Programming III

Centennial College

Week#11- ONLINE 2020 Winter

Topic: Asynchronous Programming

Objectives

- What is Thread and Task?
- What is Asynchronous programming?
- Why is asynchronous programming is important?
- How to adapt this in my applications

Thread

- Thread is the fundamental unit of execution
- More than one thread can be executing inside the same process(i.e., application)
- On a single-processor machine, the operating system is switching rapidly between the threads, giving the appearance of simultaneous execution
- Contained in namespace ***System.Threading***

Task

- An abstraction mechanism to threads
- A task represents some unit of work, that can run in a separate thread, that should be done
- Allow to build relations between tasks, e.g., one task should continue when the first one is completed; can organize tasks in hierarchy
- Contained in ***System.Threading.Task***

What is Asynchronous Programming

- A technique for writing apps containing tasks that can execute asynchronously to improve app performance and GUI responsiveness in apps with long-running or computer-intensive tasks
- It is suitable for I/O operations(i.e.,, Disk IO, database I/O, Web/API)

Asynchronous Programming Example

- Show a downloading indicator
- Start asynchronous operation
  - Down the file
  - Modify the indicator
  - Asynchronous operation completed
- Render the result on screen
- Remove indicator

Performance Tip 23.1

A problem with single-threaded applications that can lead to poor responsiveness is that lengthy activities must complete before *others can begin. In a multithreaded application, threads can be distributed across multiple cores (if available) so that multiple tasks execute in parallel and the application can operate more efficiently. Multithreading can also increase performance on single-processor systems—when one thread cannot proceed (because, for example, it's waiting for the result of an I/O operation), another can use the processor.*

async and await

- async modifier indicates that a method or lambda expression contains at least one await expression
- 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
- The await keyword will pause execution of the method until a result is available without blocking the calling (UI) thread
- Mark a method with async does not automatically make the code inside it asynchronous, async and await have to work together
- The async and await mechanism does not create new threads

Software Engineering Observation 23.1

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

```csharp
// Fig. 23.1: FibonacciForm.cs
// Performing a compute-intensive calculation from a GUI app
using System;
using System.Threading.Tasks;
using System.Windows.Forms;

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();
      }

```

Fig. 23.1 | Performing a compute-intensive calculation from a GUI app. (Part 1 of 6.)

```
      // 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 = int.Parse(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();
      }

```

Fig. 23.1 | Performing a compute-intensive calculation from a GUI app. (Part 2 of 6.)

```
    // 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 = $"Fibonacci of {count}:";
       syncResultLabel.Text = n2.ToString();
    }

```

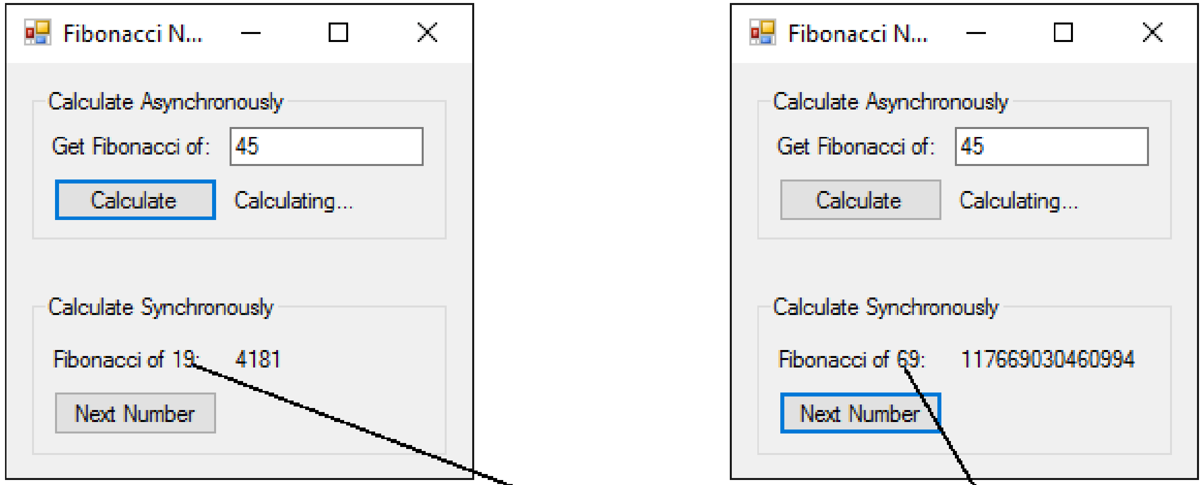
Fig. 23.1 | Performing a compute-intensive calculation from a GUI app. (Part 3 of 6.)

```
            // recursive method Fibonacci; calculates nth Fibonacci number
            public long Fibonacci(long n)
            {
               if (n == 0 || n == 1)
               {
                  return n;
               }
               else
               {
                  return Fibonacci(n - 1) + Fibonacci(n - 2);
               }
            }
         }
      }
```

Fig. 23.1 | Performing a compute-intensive calculation from a GUI app. (Part 4 of 6.)

a) GUI after `Fibonacci(45)` began executing in a separate thread

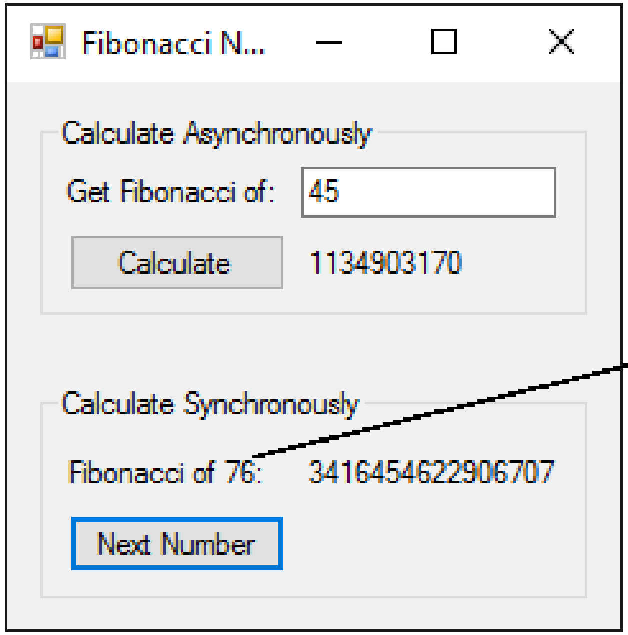
b) GUI while `Fibonacci(45)` was still executing in a separate thread



Each time you click **Next Number** the app updates this `Label` to indicate the next Fibonacci number being calculated, then immediately displays the result to the right.

Fig. 23.1 | Performing a compute-intensive calculation from a GUI app. (Part 5 of 6.)

c) GUI after `Fibonacci(45)` completed



Each time you click **Next Number** the app updates this `Label` to indicate the next Fibonacci number being calculated, then immediately displays the result to the right.

Fig. 23.1 | Performing a compute-intensive calculation from a GUI app. (Part 6 of 6.)

```
// Fig. 23.2: SynchronousTestForm.cs
// Fibonacci calculations performed sequentially
using System;
using System.Windows.Forms;

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

```

Fig. 23.2 | Fibonacci calculations performed sequentially. (Part 1 of 6.)

```
        // 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($"Fibonacci(46) = {result1}\r\n");
           double minutes = (endTime1 - startTime1).TotalMinutes;
           outputTextBox.AppendText(
              $"Calculation time = {minutes:F6} minutes\r\n\r\n");
```

Fig. 23.2 | Fibonacci calculations performed sequentially. (Part 2 of 6.)

```
         // 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($"Fibonacci(45) = {result2}\r\n");
         minutes = (endTime2 - startTime2).TotalMinutes;
         outputTextBox.AppendText(
            $"Calculation time = {minutes:F6} minutes\r\n\r\n");

         // show total calculation time
         double totalMinutes = (endTime2 - startTime1).TotalMinutes;
         outputTextBox.AppendText(
            $"Total calculation time = {totalMinutes:F6} minutes\r\n");
      }
```

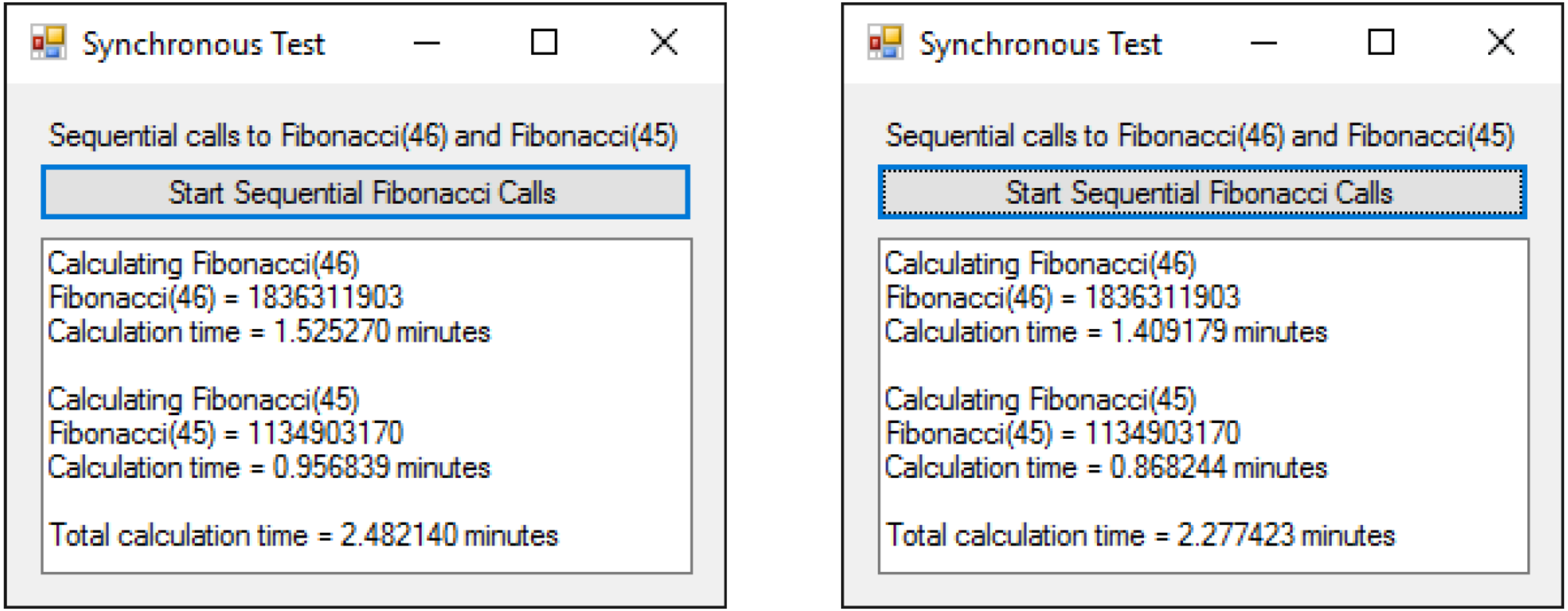
Fig. 23.2 | Fibonacci calculations performed sequentially. (Part 3 of 6.)

```

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

Fig. 23.2 | Fibonacci calculations performed sequentially. (Part 4 of 6.)

a) Outputs on a Dual-Core Windows 10 Computer



b) Outputs on a Single-Core Windows 10 Computer

Fig. 23.2 | Fibonacci calculations performed sequentially. (Part 5 of 6.)

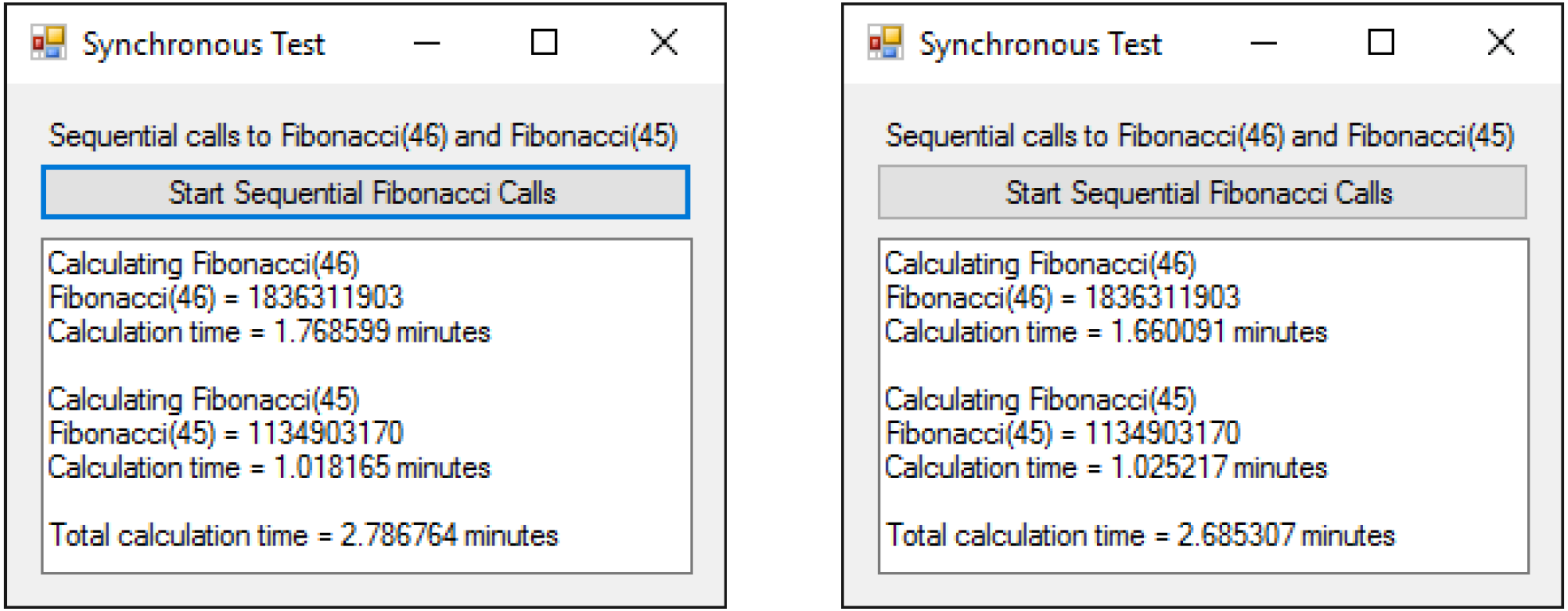


Fig. 23.2 | Fibonacci calculations performed sequentially. (Part 6 of 6.)

```
// Fig. 23.3: 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();
      }

```

Fig. 23.3 | Fibonacci calculations performed in separate threads. (Part 1 of 8.)

```
        // 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));

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

```

Fig. 23.3 | Fibonacci calculations performed in separate threads. (Part 2 of 8.)

```
         // 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
         double totalMinutes = (endTime - startTime).TotalMinutes;
         outputTextBox.AppendText(
            $"Total calculation time = {totalMinutes:F6} minutes\r\n");
      }

```

Fig. 23.3 | Fibonacci calculations performed in separate threads. (Part 3 of 8.)

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

         AppendText($"Calculating Fibonacci({n})");
         result.StartTime = DateTime.Now;
         long fibonacciValue = Fibonacci(n);
         result.EndTime = DateTime.Now;

         AppendText($"Fibonacci({n}) = {fibonacciValue}");
         double minutes =
            (result.EndTime - result.StartTime).TotalMinutes;
         AppendText($"Calculation time = {minutes:F6} minutes\r\n");

         return result;
      }

```

Fig. 23.3 | Fibonacci calculations performed in separate threads. (Part 4 of 8.)

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

```

Fig. 23.3 | Fibonacci calculations performed in separate threads. (Part 5 of 8.)

```
        // append text to outputTextBox in UI thread
        public void AppendText(String text)
        {
            if (InvokeRequired) // not GUI thread, so add to GUI thread
            {
                Invoke(new MethodInvoker(() => AppendText(text)));
            }
            else // GUI thread so append text
            {
                outputTextBox.AppendText(text + "\r\n");
            }
        }
    }
}
```

Fig. 23.3 | Fibonacci calculations performed in separate threads. (Part 6 of 8.)

a) Outputs on a Dual-Core Windows 10 Computer

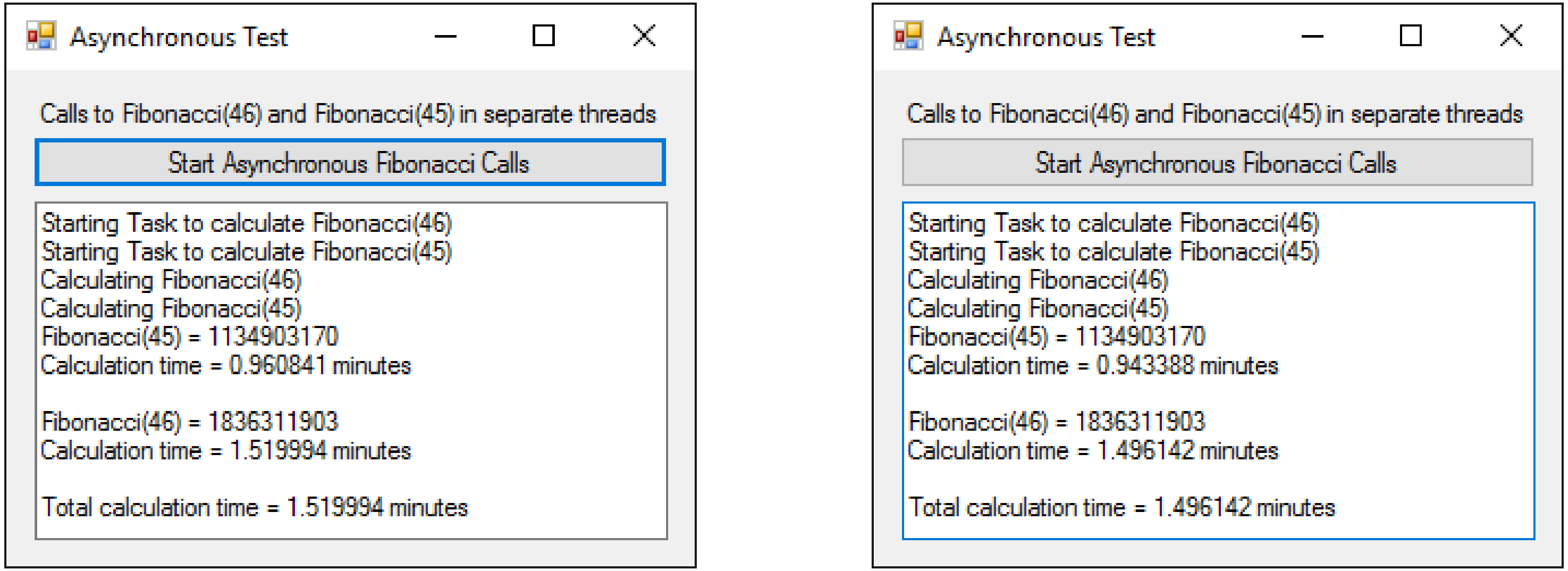
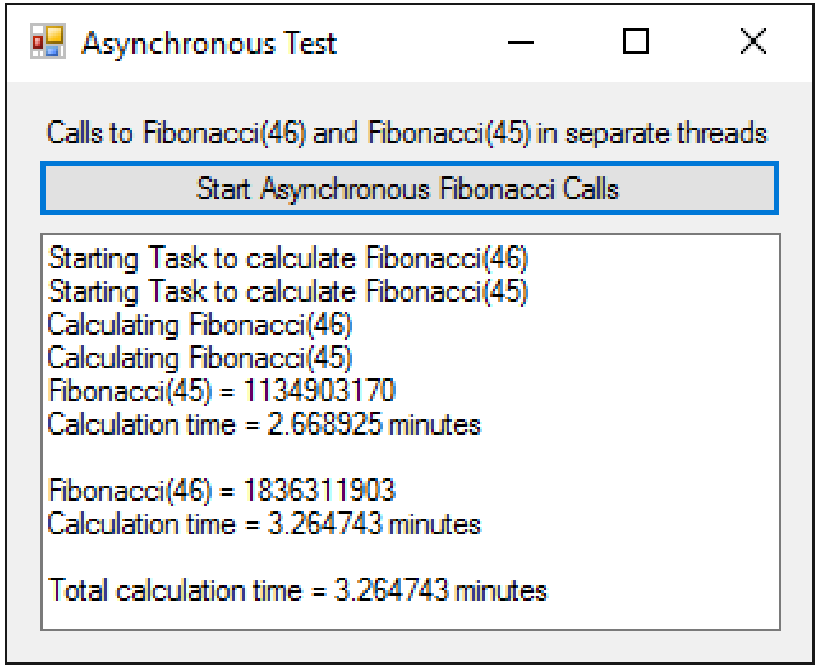


Fig. 23.3 | Fibonacci calculations performed in separate threads. (Part 7 of 8.)

b) Outputs on a Single-Core Windows 10 Computer



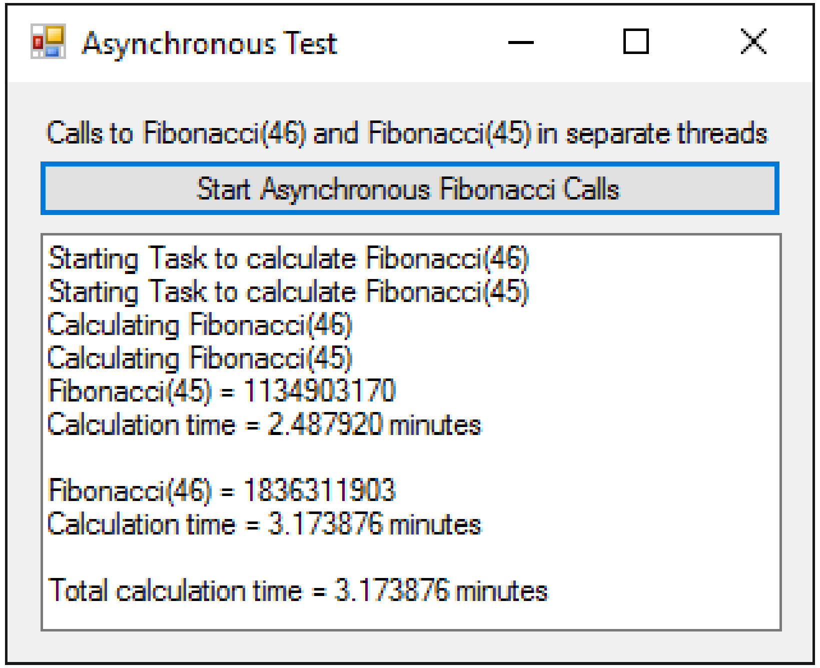


Fig. 23.3 | Fibonacci calculations performed in separate threads. (Part 8 of 8.)

```
// Fig. 23.6: FindPrimes.cs
// Displaying an asynchronous task's progress and intermediate results
using System;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace FindPrimes
{
   public partial class FindPrimesForm : Form
   {
      // used to enable cancelation of the async task
      private bool Canceled { get; set; } = false;
      private bool[] primes; // array used to determine primes

      public FindPrimesForm()
      {
         InitializeComponent();
         progressBar.Minimum = 2; // 2 is the smallest prime number
         percentageLabel.Text = $"{0:P0}"; // display 0 %
      }

```

Fig. 23.6 | Displaying an asynchronous task's progress and intermediate results. (Part 1 of 7.)

```
      // handles getPrimesButton's click event
      private async void getPrimesButton_Click(object sender, EventArgs e)
      {
         // get user input
         var maximum = int.Parse(maxValueTextBox.Text);

         // create array for determining primes
         primes = Enumerable.Repeat(true, maximum).ToArray();

         // reset Canceled and GUI
         Canceled = false;
         getPrimesButton.Enabled = false; // disable getPrimesButton
         cancelButton.Enabled = true; // enable cancelButton
         primesTextBox.Text = string.Empty; // clear primesTextBox
         statusLabel.Text = string.Empty; // clear statusLabel
         percentageLabel.Text = $"{0:P0}"; // display 0 %
         progressBar.Value = progressBar.Minimum; // reset progressBar min
         progressBar.Maximum = maximum; // set progressBar max

         // show primes up to maximum
         int count = await FindPrimes(maximum);
         statusLabel.Text = $"Found {count} prime(s)";
      }

```

Fig. 23.6 | Displaying an asynchronous task's progress and intermediate results. (Part 2 of 7.)

```
        // displays prime numbers in primesTextBox
        private async Task<int> FindPrimes(int maximum)
        {
           var primeCount = 0;

           // find primes less than maximum
           for (var i = 2; i < maximum && !Canceled; ++i)
           {
              // if i is prime, display it
              if (await Task.Run(() => IsPrime(i)))
              {
                 ++primeCount; // increment number of primes found
                 primesTextBox.AppendText($"{i}{Environment.NewLine}");
              }

              var percentage = (double)progressBar.Value /
                 (progressBar.Maximum - progressBar.Minimum + 1);
              percentageLabel.Text = $"{percentage:P0}";
              progressBar.Value = i + 1; // update progress
           }
```

Fig. 23.6 | Displaying an asynchronous task's progress and intermediate results. (Part 3 of 7.)

```

         // display message if operation was canceled
         if (Canceled)
         {
            primesTextBox.AppendText($"Canceled{Environment.NewLine}");
         }

         getPrimesButton.Enabled = true; // enable getPrimesButton
         cancelButton.Enabled = false; // disable cancelButton
         return primeCount;
      }

```

Fig. 23.6 | Displaying an asynchronous task's progress and intermediate results. (Part 4 of 7.)

```
        // check whether value is a prime number
        // and mark all multiples as not prime
        public bool IsPrime(int value)
        {
           // if value is prime, mark all of multiples
           // as not prime and return true
           if (primes[value])
           {
              // mark all multiples of value as not prime
              for (var i = value + value; i < primes.Length; i += value)
              {
                 primes[i] = false; // i is not prime
              }

              return true;
           }
           else
           {
              return false;
           }
        }
```

Fig. 23.6 | Displaying an asynchronous task's progress and intermediate results. (Part 5 of 7.)

```csharp

      // if user clicks Cancel Button, stop displaying primes
      private void cancelButton_Click(object sender, EventArgs e)
      {
         Canceled = true;
         getPrimesButton.Enabled = true; // enable getPrimesButton
         cancelButton.Enabled = false; // disable cancelButton
      }
   }
}
```

Fig. 23.6 | Displaying an asynchronous task's progress and intermediate results. (Part 6 of 7.)

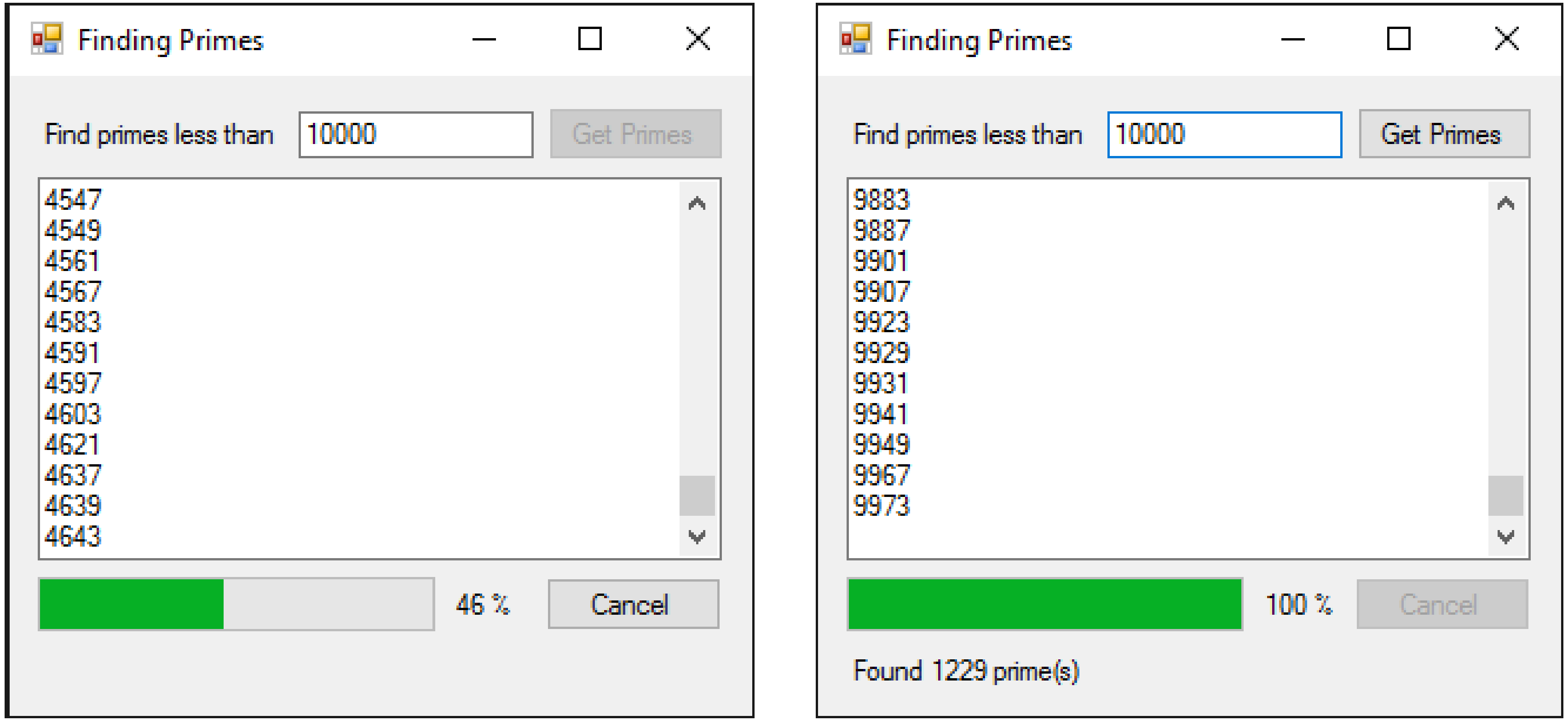


Fig. 23.6 | Displaying an asynchronous task's progress and intermediate results. (Part 7 of 7.)

Asynchronous Programming(Con't)

- Asynchronous is when work is being executed on a different thread, and does not impact the main thread of the application

Good Practices

- Always use async and await together
- Always return a Task from an asynchronous method
- Always await an asynchronous method to validate the operation
- User async and await all the way up the chain

Reference

- https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread?view=netframework-4.7.2
- https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1?view=netframework-4.7.2