Asynchronous Programming with **async** and **await**

28

Objectives

In this chapter you'll:

- Understand what asynchronous programming is and how it can improve the performance of your apps.
- Use the `async` modifier to indicate that a method is asynchronous.
- Use an `await` expression to wait for an asynchronous task to complete execution so that an `async` method can continue its execution.
- Take advantage of multicore processors by executing tasks asynchronously via features of the Task Parallel Library (TPL).
- Use `Task` method `WhenAll` to wait for multiple tasks to complete before an `async` method can continue its execution.
- Time multiple tasks running on single-core and dual-core systems (with the same processor speeds) to determine the performance improvement when these tasks are run on a dual-core system.
- Use a `WebClient` to invoke a web service asynchronously.

Outline

28.1 Introduction

It would be nice if we could focus our attention on performing only one action at a time and performing it well, but that's usually difficult to do. The human body performs a great variety of operations *in parallel*—or **concurrently**. Respiration, blood circulation, digestion, thinking and walking, for example, can occur concurrently, as can all the senses—sight, touch, smell, taste and hearing.

Computers, too, can perform operations concurrently. It's common for your computer to compile a program, send a file to a printer and receive electronic mail messages over a network concurrently. Tasks like these that proceed independently of one another are said to execute asynchronously and are referred to as **asynchronous tasks**.

Only computers that have multiple processors or cores can *truly* execute multiple asynchronous tasks concurrently. Visual C# apps can have multiple **threads of execution**, where each thread has its own method-call stack, allowing it to execute concurrently with other threads while sharing with them application-wide resources such as memory and processors. This capability is called **multithreading**. Operating systems on single-core computers create the illusion of concurrent execution by rapidly switching between activities (threads), but on such computers only a *single* instruction can execute at once. Today's multicore computers, smartphones and tablets enable computers to perform tasks truly concurrently.

To take full advantage of multicore architecture you need to write applications that can process tasks *asynchronously*. **Asynchronous programming** is a technique for writing apps containing tasks that can execute asynchronously, which can improve app performance and GUI responsiveness in apps with long-running or compute-intensive tasks. At first, concurrency was implemented with operating system primitives available only to experienced systems programmers. Then programming languages (such as C#) began enabling app developers to specify concurrent operations. Initially these capabilities were complex to use, which led to frequent and subtle bugs. Although the human mind can perform functions concurrently, people find it difficult to jump between parallel trains of thought.

To see why concurrent programs can be difficult to write and understand, try the following experiment: Open three books to page 1 and try reading the books concurrently. Read a few words from the first book, then a few from the second, then a few from the third, then loop back and read the next few words from the first book, and so on. After this experiment, you'll appreciate many of the challenges of multithreading—switching between the books, reading briefly, remembering your place in each book, moving the book you're reading closer so that you can see it and pushing the books you're not reading aside—and, amid all this chaos, trying to comprehend the content of the books!

Visual C# 2012 introduces the `async` modifier and `await` operator to greatly simplify asynchronous programming, reduce errors and enable your apps to take advantage of the processing power in today's multicore computers, smartphones and tablets. In .NET 4.5, many classes for web access, file processing, networking, image processing and more have been updated with new methods that return `Task` objects for use with `async` and `await`, so you can take advantage of this new asynchronous programming model.

This chapter presents a simple introduction to asynchronous programming with `async` and `await`. It's designed to help you evaluate and start using these capabilities. Our `async` and `await` resource center

```
http://www.deitel.com/async
```

provides links to articles and other resources that will help you get a deeper understanding of these capabilities.

28.2 Basics of async and await

Before `async` and `await`, it was common for a method that was called *synchronously* (i.e., performing tasks one after another in order) in the calling thread to launch a long-running task *asynchronously* and to provide that task with a *callback method* (or, in some cases, register an event handler) that would be invoked once the asynchronous task *completed*. This style of coding is simplified with `async` and `await`.

`async` Modifier

The **`async` modifier** indicates that a method or lambda expression (introduced in Section 22.5) contains at least one `await` expression. An `async` method executes its body in the same thread as the calling method. (Throughout the remainder of this discussion, we'll use the term "method" to mean "method or lambda expression.")

`await` Expression

An **`await` expression**, which can appear *only* in an `async` method, consists of the **`await` operator** followed by an expression that returns an *awaitable entity*—typically a `Task` object (as you'll see in Section 28.3), though it is possible to create your own awaitable entities. Creating awaitable entities is beyond the scope of our discussion. For more information, see

```
http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115642.aspx
```

When an `async` method encounters an **`await`** expression:

- If the asynchronous task has already completed, the `async` method simply continues executing.
- Otherwise, program control returns to the `async` method's caller until the asynchronous task completes execution. This allows the caller to perform other work that does not depend on the results of the asynchronous task.

When the asynchronous task completes, control returns to the `async` method and continues with the next statement after the `await` expression.

The mechanisms for determining whether to return control to the `async` method's caller or continue executing the `async` method, and for continuing the `async` method's execution when the asynchronous task completes are handled entirely by code that's written for you by the compiler.

`async`, `await` and Threads

The `async` and `await` mechanism does *not* create new threads. If any threads are required, the method that you call to start an asynchronous task on which you `await` the results is responsible for creating the threads that are used to perform the asynchronous task. For example, we'll show how to use class `Task`'s `Run` method in several examples to start new threads of execution for executing tasks asynchronously. `Task` method `Run` returns a `Task` on which a method can `await` the result.

28.3 Executing an Asynchronous Task from a GUI App

This section demonstrates the benefits of executing compute-intensive tasks asynchronously in a GUI app.

28.3.1 Performing a Task Asynchronously

Figure 28.1 demonstrates executing an asynchronous task from a GUI app. Consider the GUI at the end of Fig. 28.1. In the GUI's top half, you can enter an integer then click **Calculate** to calculate that integer's Fibonacci value using a compute-intensive recursive implementation (Section 28.3.2). Starting with integers in the 40s (on our test computer), the recursive calculation can take seconds or even minutes to calculate. If this calculation were to be performed *synchronously*, the GUI would *freeze* for that amount of time and the user would not be able to interact with the app (as we'll demonstrate in Fig. 28.2). We launch the calculation *asynchronously* and have it execute on a *separate* thread so the GUI remains *responsive*. To demonstrate this, in the GUI's bottom half, you can click **Next Number** repeatedly to calculate the next Fibonacci number by simply adding the two previous numbers in the sequence. For the screen captures in Fig. 28.1, we used the top-half of the GUI to calculate `Fibonacci(45)`, which took over a minute on our test computer. While that calculation proceeded in a separate thread, we clicked **Next Number** repeatedly to demonstrate that we could still interact with the GUI. Along the way, we were able to demonstrate that the iterative Fibonacci calculation is much more efficient.

```
// Fig. 28.1: FibonacciForm.cs
// Performing a compute-intensive calculation from a GUI app
using System;
using System.Threading.Tasks;
using System.Windows.Forms;

```

Fig. 28.1 | Performing a compute-intensive calculation from a GUI app. (Part 1 of 3.)

```
namespace FibonacciTest
{
   public partial class FibonacciForm : Form
   {
      private long n1 = 0; // initialize with first Fibonacci number
      private long n2 = 1; // initialize with second Fibonacci number
      private int count = 1; // current Fibonacci number to display

      public FibonacciForm()
      {
         InitializeComponent();
      } // end constructor

      // start an async Task to calculate specified Fibonacci number
      private async void calculateButton_Click(
         object sender, EventArgs e )
      {
         // retrieve user's input as an integer
         int number = Convert.ToInt32( inputTextBox.Text );

         asyncResultLabel.Text = "Calculating...";

         // Task to perform Fibonacci calculation in separate thread
         Task< long > fibonacciTask =
            Task.Run( () => Fibonacci( number ) );

         // wait for Task in separate thread to complete
         await fibonacciTask;

         // display result after Task in separate thread completes
         asyncResultLabel.Text = fibonacciTask.Result.ToString();
      } // end method calculateButton_Click

      // calculate next Fibonacci number iteratively
      private void nextNumberButton_Click( object sender, EventArgs e )
      {
         // calculate the next Fibonacci number
         long temp = n1 + n2; // calculate next Fibonacci number
         n1 = n2; // store prior Fibonacci number in n1
         n2 = temp; // store new Fibonacci
         ++count;

         // display the next Fibonacci number
         displayLabel.Text = string.Format( "Fibonacci of {0}:", count );
         syncResultLabel.Text = n2.ToString();
      } // end method nextNumberButton_Click

      // recursive method Fibonacci; calculates nth Fibonacci number
      public long Fibonacci( long n )
      {
         if ( n == 0 || n == 1 )
            return n;
```

Fig. 28.1 | Performing a compute-intensive calculation from a GUI app. (Part 2 of 3.)

```
59            else
60               return Fibonacci( n - 1 ) + Fibonacci( n - 2 );
61         } // end method Fibonacci
62      } // end class FibonacciForm
63   } // end namespace FibonacciTest
```

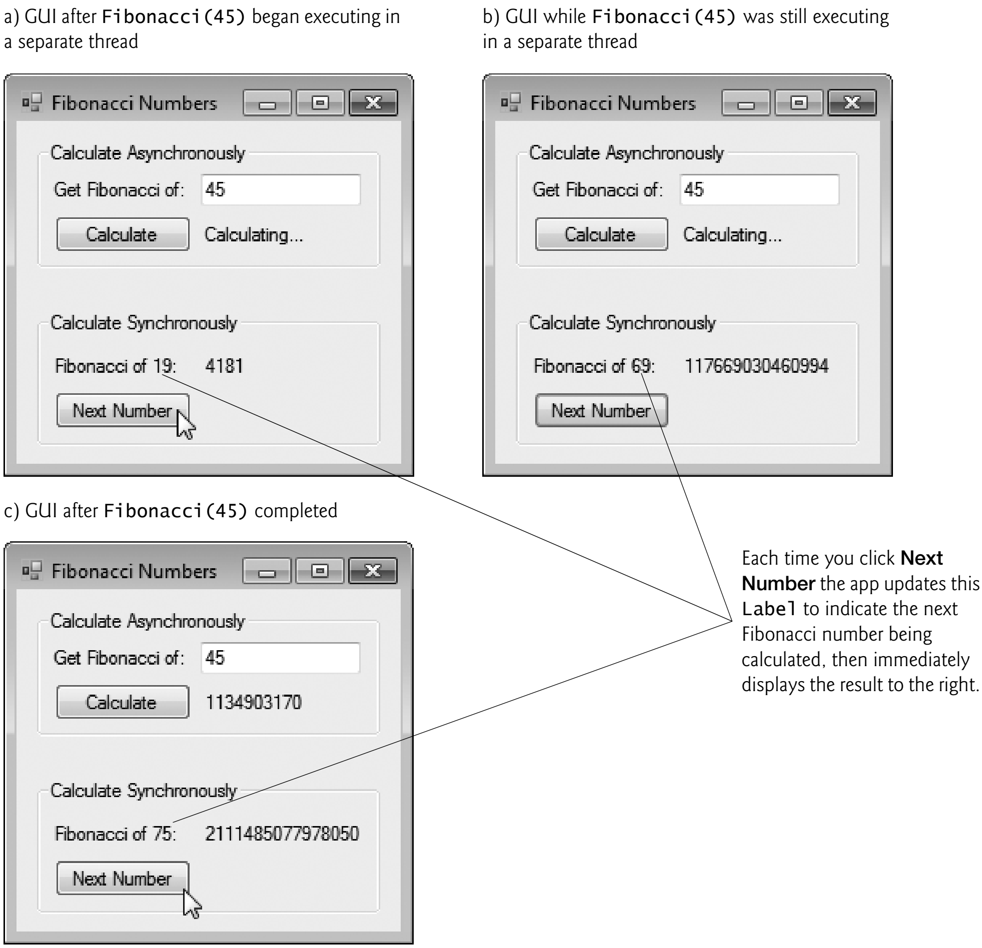


Fig. 28.1 | Performing a compute-intensive calculation from a GUI app. (Part 3 of 3.)

A Compute-Intensive Algorithm: Calculating Fibonacci Numbers Recursively

The powerful technique of recursion was introduced in Section 7.15. The examples in this section and in Sections 28.4–28.5 each perform a compute-intensive *recursive* Fibonacci calculation (defined in the `Fibonacci` method at lines 55–61). The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, …

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

The Fibonacci series can be defined *recursively* as follows:

Fibonacci(0) = 0
Fibonacci(1) = 1
Fibonacci(n) = Fibonacci(n – 1) + Fibonacci(n – 2)

A word of caution is in order about recursive methods like the one we use here to generate Fibonacci numbers. The number of recursive calls that are required to calculate the nth Fibonacci number is on the order of 2^n. This rapidly gets out of hand as n gets larger. Calculating only the 20th Fibonacci number would require on the order of 2^{20} or about a million calls, calculating the 30th Fibonacci number would require on the order of 2^{30} or about a billion calls, and so on. This **exponential complexity** can humble even the world's most powerful computers! Calculating just Fibonacci(47)—even on today's most recent desktop and notebook computers—can take many minutes.

28.3.2 Method `calculateButton_Click`

The **Calculate** button's event handler (lines 21–38) initiates the call to method `Fibonacci` in a separate thread and displays the results when the call completes. The method is declared `async` (line 21) to indicate to the compiler that the method will initiate an asynchronous task and `await` the results. In effect, an `async` method allows you to write code that looks like it executes sequentially, while the compiler deals with the complicated issues of managing asynchronous execution. This makes your code easier to write, modify and maintain, and reduces errors.

28.3.3 Task Method Run: Executing Asynchronously in a Separate Thread

Lines 30–31 create and start a **`Task`** (namespace **`System.Threading.Tasks`**). A `Task` promises to return a result *at some point* in the future. Class `Task` is part of .NET's *Task Parallel Library (TPL)* for asynchronous programming. The version of class `Task`'s `static` method **`Run`** used in line 31 receives a **`Func<TResult>` delegate** (delegates were introduced in Section 14.3.3) as an argument and executes a method in a *separate thread*. The delegate `Func<TResult>` represents any method that takes *no arguments* and returns a *result*, so the name of any method that takes no arguments and returns a result can be passed to `Run`. However, `Fibonacci` requires an argument, so line 31 use the *lambda expression*

```
() => Fibonacci( number )
```

which takes *no arguments* to encapsulate the call to `Fibonacci` with the argument `number`. The lambda expression *implicitly* returns the result of the `Fibonacci` call (a `long`), so it meets the `Func<TResult>` delegate's requirements. In this example, `Task`'s `static` method `Run` creates and returns a `Task<long>` that represents the task being performed in a separate thread. The compiler *infers* the type `long` from the return type of method `Fibonacci`.

28.3.4 `await`ing the Result

Next, line 34 `await`s the result of the `fibonacciTask` that's executing asynchronously. If the `fibonacciTask` is *already complete*, execution continues with line 37. Otherwise, control returns to `calculateButton_Click`'s caller (the GUI event handling thread) until the result of the `fibonacciTask` is available. This allows the GUI to remain *responsive* while the `Task` executes. Once the `Task` completes, `calculateButton_Click` continues execution at line 37, which uses `Task` property **`Result`** to get the value returned by `Fibonacci` and display it on `asyncResultLabel`.

It's important to note that an `async` method can perform other statements between those that launch an asynchronous `Task` and `await` the `Task`'s results. In such a case, the method continues executing those statements after launching the asynchronous `Task` until it reaches the `await` expression.

Lines 30–34 can be written more concisely as

```
long result = await Task.Run( () => Fibonacci( number ) );
```

In this case, the `await` operator unwraps and returns the `Task`'s result—the `long` returned by method `Fibonacci`. You can then use the `long` value directly without accessing the `Task`'s `Result` property.

28.3.5 Calculating the Next Fibonacci Value Synchronously

When you click **Next Number**, the event handler registered in lines 41–52 executes. Lines 44–47 add the previous two Fibonacci numbers stored in instance variables `n1` and `n2` to determine the next number in the sequence, update `n1` and `n2` to their new values and increment instance variable `count`. Then lines 50–51 update the GUI to display the Fibonacci number that was just calculated. The code in the **Next Number** event handler is performed in the GUI thread of execution that processes user interactions with controls. Handling such short computations in this thread does *not* cause the GUI to become unresponsive. Because the longer Fibonacci computation is performed in a *separate thread*, it's possible to get the next Fibonacci number *while the recursive computation is still in progress*.

28.4 Sequential Execution of Two Compute-Intensive Tasks

Figure 28.2 uses the recursive `Fibonacci` method that we introduced in Section 28.3. The example sequentially performs the calculations `fibonacci(46)` (line 22) and `Fibonacci(45)` (line 37) when the user clicks the **Start Sequential Fibonacci Calls** `Button`. Before and after each `Fibonacci` call, we capture the time so that we can calculate the total time required for *that* calculation and the total time required for *both* calculations.

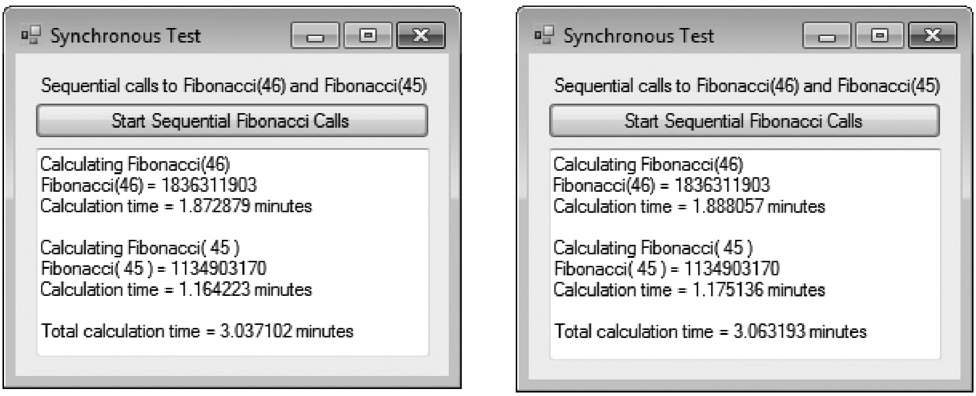
The first two outputs show the results of executing the app on a *dual-core* Windows 7 computer. The last two outputs show the results of executing the app on a single-core Windows 7 computer. In all cases, the cores operated at the same speed. The app *always* took longer to execute (in our testing) on the single-core computer, because the processor was being *shared* between this app and all the others that happened to be executing on the computer at the same time. On the dual-core system, one of the cores could have been handling the "other stuff" executing on the computer, reducing the demand on the core that's doing the synchronous calculation. Results may vary across systems based on processor speeds, the number of cores, apps currently executing and the chores the operating system is performing.

```
// Fig. 28.2: SynchronousTestForm.cs
// Fibonacci calculations performed sequentially
using System;
using System.Windows.Forms;

namespace FibonacciSynchronous
{
   public partial class SynchronousTestForm : Form
   {
      public SynchronousTestForm()
      {
         InitializeComponent();
      } // end constructor

      // start sequential calls to Fibonacci
      private void startButton_Click( object sender, EventArgs e )
      {
         // calculate Fibonacci (46)
         outputTextBox.Text = "Calculating Fibonacci(46)\r\n";
         outputTextBox.Refresh(); // force outputTextBox to repaint
         DateTime startTime1 = DateTime.Now; // time before calculation
         long result1 = Fibonacci( 46 ); // synchronous call
         DateTime endTime1 = DateTime.Now; // time after calculation

         // display results for Fibonacci(46)
         outputTextBox.AppendText(
            String.Format( "Fibonacci(46) = {0}\r\n", result1 ) );
         outputTextBox.AppendText( String.Format(
            "Calculation time = {0:F6} minutes\r\n\r\n",
            endTime1.Subtract( startTime1 ).TotalMilliseconds /
            60000.0 ) );

         // calculate Fibonacci (45)
         outputTextBox.AppendText( "Calculating Fibonacci(45)\r\n" );
         outputTextBox.Refresh(); // force outputTextBox to repaint
         DateTime startTime2 = DateTime.Now;
         long result2 = Fibonacci( 45 ); // synchronous call
         DateTime endTime2 = DateTime.Now;

         // display results for Fibonacci(45)
         outputTextBox.AppendText(
            String.Format( "Fibonacci( 45 ) = {0}\r\n", result2 ));
         outputTextBox.AppendText( String.Format(
            "Calculation time = {0:F6} minutes\r\n\r\n",
            endTime2.Subtract( startTime2 ).TotalMilliseconds /
            60000.0 ) );

         // show total calculation time
         outputTextBox.AppendText( String.Format(
            "Total calculation time = {0:F6} minutes\r\n",
            endTime2.Subtract( startTime1 ).TotalMilliseconds /
            60000.0 ) );
      } // end method startButton_Click
```

Fig. 28.2 | Fibonacci calculations performed sequentially. (Part 1 of 2.)

```

      // Recursively calculates Fibonacci numbers
      public long Fibonacci( long n )
      {
         if ( n == 0 || n == 1 )
            return n;
         else
            return Fibonacci( n - 1 ) + Fibonacci( n - 2 );
      } // end method Fibonacci
   } // end class SynchronousTestForm
} // end namespace FibonacciSynchronous
```

a) Outputs on a Dual Core Windows 7 Computer



b) Outputs on a Single Core Windows 7 Computer

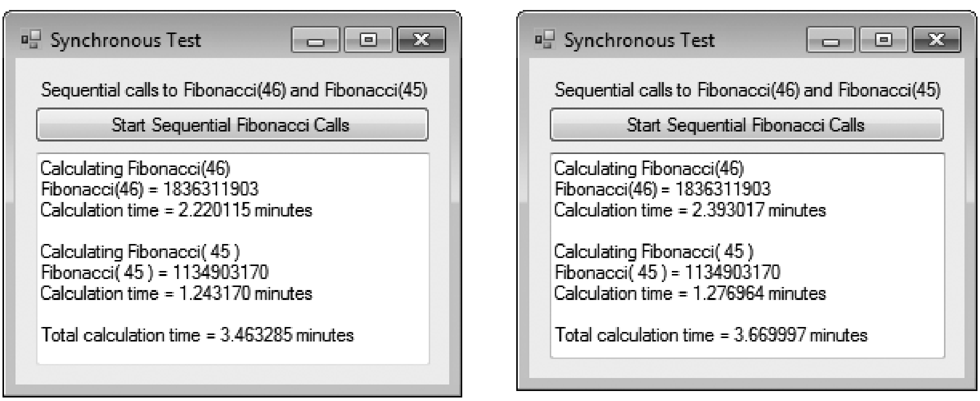


Fig. 28.2 | Fibonacci calculations performed sequentially. (Part 2 of 2.)

28.5 Asynchronous Execution of Two Compute-Intensive Tasks

When you run any program, your program's tasks compete for processor time with the operating system, other programs and other activities that the operating system is running on your behalf. When you execute the next example, the time to perform the Fibonacci calculations can vary based on your computer's processor speed, number of cores and what else is running on your computer. It's like a drive to the supermarket—the time it takes can vary based on traffic conditions, weather, timing of traffic lights and other factors.

Figure 28.3 also uses the recursive `Fibonacci` method, but the two initial calls to `Fibonacci` execute in *separate threads*. The first two outputs show the results on a *dual-core* computer. Though execution times varied, the total time to perform both `Fibonacci` calculations (in our tests) was typically *significantly less* than the total time of the sequential execution in Fig. 28.2. Dividing the compute-intensive calculations into threads and running them on a dual-core system does *not* perform the calculations *twice* as fast, but they'll typically run *faster* than performing the calculations *in sequence* on one core. Though the total time was the compute time for the longer calculation, this is not always the case as there's overhead inherent in using threads to perform separate `Tasks`.

The last two outputs show that executing calculations in multiple threads on a single-core processor can actually take *longer* than simply performing them synchronously, due to the overhead of sharing *one* processor among the app's threads, all the other apps executing on the computer at the same time and the chores the operating system was performing.

```
// Fig. 28.3: AsynchronousTestForm.cs
// Fibonacci calculations performed in separate threads
using System;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace FibonacciAsynchronous
{
   public partial class AsynchronousTestForm : Form
   {
      public AsynchronousTestForm()
      {
         InitializeComponent();
      } // end constructor

      // start asynchronous calls to Fibonacci
      private async void startButton_Click( object sender, EventArgs e )
      {
         outputTextBox.Text =
            "Starting Task to calculate Fibonacci(46)\r\n";

         // create Task to perform Fibonacci(46) calculation in a thread
         Task< TimeData > task1 =
            Task.Run( () => StartFibonacci( 46 ) );

         outputTextBox.AppendText(
            "Starting Task to calculate Fibonacci(45)\r\n" );

         // create Task to perform Fibonacci(45) calculation in a thread
         Task< TimeData > task2 =
            Task.Run( () => StartFibonacci( 45 ) );
```

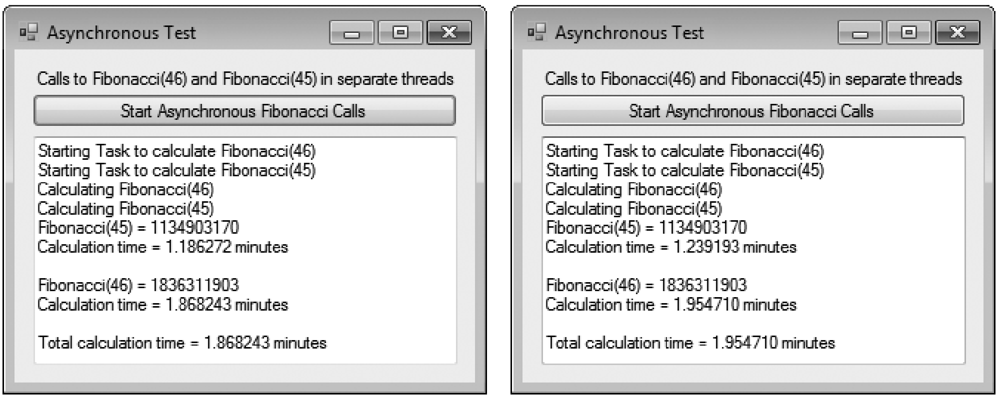
Fig. 28.3 | Fibonacci calculations performed in separate threads. (Part 1 of 3.)

```

         await Task.WhenAll( task1, task2 ); // wait for both to complete

         // determine time that first thread started
         DateTime startTime =
            ( task1.Result.StartTime < task2.Result.StartTime ) ?
            task1.Result.StartTime : task2.Result.StartTime;

         // determine time that last thread ended
         DateTime endTime =
            ( task1.Result.EndTime > task2.Result.EndTime ) ?
            task1.Result.EndTime : task2.Result.EndTime;

         // display total time for calculations
         outputTextBox.AppendText( String.Format(
            "Total calculation time = {0:F6} minutes\r\n",
            endTime.Subtract( startTime ).TotalMilliseconds /
            60000.0 ) );
      } // end method startButton_Click

      // starts a call to fibonacci and captures start/end times
      TimeData StartFibonacci( int n )
      {
         // create a TimeData object to store start/end times
         TimeData result = new TimeData();

         AppendText( String.Format( "Calculating Fibonacci({0})", n ) );
         result.StartTime = DateTime.Now;
         long fibonacciValue = Fibonacci( n );
         result.EndTime = DateTime.Now;

         AppendText( String.Format( "Fibonacci({0}) = {1}",
            n, fibonacciValue ) );
         AppendText( String.Format(
            "Calculation time = {0:F6} minutes\r\n",
            result.EndTime.Subtract(
               result.StartTime ).TotalMilliseconds / 60000.0 ) );

         return result;
      } // end method StartFibonacci

      // Recursively calculates Fibonacci numbers
      public long Fibonacci( long n )
      {
         if ( n == 0 || n == 1 )
            return n;
         else
            return Fibonacci( n - 1 ) + Fibonacci( n - 2 );
      } // end method Fibonacci

      // append text to outputTextBox in UI thread
      public void AppendText( String text )
      {
```

Fig. 28.3 | Fibonacci calculations performed in separate threads. (Part 2 of 3.)

```
85            if ( InvokeRequired ) // not GUI thread, so add to GUI thread
86               Invoke( new MethodInvoker( () => AppendText( text ) ) );
87            else // GUI thread so append text
88               outputTextBox.AppendText( text + "\r\n" );
89         } // end method AppendText
90      } // end class AsynchronousTestForm
91   } // end namespace FibonacciAsynchronous
```

a) Outputs on a Dual Core Windows 7 Computer



b) Outputs on a Single Core Windows 7 Computer

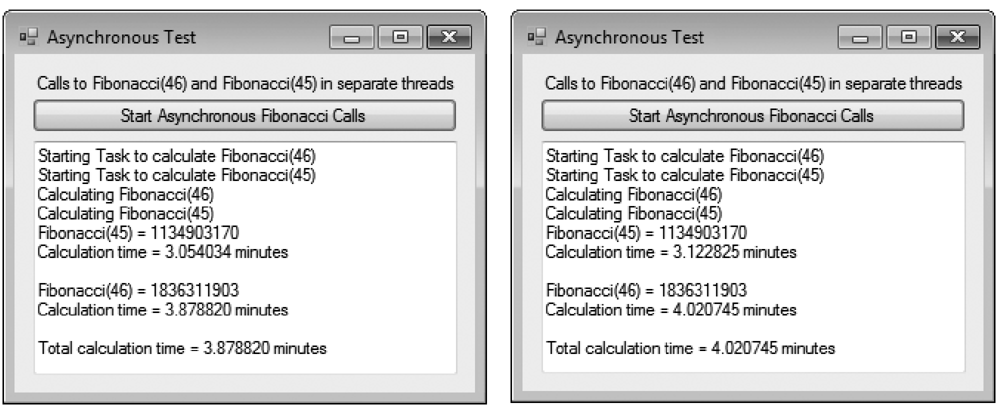


Fig. 28.3 | Fibonacci calculations performed in separate threads. (Part 3 of 3.)

28.5.1 Method `startButton_Click`: awaiting Multiple Tasks with Task Method `WhenAll`

In method `startButton_Click`, lines 23–24 and 30–31 use `Task` method `Run` to create and start `Task`s that execute method `StartFibonacci` (lines 53–71)—one to calculate `Fibonacci(46)` and one to calculate `Fibonacci(45)`. To show the total calculation time, the app must wait for *both* `Task`s to complete *before* executing lines 36–49. You can wait for *multiple* `Task`s to complete by `await`ing the result of `Task static` method **`WhenAll`** (line 33), which returns a `Task` that waits for *all* of `WhenAll`'s argument `Task`s to complete and places *all* the results in an array. In this app, the Task's Result is a TimeData[], because both of WhenAll's argument Tasks execute methods that return TimeData objects. This array can be used to iterate through the results of the awaited Tasks. In this example, we have only two Tasks, so we interact with the task1 and task2 objects directly in the remainder of the event handler.

28.5.2 Method StartFibonacci

Method StartFibonacci (lines 53–71) specifies the task to perform—in this case, to call Fibonacci (line 60) to perform the recursive calculation, to time the calculation (lines 59 and 61), to display the calculation's result (lines 63–64) and to display the time the calculation took (lines 65–68). The method returns a TimeData object (defined in this project's TimeData.cs file) that contains the time before and after each thread's call to Fibonacci. Class TimeData contains public auto-implemented properties StartTime and EndTime, which we use in our timing calculations.

28.5.3 Method AppendText: Modifying a GUI from a Separate Thread

Lines 58, 63 and 65 in StartFibonacci call our AppendText method (lines 83–89) to append text to the outputTextBox. GUI controls are designed to be manipulated *only* by the GUI thread—modifying a control from a non-GUI thread can corrupt the GUI, making it unreadable or unusable. When updating a control from a non-GUI thread, you *must schedule* that update to be performed by the GUI thread. To do so in Windows Forms, you check the **InvokeRequired property** of class Form (line 85). If this property's value is true, the code is executing in a non-GUI thread and *must not* update the GUI directly. Instead, you call the **Invoke** method of class Form (line 86), which receives as an argument a Delegate representing the update to perform in the GUI thread. In this example, we pass a MethodInvoker (namespace System.Windows.Forms), which is a Delegate that invokes a method with no arguments and a void return type. The MethodInvoker is initialized here with a *lambda expression* that calls AppendText. Line 86 *schedules* this MethodInvoker to execute in the GUI thread. When that occurs, line 88 updates the outputTextBox. (Similar concepts also apply to GUIs created with WPF and Windows 8 UI.)

28.5.4 awaiting One of Several Tasks with Task Method WhenAny

Similar to WhenAll, class Task also provides static method **WhenAny**, which enables you to wait for any one of several Tasks specified as arguments to complete. WhenAny returns the Task that completes first. One use of WhenAny might be to initiate several Tasks that perform the same complex calculation on computers around the Internet, then wait for any one of those computers to send results back. This would allow you to take advantage of computing power that's available to you to get the result as fast as possible. In this case, it's up to you to decide whether to cancel the remaining Tasks or allow them to continue executing. For details on how to do this, see

```
http://msdn.microsoft.com/en-us/library/vstudio/jj155758.aspx
```

Another use of WhenAny might be to download several large files—one per Task. In this case, you might want all the results eventually, but would like to immediately start processing the results from the first Task that returns. You could perform a new call to WhenAny for the remaining Tasks that are still executing.

28.6 Invoking a Flickr Web Service Asynchronously with `WebClient`

In this section, we present a **Flickr Viewer** app (Fig. 28.4) that allows you to search for photos on the photo-sharing website Flickr then browse through the results. The app uses an asynchronous method to invoke a Flickr web service. A **web service** is software that can receive method calls over a network using standard web technologies. Flickr provides a so-called REST web service that can receive method calls via standard web interactions, just like you'd use to access a web page in a web browser. (You'll learn more about REST web services in Chapter 30.) Because there can be unpredictably long delays while `await`ing a web-service response, asynchronous `Task`s are frequently used in GUI apps that invoke web services (or perform network communication in general) to ensure that the apps remain responsive.

Our **Flickr Viewer** app allows you to search by *tag* for photos that users worldwide have uploaded to Flickr. *Tagging*—or labeling content—is part of the collaborative nature of social media. A tag is any user-supplied word or phrase that helps organize web content. Tagging items with self-chosen words or phrases creates a strong identification of the content. Flickr uses tags on uploaded files to improve its photo-search service, giving the user better results. *To run this example on your computer, you must obtain your own Flickr API key at*

```
http://www.flickr.com/services/apps/create/apply
```

and use it to replace the words `YOUR API KEY HERE` inside the quotes in line 18. This key is a unique string of characters and numbers that enables Flickr to track the usage of its APIs.

```
1  // Fig. 28.4: FickrViewerForm.cs
2  // Invoking a web service asynchronously with class WebClient
3  using System;
4  using System.Drawing;
5  using System.IO;
6  using System.Linq;
7  using System.Net;
8  using System.Threading.Tasks;
9  using System.Windows.Forms;
10 using System.Xml.Linq;
11
12 namespace FlickrViewer
13 {
14    public partial class FickrViewerForm : Form
15    {
16       // Use your Flickr API key here--you can get one at:
17       // http://www.flickr.com/services/apps/create/apply
18       private const string KEY = "YOUR API KEY HERE";
19
20       // object used to invoke Flickr web service
21       private WebClient flickrClient = new WebClient();
22
23       Task<string> flickrTask = null; // Task<string> that queries Flickr
24
```

Fig. 28.4 | Invoking a web service asynchronously with class `WebClient`. (Part 1 of 4.) [Photos used in this example courtesy of Paul Deitel. All rights reserved.]

```
      public FickrViewerForm()
      {
         InitializeComponent();
      } // end constructor

      // initiate asynchronous Flickr search query;
      // display results when query completes
      private async void searchButton_Click( object sender, EventArgs e )
      {
         // if flickrTask already running, prompt user
         if ( flickrTask != null &&
            flickrTask.Status != TaskStatus.RanToCompletion )
         {
            var result = MessageBox.Show(
               "Cancel the current Flickr search?",
               "Are you sure?", MessageBoxButtons.YesNo,
               MessageBoxIcon.Question );

            // determine whether user wants to cancel prior search
            if ( result == DialogResult.No )
                return;
            else
                flickrClient.CancelAsync(); // cancel current search
         } // end if

         // Flickr's web service URL for searches
         var flickrURL = string.Format( "http://api.flickr.com/services" +
            "/rest/?method=flickr.photos.search&api_key={0}&tags={1}" +
            "&tag_mode=all&per_page=500&privacy_filter=1", KEY,
            inputTextBox.Text.Replace( " ", "," ) );

         imagesListBox.DataSource = null; // remove prior data source
         imagesListBox.Items.Clear(); // clear imagesListBox
         pictureBox.Image = null; // clear pictureBox
         imagesListBox.Items.Add( "Loading..." ); // display Loading...

         try
         {
            // invoke Flickr web service to search Flick with user's tags
            flickrTask =
               flickrClient.DownloadStringTaskAsync( flickrURL );

            // await flickrTask then parse results with XDocument and LINQ
            XDocument flickrXML = XDocument.Parse( await flickrTask );

            // gather information on all photos
            var flickrPhotos =
               from photo in flickrXML.Descendants( "photo" )
               let id = photo.Attribute( "id" ).Value
               let title = photo.Attribute( "title" ).Value
               let secret = photo.Attribute( "secret" ).Value
```

Fig. 28.4 | Invoking a web service asynchronously with class `WebClient`. (Part 2 of 4.) [Photos used in this example courtesy of Paul Deitel. All rights reserved.]

```
                  let server = photo.Attribute( "server" ).Value
                  let farm = photo.Attribute( "farm" ).Value
                  select new FlickrResult
                  {
                     Title = title,
                     URL = string.Format(
                        "http://farm{0}.staticflickr.com/{1}/{2}_{3}.jpg",
                        farm, server, id, secret )
                  };
               imagesListBox.Items.Clear(); // clear imagesListBox
               // set ListBox properties only if results were found
               if ( flickrPhotos.Any() )
               {
                  imagesListBox.DataSource = flickrPhotos.ToList();
                  imagesListBox.DisplayMember = "Title";
               } // end if
               else // no matches were found
                  imagesListBox.Items.Add( "No matches" );
            } // end try
            catch ( WebException )
            {
               // check whether Task failed
               if ( flickrTask.Status == TaskStatus.Faulted )
                  MessageBox.Show( "Unable to get results from Flickr",
                     "Flickr Error", MessageBoxButtons.OK,
                     MessageBoxIcon.Error );
               imagesListBox.Items.Clear(); // clear imagesListBox
               imagesListBox.Items.Add( "Error occurred" );
            } // end catch
         } // end method searchButton_Click

         // display selected image
         private async void imagesListBox_SelectedIndexChanged(
            object sender, EventArgs e )
         {
            if ( imagesListBox.SelectedItem != null )
            {
               string selectedURL =
                  ( ( FlickrResult ) imagesListBox.SelectedItem ).URL;

               // use WebClient to get selected image's bytes asynchronously
               WebClient imageClient = new WebClient();
               byte[] imageBytes = await imageClient.DownloadDataTaskAsync(
                  selectedURL );

               // display downloaded image in pictureBox
               MemoryStream memoryStream = new MemoryStream( imageBytes );
               pictureBox.Image = Image.FromStream( memoryStream );
            } // end if
         } // end method imagesListBox_SelectedIndexChanged
      } // end class FlickrViewerForm
   } // end namespace FlickrViewer
```

Fig. 28.4 | Invoking a web service asynchronously with class `WebClient`. (Part 3 of 4.) [Photos used in this example courtesy of Paul Deitel. All rights reserved.]

Fig. 28.4 | Invoking a web service asynchronously with class `WebClient`. (Part 4 of 4.) [Photos used in this example courtesy of Paul Deitel. All rights reserved.]

As shown in the screen captures of Fig. 28.4, you can type one or more tags (e.g., "pdeitel flowers") into the `TextBox`. When you click the **Search** `Button`, the application invokes the Flickr web service that searches for photos, which returns an XML document containing links to the first 500 (or fewer if there are not 500) results that match the tags you specify. We use LINQ to XML (Chapter 24) to parse the results and display a list of photo titles in a `ListBox`. When you select an image's title in the `ListBox`, the app uses another asynchronous `Task` to download the full-size image from Flickr and display it in a `PictureBox`.

Using Class `WebClient` to Invoke a Web Service

This app uses class **`WebClient`** (namespace `System.Net`) to interact with Flickr's web service and retrieve photos that match the tags you enter. Line 21 creates object `flickrClient` of class `WebClient` that can be used, among other things, to download data from a website. Class `WebClient` is one of many .NET classes that have been updated with new methods in .NET 4.5 to support asynchronous programming with `async` and `await`. In the `searchButton_Click` event handler (lines 32–105), we'll use class `WebClient`'s `DownloadStringTaskAsync` method to start a new `Task` in a separate thread. When we create that `Task`, we'll assign it to instance variable `flickrTask` (declared in line 23) so that we can test whether the `Task` is still executing when the user initiates a new search.

Method `searchButton_Click`

Method `searchButton_Click` (lines 32–105) initiates the *asynchronous* Flickr search, so it's declared as an `async` method. First lines 35–48 check whether you started a search previously (i.e., `flickrTask` is not `null` and the prior search has not completed) and, if so, whether that search has already completed. If an existing search is still being performed, we display a dialog asking if you wish to cancel the search. If you click **No**, the event handler simply returns. Otherwise, we call the `WebClient`'s **`CancelAsync`** method to terminate the search.

Invoking the Flickr Web Service's `flickr.photos.search` Method

Lines 51–54 create the URL required for invoking the Flickr web service's method `flickr.photos.search`. You can learn more about this web-service method's parameters and the format of the URL for invoking the method at

```
http://www.flickr.com/services/api/flickr.photos.search.html
```

In this example, we specify values for the following parameters:

- `api_key`—Your Flickr API key that you obtained from `www.flickr.com/services/apps/create/apply`.
- `tags`—A comma-separated list of the tags for which to search. In our sample executions it was `"pdeitel,flowers"`.
- `tag_mode`—`all` to get results that match *all* the tags you specified in your search (or `any` to get results that match *one or more* of the tags).
- `per_page`—The maximum number of results to return (up to 500).
- `privacy_filter`—The value `1` indicates that only *publicly accessible* photos should be returned.

Lines 64–65 call class `WebClient`'s **`DownloadStringTaskAsync`** method using the URL specified as the method's `string` argument to request information from a web server. Because this URL represents a call to a web service method, calling `DownloadStringTaskAsync` will invoke the Flickr web service to perform the search. `DownloadStringTaskAsync` creates and starts a *new thread* for you and returns a `Task<string>` representing a promise to eventually return a `string` containing the search results. The app then `await`s the results of the `Task` (line 68). At this point, if the `Task` is complete, method `searchButton_Click`'s execution continues at line 71; otherwise, program control returns to method `searchButton_Click`'s caller until the results are received. This allows the GUI thread of execution to handle other events, so the GUI remains *responsive* while the search is ongoing. Thus, you could decide to start a *different* search at any time (which cancels the original search in this app).

Processing the XML Response

When `Task` completes, program control continues in method `searchButton_Click`. Lines 71–84 process the search results, which are returned in XML format. A sample of the XML is shown in Fig. 28.5.

```
<rsp stat="ok">
   <photos page="1" pages="1" perpage="500" total="5">
   <photo id="2608518370" owner="8832668@N04" secret="0099e12778"
      server="3076" farm="4" title="Fuscia Flowers" ispublic="1"
      isfriend="0" isfamily="0"/>
   <photo id="2608518732" owner="8832668@N04" secret="76dab8eb42"
      server="3185" farm="4" title="Red Flowers 1" ispublic="1"
      isfriend="0" isfamily="0"/>
   <photo id="2607687273" owner="8832668@N04" secret="4b630e31ba"
      server="3283" farm="4" title="Red Flowers 2" ispublic="1"
      isfriend="0" isfamily="0"/>
   <photo id="2608518890" owner="8832668@N04" secret="98fcb5fb42"
      server="3121" farm="4" title="Yellow Flowers" ispublic="1"
      isfriend="0" isfamily="0"/>
   <photo id="2608518654" owner="8832668@N04" secret="57d35c8f64"
      server="3293" farm="4" title="Lavender Flowers" ispublic="1"
      isfriend="0" isfamily="0"/>
   </photos>
</rsp>
```

Fig. 28.5 | Sample XML response from the Flickr APIs.

Once the app receives the XML response from Flickr, line 68 converts the XML `string` returned by the `await` expression into an `XDocument` that we can use with LINQ to XML. The LINQ query (lines 71–84) gathers from each `photo` element in the XML the `id`, `title`, `secret`, `server` and `farm` attributes, then creates an object of our class `FlickrResult` (located in this project's `FlickrResult.cs` file). Each `FlickrResult` contains:

- A `Title` property—initialized with the `photo` element's `title` attribute.
- A `URL` property—assembled from the `photo` element's `id`, `secret`, `server` and `farm` (a *farm* is a collection of servers on the Internet) attributes.

The format of the URL for each image is specified at

```
http://www.flickr.com/services/api/misc.urls.html
```

We use these URLs in the `imagesListBox_SelectedIndexChanged` event handler to download an image that you select from the `ListBox` at the left side of the app.

Binding the Photo Titles to the `ListBox`

If there are any results, lines 87–91 clear any prior results from the `ListBox`, then *bind* the titles of all the new results to the `ListBox`. You cannot bind a LINQ query's result directly to a `ListBox`, so line 89 invokes `ToList` on the `flickrPhotos` LINQ query to convert it to a `List` first, then assigns the result to the `ListBox`'s `DataSource` property. This indicates that the `List`'s data should be used to populate the `ListBox`'s `Items` collection. The `List` contains `FlickrResult` objects, so line 90 sets the `ListBox`'s `DisplayMember` property to indicate that the `Title` property of a `FlickrResult` should be displayed for each item in the `ListBox`.

Method `imagesListBox_SelectedIndexChanged`

Method `imagesListBox_SelectedIndexChanged` (lines 108–125) is declared `async` because it `await`s an *asynchronous* download of a photo. Lines 113–114 get the `URL` property of the selected `ListBox` item. Line 117 creates a `WebClient` object for downloading the selected photo from `Flickr`. Lines 118–119 invoke the `WebClient`'s **`DownloadDataTaskAsync`** method to get a `byte` array containing the photo and `await` the results. The method uses the URL specified as the method's `string` argument to request the photo from Flickr and returns a `Task<byte[]>`—a promise to return a `byte[]` once the task completes execution. The event handler then `await`s the result. When the `Task` completes, the `await` expression returns the `byte[]`, which is then assigned to `imageBytes`. Line 122 creates a `MemoryStream` from the `byte[]` (which allows reading `byte`s as a stream from an array in memory), then line 123 uses the `Image` class's `static` `FromStream` method to create an `Image` from the `byte` array and assign it to the `PictureBox`'s `Image` property to display the selected photo.

28.7 Wrap-Up

In this chapter, you learned how to use the `async` modifier, `await` operator and `Task`s to perform long-running or compute-intensive tasks asynchronously. You learned that tasks that proceed independently of one another are said to execute asynchronously and are referred to as asynchronous tasks.

We showed that multithreading enables threads to execute concurrently with other threads while sharing application-wide resources such as memory and processors. To take full advantage of multicore architecture, we wrote applications that processed tasks asynchronously. You learned that asynchronous programming is a technique for writing apps containing tasks that can execute asynchronously, which can improve app performance and GUI responsiveness in apps with long-running or compute-intensive tasks.

To provide a convincing demonstration of asynchronous programming, we presented several apps:

- The first showed how to execute a compute-intensive calculation asynchronously in a GUI app so that the GUI remained responsive while the calculation executed.
- The second app performed two compute-intensive calculations synchronously (sequentially). When that app executed, the GUI froze because the calculations

were performed in the GUI thread. The third app executed the same compute-intensive calculations asynchronously. We executed these two apps on single-core and dual-core computers to demonstrate the performance of each program in each scenario.

- Finally, the fourth app used class `WebClient` to interact with the Flickr website to search for photos. You learned that class `WebClient` is one of many built-in .NET Framework classes that can initiate asynchronous tasks for use with `async` and `await`.

In the next chapter, we continue our discussion of ASP.NET that began in Chapter 23.

Summary

Section 28.1 Introduction

- Computers can perform operations concurrently.
- Tasks that proceed independently of one another are said to execute asynchronously and are referred to as asynchronous tasks.
- Only computers that have multiple processors or cores can truly execute multiple asynchronous tasks concurrently.
- Visual C# apps can have multiple threads of execution, where each thread has its own method-call stack, allowing it to execute concurrently with other threads while sharing with them application-wide resources such as memory and processors. This capability is called multithreading.
- Operating systems on single-core computers create the illusion of concurrent execution by rapidly switching between activities (threads).
- Today's multicore computers, smartphones and tablets enable computers to perform tasks truly concurrently.
- Asynchronous programming is a technique for writing apps containing tasks that can execute asynchronously, which can improve app performance and GUI responsiveness in apps with long-running or compute-intensive tasks.
- Visual C# 2012 introduces the `async` modifier and `await` operator to greatly simplify asynchronous programming, reduce errors and enable your apps to take advantage of the processing power in today's multicore computers, smartphones and tablets.
- In .NET 4.5, many classes for web access, file processing, networking, image processing and more have been updated with new methods that return `Task` objects for use with `async` and `await`, so you can take advantage of this new asynchronous programming model.

Section 28.2 Basics of `async` and `await`

- The `async` modifier indicates that a method or lambda expression contains at least one `await` expression.
- An `async` method executes its body in the same thread as the calling method.
- When an `async` method encounters an `await` expression: If the asynchronous task has already completed, the `async` method simply continues executing. Otherwise, program control returns to the `async` method's caller until the asynchronous task completes execution. When the asynchronous task completes, control returns to the `async` method and continues with the next statement after the `await` expression.

- The mechanisms for determining whether to return control to the async method's caller or continue executing the async method, and for continuing the async method's execution when the asynchronous task completes are handled entirely by code that's written for you by the compiler.
- The async and await mechanism does not create new threads. The method that you call to start an asynchronous task on which you await the results is responsible for creating any threads that are used to perform the asynchronous task.

Section 28.3.1 Performing a Task Asynchronously

- If a long-running calculation were to be performed synchronously in a GUI app, the GUI would freeze until the calculation completed and the user would not be able to interact with the app.
- Launching a calculation asynchronously and executing it on a separate thread keeps the GUI responsive.
- The recursive implementation of the Fibonacci calculation is a compute-intensive calculation.

Section 28.3.2 Method `calculateButton_Click`

- A method is declared async to indicate to the compiler that the method will await an asynchronous task.
- In effect, an async method allows you to write code that looks like it executes sequentially, while the compiler deals with the complicated issues of managing asynchronous execution.

Section 28.3.3 `Task` *Method* `Run`*: Executing Asynchronously in a Separate Thread*

- A Task promises to return a result at some point in the future.
- Class Task is part of .NET's Task Parallel Library (TPL) for asynchronous programming.
- Task static method Run receives a Func<TResult> delegate as an argument and executes a method in a separate thread. The method returns a Task<TResult> where TResult represents the type of value returned by the method being executed.
- The Func<TResult> delegate represents any method that takes no arguments and returns a result.

Section 28.3.4 `await`*ing the Result*

- When you await a Task, if that Task has already completed, execution simply continues. Otherwise, control returns to the async method's caller until the result of the Task is available. Once the Task completes, the async method continues execution.
- Task property Result returns the value returned by a Task.
- An async method can perform other statements between those that launch an asynchronous Task and await the Task's results. In such as case, the method continues executing those statements after launching the asynchronous Task until it reaches the await expression.
- You can place the await expression on the right side of an assignment. The await operator unwraps and returns the Task's result so you can use it directly without accessing the Task's Result property.

Section 28.3.5 Calculating the Next Fibonacci Value Synchronously

- Handling short computations in the GUI thread does not cause the GUI to become unresponsive.

Section 28.4 Sequential Execution of Two Compute-Intensive Tasks

- An app that performs synchronous tasks on a single-core computer often takes longer to execute than on a multi-core computer, because the processor is shared between the app and all the others that happened to be executing on the computer at the same time. On the dual-core system, one of the cores could have been handling the "other stuff" executing on the computer, reducing the demand on the core that's doing the synchronous calculation.

Section 28.5 Asynchronous Execution of Two Compute-Intensive Tasks

- When you run any program, your program's tasks compete for processor time with the operating system, other programs and other activities that the operating system is running on your behalf. In any app, the time to perform the app's tasks can vary based on your computer's processor speed, number of cores and what else is running on your computer.
- Executing asynchronous methods in separate threads on a dual-core computer typically takes less time than executing the same tasks sequentially.
- Executing asynchronous method in multiple threads on a single-core processor can actually take longer than simply performing them synchronously, due to the overhead of sharing one processor among the app's threads, all the other apps executing on the computer at the same time and the chores the operating system was performing.

Section 28.5.1 Method `startButton_Click`: `await`ing Multiple `Task`s with `Task` Method `WhenAll`

- You can wait for multiple `Task`s to complete by `await`ing the result of `Task` `static` method `WhenAll`, which returns a `Task` that waits for all of `WhenAll`'s argument `Task`s to complete and places all the results in an array. This array can be used to iterate through the results of the `await`ed `Task`s.

Section 28.5.3 Method `AppendText`: Modifying a GUI from a Separate Thread

- GUI controls are designed to be manipulated only by the GUI thread—modifying a control from a non-GUI thread can corrupt the GUI, making it unreadable or unusable.
- When updating a control from a non-GUI thread, you must schedule that update to be performed by the GUI thread. To do so in Windows `Forms`, you check the `InvokeRequired` property of class `Form`. If this property's value is `true`, the code is executing in a non-GUI thread and must not update the GUI directly. Instead, you call the `Invoke` method of class `Form`, which receives as an argument a `Delegate` representing the update to perform in the GUI thread.
- A `MethodInvoker` (namespace `System.Windows.Forms`) is a `Delegate` that invokes a method with no arguments and a `void` return type.

Section 28.5.4 `await`ing One of Several Tasks with `Task` Method `WhenAny`

- `Task` `static` method `WhenAny` enables you to wait for any one of several `Task`s specified as arguments to complete. `WhenAny` returns the `Task` that completes first.

Section 28.6 Invoking a Flickr Web Service Asynchronously with `WebClient`

- A **web service** is software that can receive method calls over a network using standard web technologies. Because there can be unpredictably long delays while `await`ing a web-service response, asynchronous `Task`s are frequently used in GUI apps that invoke web services (or perform network communication in general) to ensure that the apps remain responsive.
- Class `WebClient` (namespace `System.Net`) can be used to invoke a web service. Class `WebClient` is one of many .NET classes that have been updated with new methods in .NET 4.5 to support asynchronous programming with `async` and `await`.
- Class `WebClient`'s `DownloadStringTaskAsync` method starts a new `Task<string>` in a separate thread and uses the URL specified as the method's `string` argument to request information from a web server.
- `WebClient`'s `CancelAsync` method terminates it's executing asynchronous task.

- You cannot bind a LINQ query's result directly to a `ListBox`. You must first convert the results to a `List` with method `ToList`.
- A `ListBox`'s `DataSource` property indicates the source of the data that populates the `ListBox`'s `Items` collection. A `ListBox`'s `DisplayMember` property indicates which property of each item in the data source should be displayed in the `ListBox`.
- `WebClient`'s `DownloadDataTaskAsync` method launches in a separate thread a `Task<byte[]>` that gets a `byte[]` from the URL specified as the method's `string` argument.

Terminology

`async` modifier
asynchronous call
asynchronous programming
asynchronous task
`await` expression
`await` multiple `Tasks`
`await` operator
awaitable entity
block a calling method
callback method
`CancelAsync` method of class `WebClient`
concurrency
concurrent operations
`DownloadDataTaskAsync` method of class `WebClient`
`DownloadStringTaskAsync` method of class `WebClient`
exponential complexity
Fibonacci series
`Func<TResult>` delegate
`Invoke` method of class `Control`
`InvokeRequired` property of class `Control`
`MethodInvoker` delegate
multithreading
parallel operations
performing operations concurrently
responsive GUI
REST web service
`Result` property of class `Task`
`Run` method of class `Task`
`System.Net` namespace
`System.Threading.Tasks` namespace
`Task` class
Task Parallel Library
thread of execution
web service
`WebClient` class
`WhenAll` method of class `Task`
`WhenAny` method of class `Task`
`XDocument` class

Self-Review Exercises

28.1 What does it mean to process tasks asynchronously?

28.2 What is the key advantage of programming your apps for multicore systems?

28.3 Suppose you program an app with two compute-intensive tasks that run on one core of a dual-core system. Then, suppose you program the app so that the two compute-intensive tasks run asynchronously in separate threads on a dual-core system. Should you expect the latter program to run in half the time? Why?

28.4 Does the `async` and `await` mechanism create new threads?

28.5 (True/False) You can update a GUI from any thread of execution. If *false*, explain why.

Answers to Self-Review Exercises

28.1 Applications that can process tasks asynchronously typically perform tasks in separate threads of execution, so that the operating system on a multicore computer can run those threads in parallel by assigning them to different cores.

28.2 In multicore systems, the hardware can put multiple cores to work simultaneously, thereby enabling faster completion of programs that can be implemented with asynchronous tasks.

28.3 No. There's overhead inherent in using threads to perform separate tasks. Simply performing the tasks asynchronously on a dual-core system does not complete the tasks in half the time, though they'll often run faster than if you perform the tasks in sequence.

28.4 No. The asynchronous method on which you `await` is responsible for creating any new threads.

28.5 False. When updating a control from a non-GUI thread, you must schedule that update to be performed by the GUI thread. To do so, you check the `Form`'s inherited `InvokeRequired` property. If this property's value is true, the code is executing in a non-GUI thread and must not update the GUI directly. Instead, you call `Form`'s inherited `Invoke` method, which receives as an argument a `Delegate` representing a method to invoke. In this example, we pass a `MethodInvoker`, which is a `Delegate` that invokes a method. The `MethodInvoker` is initialized here with a lambda expression that calls `AppendText`. Line 86 schedules this `MethodInvoker` to execute in the GUI thread. When that occurs, line 88 updates the `outputTextBox`.

Exercises

28.6 Investigate other compute-intensive calculations, then modify the example of Fig. 28.1 to perform a different compute-intensive calculation asynchronously.

28.7 Modify the example of Fig. 28.3 to process the results by using the array returned by the `Task` produced by `Task` method `WhenAll`.

28.8 Investigate other web services at a site like `www.programmableweb.com`. Locate a REST web service that returns XML, then modify the example of Fig. 28.4 to invoke the web service asynchronously using the methods of class `WebClient`. Parse the results using LINQ to XML, then display the results as appropriate for the type of data returned.