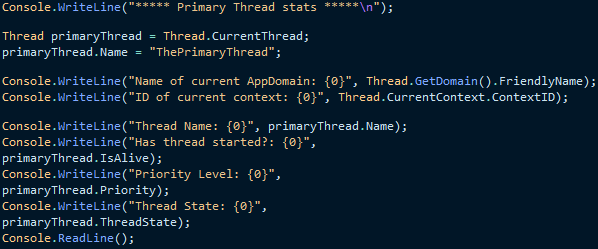
**The Role of the AsyncResult Class** Currently, the AddComplete() method is not printing the actual result of the operation (adding two numbers). The reason is that the target of the AsyncCallback delegate (AddComplete(), in this example) does not have access to the original BinaryOp delegate created in the scope of Main() and, therefore, you can’t call EndInvoke() from within AddComplete()! While you could simply declare the BinaryOp variable as a static member variable in the class to allow both methods to access the same object, a more elegant solution is to use the incoming IAsyncResult parameter.

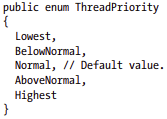
**Passing and Receiving Custom State Data** The final aspect of asynchronous delegates you need to address is the final argument to the BeginInvoke() method (which has been null up to this point). This parameter allows you to pass additional state information to the callback method from the primary thread.

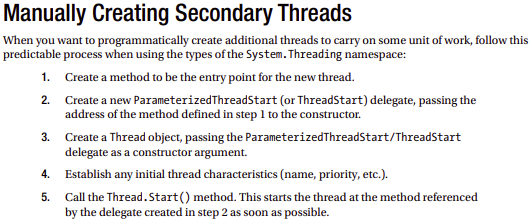
**The System.Threading Namespace** Under the .NET platform, the System.Threading namespace provides a number of types that enable the direct construction of multithreaded applications. In addition to providing types that allow you to interact with a particular CLR thread, this namespace defines types that allow access to the CLR- maintained thread pool, a simple (non-GUI-based) Timer class, and numerous types used to provide synchronized access to shared resources. Table 19-1 lists some of the important members of this namespace. (Be sure to consult the .NET Framework 4.6 SDK documentation for full details.

**The System.Threading.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 instance level) that allow you to create new threads within the current AppDomain, as well as to suspend, stop, and destroy a particular thread.

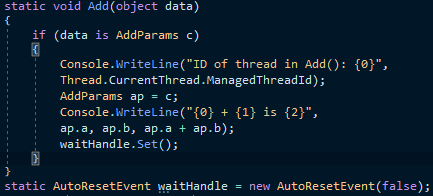
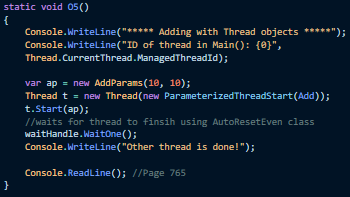
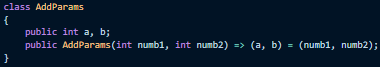
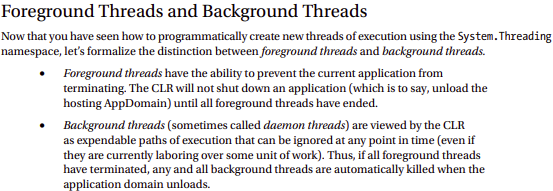
**Obtaining Statistics About the Current Thread of Execution** Recall that the entry point of an executable assembly (i.e., the Main() method) runs on the primary thread of execution. To illustrate the basic use of the Thread type, assume you have a new Console Application project named ThreadStats. As you know, the static Thread.CurrentThread property retrieves a Thread object that represents the currently executing thread

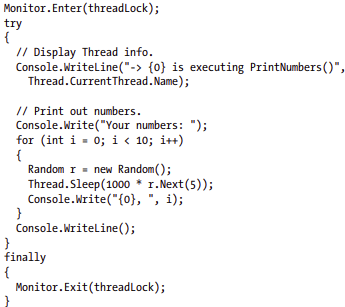
**The Name Property** While this code is more or less self-explanatory, do notice that the Thread class supports a property called Name. If you do not set this value, Name will return an empty string. However, once you assign a friendly string moniker to a given Thread object, you can greatly simplify your debugging endeavors. If you are using Visual Studio, you may access the Threads window during a debugging session (select Debug ➤ Windows ➤ Threads). As you can see from Figure 19-1, you can quickly identify the thread you want to diagnose.

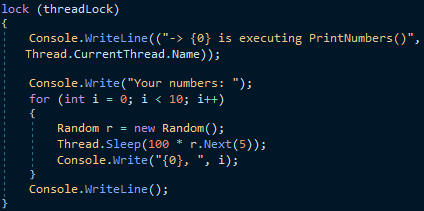


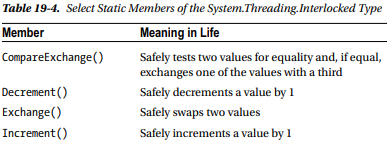
**Working with the ThreadStart Delegate** To illustrate the process of building a multithreaded application (as well as to demonstrate the usefulness of doing so), assume you have a Console Application project (SimpleMultiThreadApp) that allows the end user to choose whether the application will perform its duties using the single primary thread or split its workload using two separate threads of execution.

**Working with the ParameterizedThreadStart** **Delegate** Recall that the ThreadStart delegate can point only to methods that return void and take no arguments. While this might fit the bill in some cases, if you want to pass data to the method executing on the secondary thread, you will need to use the ParameterizedThreadStart delegate type. To illustrate, let’s re-create the logic of the AsyncCallbackDelegate project created earlier in this chapter, this time using the ParameterizedThreadStart delegate type.

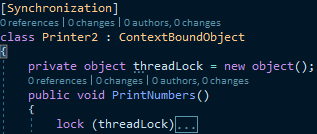
**The AutoResetEvent Class** In these first few examples, you have used a few crude ways to inform the primary thread to wait until the secondary thread has completed. During your examination of asynchronous delegates, you used a simple bool variable as a toggle; however, this is not a recommended solution, as both threads can access the same point of data, and this can lead to data corruption. A safer but still undesirable alternative is to call Thread.Sleep() for a fixed amount of time. The problem here is that you don’t want to wait longer than necessary.

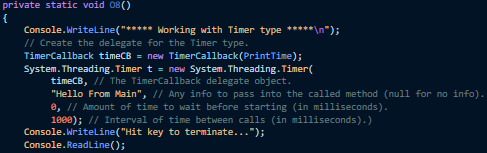
**The Issue of Concurrency** When you build multithreaded applications, your program needs to ensure that any piece of shared data is protected against the possibility of numerous threads changing its value. Given that all threads in an AppDomain have concurrent access to the shared data of the application, imagine what might happen if multiple threads were accessing the same point of data. As the thread scheduler will force threads to suspend their work at random, what if thread A is kicked out of the way before it has fully completed its work? Thread B is now reading unstable data.

**Synchronization Using the C# lock Keyword** The first technique you can use to synchronize access to shared resources is the C# lock keyword. This keyword allows you to define a scope of statements that must be synchronized between threads. By doing so, incoming threads cannot interrupt the current thread, thus preventing it from finishing its work.

**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.

**Synchronization Using the System.Threading.Interlocked Type** Although it always is hard to believe until you look at the underlying CIL code, assignments and simple arithmetic operations are not atomic. For this reason, the System.Threading namespace provides a type that allows you to operate on a single point of data atomically with less overhead than with the Monitor type. T

**Synchronization Using the [Synchronization] Attribute** The final synchronization primitive examined here is the [Synchronization] attribute, which is a member of the System.Runtime.Remoting.Contexts namespace. In essence, this class-level attribute effectively locks down all instance member code of the object for thread safety. When the CLR allocates objects attributed with [Synchronization], it will place the object within a synchronized context.

In some ways, this approach can be seen as the lazy way to write thread-safe code, given that you 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.

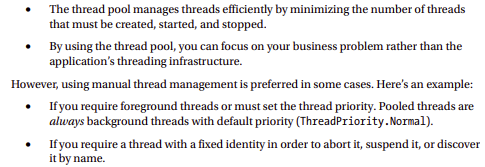
**Programming with Timer Callbacks** Many applications have the need to call a specific method during regular intervals of time. For example, you might have an application that needs to display the current time on a status bar via a given helper function. As another example, you might want to have your 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, you can use the System.Threading.Timer type in conjunction with a related delegate named TimerCallback.

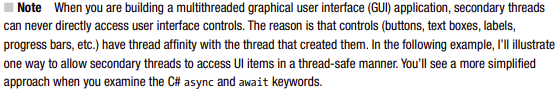
**Understanding the CLR ThreadPool** The next thread-centric topic you will examine in this chapter is the role of the CLR thread pool. When you invoke a method asynchronously using delegate types (via the BeginInvoke() method), the CLR does not literally create a new thread. For purposes of efficiency, a delegate’s BeginInvoke() method leverages a pool of worker threads that is maintained by the runtime. To allow you to interact with this pool of waiting threads, the System.Threading namespace provides the ThreadPool class type.

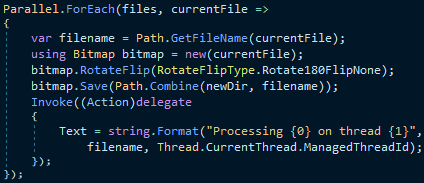
**Parallel Programming Using the Task Parallel Library** At this point in the chapter, you have examined two programming techniques (using asynchronous delegates and via the members of System.Threading) that allow you to build multithreaded software. Recall that both of these approaches will work under any version of the .NET platform.

**The System.Threading.Tasks Namespace** Collectively speaking, the types of System.Threading.Tasks are referred to as the Task Parallel Library. The TPL will automatically distribute your application’s workload across available CPUs dynamically, using the CLR thread pool. The TPL handles the partitioning of the work, thread scheduling, state management, and other low-level details.

**The Role of the Parallel Class** A key class of the TPL is System.Threading.Tasks.Parallel. This class supports a number of methods that allow you to iterate over a collection of data (specifically, an object implementing IEnumerable) in a parallel fashion. If you were to look up the Parallel class in the .NET Framework 4.6 SDK documentation, you would see that this class supports two primary static methods, Parallel.For() and Parallel.ForEach(), each of which defines numerous overloaded versions

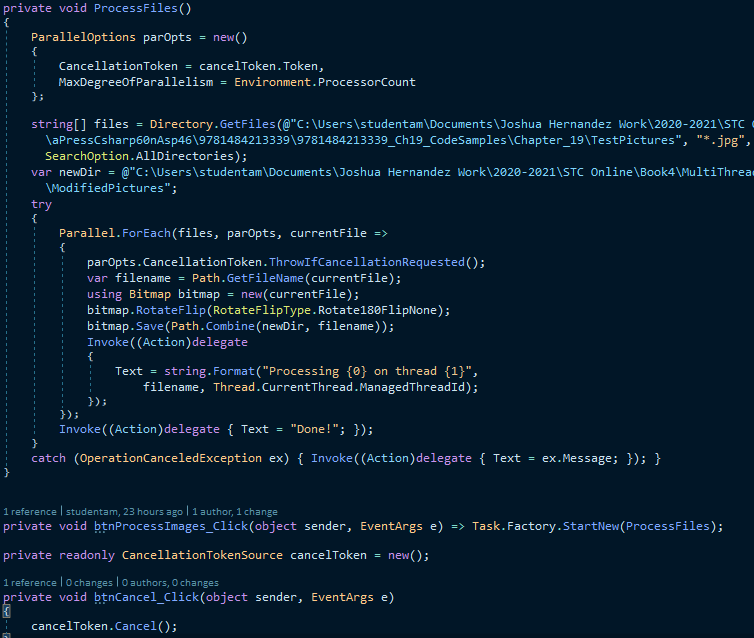
**Data Parallelism with the Parallel Class** The first way to use the TPL is to perform data parallelism. Simply put, this term refers to the task of iterating over an array or collection in a parallel manner using the Parallel.For() or Parallel.ForEach() method. Assume you need to perform some labor-intensive file I/O operations. Specifically, you need to load a large number of \*.jpg files into memory, flip them upside down, and save the modified image data to a new location.

**Accessing UI Elements on Secondary Threads**

You’ll notice that I’ve commented out the previous line of code that updated the caption of the main window with the ID of the currently executing thread. As noted previously, GUI controls have “thread affinity” with the thread that created it. If secondary threads attempt to access a control they did not directly create, you are bound to run into runtime errors when debugging your software. On the flip side, if you were to run the application (via Ctrl+F5), you might not ever find any problems whatsoever with the original code.

**The Task Class** The Task class allows you to easily invoke a method on a secondary thread and can be used as a simple alternative to working with asynchronous delegates.

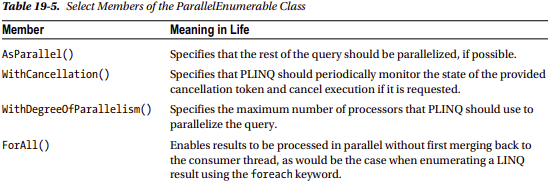




**Handling Cancellation Request** One improvement you can make to the current example is to provide a way for the user to stop the processing of the image data, via a second (aptly named) Cancel button. Thankfully, the Parallel.For() and Parallel.ForEach() methods both support cancellation through the use of cancellation tokens. When you invoke methods on Parallel, you can pass in a ParallelOptions object, which in turn contains a CancellationTokenSource object.

**Task Parallelism Using the Parallel Class** In addition to data parallelism, the TPL can also be used to easily fire off any number of asynchronous tasks using the Parallel.Invoke() method. This approach is a bit more straightforward than using delegates or members from System.Threading; however, if you require more control over the way tasks are executed, you could forgo use of Parallel.Invoke() and use the Task class directly, as you did in the previous example.

**The Parallel.Invoke()** method expects a parameter array of Action<> delegates, which you have supplied indirectly using lambda expressions. Again, while the output is identical, the benefit is that the TPL will now use all possible processors on the machine to invoke each method in parallel if possible.

**Parallel LINQ Queries (PLINQ)** To wrap up your look at the TPL, be aware that there is another way you can incorporate parallel tasks into your .NET applications. If you choose, you can use a set of extension methods, which allow you to construct a LINQ query that will perform its workload in parallel (if possible). Fittingly, LINQ queries that are designed to run in parallel are termed PLINQ queries. Like parallel code authored using the Parallel class, PLINQ has the option of ignoring your request to process the collection in parallel if need be. The PLINQ framework has been optimized in numerous ways, which includes determining whether a query would, in fact, perform faster in a synchronous manner. At runtime, PLINQ analyzes the overall structure of the query, and if the query is likely to benefit from parallelization, it will run concurrently. However, if parallelizing a query would hurt performance, PLINQ just runs the query sequentially. If PLINQ has a choice between a potentially expensive parallel algorithm or an inexpensive sequential algorithm, it chooses the sequential algorithm by default.

