CHAPTER 1

Manage program flow

If you could build only programs that execute all their logic from top to bottom, it would not be feasible to build complex applications. Fortunately, C# and the .NET Framework offer you a lot of options for creating complex programs that don't have a fixed program flow.

This chapter starts with looking at how to create *multithreaded* applications. Those applications can scale well and remain responsive to the user while doing their work. You will also look at the new language feature *async/await* that was added to C# 5.

IMPORTANT

Have you read page xxi?

It contains valuable information regarding the skills you need to pass the exam.

You will learn about the basic C# language constructs to make decisions and execute a piece of code multiple times, depending on the circumstances. These constructs form the basic language blocks of each application, and you will use them often.

After that, you will learn how to create applications that are loosely coupled by using *delegates* and *events*. With events, you can build objects that can notify each other when something happens and that can respond to those notifications. Frameworks such as ASP.NET, Windows Presentation Foundation (WPF), and WinForms make heavy use of events; understanding events thoroughly will help you build great applications.

Unfortunately, your program flow can also be interrupted by errors. Such errors can happen in areas that are out of your control but that you need to respond to. Sometimes you want to raise such an error yourself. You will learn how exceptions can help you implement a robust error-handling strategy in your applications.

Objectives in this chapter:

- Objective 1.1: Implement multithreading and asynchronous processing
- Objective 1.2: Manage multithreading
- Objective 1.3: Implement program flow
- Objective 1.4: Create and implement events and callbacks
- Objective 1.5. Implement exception handling

Objective 1.1: Implement multithreading and asynchronous processing

Applications are becoming more and more complex as user expectations rise. To fully take advantage of multicore systems and stay responsive, you need to create applications that use multiple threads, often called *parallelism*.

The .NET Framework and the C# language offer a lot of options that you can use to create multithreaded applications.

This objective covers how to:

- Understand threads.
- Use the Task Parallel Library.
- Use the *Parallel* class.
- Use the new *async* and *await* keywords.
- Use Parallel Language Integrated Query.
- Use concurrent collections.

Understanding threads

Imagine that your computer has only one *central processing unit* (CPU) that is capable of executing only one operation at a time. Now, imagine what would happen if the CPU has to work hard to execute a task that takes a long time.

While this operation runs, all other operations would be paused. This means that the whole machine would freeze and appear unresponsive to the user. Things get even worse when that long-running operation contains a bug so it never ends. Because the rest of the machine is unusable, the only thing you can do is restart the machine.

To remedy this problem, the concept of a *thread* is used. In current versions of Windows, each application runs in its own *process*. A process isolates an application from other applications by giving it its own *virtual memory* and by ensuring that different processes can't influence each other. Each process runs in its own *thread*. A *thread* is something like a virtualized CPU. If an application crashes or hits an infinite loop, only the application's process is affected.

Windows must manage all of the threads to ensure they can do their work. These management tasks do come with an overhead. Each thread is allowed by Windows to execute for a certain time period. After this period ends, the thread is paused and Windows switches to another thread. This is called *context switching*.

In practice, this means that Windows has to do some work to make it happen. The current thread is using a certain area of memory; it uses CPU registers and other state data, and Windows has to make sure that the whole context of the thread is saved and restored on each switch.

But although there are certain performance hits, using threads does ensure that each process gets its time to execute without having to wait until all other operations finish. This improves the responsiveness of the system and gives the illusion that one CPU can execute multiple tasks at a time. This way you can create an application that uses *parallelism*, meaning that it can execute multiple threads on different CPUs in parallel.

Almost any device that you buy today has a CPU with multiple cores, which is similar to having multiple CPUs. Some servers not only have multicore CPUs but they also have more than one CPU. To make use of all these cores, you need multiple threads. Windows ensures that those threads are distributed over your available cores. This way you can perform multiple tasks at once and improve scalability.

Because of the associated overhead, you should carefully determine whether you need multithreading. But if you want to use threads for scalability or responsiveness, C# and .NET Framework offer you a lot of possibilities.

Using the *Thread* class

The *Thread* class can be found in the *System.Threading* namespace. This class enables you to create new treads, manage their priority, and get their status.

The *Thread* class isn't something that you should use in your applications, except when you have special needs. However, when using the *Thread* class you have control over all configuration options. You can, for example, specify the priority of your thread, tell Windows that your thread is long running, or configure other advanced options.

Listing 1-1 shows an example of using the *Thread* class to run a method on another thread. The *Console* class synchronizes the use of the output stream for you so you can write to it from multiple threads. *Synchronization* is the mechanism of ensuring that two threads don't execute a specific portion of your program at the same time. In the case of a console application, this means that no two threads can write data to the screen at the exact same time. If one thread is working with the output stream, other threads will have to wait before it's finished.

LISTING 1-1 Creating a thread with the *Thread* class

```
using System;
using System.Threading;

namespace Chapter1
{
    public static class Program
    {
        public static void ThreadMethod()
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine(“ThreadProc: {0}”, i);
                Thread.Sleep(0);
            }
        }

        public static void Main()
        {
            Thread t = new Thread(new ThreadStart(ThreadMethod));
            t.Start();

            for (int i = 0; i < 4; i++)
            {
                Console.WriteLine(“Main thread: Do some work.”);
                Thread.Sleep(0);
            }

            t.Join();

        }

    }

}

// Displays
//Main thread: Do some work.
//ThreadProc: 0
//Main thread: Do some work.
//ThreadProc: 1
//Main thread: Do some work.
//ThreadProc: 2
//Main thread: Do some work.
//ThreadProc: 3
//ThreadProc: 4
//ThreadProc: 5
//ThreadProc: 6
//ThreadProc: 7
//ThreadProc: 8
//ThreadProc: 9
//ThreadProc: 10
```

As you can see, both threads run and print their message to the console. The *Thread.Join* method is called on the main thread to let it wait until the other thread finishes.

Why the *Thread.Sleep(0)*? It is used to signal to Windows that this thread is finished. Instead of waiting for the whole time-slice of the thread to finish, it will immediately switch to another thread.

Both your process and your thread have a *priority*. Assigning a low priority is useful for applications such as a screen saver. Such an application shouldn't compete with other applications for CPU time. A higher-priority thread should be used only when it's absolutely necessary. A new thread is assigned a priority of Normal, which is okay for almost all scenarios.

Another thing that's important to know about threads is the difference between *foreground* and *background* threads. Foreground threads can be used to keep an application alive. Only when all foreground threads end does the common language runtime (CLR) shut down your application. Background threads are then terminated.

Listing 1-2 shows this difference in action.

LISTING 1-2 Using a background thread

```
using System;
using System.Threading;

namespace Chapter1
{
    public static class Program

    {
        public static void ThreadMethod()
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("ThreadProc: {0}", i);
                Thread.Sleep(1000);
            }
        }

        public static void Main()
        {
            Thread t = new Thread(new ThreadStart(ThreadMethod));
            t.IsBackground = true;
            t.Start();
        }
    }
}
```

If you run this application with the *IsBackground* property set to *true*, the application exits immediately. If you set it to *false* (creating a foreground thread), the application prints the *ThreadProc* message ten times.

The *Thread* constructor has another overload that takes an instance of a *ParameterizedThreadStart* delegate. This overload can be used if you want to pass some data through the start method of your thread to your worker method, as Listing 1-3 shows.

LISTING 1-3 Using the *ParameterizedThreadStart*

```
public static void ThreadMethod(object o)
{
    for (int i = 0; i < (int)o; i++)
    {
        Console.WriteLine("ThreadProc: {0}", i);

        Thread.Sleep(0);
    }
}

public static void Main()
{
    Thread t = new Thread(new ParameterizedThreadStart(ThreadMethod));
    t.Start(5);
    t.Join();
}
```

In this case, the value *5* is passed to the *ThreadMethod* as an object. You can cast it to the expected type to use it in your method.

To stop a thread, you can use the *Thread.Abort* method. However, because this method is executed by another thread, it can happen at any time. When it happens, a *ThreadAbortException* is thrown on the target thread. This can potentially leave a corrupt state and make your application unusable.

A better way to stop a thread is by using a shared variable that both your target and your calling thread can access. Listing 1-4 shows an example.

LISTING 1-4 Stopping a thread

```
using System;
using System.Threading;

namespace Chapter1
{
    public static class Program
    {
        public static void ThreadMethod(object o)
        {
            for (int i = 0; i < (int)o; i++)
            {
                Console.WriteLine("ThreadProc: {0}", i);
                Thread.Sleep(0);
            }
        }

        public static void Main()
        {

            bool stopped = false;
            Thread t = new Thread(new ThreadStart(() =>
            {
                while (!stopped)
                {
                    Console.WriteLine("Running...");
                    Thread.Sleep(1000);
                }
            }));

            t.Start();
            Console.WriteLine("Press any key to exit");
            Console.ReadKey();

            stopped  = true;
            t.Join();
        }
    }
}
```

In this case, the thread is initialized with a lambda expression (which in turn is just a shorthand version of a delegate). The thread keeps running until *stopped* becomes *true*. After that, the *t.Join* method causes the console application to wait till the thread finishes execution.

A thread has its own call stack that stores all the methods that are executed. Local variables are stored on the call stack and are private to the thread.

A thread can also have its own data that's not a local variable. By marking a field with the *ThreadStatic* attribute, each thread gets its own copy of a field (see Listing 1-5).

LISTING 1-5 Using the *ThreadStaticAttribute*

```
using System;
using System.Threading;

namespace Chapter1
{
    public static class Program
    {
        [ThreadStatic]
        public static int _field;
        public static void Main()
        {

            new Thread(() =>
               {
                    for(int x = 0; x < 10; x++)
                    {
                        _field++;
                        Console.WriteLine("Thread A: {0}", _field);
                    }
                }).Start();


            new Thread(() =>
                {
                    for(int x = 0; x < 10; x++)
                    {
                        _field++;
                        Console.WriteLine("Thread B: {0}", _field);
                    }
                }).Start();

            Console.ReadKey();
        }
    }
}
```

With the *ThreadStaticAttribute* applied, the maximum value of *_field* becomes *10*. If you remove it, you can see that both threads access the same value and it becomes *20*.

If you want to use local data in a thread and initialize it for each thread, you can use the *ThreadLocal<T>* class. This class takes a delegate to a method that initializes the value. Listing 1-6 shows an example.

LISTING 1-6 Using *ThreadLocal<T>*

```
using System;
using System.Threading;

namespace Chapter1
{
    public static class Program
    {
        public static ThreadLocal<int> _field =
            new ThreadLocal<int>(() =>
            {
                return Thread.CurrentThread.ManagedThreadId;

            });

        public static void Main()
        {
            new Thread(() =>

                {

                    for(int x = 0; x < _field.Value; x++)
                    {
                        Console.WriteLine("Thread A: {0}", x);
                    }

                }).Start();
            new Thread(() =>
                {
                    for (int x = 0; x < _field.Value; x++)
                    {
                        Console.WriteLine("Thread B: {0}", x);
                    }
                }).Start();

            Console.ReadKey();
        }
    }
}

// Displays
// Thread B: 0
// Thread B: 1
// Thread B: 2
// Thread B: 3
// Thread A: 0
// Thread A: 1
// Thread A: 2
```

Here you see another feature of the .NET Framework. You can use the *Thread.CurrentThread* class to ask for information about the thread that's executing. This is called the thread's *execution context*. This property gives you access to properties like the thread's *current culture* (a *CultureInfo* associated with the current thread that is used to format dates, times, numbers, currency values, the sorting order of text, casing conventions, and string comparisons), *principal* (representing the current security context), *priority* (a value to indicate how the thread should be scheduled by the operating system), and other info.

When a thread is created, the runtime ensures that the initiating thread's execution context is flowed to the new thread. This way the new thread has the same privileges as the parent thread.

This copying of data does cost some resources, however. If you don't need this data, you can disable this behavior by using the *ExecutionContext.SuppressFlow* method.

Thread pools

When working directly with the *Thread* class, you create a new thread each time, and the thread dies when you're finished with it. The creation of a thread, however, is something that costs some time and resources.

A *thread pool* is created to reuse those threads, similar to the way a database connection pooling works. Instead of letting a thread die, you send it back to the pool where it can be reused whenever a request comes in.

When you work with a thread pool from .NET, you queue a work item that is then picked up by an available thread from the pool. Listing 1-7 shows how this is done.

LISTING 1-7 Queuing some work to the thread pool

```
using System;
using System.Threading;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            ThreadPool.QueueUserWorkItem((s) =>
            {
                Console.WriteLine("Working on a thread from threadpool");
            });

            Console.ReadLine();
        }
    }
}
```

Because the thread pool limits the available number of threads, you do get a lesser degree of parallelism than using the regular *Thread* class. But the thread pool also has many advantages.

Take, for example, a web server that serves incoming requests. All those requests come in at an unknown time and frequency. The thread pool ensures that each request gets added to the queue and that when a thread becomes available, it is processed. This ensures that your server doesn't crash under the amount of requests. If you span threads manually, you can easily bring down your server if you get a lot of requests. Each request has unique characteristics in the work they need to do. What the thread pool does is map this work onto the threads available in the system. Of course, you can still get so many requests that you run out of threads. Requests then start to queue up and this leads to your web server becoming unresponsive.

The thread pool automatically manages the amount of threads it needs to keep around. When it is first created, it starts out empty. As a request comes in, it creates additional threads to handle those requests. As long as it can finish an operation before a new one comes in, no new threads have to be created. If new threads are no longer in use after some time, the thread pool can kill those threads so they no longer use any resources.

MORE INFO **THREAD POOL**

For more information on how the thread pool works and how you can configure it, see *http://msdn.microsoft.com/en-us/library/system.threading.threadpool.aspx*.

One thing to be aware of is that because threads are being reused, they also reuse their local state. You may not rely on state that can potentially be shared between multiple operations.

Using *Tasks*

Queuing a work item to a thread pool can be useful, but it has its shortcomings. There is no built-in way to know when the operation has finished and what the return value is.

This is why the .NET Framework introduces the concept of a *Task*, which is an object that represents some work that should be done. The *Task* can tell you if the work is completed and if the operation returns a result, the *Task* gives you the result.

A *task scheduler* is responsible for starting the *Task* and managing it. By default, the *Task* scheduler uses threads from the thread pool to execute the *Task*.

Tasks can be used to make your application more responsive. If the thread that manages the user interface offloads work to another thread from the thread pool, it can keep processing user events and ensure that the application can still be used. But it doesn't help with scalability. If a thread receives a web request and it would start a new *Task*, it would just consume another thread from the thread pool while the original thread waits for results.

Executing a *Task* on another thread makes sense only if you want to keep the user interface thread free for other work or if you want to parallelize your work on to multiple processors.

Listing 1-8 shows how to start a new *Task* and wait until it's finished.

LISTING 1-8 Starting a new *Task*

```
using System;
using System.Threading.Tasks;

namespace Chapter1
{

    public static class Program
    {
        public static void Main()

        {

            Task t = Task.Run(() =>
            {
                for (int x = 0; x < 100; x++)
                {

                    Console.Write('*');
                }
            });


            t.Wait();

        }
    }
}
```

This example creates a new *Task* and immediately starts it. Calling *Wait* is equivalent to calling *Join* on a thread. It waits till the *Task* is finished before exiting the application.

Next to *Task*, the .NET Framework also has the *Task<T>* class that you can use if a *Task* should return a value. Listing 1-9 shows how this works.

LISTING 1-9 Using a *Task* that returns a value.

```
using System;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            Task<int> t = Task.Run(() =>
            {
                return 42;
            });
            Console.WriteLine(t.Result); // Displays 42
        }
    }
}
```

Attempting to read the *Result* property on a *Task* will force the thread that's trying to read the result to wait until the *Task* is finished before continuing. As long as the *Task* has not finished, it is impossible to give the result. If the *Task* is not finished, this call will block the current thread.

Because of the object-oriented nature of the *Task* object, one thing you can do is add a *continuation task*. This means that you want another operation to execute as soon as the *Task* finishes.

Listing 1-10 shows an example of creating such a continuation.

LISTING 1-10 Adding a continuation

```
Task<int> t = Task.Run(() =>
{
    return 42;
}).ContinueWith((i) =>
{
    return i.Result * 2;
});

Console.WriteLine(t.Result); // Displays 84
```

The *ContinueWith* method has a couple of overloads that you can use to configure when the continuation will run. This way you can add different continuation methods that will run when an exception happens, the *Task* is canceled, or the *Task* completes successfully. Listing 1-11 shows how to do this.

LISTING 1-11 Scheduling different continuation tasks

```
Task<int> t = Task.Run(() =>
{
    return 42;
});

t.ContinueWith((i) =>
{
    Console.WriteLine("Canceled");
}, TaskContinuationOptions.OnlyOnCanceled);

t.ContinueWith((i) =>
{
    Console.WriteLine("Faulted");
}, TaskContinuationOptions.OnlyOnFaulted);

var completedTask =  t.ContinueWith((i) =>
 {
     Console.WriteLine("Completed");
 }, TaskContinuationOptions.OnlyOnRanToCompletion);

completedTask.Wait();
```

Next to continuation *Tasks*, a *Task* can also have several *child Tasks*. The *parent Task* finishes when all the child tasks are ready. Listing 1-12 shows how this works.

LISTING 1-12 Attaching child tasks to a parent task

```
using System;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            Task<Int32[]> parent = Task.Run(() =>
            {
                var results = new Int32[3];
                new Task(() => results[0] = 0,
                    TaskCreationOptions.AttachedToParent).Start();
                new Task(() => results[1] = 1,
                    TaskCreationOptions.AttachedToParent).Start();
                new Task(() => results[2] = 2,
                    TaskCreationOptions.AttachedToParent).Start();

                return results;
            });

            var finalTask = parent.ContinueWith(
               parentTask => {
                   foreach(int i in parentTask.Result)
                       Console.WriteLine(i);
               });
            finalTask.Wait();
        }
    }
}
```

The *finalTask* runs only after the parent *Task* is finished, and the parent *Task* finishes when all three children are finished. You can use this to create quite complex *Task* hierarchies that will go through all the steps you specified.

In the previous example, you had to create three *Tasks* all with the same options. To make the process easier, you can use a *TaskFactory*. A *TaskFactory* is created with a certain configuration and can then be used to create *Tasks* with that configuration. Listing 1-13 shows how you can simplify the previous example with a factory.

LISTING 1-13 Using a *TaskFactory*

```
using System;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            Task<Int32[]> parent = Task.Run(() =>
            {
                var results = new Int32[3];

                TaskFactory tf = new TaskFactory(TaskCreationOptions.AttachedToParent,
                   TaskContinuationOptions.ExecuteSynchronously);

                tf.StartNew(() => results[0] = 0);
                tf.StartNew(() => results[1] = 1);
                tf.StartNew(() => results[2] = 2);
                return results;
            });

            var finalTask = parent.ContinueWith(
               parentTask => {
                   foreach(int i in parentTask.Result)
                   Console.WriteLine(i);
               });

            finalTask.Wait();
        }
    }
}
```

Next to calling *Wait* on a single *Task*, you can also use the method *WaitAll* to wait for multiple *Tasks* to finish before continuing execution. Listing 1-14 shows how to use this.

LISTING 1-14 Using *Task.WaitAll*

```
using System.Threading;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            Task[] tasks = new Task[3];

            tasks[0] = Task.Run(() => {
                                  Thread.Sleep(1000);
                                  Console.WriteLine("1");
                                  return 1;
                              });
            tasks[1] = Task.Run(() => {
                                  Thread.Sleep(1000);
                                  Console.WriteLine("2");
                                  return 2;
                              });
            tasks[2] = Task.Run(() => {
                                  Thread.Sleep(1000);
                                  Console.WriteLine("3");
                                  return 3; }
                              );

           Task.WaitAll(tasks);
        }
    }
}
```

In this case, all three *Tasks* are executed simultaneously, and the whole run takes approximately 1000ms instead of 3000. Next to *WaitAll*, you also have a *WhenAll* method that you can use to schedule a continuation method after all *Tasks* have finished.

Instead of waiting until all tasks are finished, you can also wait until one of the tasks is finished. You use the *WaitAny* method for this. Listing 1-15 shows how this works.

LISTING 1-15 Using *Task.WaitAny*

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            Task<int>[] tasks = new Task<int>[3];
            tasks[0] = Task.Run(() => { Thread.Sleep(2000); return 1; });
            tasks[1] = Task.Run(() => { Thread.Sleep(1000); return 2; });
            tasks[2] = Task.Run(() => { Thread.Sleep(3000); return 3; });

            while (tasks.Length > 0)
            {
                int i = Task.WaitAny(tasks);
                Task<int> completedTask = tasks[i];

                Console.WriteLine(completedTask.Result);

                var temp = tasks.ToList();
                temp.RemoveAt(i);
                tasks = temp.ToArray();

            }
        }
    }
}
```

In this example, you process a completed *Task* as soon as it finishes. By keeping track of which *Tasks* are finished, you don't have to wait until all *Tasks* have completed.

Using the *Parallel* class

The *System.Threading.Tasks* namespace also contains another class that can be used for parallel processing. The *Parallel* class has a couple of static methods—*For*, *ForEach*, and *Invoke*—that you can use to parallelize work.

Parallelism involves taking a certain task and splitting it into a set of related tasks that can be executed concurrently. This also means that you shouldn't go through your code to replace all your loops with parallel loops. You should use the *Parallel* class only when your code doesn't have to be executed sequentially.

Increasing performance with parallel processing happens only when you have a lot of work to be done that can be executed in parallel. For smaller work sets or for work that has to synchronize access to resources, using the *Parallel* class can hurt performance.

The best way to know whether it will work in your situation is to measure the results.

Listing 1-16 shows an example of using *Parallel.For* and *Parallel.ForEach*.

LISTING 1-16 Using *Parallel.For* and *Parallel.Foreach*

```
Parallel.For(0, 10, i =>
{
    Thread.Sleep(1000);
});

var numbers = Enumerable.Range(0, 10);
Parallel.ForEach(numbers, i =>
{
    Thread.Sleep(1000);
});
```

You can cancel the loop by using the *ParallelLoopState* object. You have two options to do this: *Break* or *Stop*. *Break* ensures that all iterations that are currently running will be finished. *Stop* just terminates everything. Listing 1-17 shows an example.

LISTING 1-17 Using *Parallel.Break*

```
ParallelLoopResult result = Parallel.
    For(0, 1000, (int i, ParallelLoopState loopState) =>
{
    if (i == 500)
    {
          Console.WriteLine("Breaking loop");
       loopState.Break();

 }
    return;
});
```

When breaking the parallel loop, the result variable has an *IsCompleted* value of *false* and a *LowestBreakIteration* of *500*. When you use the *Stop* method, the *LowestBreakIteration* is *null*.

Using *async* and *await*

As you have seen, long-running CPU-bound tasks can be handed to another thread by using the *Task* object. But when doing work that's input/output (I/O)–bound, things go a little differently.

When your application is executing an I/O operation on the primary application thread, Windows notices that your thread is waiting for the I/O operation to complete. Maybe you are accessing some file on disk or over the network, and this could take some time.

Because of this, Windows pauses your thread so that it doesn't use any CPU resources. But while doing this, it still uses memory, and the thread can't be used to serve other requests, which in turn will lead to new threads being created if requests come in.

Asynchronous code solves this problem. Instead of blocking your thread until the I/O operation finishes, you get back a *Task* object that represents the result of the asynchronous operation. By setting a continuation on this *Task*, you can continue when the I/O is done. In the meantime, your thread is available for other work. When the I/O operation finishes, Windows notifies the runtime and the continuation *Task* is scheduled on the thread pool.

But writing asynchronous code is not easy. You have to make sure that all edge cases are handled and that nothing can go wrong. Because of this predicament, C# 5 has added two new keywords to simplify writing asynchronous code. Those keywords are *async* and *await*.

You use the *async* keyword to mark a method for asynchronous operations. This way, you signal to the compiler that something asynchronous is going to happen. The compiler responds to this by transforming your code into a *state machine*.

A method marked with *async* just starts running synchronously on the current thread. What it does is enable the method to be split into multiple pieces. The boundaries of these pieces are marked with the *await* keyword.

When you use the *await* keyword, the compiler generates code that will see whether your asynchronous operation is already finished. If it is, your method just continues running synchronously. If it's not yet completed, the state machine will hook up a continuation method that should run when the *Task* completes. Your method yields control to the calling thread, and this thread can be used to do other work.

Listing 1-18 shows a simple example of an asynchronous method.

LISTING 1-18 *async* and *await*

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

namespace Chapter1.Threads
{
    public static class Program
    {
        public static void Main()
        {
            string result = DownloadContent().Result;
            Console.WriteLine(result);
        }

        public static async Task<string> DownloadContent()
        {
            using(HttpClient client = new HttpClient())
            {

                string result = await client.GetStringAsync("http://www.microsoft.com");
                return result;
           }
        }
    }

}
```

Because the entry method of an application can't be marked as *async*, the example uses the *Wait* method in *Main*. This class uses both the *async* and *await* keywords in the *DownloadContent* method.

The *GetStringAsync* uses asynchronous code internally and returns a *Task<string>* to the caller that will finish when the data is retrieved. In the meantime, your thread can do other work.

The nice thing about *async* and *await* is that they let the compiler do the thing it's best at: generate code in precise steps. Writing correct asynchronous code by hand is difficult, especially when trying to implement exception handling. Doing this correctly can become difficult quickly. Adding continuation tasks also breaks the logical flow of the code. Your code doesn't read top to bottom anymore. Instead, program flow jumps around, and it's harder to follow when debugging your code. The *await* keyword enables you to write code that looks synchronous but behaves in an asynchronous way. The Visual Studio debugger is even clever enough to help you in debugging asynchronous code as if it were synchronous.

So doing a CPU-bound task is different from an I/O-bound task. CPU-bound tasks always use some thread to execute their work. An asynchronous I/O-bound task doesn't use a thread until the I/O is finished.

If you are building a client application that needs to stay responsive while background operations are running, you can use the *await* keyword to offload a long-running operation to another thread. Although this does not improve performance, it does improve responsiveness. The *await* keyword also makes sure that the remainder of your method runs on the correct user interface thread so you can update the user interface.

Making a scalable application that uses fewer threads is another story. Making code scale better is about changing the actual implementation of the code. Listing 1-19 shows an example of this.

LISTING 1-19 Scalability versus responsiveness

```
public Task SleepAsyncA(int millisecondsTimeout)
{
    return Task.Run(() => Thread.Sleep(millisecondsTimeout));
}

public Task SleepAsyncB(int millisecondsTimeout)
{
    TaskCompletionSource<bool> tcs = null;
    var t = new Timer(delegate { tcs.TrySetResult(true); }, null, -1, -1);
    tcs = new TaskCompletionSource<bool>(t);
    t.Change(millisecondsTimeout, -1);
    return tcs.Task;
}
```

The *SleepAsyncA* method uses a thread from the thread pool while sleeping. The second method, however, which has a completely different implementation, does not occupy a thread while waiting for the timer to run. The second method gives you scalability.

When using the *async* and *await* keywords, you should keep this in mind. Just wrapping each and every operation in a task and awaiting them won't make your application perform any better. It could, however, improve responsiveness, which is very important in client applications.

The *FileStream* class, for example, exposes asynchronous methods such as *WriteAsync* and *ReadAsync*. They use an implementation that makes use of actual asynchronous I/O. This way, they don't use a thread while they are waiting on the hard drive of your system to read or write some data.

When an exception happens in an asynchronous method, you normally expect an *AggregateException*. However, the generated code helps you unwrap the *AggregateException* and throws the first of its inner exceptions. This makes the code more intuitive to use and easier to debug.

One other thing that's important when working with asynchronous code is the concept of a *SynchronizationContext*, which connects its application model to its threading model. For example, a WPF application uses a single user interface thread and potentially multiple background threads to improve responsiveness and distribute work across multiple CPUs. An ASP.NET application, however, uses threads from the thread pool that are initialized with the correct data, such as current user and culture to serve incoming requests.

The *SynchronizationContext* abstracts the way these different applications work and makes sure that you end up on the right thread when you need to update something on the UI or process a web request.

The *await* keyword makes sure that the current *SynchronizationContext* is saved and restored when the task finishes. When using *await* inside a WPF application, this means that after your *Task* finishes, your program continues running on the user interface thread. In an ASP.NET application, the remaining code runs on a thread that has the client's cultural, principal, and other information set.

If you want, you can disable the flow of the *SynchronizationContext*. Maybe your continuation code can run on any thread because it doesn't need to update the UI after it's finished. By disabling the *SynchronizationContext*, your code performs better. Listing 1-20 shows an example of a button event handler in a WPF application that downloads a website and then puts the result in a label.

LISTING 1-20 Using *ConfigureAwait*

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    HttpClient httpClient = new HttpClient();

    string content = await httpClient
        .GetStringAsync("http://www.microsoft.com")
        .ConfigureAwait(false);

    Output.Content = content;
}
```

This example throws an exception; the *Output.Content* line is not executed on the UI thread because of the *ConfigureAwait(false)*. If you do something else, such as writing the content to file, you don't need to set the *SynchronizationContext* to be set (see Listing 1-21).

LISTING 1-21 Continuing on a thread pool instead of the UI thread

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    HttpClient httpClient = new HttpClient();

    string content = await httpClient
        .GetStringAsync("http://www.microsoft.com")
        .ConfigureAwait(false);
    using (FileStream sourceStream = new FileStream("temp.html",
            FileMode.Create, FileAccess.Write, FileShare.None,
            4096, useAsync: true))
    {
        byte[] encodedText = Encoding.Unicode.GetBytes(content);
        await sourceStream.WriteAsync(encodedText, 0, encodedText.Length)
            .ConfigureAwait(false);
    };
}
```

Both *await*s use the *ConfigureAwait(false)* method because if the first method is already finished before the *awaiter* checks, the code still runs on the UI thread.

When creating *async* methods, it's important to choose a return type of *Task* or *Task<T>*. Avoid the *void* return type. A *void* returning *async* method is effectively a fire-and-forget method. You can never inspect the *return* type, and you can't see whether any exceptions were thrown. You should use *async void* methods only when dealing with asynchronous events.

The use of the new *async/await* keywords makes it much easier to write asynchronous code. In today's world with multiple cores and requirements for responsiveness and scalability, it's important to look for opportunities to use these new keywords to improve your applications.

EXAM TIP

When using *async* and *await* keep in mind that you should never have a method marked *async* without any *await* statements. You should also avoid returning void from an *async* method except when it's an event handler.

Using Parallel Language Integrated Query (PLINQ)

Language-Integrated Query (LINQ) is a popular addition to the C# language. You can use it to perform queries over all kinds of data.

Parallel Language-Integrated Query (PLINQ) can be used on objects to potentially turn a sequential query into a parallel one.

Extension methods for using PLINQ are defined in the *System.Linq.ParallelEnumerable* class. Parallel versions of LINQ operators, such as *Where, Select, SelectMany, GroupBy, Join, OrderBy, Skip*, and *Take,* can be used.

Listing 1-22 shows how you can convert a query to a parallel query.

LISTING 1-22 Using *AsParallel*

```
var numbers = Enumerable.Range(0, 100000000);
var parallelResult = numbers.AsParallel()
    .Where(i => i % 2 == 0)
    .ToArray();
```

The runtime determines whether it makes sense to turn your query into a parallel one. When doing this, it generates *Task* objects and starts executing them. If you want to force PLINQ into a parallel query, you can use the *WithExecutionMode* method and specify that it should always execute the query in parallel.

You can also limit the amount of parallelism that is used with the *WithDegreeOfParallelism* method. You pass that method an integer that represents the number of processors that you want to use. Normally, PLINQ uses all processors (up to 64), but you can limit it with this method if you want.

One thing to keep in mind is that parallel processing does not guarantee any particular order. Listing 1-23 shows what can happen.

LISTING 1-23 Unordered parallel query

```
using System;
using System.Linq;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            var numbers = Enumerable.Range(0, 10);
            var parallelResult = numbers.AsParallel()
                .Where(i => i % 2 == 0)
                .ToArray();

            foreach (int i in parallelResult)
                Console.WriteLine(i);
        }
    }
}

// Displays
// 2
// 0
// 4
// 6
// 8
```

As you can see, the returned results from this query are in no particular order. The results of this code vary depending on the amount of CPUs that are available. If you want to ensure that the results are ordered, you can add the *AsOrdered* operator. Your query is still processed in parallel, but the results are buffered and sorted. Listing 1-24 shows how this works.

LISTING 1-24 Ordered parallel query

```
using System;
using System.Linq;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            var numbers = Enumerable.Range(0, 10);
            var parallelResult = numbers.AsParallel().AsOrdered()
                .Where(i => i % 2 == 0)
                .ToArray();

            foreach (int i in parallelResult)
                Console.WriteLine(i);
        }
    }
}

// Displays
// 0
// 2
// 4
// 6
// 8
```

If you have a complex query that can benefit from parallel processing but also has some parts that should be done sequentially, you can use the *AsSequential* to stop your query from being processed in parallel.

One scenario where this is required is to preserve the ordering of your query. Listing 1-25 shows how you can use the *AsSequential* operator to make sure that the *Take* method doesn't mess up your order.

LISTING 1-25 Making a parallel query sequential

```
var numbers = Enumerable.Range(0, 20);

var parallelResult = numbers.AsParallel().AsOrdered()
    .Where(i => i % 2 == 0).AsSequential();

foreach (int i in parallelResult.Take(5))
    Console.WriteLine(i);

// Displays
// 0
// 2
// 4
// 6
// 8
```

When using PLINQ, you can use the *ForAll* operator to iterate over a collection when the iteration can also be done in a parallel way. Listing 1-26 shows how to do this.

LISTING 1-26 Using *ForAll*

```
var numbers = Enumerable.Range(0, 20);

var parallelResult = numbers.AsParallel()
    .Where(i => i % 2 == 0);

parallelResult.ForAll(e => Console.WriteLine(e));
```

In contrast to *foreach*, *ForAll* does not need all results before it starts executing. In this example, *ForAll* does, however, remove any sort order that is specified.

Of course, it can happen that some of the operations in your parallel query throw an exception. The .NET Framework handles this by aggregating all exceptions into one *Aggrega-teException*. This exception exposes a list of all exceptions that have happened during parallel execution. Listing 1-27 shows how you can handle this.

LISTING 1-27 Catching *AggregateException*

```
using System;
using System.Linq;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()

        {
            var numbers = Enumerable.Range(0, 20);

            try
            {

                var parallelResult = numbers.AsParallel()
                    .Where(i => IsEven(i));

                parallelResult.ForAll(e => Console.WriteLine(e));
            }
            catch (AggregateException e)
            {
                Console.WriteLine("There where {0} exceptions",
                                    e.InnerExceptions.Count);
            }
        }

        public static bool IsEven(int i)
        {
            if (i % 10 == 0) throw new ArgumentException("i");

            return i % 2 == 0;
        }
    }
}

// Displays
// 4
// 6
// 8
// 2
// 12
// 14
// 16
// 18
// There where 2 exceptions
```

As you can see, two exceptions were thrown while processing the data. You can inspect those exceptions by looping through the *InnerExceptions* property.

Using concurrent collections

When working in a multithreaded environment, you need to make sure that you are not manipulating shared data at the same time without synchronizing access.

The .NET Framework offers some collection classes that are created specifically for use in concurrent environments, which is what you have when you're using multithreading. These collections are thread-safe, which means that they internally use synchronization to make sure that they can be accessed by multiple threads at the same time.

Those collections are the following:

- *BlockingCollection<T>*
- *ConcurrentBag<T>*
- *ConcurrentDictionary<TKey,T>*
- *ConcurrentQueue<T>*
- *ConcurrentStack<T>*

BlockingCollection<T>

This collection is thread-safe for adding and removing data. Removing an item from the collection can be blocked until data becomes available. Adding data is fast, but you can set a maximum upper limit. If that limit is reached, adding an item blocks the calling thread until there is room.

BlockingCollection is in reality a wrapper around other collection types. If you don't give it any specific instructions, it uses the *ConcurrentQueue* by default.

A regular collection blows up when being used in a multithreaded scenario because an item might be removed by one thread while the other thread is trying to read it.

Listing 1-28 shows an example of using a *BlockingCollection*. One *Task* listens for new items being added to the collection. It blocks if there are no items available. The other *Task* adds items to the collection.

LISTING 1-28 Using *BlockingCollection<T>*

```
using System;
using System.Collections.Concurrent;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            BlockingCollection<string> col = new BlockingCollection<string>();
            Task read = Task.Run(() =>
                {
                    while (true)
                    {
                        Console.WriteLine(col.Take());
                    }
                });

            Task write = Task.Run(() =>
                {
                    while (true)
                    {
                        string s = Console.ReadLine();
                        if (string.IsNullOrWhiteSpace(s)) break;
                        col.Add(s);
                    }
                });

            write.Wait();
        }
    }
}
```

The program terminates when the user doesn't enter any data. Until that, every string entered is added by the write *Task* and removed by the read *Task*.

You can use the *CompleteAdding* method to signal to the *BlockingCollection* that no more items will be added. If other threads are waiting for new items, they won't be blocked anymore.

You can even remove the *while(true)* statements from Listing 1-28. By using the *GetConsumingEnumerable* method, you get an *IEnumerable* that blocks until it finds a new item. That way, you can use a *foreach* with your *BlockingCollection* to enumerate it (see Listing 1-29).

LISTING 1-29 Using *GetConsumingEnumerable* on a *BlockingCollection*

```
Task read = Task.Run(() =>
    {

        foreach (string v in col.GetConsumingEnumerable())
            Console.WriteLine(v);
    });
```

MORE INFO **IENUMERABLE**

For more information about using *IEnumerable*, see Chapter 2.

ConcurrentBag

A *ConcurrentBag* is just a bag of items. It enables duplicates and it has no particular order. Important methods are *Add*, *TryTake*, and *TryPeek*.

Listing 1-30 shows how to work with the *ConcurrentBag*.

LISTING 1-30 Using a *ConcurrentBag*

```
ConcurrentBag<int> bag = new ConcurrentBag<int>();

bag.Add(42);
bag.Add(21);

int result;
if (bag.TryTake(out result))
    Console.WriteLine(result);

if (bag.TryPeek(out result))
    Console.WriteLine("There is a next item: {0}", result);
```

One thing to keep in mind is that the *TryPeek* method is not very useful in a multithreaded environment. It could be that another thread removes the item before you can access it.

ConcurrentBag also implements *IEnumerable<T>*, so you can iterate over it. This operation is made thread-safe by making a snapshot of the collection when you start iterating it, so items added to the collection after you started iterating it won't be visible. Listing 1-31 shows this in practice.

LISTING 1-31 Enumerating a *ConcurrentBag*

```
ConcurrentBag<int> bag = new ConcurrentBag<int>();
Task.Run(() =>
{
    bag.Add(42);
    Thread.Sleep(1000);
    bag.Add(21);
});
Task.Run(() =>
{
    foreach (int i in bag)
        Console.WriteLine(i);
}).Wait();

// Displays
// 42
```

This code only displays *42* because the other value is added after iterating over the bag has started.

ConcurrentStack and *ConcurrentQueue*

A stack is a *last in, first out* (LIFO) collection. A queue is a *first in, first out* (FIFO) collection.

ConcurrentStack has two important methods: *Push* and *TryPop*. *Push* is used to add an item to the stack; *TryPop* tries to get an item off the stack. You can never be sure whether there are items on the stack because multiple threads might be accessing your collection at the same time.

You can also add and remove multiple items at once by using *PushRange* and *TryPopRange*. When you enumerate the collection, a snapshot is taken.

Listing 1-32 shows how these methods work.

LISTING 1-32 Using a *ConcurrentStack*

```
ConcurrentStack<int> stack = new ConcurrentStack<int>();

stack.Push(42);

int result;
if (stack.TryPop(out result))
    Console.WriteLine("Popped: {0}", result);

stack.PushRange(new int[] { 1, 2, 3 });

int[] values = new int[2];
stack.TryPopRange(values);


foreach (int i in values)
    Console.WriteLine(i);


// Popped: 42
// 3
// 2
```

ConcurrentQueue offers the methods *Enqueue* and *TryDequeue* to add and remove items from the collection. It also has a *TryPeek* method and it implements *IEnumerable* by making a snapshot of the data. Listing 1-33 shows how to use a *ConcurrentQueue*.

LISTING 1-33 Using a *ConcurrentQueue*.

```
ConcurrentQueue<int> queue = new ConcurrentQueue<int>();
queue.Enqueue(42);

int result;
if (queue.TryDequeue(out result))
    Console.WriteLine("Dequeued: {0}", result);

// Dequeued: 42
```

ConcurrentDictionary

A *ConcurrentDictionary* stores key and value pairs in a thread-safe manner. You can use methods to add and remove items, and to update items in place if they exist.

Listing 1-34 shows the methods that you can use on a *ConcurrentDictionary*.

LISTING 1-34 Using a *ConcurrentDictionary*

```
var dict = new ConcurrentDictionary<string, int>();
if (dict.TryAdd("k1", 42))
{
    Console.WriteLine("Added");
}

if (dict.TryUpdate("k1", 21, 42))
{
    Console.WriteLine("42 updated to 21");
}

dict["k1"] = 42; // Overwrite unconditionally

int r1 = dict.AddOrUpdate("k1", 3, (s, i) => i * 2);
int r2 = dict.GetOrAdd("k2", 3);
```

When working with a *ConcurrentDictionary* you have methods that can atomically add, get, and update items. An *atomic operation* means that it will be started and finished as a single step without other threads interfering. *TryUpdate* checks to see whether the current value is equal to the existing value before updating it. *AddOrUpdate* makes sure an item is added if it's not there, and updated to a new value if it is. *GetOrAdd* gets the current value of an item if it's available; if not, it adds the new value by using a factory method.

Thought experiment

Implementing multithreading

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You need to build a new application, and you look into multithreading capabilities. Your application consists of a client application that communicates with a web server.

1. **Explain how multithreading can help with your client application.**
2. **What is the difference between CPU and I/O bound operations?**
3. **Does using multithreading with the TPL offer the same advantages for your server application?**

Objective summary

- A thread can be seen as a virtualized CPU.
- Using multiple threads can improve responsiveness and enables you to make use of multiple processors.
- The *Thread* class can be used if you want to create your own threads explicitly. Otherwise, you can use the *ThreadPool* to queue work and let the runtime handle things.
- A *Task* object encapsulates a job that needs to be executed. Tasks are the recommended way to create multithreaded code.
- The *Parallel* class can be used to run code in parallel.
- PLINQ is an extension to LINQ to run queries in parallel.
- The new *async* and *await* operators can be used to write asynchronous code more easily.
- Concurrent collections can be used to safely work with data in a multithreaded (concurrent access) environment.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. You have a lot of items that need to be processed. For each item, you need to perform a complex calculation. Which technique should you use?

   A. You create a *Task* for each item and then wait until all tasks are finished.

   B. You use *Parallel.For* to process all items concurrently.

   C. You use *async/await* to process all items concurrently.

   D. You add all items to a *BlockingCollection* and process them on a thread created by the *Thread* class.

2. You are creating a complex query that doesn't require any particular order and you want to run it in parallel. Which method should you use?

   A. *AsParallel*

   B. *AsSequential*

   C. *AsOrdered*

   D. *WithDegreeOfParallelism*

3. You are working on an ASP.NET application that retrieves some data from another web server and then writes the response to the database. Should you use *async/await*?
   - **A.** No, both operations depend on external factors. You need to wait before they are finished.
   - **B.** No, in a server application you don't have to use *async/await*. It's only for responsiveness on the client.
   - **C.** Yes, this will free your thread to serve other requests while waiting for the I/O to complete.
   - **D.** Yes, this put your thread to sleep while waiting for I/O so that it doesn't use any CPU.

Objective 1.2: Manage multithreading

Although multithreading can give you a lot of advantages, it's not easy to write a multithreaded application. Problems can happen when different threads access some shared data. What should happen when both try to change something at the same time? To make this work successfully, *synchronizing* resources is important.

This objective covers how to:

- Synchronize resources.
- Cancel long-running tasks.

Synchronizing resources

As you have seen, with the TPL support in .NET, it's quite easy to create a multithreaded application. But when you build real-world applications with multithreading, you run into problems when you want to access the same data from multiple threads simultaneously. Listing 1-35 shows an example of what can go wrong.

LISTING 1-35 Accessing shared data in a multithreaded application

```
using System;
using System.Threading.Tasks;

namespace Chapter1
{
    public class Program
    {
        static void Main()
        {
            int n = 0;

            var up = Task.Run(() =>
            {
                for (int i = 0; i < 1000000; i++)
                    n++;
            });

            for (int i = 0; i < 1000000; i++)
                n--;

            up.Wait();
            Console.WriteLine(n);
        }
    }
}
```

What would the output of Listing 1-35 be? The answer is, it depends. When you run this application, you get a different output each time. The seemingly simple operation of incrementing and decrementing the variable *n* results in both a lookup (check the value of *n*) and add or subtract 1 from *n*. But what if the first task reads the value and adds 1, and at the exact same time task 2 reads the value and subtracts 1? This is what happens in this example and that's why you never get the expected output of *0*.

This is because the operation is not *atomic*. It consists of both a read and a write that happen at different moments. This is why access to the data you're working with needs to be *synchronized*, so you can reliably predict how your data is affected.

It's important to synchronize access to shared data. One feature the C# language offers is the *lock* operator, which is some syntactic sugar that the compiler translates in a call to *System.Thread.Monitor*. Listing 1-36 shows the use of the *lock* operator to fix the previous example.

LISTING 1-36 Using the *lock* keyword

```
using System;
using System.Threading.Tasks;

namespace Chapter1
{
    public class Program
    {
        static void Main()
        {
            int n = 0;

            object _lock = new object();

            var up = Task.Run(() =>
            {

                for (int i = 0; i < 1000000; i++)
                    lock (_lock)
                        n++;
            });

            for (int i = 0; i < 1000000; i++)
```

```
                lock (_lock)
                    n--;

            up.Wait();
            Console.WriteLine(n);
        }
    }
}
```

After this change, the program always outputs *0* because access to the variable *n* is now synchronized. There is no way that one thread could change the value while the other thread is working with it.

However, it also causes the threads to *block* while they are waiting for each other. This can give performance problems and it could even lead to a *deadlock*, where both threads wait on each other, causing neither to ever complete. Listing 1-37 shows an example of a deadlock.

LISTING 1-37 Creating a deadlock

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace Chapter1
{
    public class Program
    {
        static void Main()
        {
            object lockA = new object();
            object lockB = new object();

            var up = Task.Run(() =>
            {
                lock (lockA)
                {
                    Thread.Sleep(1000);
                    lock (lockB)
                    {
                        Console.WriteLine("Locked A and B");
                    }
                }
            });

            lock (lockB)
            {
                lock (lockA)
                {
                    Console.WriteLine("Locked A and B");
                }
            }
            up.Wait();
        }
    }
}
```

Because both locks are taken in reverse order, a deadlock occurs. The first *Task* locks *A* and waits for *B* to become free. The main thread, however, has *B* locked and is waiting for *A* to be released.

You need to be careful to avoid deadlocks in your code. You can avoid a deadlock by making sure that locks are requested in the same order. That way, the first thread can finish its work, after which the second thread can continue.

The lock code is translated by the compiler into something that looks like Listing 1-38.

LISTING 1-38 Generated code from a *lock* statement

```
object gate = new object();
bool __lockTaken = false;
try
{
    Monitor.Enter(gate, ref __lockTaken);
}
finally
{
    if (__lockTaken)
        Monitor.Exit(gate);
}
```

You shouldn't write this code by hand; let the compiler generate it for you. The compiler takes care of tricky edge cases that can happen.

It's important to use the *lock* statement with a reference object that is private to the class. A public object could be used by other threads to acquire a lock without your code knowing.

It should also be a reference type because a value type would get boxed each time you acquired a lock. In practice, this generates a completely new lock each time, losing the locking mechanism. Fortunately, the compiler helps by raising an error when you accidentally use a value type for the *lock* statement.

You should also avoid locking on the *this* variable because that variable could be used by other code to create a lock, causing deadlocks.

For the same reason, you should not lock on a string. Because of *string-interning* (the process in which the compiler creates one object for several strings that have the same content) you could suddenly be asking for a lock on an object that is used in multiple places.

Volatile class

The C# compiler is pretty good at optimizing code. The compiler can even remove complete statements if it discovers that certain code would never be executed.

The compiler sometimes changes the order of statements in your code. Normally, this wouldn't be a problem in a single-threaded environment. But take a look at Listing 1-39, in which a problem could happen in a multithreaded environment.

LISTING 1-39 A potential problem with multithreaded code

```
private static int _flag = 0;
private static int _value = 0;

public static void Thread1()
{
    _value = 5;
    _flag = 1;
}

public static void Thread2()
{
    if (_flag == 1)
        Console.WriteLine(_value);
}
```

Normally, if you would run *Thread1* and *Thread2*, you would expect no output or an output of 5. It could be, however, that the compiler switches the two lines in *Thread1*. If *Thread2* then executes, it could be that *_flag* has a value of *1* and *_value* has a value of *0*.

You can use locking to fix this, but there is also another class in the .NET Framework that you can use: *System.Threading.Volatile*. This class has a special *Write* and *Read* method, and those methods disable the compiler optimizations so you can force the correct order in your code. Using these methods in the correct order can be quite complex, so .NET offers the *volatile* keyword that you can apply to a field. You would then change the declaration of your field to this:

```
private static volatile int _flag = 0;
```

It's good to be aware of the existence of the *volatile* keyword, but it's something you should use only if you really need it. Because it disables certain compiler optimizations, it will hurt performance. It's also not something that is supported by all .NET languages (Visual Basic doesn't support it), so it hinders language interoperability.

The *Interlocked* class

Referring to Listing 1-35, the essential problem was that the operations of adding and subtracting were not atomic. This because *n++* is translated into *n = n + 1*, both a read and a write.

Making operations atomic is the job of the *Interlocked* class that can be found in the *System.Threading* namespace. When using the *Interlocked.Increment* and *Interlocked.Decrement*, you create an atomic operation, as Listing 1-40 shows.

LISTING 1-40 Using the *Interlocked* class

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace Chapter1
{
    public class Program
    {
        static void Main()
        {
            int n = 0;

            var up = Task.Run(() =>
            {
                for (int i = 0; i < 1000000; i++)
                    Interlocked.Increment(ref n);
            });

            for (int i = 0; i < 1000000; i++)
                Interlocked.Decrement(ref n);


            up.Wait();
            Console.WriteLine(n);

        }

    }
}
```

Interlocked guarantees that the increment and decrement operations are executed atomically. No other thread will see any intermediate results. Of course, adding and subtracting is a simple operation. If you have more complex operations, you would still have to use a lock.

Interlocked also supports switching values by using the *Exchange* method. You use this method as follows:

```
if ( Interlocked.Exchange(ref isInUse, 1) == 0) { }
```

This code retrieves the current value and immediately sets it to the new value in the same operation. It returns the previous value before changing it.

You can also use the *CompareExchange* method. This method first checks to see whether the expected value is there; if it is, it replaces it with another value.

Listing 1-41 shows what can go wrong when comparing and exchanging a value in a nonatomic operation.

LISTING 1-41 Compare and exchange as a nonatomic operation

```
using System;
using System.Threading;
using System.Threading.Tasks;

public static class Program
{
    static int value = 1;

    public static void Main()
    {
        Task t1 = Task.Run(() =>
        {
            if (value == 1)
            {
                // Removing the following line will change the output
                Thread.Sleep(1000);
                value = 2;
            }
        });

        Task t2 = Task.Run(() =>
        {
            value = 3;
        });

        Task.WaitAll(t1, t2);
        Console.WriteLine(value); // Displays 2
    }
}
```

Task t1 starts running and sees that *value* is equal to *1*. At the same time, *t2* changes the value to *3* and then *t1* changes it back to *2*. To avoid this, you can use the following *Interlocked* statement:

```
Interlocked.CompareExchange(ref value, newValue, compareTo);
```

This makes sure that comparing the value and exchanging it for a new one is an atomic operation. This way, no other thread can change the value between comparing and exchanging it.

Canceling tasks

When working with multithreaded code such as the TPL, the *Parallel* class, or PLINQ, you often have long-running tasks. The .NET Framework offers a special class that can help you in canceling these tasks: *CancellationToken*.

You pass a *CancellationToken* to a *Task*, which then periodically monitors the token to see whether cancellation is requested.

Listing 1-42 shows how you can use a *CancellationToken* to end a task.

LISTING 1-42 Using a *CancellationToken*

```
CancellationTokenSource cancellationTokenSource =
    new CancellationTokenSource();
CancellationToken token = cancellationTokenSource.Token;

Task task = Task.Run(() =>
{
    while(!token.IsCancellationRequested)
    {

        Console.Write("*");
        Thread.Sleep(1000);
    }

}, token);
```

```
Console.WriteLine("Press enter to stop the task");
Console.ReadLine();
cancellationTokenSource.Cancel();

Console.WriteLine("Press enter to end the application");
Console.ReadLine();
```

The *CancellationToken* is used in the asynchronous *Task*. The *CancellationTokenSource* is used to signal that the *Task* should cancel itself.

In this case, the operation will just end when cancellation is requested. Outside users of the *Task* won't see anything different because the *Task* will just have a *RanToCompletion* state. If you want to signal to outside users that your task has been canceled, you can do this by throwing an *OperationCanceledException*. Listing 1-43 shows how to do this.

LISTING 1-43 Throwing *OperationCanceledException*

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace Chapter1.Threads
{
    public class Program
    {
        static void Main()
        {
            CancellationTokenSource cancellationTokenSource =
                new CancellationTokenSource();
            CancellationToken token = cancellationTokenSource.Token;

            Task task = Task.Run(() =>
            {
                while (!token.IsCancellationRequested)
                {

                    Console.Write("*");
                    Thread.Sleep(1000);
                }


                token.ThrowIfCancellationRequested();

            }, token);

            try
            {

                Console.WriteLine("Press enter to stop the task");
                Console.ReadLine();

                cancellationTokenSource.Cancel();
                task.Wait();
            }
            catch (AggregateException e)
            {
                Console.WriteLine(e.InnerExceptions[0].Message);
            }
            Console.WriteLine("Press enter to end the application");
            Console.ReadLine();

        }

    }
}
// Displays
// Press enter to stop the task
// **
// A task was canceled.
// Press enter to end the application
```

Instead of catching the exception, you can also add a continuation *Task* that executes only when the *Task* is canceled. In this *Task*, you have access to the exception that was thrown, and you can choose to handle it if that's appropriate. Listing 1-44 shows what such a continuation task would look like.

LISTING 1-44 Adding a continuation for canceled tasks

```
Task task = Task.Run(() =>
{
    while (!token.IsCancellationRequested)
    {
        Console.Write("*");
        Thread.Sleep(1000);
    }


}, token).ContinueWith((t) =>
{
    t.Exception.Handle((e) => true);
    Console.WriteLine("You have canceled the task");
}, TaskContinuationOptions.OnlyOnCanceled);
```

If you want to cancel a *Task* after a certain amount of time, you can use an overload of *Task.WaitAny* that takes a timeout. Listing 1-45 shows an example.

LISTING 1-45 Setting a timeout on a task

```
Task longRunning = Task.Run(() =>
{
    Thread.Sleep(10000);
});

int index = Task.WaitAny(new[] { longRunning }, 1000);

if (index == -1)
    Console.WriteLine("Task timed out");
```

If the returned *index* is *-1*, the task timed out. It's important to check for any possible errors on the other tasks. If you don't catch them, they will go unhandled.

Thought experiment

Implementing multithreading

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are experiencing deadlocks in your code. It's true that you have a lot of locking statements and you are trying to improve your code to avoid the deadlocks.

1. **How can you orchestrate your locking code to avoid deadlocks?**
2. **How can the *Interlocked* class help you?**

Objective summary

- When accessing shared data in a multithreaded environment, you need to synchronize access to avoid errors or corrupted data.
- Use the *lock* statement on a private object to synchronize access to a piece of code.
- You can use the *Interlocked* class to execute simple atomic operations.
- You can cancel tasks by using the *CancellationTokenSource* class with a *CancellationToken*.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. You want to synchronize access by using a *lock* statement. On which member do you lock?
   - **A.** *this*
   - **B.** *string _lock = "mylock"*
   - **C.** *int _lock = 42;*
   - **D.** *object _lock = new object();*