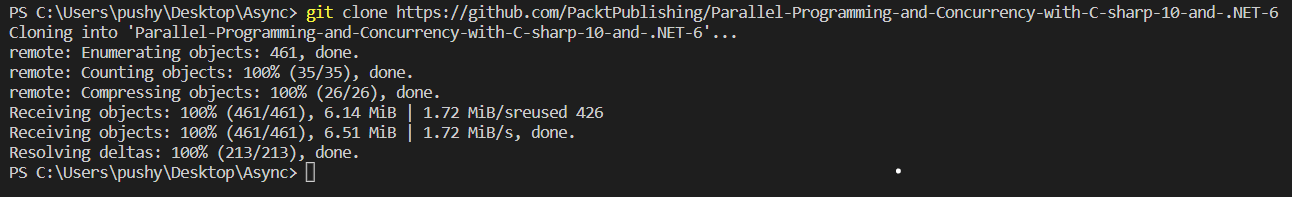
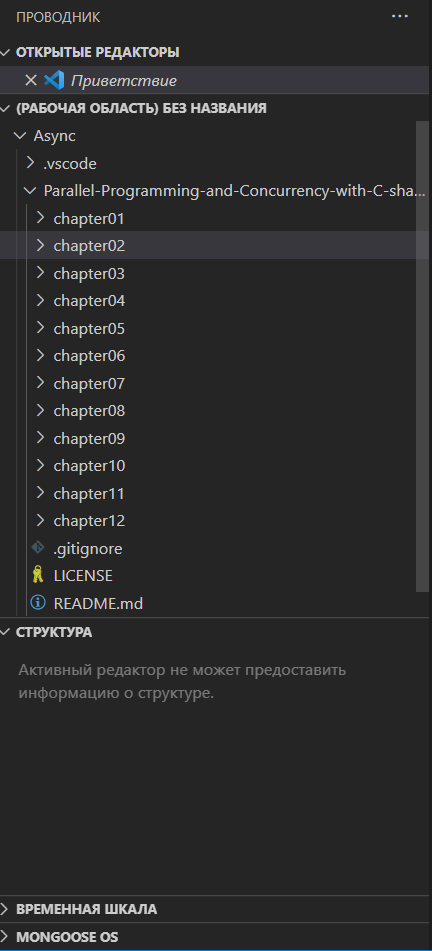
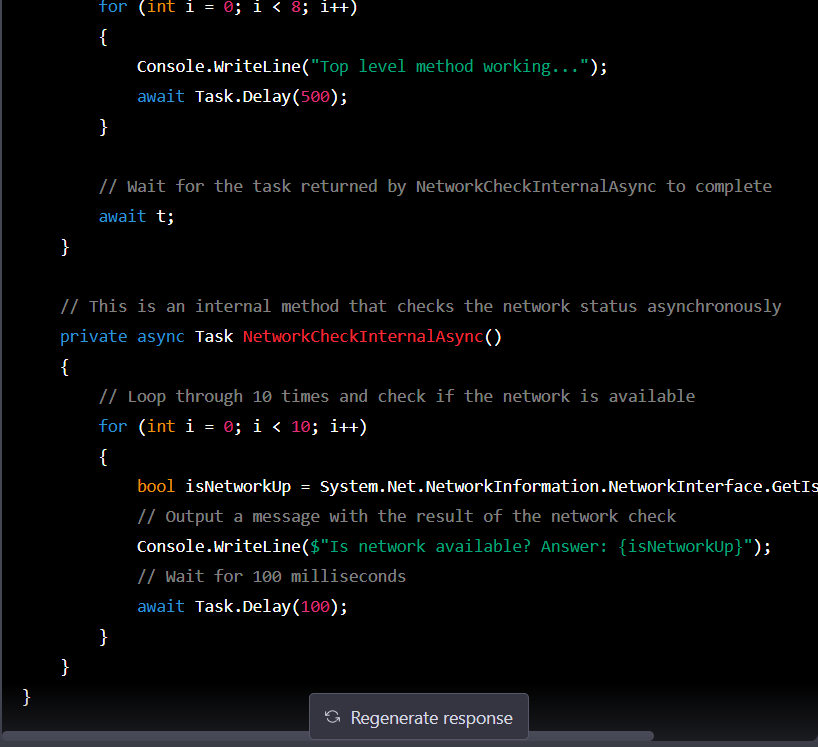
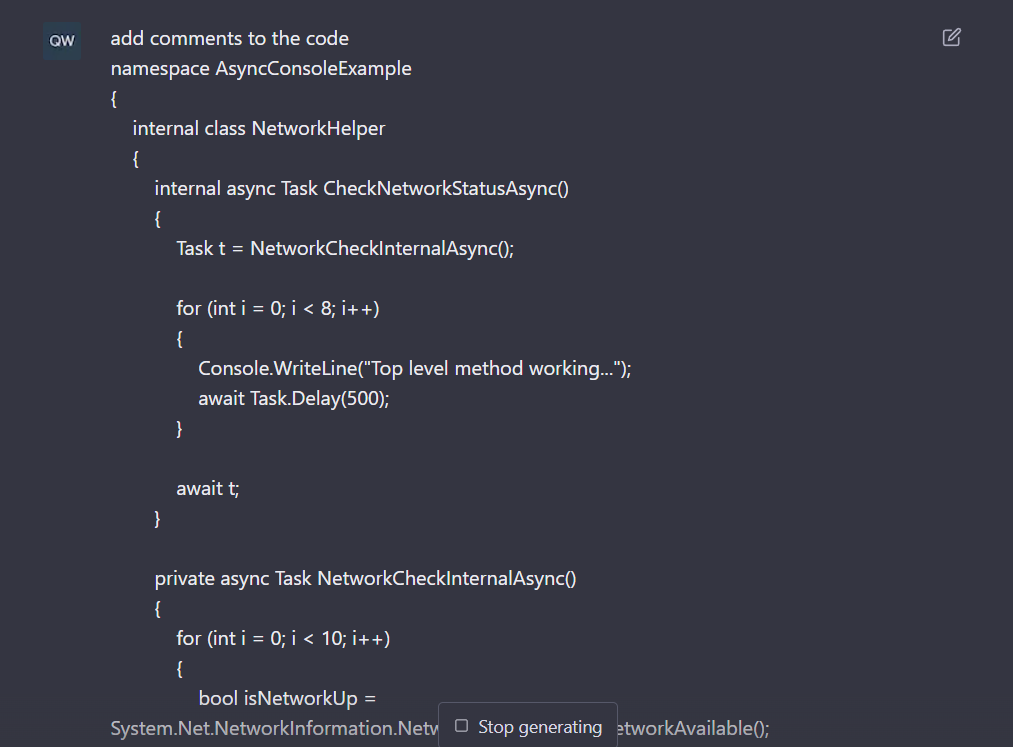
Async and Await Assignm ent

TASK 1. Provide screenshots proving that part 1 was completed.





TASK 2. Make the chatGPT generate comments for your code. Read through the commented programs to make sure all logic is transparent and clear. If some explanation is missing/needed, request for more clarifcation. Run the code. Provide Source Code and Output fully visible. Your source code should demonstrate the comments generated by chatGPT and run successfully. Explain in your own words what programs from Chapter 2 do and what you learn from them.



**AsyncConsoleExample**

**NetworkHelper.cs**

// This is a namespace for an example of an async console application

namespace AsyncConsoleExample

{

    // This is a class that helps with network related tasks

    internal class NetworkHelper

    {

        // This method checks the network status asynchronously

        internal async Task CheckNetworkStatusAsync()

        {

            // Call the NetworkCheckInternalAsync method and store the returned task in a variable

            Task t = NetworkCheckInternalAsync();

            // Loop through 8 times and output a message every half a second

            for (int i = 0; i < 8; i++)

            {

                Console.WriteLine("Top level method working...");

                await Task.Delay(500);

            }

            // Wait for the task returned by NetworkCheckInternalAsync to complete

            await t;

        }

        // This is an internal method that checks the network status asynchronously

        private async Task NetworkCheckInternalAsync()

        {

            // Loop through 10 times and check if the network is available

            for (int i = 0; i < 10; i++)

            {

                bool isNetworkUp = System.Net.NetworkInformation.NetworkInterface.GetIsNetworkAvailable();

                // Output a message with the result of the network check

                Console.WriteLine($"Is network available? Answer: {isNetworkUp}");

                // Wait for 100 milliseconds

                await Task.Delay(100);

            }

        }

    }

}

**Program.cs**

// Import the namespace that contains the NetworkHelper class

using AsyncConsoleExample;

// Output a message to the console

Console.WriteLine("Hello, async!");

// Create a new instance of the NetworkHelper class

var networkHelper = new NetworkHelper();

// Call the CheckNetworkStatusAsync method asynchronously and wait for it to complete

await networkHelper.CheckNetworkStatusAsync();

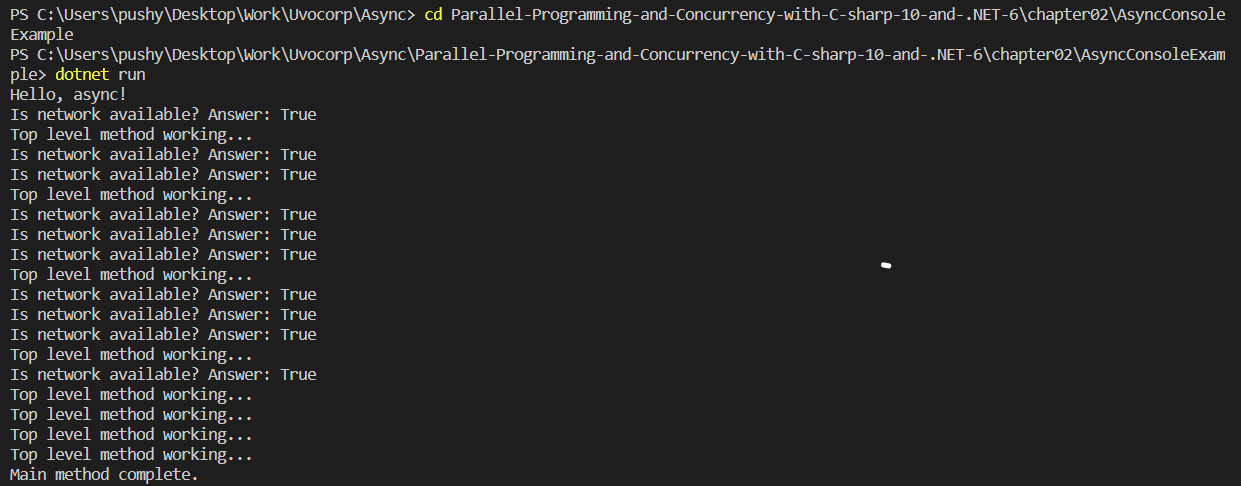
// Output a message to the console indicating that the main method has completed

Console.WriteLine("Main method complete.");

// Wait for a key press before exiting the console application

Console.ReadKey();

**Output**

****

This is an illustration of an asynchronous desktop application written in C#. The network status is checked asynchronously by the application using the CheckNetworkStatusAsync and NetworkCheckInternalAsync methods of the NetworkHelper class. The NetworkCheckInternalAsync method is called by the CheckNetworkStatusAsync method; this method continuously checks the network status in a cycle and sends the results to the console. The top-level CheckNetworkStatusAsync function, which performs its own cycle and outputs a message every half-second, runs in the background.

In summary, the program demonstrates how to use C#'s async/await syntax to perform non-blocking actions and maintain program responsiveness while waiting for asynchronous tasks to complete. It also demonstrates how to develop and use a helper class to contain features unique to a particular domain. (in this case, network-related tasks). Overall, the program emphasizes how asynchronous programming can improve the speed and user interface of console applications.

**BackgroundPingConsoleApp\_pool**

**Program.cs**

// Output a message to the console

Console.WriteLine("Hello, World!");

// Queue a new work item to the thread pool

ThreadPool.QueueUserWorkItem((o) =>

{

    // Loop through 20 times and check if the network is available

    for (int i = 0; i < 20; i++)

    {

        bool isNetworkUp = System.Net.NetworkInformation.NetworkInterface.GetIsNetworkAvailable();

        // Output a message with the result of the network check

        Console.WriteLine($"Is network available? Answer: {isNetworkUp}");

        // Pause the thread for 100 milliseconds

        Thread.Sleep(100);

    }

});

// Loop through 10 times and output a message every half second

for (int i = 0; i < 10; i++)

{

    Console.WriteLine("Main thread working...");

    Task.Delay(500);

}

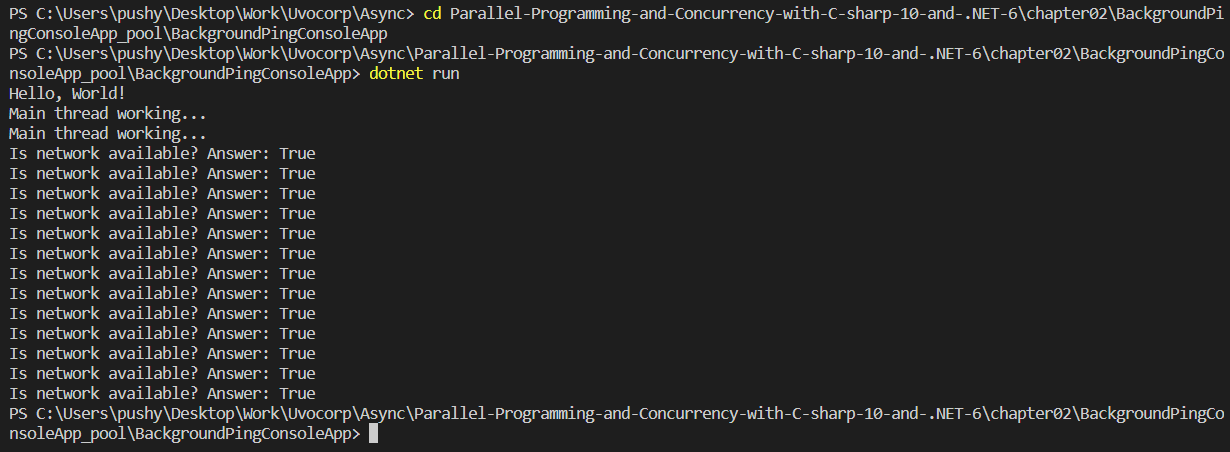
// Output a message indicating that the work is done

Console.WriteLine("Done");

// Wait for a key press before exiting the console application

Console.ReadKey();

Output



This software is a console tool that performs a similar task-checking network availability-but it does so in a different way. Instead of using async/await, it queues a work item in the thread pool using the ThreadPool.QueueUserWorkItem function. The thread pool then executes the code provided by the delegate on a separate thread. The work item in the list has a loop that sends the network state to the console 20 times. The main thread and this loop run simultaneously. The main thread's loop produces a message every half second for ten times.

The use of Thread is a major departure from the previous software: instead of working, sleep.Pause pauses the thread's processing by delaying it. Thread.Task allows the process to continue while Sleep prevents it from doing any more work.While waiting for the delay to end, the thread is able to focus on other tasks. As a result, the software is slower and uses resources less efficiently. Overall, the two programs check the network state in the same way, but they use different strategies and make different tradeoffs. While the second program uses the thread pool to perform background tasks, but may stop the thread and reduce parallelism, the first program uses async/await to perform non-blocking I/O operations and keep the program functional.

**ParallelExamples**

**ParallelForEachExample.cs**

// This code is contained in the ParallelExamples namespace and defines a class called ParallelForEachExample

namespace ParallelExamples

{

    internal class ParallelForEachExample

    {

        // This method executes a parallel loop over the provided list of numbers

        internal void ExecuteParallelForEach(IList<int> numbers)

        {

            // Use the Parallel.ForEach method to loop over the numbers list in parallel

            Parallel.ForEach(numbers, number =>

            {

                // Check if the current time contains the current number in the loop

                bool timeContainsNumber = DateTime.Now.ToLongTimeString().Contains(number.ToString());

                // Output a message to the console depending on whether or not the current time contains the current number

                if (timeContainsNumber)

                {

                    Console.WriteLine($"The current time contains number {number}. Thread id: {Thread.CurrentThread.ManagedThreadId}");

                }

                else

                {

                    Console.WriteLine($"The current time does not contain number {number}. Thread id: {Thread.CurrentThread.ManagedThreadId}");

                }

            });

        }

    }

}

**ParallelInvokeExample.cs**

//namespace for examples related to parallel programming

namespace ParallelExamples

{

    //class for parallel invocation example

    internal class ParallelInvokeExample

    {

        //method to do work in parallel

        internal void DoWorkInParallel()

        {

            //invoke multiple delegates in parallel using Parallel.Invoke method

            Parallel.Invoke(

                DoComplexWork,                                              //1st delegate

                () => {                                                     //2nd delegate with lambda expression

                    Console.WriteLine($"Hello from lambda expression. Thread id: {Thread.CurrentThread.ManagedThreadId}");

                },

                new Action(() =>                                            //3rd delegate using Action delegate

                {

                    Console.WriteLine($"Hello from Action. Thread id: {Thread.CurrentThread.ManagedThreadId}");

                }),

                delegate ()                                                 //4th delegate using anonymous method

                {

                    Console.WriteLine($"Hello from delegate. Thread id: {Thread.CurrentThread.ManagedThreadId}");

                }

            );

        }

        //method to do complex work

        private void DoComplexWork()

        {

            Console.WriteLine($"Hello from DoComplexWork method. Thread id: {Thread.CurrentThread.ManagedThreadId}");

        }

    }

}

**ParallelLinqExample.cs**

namespace ParallelExamples

{

    internal class ParallelLinqExample

    {

        // Executes a LINQ query to get even numbers from a list

        internal void ExecuteLinqQuery(IList<int> numbers)

        {

            var evenNumbers = numbers.Where(n => n % 2 == 0);

            // Outputs even numbers obtained using regular LINQ query

            OutputNumbers(evenNumbers, "Regular");

        }

        // Executes a Parallel LINQ query to get even numbers from a list

        internal void ExecuteParallelLinqQuery(IList<int> numbers)

        {

            var evenNumbers = numbers.AsParallel().Where(n => IsEven(n));

            // Outputs even numbers obtained using Parallel LINQ query

            OutputNumbers(evenNumbers, "Parallel");

        }

        // Method to check if a number is even

        private bool IsEven(int number)

        {

            Task.Delay(100); // delay to simulate a computationally intensive operation

            return number % 2 == 0;

        }

        // Method to output a string representation of a list of numbers

        private void OutputNumbers(IEnumerable<int> numbers, string loopType)

        {

            var numberString = string.Join(",", numbers);

            Console.WriteLine($"{loopType} number string: {numberString}");

        }

    }

}

**Program.cs**

// This code imports the ParallelExamples namespace

using ParallelExamples;

// Create an instance of the ParallelInvokeExample class and call the DoWorkInParallel method to execute a set of tasks in parallel

var parallelExample = new ParallelInvokeExample();

parallelExample.DoWorkInParallel();

// Create a list of integers and an instance of the ParallelForEachExample class, and call the ExecuteParallelForEach method to execute a loop over the list in parallel

var numbers = new List<int> { 1, 3, 5, 7, 9, 0 };

var foreachExample = new ParallelForEachExample();

foreachExample.ExecuteParallelForEach(numbers);

// Create a list of integers and an instance of the ParallelLinqExample class, and call the ExecuteLinqQuery and ExecuteParallelLinqQuery methods to execute LINQ queries on the list

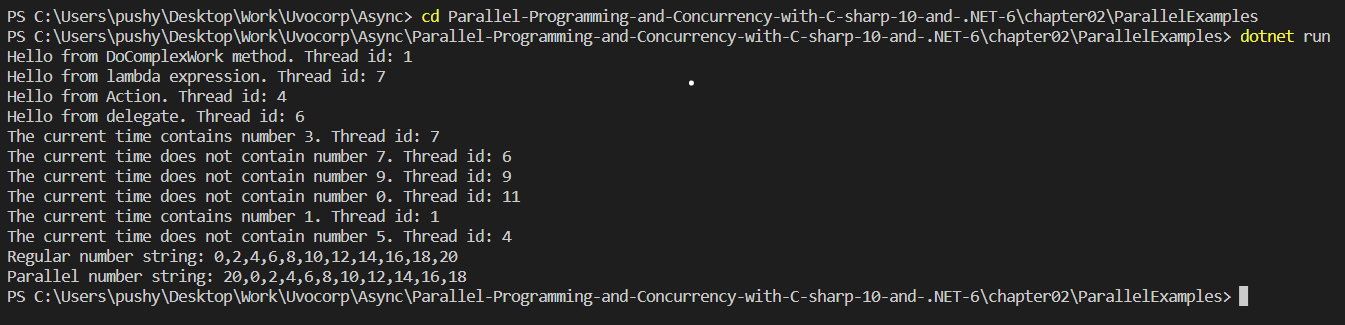
var linqNumbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };

var linqExample = new ParallelLinqExample();

linqExample.ExecuteLinqQuery(linqNumbers);

linqExample.ExecuteParallelLinqQuery(linqNumbers);

**Output**

****

The scripts present three parallel computing examples using Parallel.ForEach, Parallel.Invoke, and Parallel LINQ. Each application is called from the application.cs file and is housed in its own class within the ParallelExamples namespace.

In the first application, ParallelForEachExample.cs, a class called ExecuteParallelForEach is defined, which accepts an array of integer inputs and has a function called ExecuteParallelForEach. This method uses the Parallel.ForEach method to iterate through the list of numbers in parallel and determine whether the current time includes the current number in the iteration. The function sends a notification to the console based on whether or not the current time includes the current number.

The second program, ParallelInvokeExample.cs, defines a class with a function named DoWorkInParallel. This approach uses the Parallel.Initiate function to invoke multiple delegates simultaneously. The delegates consist of a lambda expression, an Action delegate, an unnamed method, and a method called DoComplexWork. Each delegate writes a notification to the console.

ExecuteLinqQuery and ExecuteParallelLinqQuery are two functions defined in a class in the third application, ParallelLinqExample.cs. Both of these techniques use LINQ lookups to extract even numbers from lists of integers as input. ExecuteParallelLinqQuery uses a parallel LINQ query, while ExecuteLinqQuery uses a regular LINQ query.

In addition, the class has a secret method called IsEven that mimics a numerically expensive process with a pause while determining whether an integer is even. The class also has a secret function called OutputNumbers, which sends a collection of numbers to the console as a string.

To run the parallel programming examples, the Program.cs file creates copies of each class and uses its methods.

This code demonstrates how to perform tasks simultaneously and increase speed by using parallel programming methods in C#. To achieve parallelism in various situations, ForEach, Parallel.Invoke, and Parallel LINQ are used.

**WorkingWithTimers**

**Program.cs**

// The namespace WorkingWithTimers holds the TimerForm class and the Program class that will execute the form.

namespace WorkingWithTimers

{

        // The Program class is internal as it only executes the TimerForm class.

    internal static class Program

    {

        /// <summary>

        ///  The main entry point for the application.

        /// </summary>

        [STAThread]

        static void Main()

        {

            // To customize application configuration such as set high DPI settings or default font,

            // see https://aka.ms/applicationconfiguration.

            // Initialize the application configuration.

            ApplicationConfiguration.Initialize();

            // Run the TimerForm.

            Application.Run(new TimerForm());

        }

    }

}

**ThreadingTimerSample.cs**

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WorkingWithTimers

{

    internal class ThreadingTimerSample

    {

        private System.Threading.Timer? \_timer;

        // Start the timer if it hasn't already been started

        public void StartTimer()

        {

            if (\_timer == null)

            {

                InitializeTimer();

            }

        }

        // Dispose the timer asynchronously

        public async Task DisposeTimerAsync()

        {

            if (\_timer != null)

            {

                await \_timer.DisposeAsync();

            }

        }

        // Initialize the timer

        private void InitializeTimer()

        {

            // Create a new instance of the MessageUpdater class

            var updater = new MessageUpdater();

            // Create a new System.Threading.Timer that will call the TimerFired method every 1 second

            \_timer = new System.Threading.Timer(

                callback: new TimerCallback(TimerFired),

                state: updater,

                dueTime: 500,

                period: 1000);

        }

        // This method is called by the timer every 1 second

        private void TimerFired(object? state)

        {

            // Check for new messages

            int messageCount = CheckForNewMessageCount();

            // If there are new messages and the state object is a MessageUpdater instance, update the UI

            if (messageCount > 0 && state is MessageUpdater updater)

            {

                updater.Update(messageCount);

            }

        }

        // This method generates a random number of new messages to simulate checking for new messages

        private int CheckForNewMessageCount()

        {

            // Generate a random number of messages to return

            return new Random().Next(100);

        }

    }

    // This class is responsible for updating the UI with the new message count

    internal class MessageUpdater

    {

        internal void Update(int messageCount)

        {

            Debug.WriteLine($"You have {messageCount} new messages!");

        }

    }

}

**TimerForm.cs**

namespace WorkingWithTimers

{

    public partial class TimerForm : Form

    {

        private TimerSample \_timerSample;

        private ThreadingTimerSample \_threadingTimerSample;

        // Constructor

        public TimerForm()

        {

            InitializeComponent();

            // Initialize the TimerSample and ThreadingTimerSample instances

            \_timerSample = new TimerSample();

            \_threadingTimerSample = new ThreadingTimerSample();

        }

        // Event handler for the startTimerButton Click event

        // Calls the StartTimer method of the TimerSample instance

        private void startTimerButton\_Click(object sender, EventArgs e)

        {

            \_timerSample.StartTimer();

        }

        // Event handler for the stopTimerButton Click event

        // Calls the StopTimer method of the TimerSample instance

        private void stopTimerButton\_Click(object sender, EventArgs e)

        {

            \_timerSample.StopTimer();

        }

        // Event handler for the startThreadingTimerButton Click event

        // Calls the StartTimer method of the ThreadingTimerSample instance

        private void startThreadingTimerButton\_Click(object sender, EventArgs e)

        {

            \_threadingTimerSample.StartTimer();

        }

        // Event handler for the stopThreadingTimerButton Click event

        // Calls the DisposeTimerAsync method of the ThreadingTimerSample instance asynchronously

        private async void stopThreadingTimerButton\_Click(object sender, EventArgs e)

        {

            await \_threadingTimerSample.DisposeTimerAsync();

        }

    }

}

**TimerForm.Designer.cs**

namespace WorkingWithTimers {

    partial class TimerForm {

         /// <summary>

        ///  Required designer variable.

        /// </summary>

        private System.ComponentModel.IContainer components = null;

        /// <summary>

        /// Clean up any resources being used.

        /// </summary>

        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

        protected override void Dispose(bool disposing) {

            if (disposing && (components != null)) {

                components.Dispose();

            }

            base.Dispose(disposing);

        }

        #region Windows Form Designer generated code

        /// <summary>

        /// Required method for Designer support - do not modify

        /// the contents of this method with the code editor.

        /// </summary>

        private void InitializeComponent() {

            // Create a group box for the timer

            this.timerGroupBox = new System.Windows.Forms.GroupBox();

            // Create a stop timer button

            this.stopTimerButton = new System.Windows.Forms.Button();

            // Create a start timer button

            this.startTimerButton = new System.Windows.Forms.Button();

            // Create a group box for the threading timer

            this.groupBox1 = new System.Windows.Forms.GroupBox();

            // Create a stop threading timer button

            this.stopThreadingTimerButton = new System.Windows.Forms.Button();

            // Create a start threading timer button

            this.startThreadingTimerButton = new System.Windows.Forms.Button();

            // Suspend layout for both group boxes

            this.timerGroupBox.SuspendLayout();

            this.groupBox1.SuspendLayout();

            this.SuspendLayout();

            //

            // timerGroupBox

            //

            // Add the stop and start timer buttons and set the location and size of the group box

            this.timerGroupBox.Controls.Add(this.stopTimerButton);

            this.timerGroupBox.Controls.Add(this.startTimerButton);

            this.timerGroupBox.Location = new System.Drawing.Point(35, 38);

            this.timerGroupBox.Name = "timerGroupBox";

            this.timerGroupBox.Size = new System.Drawing.Size(400, 308);

            this.timerGroupBox.TabIndex = 0;

            this.timerGroupBox.TabStop = false;

            this.timerGroupBox.Text = "Timer";

            //

            // stopTimerButton

            //

            // Set the location and size of the stop timer button and add a click event listener

            this.stopTimerButton.Location = new System.Drawing.Point(116, 150);

            this.stopTimerButton.Name = "stopTimerButton";

            this.stopTimerButton.Size = new System.Drawing.Size(150, 46);

            this.stopTimerButton.TabIndex = 1;

            this.stopTimerButton.Text = "Stop Timer";

            this.stopTimerButton.UseVisualStyleBackColor = true;

            this.stopTimerButton.Click += new System.EventHandler(this.stopTimerButton\_Click);

            //

            // startTimerButton

            //

            // Set the location and size of the start timer button and add a click event listener

            this.startTimerButton.Location = new System.Drawing.Point(116, 62);

            this.startTimerButton.Name = "startTimerButton";

            this.startTimerButton.Size = new System.Drawing.Size(150, 46);

            this.startTimerButton.TabIndex = 0;

            this.startTimerButton.Text = "Start Timer";

            this.startTimerButton.UseVisualStyleBackColor = true;

            this.startTimerButton.Click += new System.EventHandler(this.startTimerButton\_Click);

            //

            // groupBox1

            //

            // Add the stop and start threading timer buttons and set the location and size of the group box

            this.groupBox1.Controls.Add(this.stopThreadingTimerButton);

            this.groupBox1.Controls.Add(this.startThreadingTimerButton);

            this.groupBox1.Location = new System.Drawing.Point(495, 38);

            this.groupBox1.Name = "groupBox1";

            this.groupBox1.Size = new System.Drawing.Size(400, 308);

            this.groupBox1.TabIndex = 1;

            this.groupBox1.TabStop = false;

            this.groupBox1.Text = "Timer";

            //

            // stopThreadingTimerButton

            //

            // Set the location and size of the stop threading timer button and add a click event listener

            this.stopThreadingTimerButton.Location = new System.Drawing.Point(66, 150);

            this.stopThreadingTimerButton.Name = "stopThreadingTimerButton";

            this.stopThreadingTimerButton.Size = new System.Drawing.Size(280, 46);

            this.stopThreadingTimerButton.TabIndex = 1;

            this.stopThreadingTimerButton.Text = "Stop Threading Timer";

            this.stopThreadingTimerButton.UseVisualStyleBackColor = true;

            this.stopThreadingTimerButton.Click += new System.EventHandler(this.stopThreadingTimerButton\_Click);

            //

            // startThreadingTimerButton

            //

            // Set the location and size of the start threading timer button and add a click event listener

            this.startThreadingTimerButton.Location = new System.Drawing.Point(66, 62);

            this.startThreadingTimerButton.Name = "startThreadingTimerButton";

            this.startThreadingTimerButton.Size = new System.Drawing.Size(280, 46);

            this.startThreadingTimerButton.TabIndex = 0;

            this.startThreadingTimerButton.Text = "Start Threading Timer";

            this.startThreadingTimerButton.UseVisualStyleBackColor = true;

            this.startThreadingTimerButton.Click += new System.EventHandler(this.startThreadingTimerButton\_Click);

            //

            // TimerForm

            //

            // Set the size of the form and add the group boxes

            this.AutoScaleDimensions = new System.Drawing.SizeF(13F, 32F);

            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;

            this.ClientSize = new System.Drawing.Size(936, 382);

            this.Controls.Add(this.groupBox1);

            this.Controls.Add(this.timerGroupBox);

            this.Name = "TimerForm";

            this.Text = "Working with Timers";

            // Resume layout for both group boxes

            this.timerGroupBox.ResumeLayout(false);

            this.groupBox1.ResumeLayout(false);

            this.ResumeLayout(false);

        }

        #endregion

        private GroupBox timerGroupBox;

        private GroupBox groupBox1;

        private Button stopTimerButton;

        private Button startTimerButton;

        private Button stopThreadingTimerButton;

        private Button startThreadingTimerButton;

    }

}

**TimerSample.cs**

using System.Diagnostics;

namespace WorkingWithTimers

{

    internal class TimerSample : IDisposable

    {

        private System.Timers.Timer? \_timer;

        // Start the timer

        public void StartTimer()

        {

            // If the timer has not been initialized yet, initialize it

            if (\_timer == null)

            {

                InitializeTimer();

            }

            // If the timer is not enabled, enable it

            if (\_timer != null && !\_timer.Enabled)

            {

                \_timer.Enabled = true;

            }

        }

        // Stop the timer

        public void StopTimer()

        {

            // If the timer is enabled, disable it

            if (\_timer != null && \_timer.Enabled)

            {

                \_timer.Enabled = false;

            }

        }

        // Initialize the timer

        private void InitializeTimer()

        {

            \_timer = new System.Timers.Timer

            {

                Interval = 1000

            };

            // Add the elapsed event handler for the timer

            \_timer.Elapsed += \_timer\_Elapsed;

        }

        // Handle the elapsed event of the timer

        private void \_timer\_Elapsed(object? sender, System.Timers.ElapsedEventArgs e)

        {

            // Check for new messages

            int messageCount = CheckForNewMessageCount();

            // If there are new messages, alert the user

            if (messageCount > 0)

            {

                AlertUser(messageCount);

            }

        }

        // Alert the user about new messages

        private void AlertUser(int messageCount)

        {

            Debug.WriteLine($"You have {messageCount} new messages!");

        }

        // Check for new messages and return their count

        private int CheckForNewMessageCount()

        {

            // Generate a random number of messages to return

            return new Random().Next(100);

        }

        // Dispose the timer

        public void Dispose()

        {

            if (\_timer != null)

            {

                // Remove the elapsed event handler for the timer

                \_timer.Elapsed -= \_timer\_Elapsed;

                // Dispose the timer

                \_timer.Dispose();

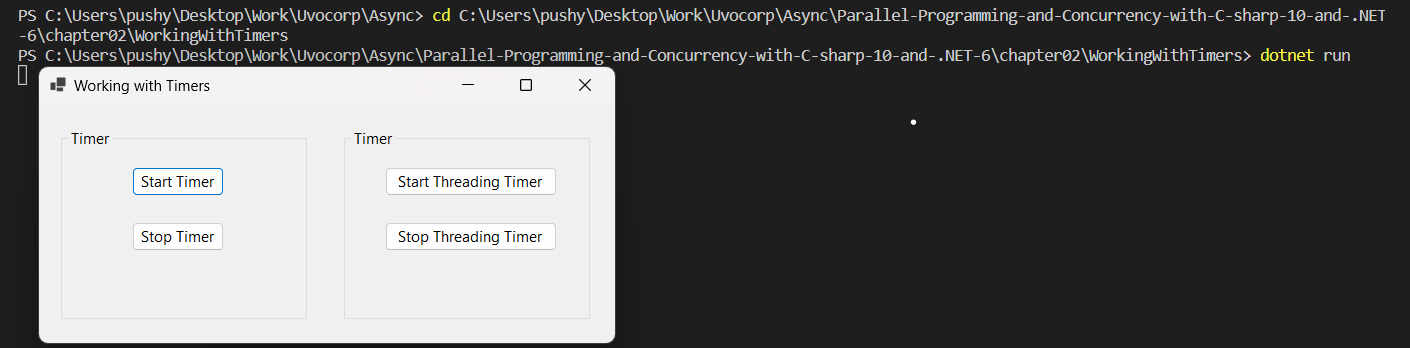
            }

        }

    }

}

**Output**

****

This C# program demonstrates the use of timers in Windows Forms applications. The program consists of three classes: ThreadingTimerSample, TimerSample, and TimerForm.

The startTimerButton, stopTimerButton, startThreadingTimerButton, and stopThreadingTimerButton buttons are all part of the TimerForm class, which is the primary form of the program. The TimerSample starts when the startTimerButton is pressed, and a timer that refreshes the UI every second is started by calling the StartTimer() function. The TimerSample is activated when the stopTimerButton is pressed. The timer is stopped by calling the StopTimer() function. The ThreadingTimerSample is started when the startThreadingTimerButton is pressed.The StartTimer() function is used to initiate a timer that uses a system to refresh the UI once every second. The ThreadingTimerSample is activated when the stopThreadingTimerButton is pressed.The timer is disposed of by calling the DisposeTimerAsync() function.

The UI is updated once every second by the TimerSample class using a System.Windows.Forms.Timer. The StartTimer() and StopTimer() functions are used to start and stop the timer, respectively. Every second, the timer calls the TimerFired() method, which changes the user interface by contacting the Update() method of the MessageUpdater class.

When using a system, the ThreadingTimerSample class is comparable to the TimerSample class.Threading.Timer as opposed to a system.Windows.Forms.Every 1 second, the UI will refresh. The StartTimer() function starts the timer, and the DisposeTimerAsync() method disposes of the timer asynchronously. Every second, the timer calls the TimerFired() method, which updates the UI by contacting the Update() method of the MessageUpdater class.

The task of updating the UI with the new message count falls under the purview of the MessageUpdater class. Using the Debug.WriteLine() function, it changes the UI by sending a message to the Debug output window.

TASK 3. Do the same as in Task 2, but with runnable examples from Chapter 3, Chapter 5, Chapter 6, and for Chapter 8.

**Chaper 3**

**InterlockedSample**

**Interlocked.cs**

namespace InterlockedSample

{

    public class InterlockedExample

    {

        // Declare a private field to store the running total.

        private long \_runningTotal;

        public void PerformCalculations()

        {

            // Initialize the running total to 3.

            \_runningTotal = 3;

            // Invoke two actions in parallel.

            Parallel.Invoke(() =>

            {

                // Call AddValue and wait for it to complete.

                AddValue().Wait();

            }, () =>

            {

                // Call MultiplyValue and wait for it to complete.

                MultiplyValue().Wait();

            });

            // Print the final running total to the console.

            Console.WriteLine($"Running total is {\_runningTotal}");

        }

        private async Task AddValue()

        {

            // Simulate a delay.

            await Task.Delay(100);

            // Add 15 to the running total atomically using Interlocked.

            Interlocked.Add(ref \_runningTotal, 15);

        }

        private async Task MultiplyValue()

        {

            // Simulate a delay.

            await Task.Delay(100);

            // Read the current value of the running total atomically using Interlocked.

            var currentTotal = Interlocked.Read(ref \_runningTotal);

            // Multiply the current value of the running total by 10 and store the result atomically using Interlocked.

            Interlocked.Exchange(ref \_runningTotal, currentTotal \* 10);

        }

    }

}

**Program.cs**

// Import the InterlockedSample namespace.

using InterlockedSample;

// Print "Hello, World!" to the console.

Console.WriteLine("Hello, World!");

// Create a new instance of the InterlockedExample class.

var interlocked = new InterlockedExample();

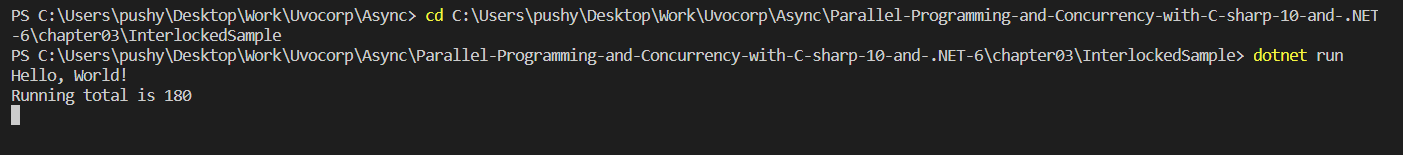
// Call the PerformCalculations method on the InterlockedExample instance.

interlocked.PerformCalculations();

// Wait for the user to press a key before exiting the application.

Console.ReadLine();

Output



This code shows how to use the C# Interlocked class to execute atomic actions from numerous threads on a common variable called \_runningTotal. It sets the initial value of \_runningTotal to 3 before using concurrent to launch two concurrent operations.Invoke(). Interlocked allows one operation to add 15 to \_runningTotal, and the other operation uses Interlocked to multiply \_runningTotal by 10 after calling Add().Exchange(). The console displays the sum of \_runningTotal. To prevent race conditions and other synchronization problems that can occur when multiple processes try to access and modify the same data at the same time, this code demonstrates how to use the Interlocked class to ensure that shared data is modified atomically.

**ThreadingStaticDataExample**

**Program.cs**

// Import the ThreadingStaticDataExample namespace.

using ThreadingStaticDataExample;

// Print "Hello, World!" to the console.

Console.WriteLine("Hello, World!");

// Print the current UTC datetime to the console.

Console.WriteLine($"Current datetime: {DateTime.UtcNow}");

// Create a new instance of the WorkstationHelper class.

var helper = new WorkstationHelper();

// await helper.GetNetworkAvailability();

//Parallel.For(1, 30, async (x) =>

//{

//    await helper.GetNetworkAvailability();

//});

// Call the GetNetworkAvailabilityFromSingleton method asynchronously.

await helper.GetNetworkAvailabilityFromSingleton();

Console.WriteLine($"Network availability last updated {WorkstationState.NetworkConnectivityLastUpdated} for computer {WorkstationState.Name} at IP {WorkstationState.IpAddress}"); // Print the network availability status to the console.

**WorkstationHelper.cs**

// Import the System.Net.NetworkInformation namespace.

using System.Net.NetworkInformation;

namespace ThreadingStaticDataExample

{

    internal class WorkstationHelper

    {

        // Create a private static object to use for locking.

        private static object \_workstationLock = new object();

        internal async Task<bool> GetNetworkAvailability()

        {

            // Wait for 100 milliseconds (async/await pattern).

            await Task.Delay(100);

            // Use the lock statement to synchronize access to the WorkstationState object.

            lock (\_workstationLock)

            {

                // Update the IsNetworkAvailable property of the WorkstationState object.

                WorkstationState.IsNetworkAvailable = NetworkInterface.GetIsNetworkAvailable();

                // Update the NetworkConnectivityLastUpdated property of the WorkstationState object.

                WorkstationState.NetworkConnectivityLastUpdated = DateTime.UtcNow;

            }

            // Return the updated value of the IsNetworkAvailable property.

            return WorkstationState.IsNetworkAvailable;

        }

        internal async Task<bool> GetNetworkAvailabilityFromSingleton()

        {

            // Wait for 100 milliseconds (async/await pattern).

            await Task.Delay(100);

            // Get a reference to the WorkstationState singleton object.

            var state = WorkstationStateSingleton.Instance;

            // Use the lock statement to synchronize access to the singleton object.

            lock (\_workstationLock)

            {

                // Update the IsNetworkAvailable property of the singleton object.

                state.IsNetworkAvailable = NetworkInterface.GetIsNetworkAvailable();

                // Update the NetworkConnectivityLastUpdated property of the singleton object.

                state.NetworkConnectivityLastUpdated = DateTime.UtcNow;

            }

            // Return the updated value of the IsNetworkAvailable property of the singleton object.

            return state.IsNetworkAvailable;

        }

    }

}

**WorkstationState.cs**

// Importing necessary libraries for networking

using System.Net;

using System.Net.NetworkInformation;

using System.Net.Sockets;

namespace ThreadingStaticDataExample

{

    // This class represents the state of a workstation.

    internal class WorkstationState

    {

        // Properties representing the name and IP address of the workstation.

        internal static string Name { get; set; }

        internal static string IpAddress { get; set; }

        // Property representing whether the network is available.

        internal static bool IsNetworkAvailable { get; set; }

        // ThreadStatic attribute ensures that each thread has its own instance of this variable.

        // Property representing the last time network connectivity was updated.

        [ThreadStatic]

        internal static DateTime? NetworkConnectivityLastUpdated;

        // Static constructor that initializes the properties.

        static WorkstationState()

        {

            // Get the name of the workstation.

            Name = Dns.GetHostName();

            // Get the IP address of the workstation.

            IpAddress = GetLocalIPAddress(Name);

            // Check if the network is available.

            IsNetworkAvailable = NetworkInterface.GetIsNetworkAvailable();

            // Set the last updated time to the current UTC time.

            NetworkConnectivityLastUpdated = DateTime.UtcNow;

            // Pause the thread for 2 seconds.

            Thread.Sleep(2000);

        }

        // Helper function to get the local IP address of the workstation.

        private static string GetLocalIPAddress(string hostName)

        {

            // Get the host entry for the given host name.

            var hostEntry = Dns.GetHostEntry(hostName);

            // Loop through the addresses to find the IPv4 address.

            foreach (var address in hostEntry.AddressList

                                    .Where(a => a.AddressFamily == AddressFamily.InterNetwork))

            {

                // Return the first IPv4 address found.

                return address.ToString();

            }

            // If no IPv4 address is found, return an empty string.

            return string.Empty;

        }

    }

}

**WorkstationStateSingleton.cs**

// Importing necessary libraries for networking

using System.Net;

using System.Net.NetworkInformation;

using System.Net.Sockets;

namespace ThreadingStaticDataExample

{

    // This class represents the singleton instance of the state of a workstation.

    public class WorkstationStateSingleton

    {

        // Singleton instance variable and lock object to ensure thread safety.

        private static WorkstationStateSingleton? \_singleton = null;

        private static readonly object \_lock = new();

        // Private constructor to prevent direct instantiation of the class.

        private WorkstationStateSingleton()

        {

            // Get the name of the workstation.

            Name = Dns.GetHostName();

            // Get the IP address of the workstation.

            IpAddress = GetLocalIPAddress(Name);

            // Check if the network is available.

            IsNetworkAvailable = NetworkInterface.GetIsNetworkAvailable();

            // Set the last updated time to the current UTC time.

            NetworkConnectivityLastUpdated = DateTime.UtcNow;

        }

        // Public property that returns the singleton instance of the class.

        public static WorkstationStateSingleton Instance

        {

            get

            {

                // Ensure thread safety using the lock object.

                lock (\_lock)

                {

                    // If the singleton instance does not exist, create it.

                    if (\_singleton == null)

                    {

                        \_singleton = new WorkstationStateSingleton();

                    }

                    // Return the singleton instance.

                    return \_singleton;

                }

            }

        }

        // Properties representing the name and IP address of the workstation.

        public string Name { get; set; }

        public string IpAddress { get; set; }

        // Property representing whether the network is available.

        public bool IsNetworkAvailable { get; set; }

        // Property representing the last time network connectivity was updated.

        public DateTime? NetworkConnectivityLastUpdated { get; set; }

        // Helper function to get the local IP address of the workstation.

        private string GetLocalIPAddress(string hostName)

        {

            // Get the host entry for the given host name.

            var hostEntry = Dns.GetHostEntry(hostName);

            // Loop through the addresses to find the IPv4 address.

            foreach (var address in hostEntry.AddressList

                                    .Where(a => a.AddressFamily == AddressFamily.InterNetwork))

            {

                // Return the first IPv4 address found.

                return address.ToString();

            }

            // If no IPv4 address is found, return an empty string.

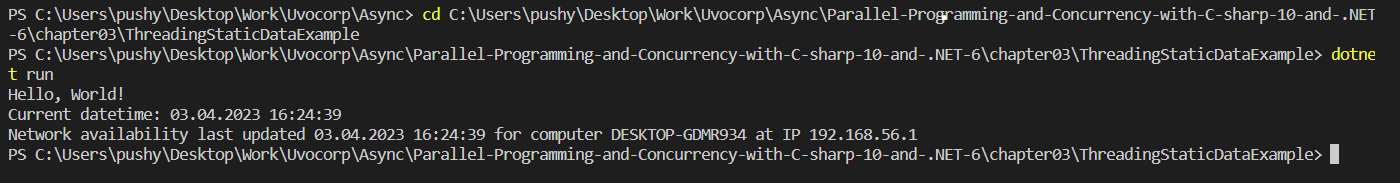
            return string.Empty;

        }

    }

}

**Output**

****

This software serves as an illustration of how immutable data can be used in a multithreaded application. The sample code develops a WorkstationHelper class that provides methods for checking network connectivity.

The identity, IP location, and network availability state of a workstation are stored in the WorkstationState class. To ensure that each thread has a unique instance of the NetworkConnectivityLastUpdated value, the class uses the ThreadStatic attribute.

The single version of the WorkstationState class is represented by the WorkstationStateSingleton class. It has a public property that returns the singleton instance of the class, and a secret function Object() { [native code] } to prevent direct instantiation of the class.

To prevent race conditions when updating static data, the sample code shows how to use the lock command to coordinate access to the WorkstationState object and the WorkstationStateSingleton instance.

To call the GetNetworkAvailability and GetNetworkAvailabilityFromSingleton methods asynchronously, the application uses the async/await paradigm. The sample also includes code that uses the parallel paradigm, but it is commented out.Use numerous threads to concurrently invoke the GetNetworkAvailability method.

**DeadlockAndRaceSample.cs**

public class DeadlockSample

{

    // Declare a private object for locking and a list to store some data

    private object \_lock = new object();

    private List<string> \_data;

    // Constructor to initialize the list with some values

    public DeadlockSample()

    {

        \_data = new List<string> { "First", "Second", "Third" };

    }

    // Method to process the data with a lock

    public async Task ProcessData()

    {

        // Acquire a lock on the private object

        lock (\_lock)

        {

            // Iterate over the data and write it to the console

            foreach (var item in \_data)

            {

                Console.WriteLine(item);

            }

            // Call the AddData method asynchronously and await its completion

            await AddData();

        }

    }

    // Method to add more data to the list with a lock

    private async Task AddData()

    {

        // Acquire a lock on the private object

        lock (\_lock)

        {

            // Add more data to the list

            \_data.AddRange(GetMoreData());

            // Wait for 100ms asynchronously

            await Task.Delay(100);

        }

    }

    // Method to process the data with a Monitor

    public void ProcessDataWithMonitor()

    {

        // Acquire a lock on the private object

        lock (\_lock)

        {

            // Iterate over the data and write it to the console

            foreach (var item in \_data)

            {

                Console.WriteLine(item);

            }

            // Call the AddDataWithMonitor method

            AddDataWithMonitor();

        }

    }

    // Method to add more data to the list with a Monitor

    private void AddDataWithMonitor()

    {

        // Try to acquire a lock on the private object for 1 second

        if (Monitor.TryEnter(\_lock, 1000))

        {

            try

            {

                // Add more data to the list

                \_data.AddRange(GetMoreData());

            }

            finally

            {

                // Release the lock on the private object

                Monitor.Exit(\_lock);

            }

        }

        else

        {

            // Write a message to the console if the lock cannot be acquired

            Console.WriteLine($"AddData: Unable to acquire lock. Stack trace: {Environment.StackTrace}");

        }

    }

    // Method to perform calculations with a race condition

    private int \_runningTotal;

    public void PerformCalculationsRace()

    {

        // Initialize the running total to 3

        \_runningTotal = 3;

        // Invoke two methods asynchronously and wait for their completion

        Parallel.Invoke(() =>

        {

            AddValue().Wait();

        }, () =>

        {

            MultiplyValue().Wait();

        });

        // Write the running total to the console

        Console.WriteLine($"Running total is {\_runningTotal}");

    }

    // Method to add and multiply values asynchronously without a race condition

    public async Task PerformCalculations()

    {

        // Initialize the running total to 3

        \_runningTotal = 3;

        // Multiply the value asynchronously and then add to it

        await MultiplyValue().ContinueWith(async (Task) =>

        {

            await AddValue();

        });

        // Write the running total to the console

        Console.WriteLine($"Running total is {\_runningTotal}");

    }

    // Method to get some more data

    private List<string> GetMoreData()

    {

        return new List<string> { "Fourth", "Fifth", "Sixth" };

    }

    private int \_runningTotal;

    // Executes AddValue() and MultiplyValue() concurrently and waits for both to complete

    public void PerformCalculationsRace()

    {

        // Initialize \_runningTotal to 3

        \_runningTotal = 3;

        Parallel.Invoke(() =>

        {

            // Wait for AddValue() to complete and add 15 to \_runningTotal

            AddValue().Wait();

        }, () =>

        {

            // Wait for MultiplyValue() to complete and multiply \_runningTotal by 10

            MultiplyValue().Wait();

        });

        // Print the final running total

        Console.WriteLine($"Running total is {\_runningTotal}");

    }

    // Add 15 to \_runningTotal after waiting for 100ms

    private async Task AddValue()

    {

        await Task.Delay(100);

        \_runningTotal += 15;

    }

    // Multiply \_runningTotal by 10 after waiting for 100ms

    private async Task MultiplyValue()

    {

        await Task.Delay(100);

        \_runningTotal = \_runningTotal \* 10;

    }

    // Waits for MultiplyValue() to complete, and then adds 15 to \_runningTotal

    public async Task PerformCalculations()

    {

        // Initialize \_runningTotal to 3

        \_runningTotal = 3;

        await MultiplyValue().ContinueWith(async (Task) =>

        {

            await AddValue();

        });

        // Print the final running total

        Console.WriteLine($"Running total is {\_runningTotal}");

    }

}

**Interlocked.cs**

public class InterlockedSample

{

    private long \_runningTotal;

    public void PerformCalculations()

    {

        // Set the initial value of \_runningTotal to 3

        \_runningTotal = 3;

        // Executes AddValue() and MultiplyValue() concurrently and waits for both to complete

        Parallel.Invoke(() =>

        {

            AddValue().Wait();

        }, () =>

        {

            MultiplyValue().Wait();

        });

        // Outputs the final value of \_runningTotal

        Console.WriteLine($"Running total is {\_runningTotal}");

    }

    private async Task AddValue()

    {

        // Wait for 100 milliseconds

        await Task.Delay(100);

        // Adds 15 to the \_runningTotal using Interlocked.Add method to ensure atomicity

        Interlocked.Add(ref \_runningTotal, 15);

    }

    private async Task MultiplyValue()

    {

        // Wait for 100 milliseconds

        await Task.Delay(100);

        // Read the current value of \_runningTotal using Interlocked.Read method

        var currentTotal = Interlocked.Read(ref \_runningTotal);

        // Multiply the current value of \_runningTotal by 10 using Interlocked.Exchange method to ensure atomicity

        Interlocked.Exchange(ref \_runningTotal, currentTotal \* 10);

    }

}

**ThreadingLimits.cs**

// This class demonstrates techniques for limiting the number of threads used in parallel processing

public class ThreadingLimitsSample

{

    // Uses Parallel.ForEach to process a list of items, limiting the degree of parallelism to half the processor count

    public void ProcessParallelForEachWithLimits(List<string> items)

    {

        int max = Environment.ProcessorCount > 1 ?

        Environment.ProcessorCount / 2 : 1;

        var options = new ParallelOptions

        {

            MaxDegreeOfParallelism = max

        };

        Parallel.ForEach(items, options, y =>

        {

            // Process items

        });

    }

    // Uses PLINQ to process a list of items, limiting the degree of parallelism to half the processor count,

    // and returns true if any items meet a specific criteria (in this case, if they are not null or whitespace)

    public bool ProcessPlinqWithLimits(List<string> items)

    {

        int max = Environment.ProcessorCount > 1 ? Environment.ProcessorCount / 2 : 1;

        return items.AsParallel()

            .WithDegreeOfParallelism(max)

            .Any(i => CheckString(i));

    }

    // Updates the maximum number of threads available in the thread pool to twice the processor count, or twice the minimum number of threads, whichever is greater

    private void UpdateThreadPoolMax()

    {

        ThreadPool.GetMinThreads(out int workerMin, out int completionMin);

        int workerMax = GetProcessingMax(workerMin);

        int completionMax = GetProcessingMax(completionMin);

        ThreadPool.SetMaxThreads(workerMax, completionMax);

    }

    // Calculates the maximum number of threads to use based on the minimum number of threads available

    private int GetProcessingMax(int min)

    {

        return min < Environment.ProcessorCount ? Environment.ProcessorCount \* 2 : min \* 2;

    }

}

Also in chapter03 there are files without an executable file.

The code in the files Interlocked.cs and DeadlockAndRaceSample.cs shows how to handle concurrency problems in C#, including deadlocks, race conditions, and memory integrity errors.

The DeadlockSample class provides an example of how to use locks and monitors to avoid deadlocks and race conditions in the DeadlockAndRaceSample.cs file. Before accessing the shared \_data field, the ProcessData and AddData methods lock a private object using the Lock keyword, while the ProcessDataWithMonitor and AddDataWithMonitor methods lock and unlock the same object using a Monitor object. By setting a secret \_runningTotal field to 3 and then using Parallel to call the AddValue and MultiplyValue methods asynchronously, the PerformCalculationsRace function illustrates a race situation.Invoke. By using the async and await keywords to ensure that the MultiplyValue method completes before the AddValue method changes the \_runningTotal field, the PerformCalculations method demonstrates how to resolve the race situation.

The InterlockedSample class shows how to use the Interlocked class to ensure memory consistency when multiple processes are reading a shared variable in the Interlocked.cs file. The AddValue and MultiplyValue methods are called asynchronously using Parallel by the PerformCalculations function after the private \_runningTotal field is set to a starting value of 3.Invoke. The \_runningTotal field is modified atomically by the AddValue and MultiplyValue methods using the Interlocked.Add and Interlocked.Multiply methods, respectively, to ensure memory consistency even if multiple processes modify the field simultaneously. The \_runningTotal final number is printed to the console.

**chapter05**

**AsyncSamples**

**AsyncSample.cs**

using AsyncSamples;

using System.Text;

using System.Text.Json;

using System.Xml.Serialization;

public class AsyncSample

{

// This method reads a file asynchronously and returns a list of strings

// Each string represents a line of text in the file

public async Task<List<string>> GetDataAsync(string filePath)

{

using var file = File.OpenText(filePath);

var data = await file.ReadToEndAsync();

return data.Split(new[] { Environment.NewLine },

StringSplitOptions.RemoveEmptyEntries).ToList();

}

// This method gets data from an online resource asynchronously and returns a list of strings

// Each string represents a line of text from the resource

public async Task<List<string>> GetOnlineDataAsync(string url)

{

    var httpClient = new HttpClient();

    var data = await httpClient.GetStringAsync(url);

    return data.Split(new[] { Environment.NewLine },

        StringSplitOptions.RemoveEmptyEntries).ToList();

}

// This method deserializes a list of XML strings into a list of JournalEntry objects asynchronously

public async Task<List<JournalEntry>> DeserializeJournalDataAsync(List<string> journalData)

{

    return await Task.Run(() => DeserializeEntries(journalData));

}

// This is a helper method for DeserializeJournalDataAsync that deserializes a single XML string

private List<JournalEntry> DeserializeEntries(List<string> journalData)

{

    var deserializedEntries = new List<JournalEntry>();

    var serializer = new XmlSerializer(typeof(JournalEntry));

    foreach (var xmlEntry in journalData)

    {

        if (xmlEntry == null) continue;

        using var reader = new StringReader(xmlEntry);

        var entry = (JournalEntry)serializer.Deserialize(reader)!;

        if (entry == null) continue;

        deserializedEntries.Add(entry);

    }

    return deserializedEntries;

}

// This method deserializes a list of JSON strings into a list of JournalEntry objects

public List<JournalEntry> DeserialzeJsonEntries(List<string> journalData)

{

    var deserializedEntries = new List<JournalEntry>();

    foreach (var jsonEntry in journalData)

    {

        if (string.IsNullOrWhiteSpace(jsonEntry)) continue;

        deserializedEntries.Add(JsonSerializer.Deserialize<JournalEntry>(jsonEntry)!);

    }

    return deserializedEntries;

}

// This method deserializes a list of JSON strings into a list of JournalEntry objects asynchronously

public async Task<List<JournalEntry>> DeserializeJsonEntriesAsync(List<string> journalData)

{

    var deserializedEntries = new List<JournalEntry>();

    foreach (var jsonEntry in journalData)

    {

        if (string.IsNullOrWhiteSpace(jsonEntry)) continue;

        using var stream = new MemoryStream(Encoding.Unicode.GetBytes(jsonEntry));

        deserializedEntries.Add((await JsonSerializer.DeserializeAsync<JournalEntry>(stream))!);

    }

    return deserializedEntries;

}

// This method deserializes a list of JSON strings into a list of JournalEntry objects asynchronously using parallelism

public async Task<List<JournalEntry>> GetJournalEntriesAsync(List<string> journalData)

{

    var journalTasks = journalData.Select(entry => DeserializeJsonEntryAsync(entry));

    return (await Task.WhenAll(journalTasks)).ToList();

}

// This is a helper method for GetJournalEntriesAsync that deserializes a single JSON string

private async Task<JournalEntry> DeserializeJsonEntryAsync(string jsonEntry)

{

    if (string.IsNullOrWhiteSpace(jsonEntry)) return new JournalEntry();

    using var stream = new MemoryStream(Encoding.Unicode.GetBytes(jsonEntry));

    return (await JsonSerializer.DeserializeAsync<JournalEntry>(stream))!;

}

// Note: It's important to include error handling and dispose of resources properly in a production environment.

}

**JournalEntry.cs**

namespace AsyncSamples

{

    // This class represents a journal entry

    [Serializable]

    public class JournalEntry

    {

        // The title of the entry

        public string Title { get; set; }

        // A short description of the entry

        public string Description { get; set; }

        // The date the entry was made

        public DateTime EntryDate { get; set; }

        // The content of the entry

        public string EntryText { get; set; }

    }

}

**Program.cs**

using AsyncSamples;

// Print a message indicating that the processing is starting

Console.WriteLine("Start processing...");

// Create an instance of TaskSample and invoke its DoThingsAsync method asynchronously

var taskSample = new TaskSample();

await taskSample.DoThingsAsync();

// Print a message indicating that the processing is continuing

Console.WriteLine("Continue processing...");

// Invoke the DoingThingsWrongAsync method, which performs a blocking operation on the main thread

await taskSample.DoingThingsWrongAsync();

// Print a message indicating that the processing is continuing

Console.WriteLine("Continue processing...");

// Invoke the DoBlockingThingsAsync method, which performs a blocking operation on a background thread

await taskSample.DoBlockingThingsAsync();

// Print a message indicating that the processing is done

Console.WriteLine("Done processing...");

**TaskSample.cs**

namespace AsyncSamples

{

    public class TaskSample

    {

        // This method uses the "async/await" pattern correctly

        public async Task DoThingsAsync()

        {

            Console.WriteLine($"Doing things in {nameof(DoThingsAsync)}");

            // This call to an asynchronous method is awaited correctly

            await DoFirstThingAsync();

            // This call to an asynchronous method is awaited correctly

            await DoSecondThingAsync();

            Console.WriteLine($"Did things in {nameof(DoThingsAsync)}");

        }

        // This method calls an asynchronous method incorrectly

        public async Task DoingThingsWrongAsync()

        {

            Console.WriteLine($"Doing things in {nameof(DoingThingsWrongAsync)}");

            // This call to an asynchronous method is not awaited or assigned to a variable

            DoFirstThingAsync();

            // This call to an asynchronous method is awaited correctly

            await DoSecondThingAsync();

            Console.WriteLine($"Did things in {nameof(DoingThingsWrongAsync)}");

        }

        // This method blocks on an asynchronous method, which is not recommended

        public async Task DoBlockingThingsAsync()

        {

            Console.WriteLine($"Doing things in {nameof(DoBlockingThingsAsync)}");

            // This call to an asynchronous method blocks the current thread, which can lead to deadlocks

            DoFirstThingAsync().Wait();

            // This call to an asynchronous method is awaited correctly

            await DoSecondThingAsync();

            Console.WriteLine($"Did things in {nameof(DoBlockingThingsAsync)}");

        }

        // These methods are called correctly with the "await" keyword

        private async Task DoFirstThingAsync()

        {

            Console.WriteLine($"Doing something in {nameof(DoFirstThingAsync)}");

            // This call to an asynchronous method is awaited correctly

            await DoAnotherThingAsync();

            Console.WriteLine($"Did something in {nameof(DoFirstThingAsync)}");

        }

        private async Task DoSecondThingAsync()

        {

            Console.WriteLine($"Doing something in {nameof(DoSecondThingAsync)}");

            // This call to an asynchronous method is awaited correctly

            await Task.Delay(500);

            Console.WriteLine($"Did something in {nameof(DoSecondThingAsync)}");

        }

        private async Task DoAnotherThingAsync()

        {

            Console.WriteLine($"Doing something in {nameof(DoAnotherThingAsync)}");

            // This call to an asynchronous method is awaited correctly

            await Task.Delay(1500);

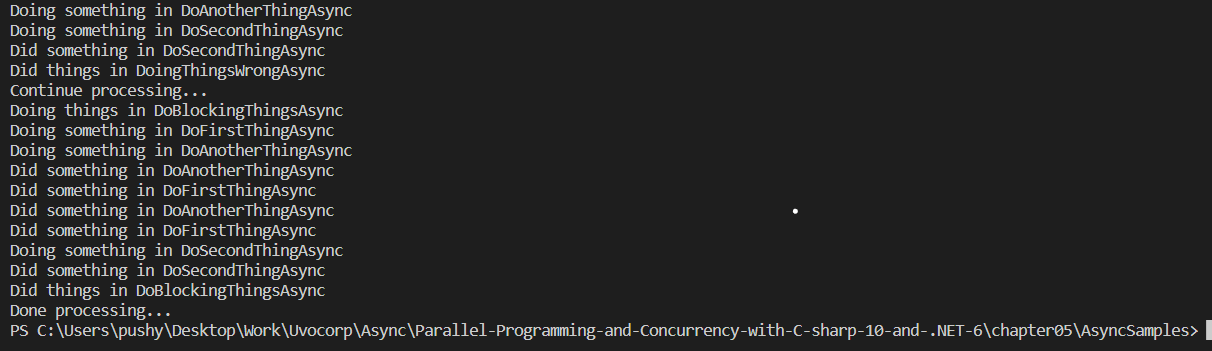
            Console.WriteLine($"Did something in {nameof(DoAnotherThingAsync)}");

        }

    }

}

**Output**



A set of classes and functions for C# asynchronous computing are present in this code.

Several methods in AsyncSample show how to retrieve data from files, retrieve data from web sources, and asynchronously deserialize XML and JSON text into a collection of JournalEntry objects.

A JournalEntry is represented by the simple JournalEntry class, which contains the content, title, summary, and date of the entry.

Another class that demonstrates how to effectively use the async/await technique to perform asynchronous tasks is TaskSample.

The primary program entry point, where an instance of TaskSample is created and its methods are called asynchronously, is in Program.cs. A string of notifications tracking the status of the asynchronous processes is the output of the application.

**SyncAndAsyncSamples**

**AsyncToSync**

**PatientLoader.cs**

using SyncAndAsyncSamples.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SyncAndAsyncSamples.AsyncToSync

{

    public class PatientLoader

    {

        private PatientService \_patientService;

        // Constructor initializes a new PatientService instance.

        public PatientLoader()

        {

            \_patientService = new PatientService();

        }

        // Method that loads patient information asynchronously and returns a Patient object.

        public async Task<Patient?> GetPatientAndMedsAsync(int patientId)

        {

            Patient? patient = null;

            try

            {

                // Load patient information asynchronously using Task.Run and GetPatientInfo method from the PatientService.

                patient = await Task.Run(() => \_patientService.GetPatientInfo(patientId));

            }

            catch (Exception e)

            {

                // Catch any exceptions thrown during loading and print an error message to the console.

                Console.WriteLine($"Error loading patient. Message: {e.Message}");

            }

            if (patient != null)

            {

                // Process patient information asynchronously using the ProcessPatientInfoAsync method.

                patient = await ProcessPatientInfoAsync(patient);

                return patient;

            }

            else

            {

                // If the patient object is null, return null.

                return null;

            }

        }

        // Method that processes patient information asynchronously and returns a Patient object.

        private async Task<Patient> ProcessPatientInfoAsync(Patient patient)

        {

            // Introduce an artificial delay of 100 milliseconds to simulate additional processing.

            await Task.Delay(100);

            // Add additional processing logic here.

            return patient;

        }

    }

}

**PatientService.cs**

using SyncAndAsyncSamples.Models;

namespace SyncAndAsyncSamples.AsyncToSync

{

    public class PatientService

    {

        // Method that retrieves patient information synchronously and returns a Patient object.

        public Patient GetPatientInfo(int patientId)

        {

            // Simulate a delay of 2 seconds to mimic a long-running operation.

            Thread.Sleep(2000);

            // Create a new Patient object with some sample data.

            Patient patient = new()

            {

                Id = patientId,

                Name = "Smith, Terry",

                PrimaryCareProvider = new Provider

                {

                    Id = 999,

                    Name = "Dr. Amy Ng"

                },

                Medications = new List<Medication>

                {

                    new Medication { Id = 1, Name = "acetaminophen" },

                    new Medication { Id = 2, Name = "hydrocortisone cream" }

                }

            };

            // Return the patient object.

            return patient;

        }

    }

}

**Models**

**Medication.cs**

namespace SyncAndAsyncSamples.Models

{

    // A class representing a medication.

    public class Medication

    {

        // The ID of the medication.

        public int Id { get; set; }

        // The name of the medication (nullable string).

        public string? Name { get; set; }

    }

}

**Patient.cs**

namespace SyncAndAsyncSamples.Models

{

    // A class representing a patient.

    public class Patient

    {

        // The ID of the patient.

        public int Id { get; set; }

        // The name of the patient (nullable string).

        public string? Name { get; set; }

        // A list of medications that the patient is currently taking (nullable list of Medication objects).

        public List<Medication>? Medications { get; set; }

        // The primary care provider for the patient (nullable Provider object).

        public Provider? PrimaryCareProvider { get; set; }

    }

}

**Provider.cs**

namespace SyncAndAsyncSamples.Models

{

    // A class representing a healthcare provider.

    public class Provider

    {

        // The ID of the healthcare provider.

        public int Id { get; set; }

        // The name of the healthcare provider (nullable string).

        public string? Name { get; set; }

    }

}

**SyncToAsync**

**HealthcareService.cs**

using SyncAndAsyncSamples.Models;

namespace SyncAndAsyncSamples.SyncToAsync

{

    public class HealthcareService

    {

        // This method asynchronously loads patient information by simulating a delay of 2 seconds using Task.Delay.

        // It then creates a new Patient object with some sample data, including an ID, name, primary care provider, and a list of medications.

        // The method returns the patient object wrapped in a Task object, which allows it to be awaited asynchronously.

        public async Task<Patient> GetPatientInfoAsync(int patientId)

        {

            await Task.Delay(2000);

            Patient patient = new()

            {

                Id = patientId,

                Name = "Smith, Terry",

                PrimaryCareProvider = new Provider

                {

                    Id = 999,

                    Name = "Dr. Amy Ng"

                },

                Medications = new List<Medication>

                {

                    new Medication { Id = 1, Name = "acetaminophen" },

                    new Medication { Id = 2, Name = "hydrocortisone cream" }

                }

            };

            return patient;

        }

    }

}

**MedicationLoader.cs**

// Import the required namespace

using SyncAndAsyncSamples.Models;

namespace SyncAndAsyncSamples.SyncToAsync

{

    // Define a class called MedicationLoader

    public class MedicationLoader

    {

        // Declare a private field of type HealthcareService

        private HealthcareService \_healthcareService;

        // Define a constructor for the MedicationLoader class

        public MedicationLoader()

        {

            // Instantiate a new HealthcareService object and assign it to the \_healthcareService field

            \_healthcareService = new HealthcareService();

        }

        // Define a public method called GetPatientAndMedications that returns a nullable Patient object

        public Patient? GetPatientAndMedications(int patientId)

        {

            // Declare a null Patient object

            Patient? patient = null;

            try

            {

                // Call the asynchronous GetPatientInfoAsync method of the \_healthcareService field and get its result

                patient = \_healthcareService.GetPatientInfoAsync(patientId).Result;

            }

            catch (AggregateException ae)

            {

                // If an exception occurs, write an error message to the console

                Console.WriteLine($"Error loading patient. Message: {ae.Flatten().Message}");

            }

            // If the patient object is not null, call the ProcessPatientInfo method and return the resulting patient object

            if (patient != null)

            {

                patient = ProcessPatientInfo(patient);

                return patient;

            }

            // Otherwise, return null

            else

            {

                return null;

            }

        }

        // Define a private method called ProcessPatientInfo that takes a Patient object as a parameter and returns a Patient object

        private Patient ProcessPatientInfo(Patient patient)

        {

            // Add additional processing to the patient object here, if needed

            return patient;

        }

    }

}

**ParallelPatientLoader.cs**

using SyncAndAsyncSamples.Models;

namespace SyncAndAsyncSamples

{

    public class ParallelPatientLoader

    {

        // Fields to store patient, provider, and medications data.

        private Patient \_patient;

        private Provider \_provider;

        private List<Medication> \_medications;

        // This method loads patient information asynchronously by calling three other asynchronous methods that load patient info, provider info, and medications.

        // The three tasks are added to a list of tasks and awaited using Task.WhenAll.

        // Once all three tasks are completed, the patient object is updated with the loaded information and returned.

        public async Task<Patient> LoadPatientAsync(int patientId)

        {

            var taskList = new List<Task>

            {

                LoadPatientInfoAsync(patientId),

                LoadProviderAsync(patientId),

                LoadMedicationsAsync(patientId)

            };

            await Task.WhenAll(taskList.ToArray());

            \_patient.Medications = \_medications;

            \_patient.PrimaryCareProvider = \_provider;

            return \_patient;

        }

        // This method loads patient information synchronously by calling three other asynchronous methods that load patient info, provider info, and medications.

        // The three tasks are added to a list of tasks and waited using Task.WaitAll.

        // Once all three tasks are completed, the patient object is updated with the loaded information and returned.

        public Patient LoadPatient(int patientId)

        {

            var taskList = new List<Task>

            {

                LoadPatientInfoAsync(patientId),

                LoadProviderAsync(patientId),

                LoadMedicationsAsync(patientId)

            };

            Task.WaitAll(taskList.ToArray());

            \_patient.Medications = \_medications;

            \_patient.PrimaryCareProvider = \_provider;

            return \_patient;

        }

        // This method loads patient information asynchronously by simulating a delay using Task.Delay and then creating a new Patient object with some sample data.

        // The patient object is stored in the \_patient field.

        public async Task LoadPatientInfoAsync(int patientId)

        {

            await Task.Delay(100);

            \_patient = new Patient { Id = patientId, Name = "Smith, Gail" };

        }

        // This method loads provider information asynchronously by simulating a delay using Task.Delay and then creating a new Provider object with some sample data.

        // The provider object is stored in the \_provider field.

        public async Task LoadProviderAsync(int patientId)

        {

            await Task.Delay(100);

            \_provider = new Provider { Id = 44, Name = "Dr. Sammy Hamm" };

        }

        // This method loads medication information asynchronously by simulating a delay using Task.Delay and then creating a new List of Medication objects with some sample data.

        // The medication list is stored in the \_medications field.

        public async Task LoadMedicationsAsync(int patientId)

        {

            await Task.Delay(100);

            \_medications = new List<Medication>

                {

                    new Medication { Id = 1, Name = "acetaminophen" },

                    new Medication { Id = 2, Name = "hydrocortisone cream" }

                };

        }

    }

}

**Program.cs**

using SyncAndAsyncSamples.AsyncToSync;

using SyncAndAsyncSamples.Models;

//using SyncAndAsyncSamples.SyncToAsync;

//Console.WriteLine("Hello, sync to async world!");

//var medLoader = new MedicationLoader();

//Patient? patient = medLoader.GetPatientAndMedications(123);

//Console.WriteLine($"Loaded patient: {patient.Name} with {patient.Medications.Count} mediations.");

// Output a greeting message for the async to sync world.

Console.WriteLine("Hello, async to sync world!");

// Create a new instance of the PatientLoader class.

var loader = new PatientLoader();

// Call the GetPatientAndMedsAsync method asynchronously and await its result.

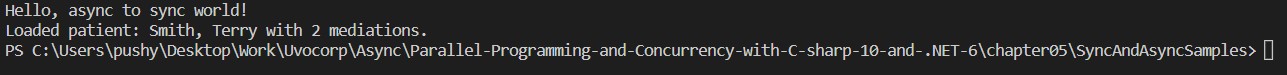
// The patient object and its medications are loaded asynchronously and then converted to synchronous code using the await keyword.

Patient? patient = await loader.GetPatientAndMedsAsync(123);

// Output a message indicating that the patient data has been loaded successfully.

Console.WriteLine($"Loaded patient: {patient.Name} with {patient.Medications.Count} medications.");

**Output**

****

Two different patient information entry methods, one using synchronous code and the other using asynchronous code, are included in the code.

The PatientLoader and PatientService classes, which import patient data asynchronously and then process it synchronously, are implemented in the AsyncToSync namespace. Patient information is loaded asynchronously using the GetPatientInfo method of PatientService, which is encapsulated in a Task by the GetPatientAndMedsAsync method of PatientLoader.Run. Once the patient data is properly loaded, it is processed asynchronously using the ProcessPatientInfoAsync function and an object named Patient is returned. To mimic additional processing, the ProcessPatientInfoAsync method adds an artificial delay of 100 milliseconds.

The HealthcareService and MedicationLoader classes that asynchronously import patient data are implemented in the SyncToAsync namespace. The GetPatientInfoAsync function of the HealthcareService asynchronously retrieves patient data via Task.Wait before creating a Patient entity with some test data. The GetPatientInfoAsync function of the HealthcareService is called by the MedicationLoader class, which then accesses the Medication property of the returned Patient object to demonstrate how to asynchronously import a patient's medications.

**TaskSamples**

**Examples.cs**

namespace TaskSamples

{

    public class Examples

    {

        // This method processes orders asynchronously using multiple tasks.

        // It first creates a task to prepare the orders by calling the PrepareOrders method.

        // Then it creates a long-running task to create labels for the orders by calling the CreateLabels method.

        // Finally, it creates a continuation task that sends the prepared orders by calling the SendOrders method.

        // The method waits for both the label and send tasks to complete by calling the WaitAll method of the Task class.

        // After both tasks complete, it sends a confirmation message to the customer by calling the SendConfirmation method.

        public void ProcessOrders(List<Order> orders, int customerId)

        {

            Task<List<Order>> processOrdersTask = Task.Run(() => PrepareOrders(orders));

            Task labelTask = Task.Factory.StartNew(() => CreateLabels(orders), TaskCreationOptions.LongRunning);

            Task sendTask = processOrdersTask.ContinueWith(task => SendOrders(task.Result));

            Task.WaitAll(new[] { labelTask, sendTask });

            SendConfirmation(customerId);

        }

        // This method processes data asynchronously using a task.

        // It first creates a task to process the data by calling the DoDataProcessing method.

        // If the data processing requires a UI thread, it runs the task synchronously on the current thread.

        // If the data processing does not require a UI thread, it starts the task on a ThreadPool thread in the background.

        public void ProcessData(object data, bool uiRequired)

        {

            Task processTask = new(() => DoDataProcessing(data));

            if (uiRequired)

            {

                // Run on current thread (UI thread assumed for example)

                processTask.RunSynchronously();

            }

            else

            {

                // Run on ThreadPool thread in background

                processTask.Start();

            }

        }

        // This method processes the data.

        // Add the data processing logic here.

        private void DoDataProcessing(object data)

        {

            // TODO: Process the data

        }

        // This method prepares the orders.

        // Add the order preparation logic here.

        private List<Order> PrepareOrders(List<Order> orders)

        {

            // TODO: Prepare orders here

            return orders;

        }

        // This method creates labels for the orders.

        // Add the label creation logic here.

        private void CreateLabels(List<Order> orders)

        {

            // TODO: Create labels here

        }

        // This method sends the prepared orders.

        // Add the order sending logic here.

        private void SendOrders(List<Order> orders)

        {

            // TODO: Send orders here

        }

        // This method sends a confirmation message to the customer.

        // Add the confirmation sending logic here.

        private void SendConfirmation(int customerId)

        {

            // TODO: Send confirmation message to customer

        }

    }

}

**Order.cs**

namespace TaskSamples

{

    // This class represents an order with an Id and a Name.

    // The Id property is an integer that uniquely identifies the order.

    // The Name property is an optional string that provides a name for the order.

    public class Order

    {

        // Gets or sets the unique identifier of the order.

        public int Id { get; set; }

        // Gets or sets the name of the order.

        // This property is optional and can be null.

        public string? Name { get; set; }

    }

}

**Program.cs**

using TaskSamples;

// Create an instance of the Examples class.

var examples = new Examples();

// Process the orders asynchronously by calling the ProcessOrders method of the examples object.

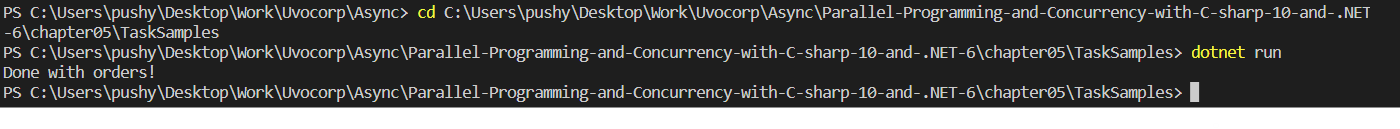
// Pass an empty list of orders and a customer ID to the method.

examples.ProcessOrders(new List<Order>(), 123);

// Write a message to the console to indicate that the order processing is complete.

Console.WriteLine("Done with orders!");

**Output**

****

This program is an example of asynchronous programming in C# that shows how to handle requests and data without stalling by using tasks. The program specifies a series of procedures for assembling, labeling, and shipping orders and for handling data. While the ProcessData method creates a single task to process data, the ProcessOrders method uses multiple tasks to perform these operations asynchronously. The software also creates an Order class that represents orders with names and IDs.

By examining this program, we can learn about asynchronous programming in C#, which allows programs to continue running while waiting for processes to complete. This can be helpful in improving the responsiveness and efficiency of programs, especially those that require I/O or long-running procedures. The software also shows how task continuations can be used to link different processes together in a pipeline. Overall, this program provides a helpful illustration of how to use tasks to perform asynchronous computation in C#.

**chapter06**

**ParallelTaskRelationshipsSample**

**ParallelWork.cs**

namespace ParallelTaskRelationshipsSample

{

    // This class provides examples of parallel work using Task.

    public class ParallelWork

    {

        // Performs three actions in parallel, each on a separate task.

        // The parent task waits for the child tasks to complete.

        public void DoAllWork()

        {

            Console.WriteLine("Starting DoAllWork");

            Task parentTask = Task.Factory.StartNew(() =>

            {

                var child1 = Task.Factory.StartNew(DoFirstItem);

                var child2 = Task.Factory.StartNew(DoSecondItem);

                var child3 = Task.Factory.StartNew(DoThirdItem);

            });

            parentTask.Wait();

            Console.WriteLine("Finishing DoAllWork");

        }

        // Performs three actions in parallel, each on a separate task.

        // The child tasks are attached to the parent task, so the parent task waits for them to complete.

        public void DoAllWorkAttached()

        {

            Console.WriteLine("Starting DoAllWorkAttached");

            Task parentTask = Task.Factory.StartNew(() =>

            {

                var child1 = Task.Factory.StartNew(DoFirstItem, TaskCreationOptions.AttachedToParent);

                var child2 = Task.Factory.StartNew(DoSecondItem, TaskCreationOptions.AttachedToParent);

                var child3 = Task.Factory.StartNew(DoThirdItem, TaskCreationOptions.AttachedToParent);

            });

            parentTask.Wait();

            Console.WriteLine("Finishing DoAllWorkAttached");

        }

        // Performs three actions in parallel, each on a separate task.

        // The child tasks are attached to the parent task, but the parent task is created with the option to deny child task attachment.

        public void DoAllWorkDenyAttach()

        {

            Console.WriteLine("Starting DoAllWorkDenyAttach");

            Task parentTask = Task.Factory.StartNew(() =>

            {

                var child1 = Task.Factory.StartNew(DoFirstItem, TaskCreationOptions.AttachedToParent);

                var child2 = Task.Factory.StartNew(DoSecondItem, TaskCreationOptions.AttachedToParent);

                var child3 = Task.Factory.StartNew(DoThirdItem, TaskCreationOptions.AttachedToParent);

            }, TaskCreationOptions.DenyChildAttach);

            parentTask.Wait();

            Console.WriteLine("Finishing DoAllWorkDenyAttach");

        }

        // Simulates work for the first task.

        public void DoFirstItem()

        {

            Console.WriteLine("Starting DoFirstItem");

            Thread.SpinWait(1000000);

            Console.WriteLine("Finishing DoFirstItem");

        }

        // Simulates work for the second task.

        public void DoSecondItem()

        {

            Console.WriteLine("Starting DoSecondItem");

            Thread.SpinWait(1000000);

            Console.WriteLine("Finishing DoSecondItem");

        }

        // Simulates work for the third task.

        public void DoThirdItem()

        {

            Console.WriteLine("Starting DoThirdItem");

            Thread.SpinWait(1000000);

            Console.WriteLine("Finishing DoThirdItem");

        }

    }

}

**Program.cs**

using ParallelTaskRelationshipsSample;

// Create an instance of the ParallelWork class.

var parallelWork = new ParallelWork();

// Call one of the methods to perform parallel work using Task.

// Uncomment one of the following lines to choose a method to call.

// parallelWork.DoAllWork();

// Performs three actions in parallel, each on a separate task.

// The parent task waits for the child tasks to complete.

// parallelWork.DoAllWorkAttached();

// Performs three actions in parallel, each on a separate task attached to the parent task.

// The parent task waits for the child tasks to complete.

parallelWork.DoAllWorkDenyAttach();

// Performs three actions in parallel, each on a separate task attached to the parent task,

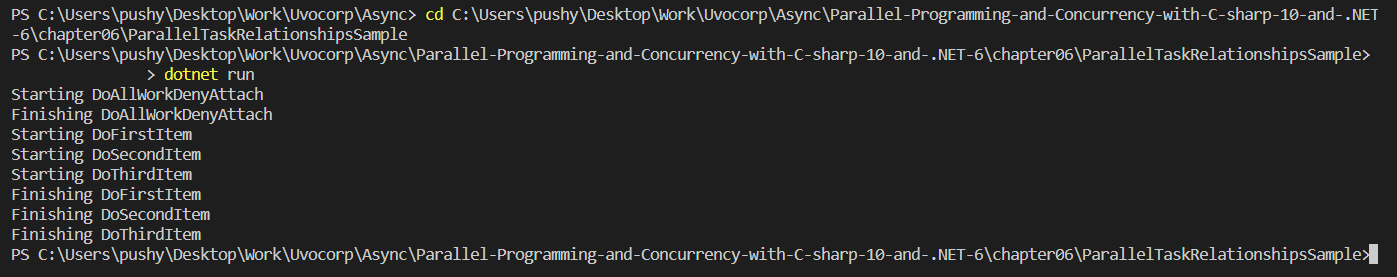
// but the parent task is created with the option to deny child task attachment.

// The parent task waits for the child tasks to complete.

// Wait for user input before exiting the program.

Console.ReadKey();

**output**

****

These applications demonstrate parallel processing using C#'s Task class. Three methods in the ParallelWork class perform three operations simultaneously on different tasks, mimicking the work done for each task. The first technique, DoAllWork, creates smaller tasks and uses the parent task to wait for them to finish. To ensure that the parent task waits for the child tasks to finish, the second technique, DoAllWorkAttached, also creates child tasks and attaches them to the parent task. DoAllWorkDenyAttach, the third technique, creates child tasks and attaches them to the parent task, but the parent task has the option to deny child task attachments when they are created.

An instance of the ParallelWork class is created in Program.cs, and one of the functions is executed based on the uncommented line. The result of each method shows the start and end times of the jobs, as well as their relationships to each other.

Taken together, these programs demonstrate how Task can be used in C# to accomplish concurrent work and how different task attachment choices can affect the interaction between parent and child tasks.

**WinFormsParallelLoopApp**

**FileData.cs**

namespace WinFormsParallelLoopApp

{

    // Represents data about a collection of FileInfo objects.

    public class FileData

    {

        // Gets or sets the list of FileInfo objects.

        public List<FileInfo> FileInfoList { get; set; } = new();

        // Gets or sets the total size of all FileInfo objects in bytes.

        public long TotalSize { get; set; } = 0;

        // Gets or sets the name of the last FileInfo object that was written to.

        public string LastWrittenFileName { get; set; } = "";

        // Gets or sets the date and time the last FileInfo object was written to.

        public DateTime LastFileWriteTime { get; set; }

    }

}

**FileProcessor.cs**

//This is the WinFormsParallelLoopApp namespace.

namespace WinFormsParallelLoopApp

{

    //This is the FileProcessor class.

    public class FileProcessor

    {

        //This is the GetInfoForFiles method that takes an array of file paths and returns file information.

        public static FileData GetInfoForFiles(string[] files)

        {

            //Create a new instance of FileData.

            var results = new FileData();

            //Create a new List object to store file information.

            var fileInfos = new List<FileInfo>();

            //Initialize the total file size to 0.

            long totalFileSize = 0;

            //Initialize lastWriteTime to the minimum DateTime value.

            DateTime lastWriteTime = DateTime.MinValue;

            //Initialize lastFileWritten to an empty string.

            string lastFileWritten = "";

            //Create an object to lock date operations.

            object dateLock = new();

            //Execute a parallel loop over the files array using Parallel.For.

            Parallel.For(0, files.Length,

                   index => {

                        //Get the file information for the current file.

                       FileInfo fi = new(files[index]);

                        //Get the size of the current file.

                       long size = fi.Length;

                        //Get the last write time of the current file.

                       DateTime lastWrite = fi.LastWriteTimeUtc;

                       //Lock date operations with dateLock.

                       lock (dateLock)

                       {

                           //If the last write time of the current file is greater than the current lastWriteTime value, update lastWriteTime and lastFileWritten.

                           if (lastWriteTime < lastWrite)

                           {

                               lastWriteTime = lastWrite;

                               lastFileWritten = fi.Name;

                           }

                       }

                       //Add the size of the current file to the total file size using Interlocked.Add.

                       Interlocked.Add(ref totalFileSize, size);

                       //Add the FileInfo object for the current file to fileInfos.

                       fileInfos.Add(fi);

                   });

            //Assign the results for total size, last file write time, last written file name, and file information List.

            results.FileInfoList = fileInfos;

            results.TotalSize = totalFileSize;

            results.LastFileWriteTime = lastWriteTime;

            results.LastWrittenFileName = lastFileWritten;

            //Return the results object.

            return results;

        }

        //This is the GetInfoForFilesThreadLocal method that takes an array of file paths and returns file information.

        public static FileData GetInfoForFilesThreadLocal(string[] files)

        {

            //Create a new instance of FileData.

            var results = new FileData();

            //Create a new List object to store file information.

            var fileInfos = new List<FileInfo>();

            //Initialize the total file size to 0.

            long totalFileSize = 0;

            //Initialize lastWriteTime to the minimum DateTime value.

            DateTime lastWriteTime = DateTime.MinValue;

            //Initialize lastFileWritten to an empty string.

            string lastFileWritten = "";

            //Create an object to lock date operations.

            object dateLock = new();

            //Execute a parallel loop over the files array using Parallel.For.

            Parallel.For<long>(0, files.Length, () => 0,

                (index, loop, subtotal) => {

                    //Get the file information for the current file.

                    FileInfo fi = new(files[index]);

                    //Get the size of the current file.

                    long size = fi.Length;

                    //Get the last write time of the current file.

                    DateTime lastWrite = fi.LastWriteTimeUtc;

                    //Lock date operations with dateLock.

                    lock (dateLock)

                    {

                        //If the last write time of the current file is greater than the current lastWriteTime value, update lastWriteTime and lastFileWritten.

                        if (lastWriteTime < lastWrite)

                        {

                            lastWriteTime = lastWrite;

                            lastFileWritten = fi.Name;

                        }

                    }

                    //Add the size of the current file to the subtotal using the accumulator.

                    subtotal += size;

                    //Add the FileInfo object for the current file to fileInfos.

                    fileInfos.Add(fi);

                    //Return the updated subtotal value as the result.

                    return subtotal;

                   },

                //Use Interlocked.Add to add all the subtotals together and update totalFileSize.

                (runningTotal) => Interlocked.Add(ref totalFileSize, runningTotal)

            );

            //Assign the results for total size, last file write time, last written file name, and file information List.

            results.FileInfoList = fileInfos;

            results.TotalSize = totalFileSize;

            results.LastFileWriteTime = lastWriteTime;

            results.LastWrittenFileName = lastFileWritten;

            //Return the results object.

            return results;

        }

        //This is the ConvertFilesToBitmaps method that takes a List of file paths and returns a List of Bitmaps.

        public static List<Bitmap> ConvertFilesToBitmaps(List<string> files)

        {

            //Create a new List object to store the Bitmaps.

            var result = new List<Bitmap>();

            //Execute a parallel loop over the files array using Parallel.ForEach.

            Parallel.ForEach(files, file =>

            {

                //Get the file information for the current file.\n

                FileInfo fi = new(file);

                //Get the file extension for the current file.

                string ext = fi.Extension.ToLower();

                //If the file is a jpeg, convert it to a Bitmap and add it to the result List.

                if (ext == ".jpg" || ext == "jpeg")

                {

                    result.Add(ConvertJpgToBitmap(file));

                }

            });

            //Return the result List.

            return result;

        }

        //This is the ConvertFilesToBitmapsAsync method that takes a List of file paths and a CancellationTokenSource and returns a List of Bitmaps asynchronously.

        public static async Task<List<Bitmap>> ConvertFilesToBitmapsAsync(List<string> files, CancellationTokenSource cts)

        {

            //Create a new ParallelOptions object and set its CancellationToken and MaxDegreeOfParallelism properties.

            ParallelOptions po = new()

            {

                CancellationToken = cts.Token,

                MaxDegreeOfParallelism = Environment.ProcessorCount == 1 ? 1

                                            : Environment.ProcessorCount - 1

            };

            //Create a new List object to store the Bitmaps.

            var result = new List<Bitmap>();

            try

            {

                //Execute a parallel forEach loop over the files array using Parallel.ForEachAsync.

                await Parallel.ForEachAsync(files, po, async (file, \_cts) =>

                {

                    //Get the file information for the current file.

                    FileInfo fi = new(file);

                    //Get the file extension for the current file.

                    string ext = fi.Extension.ToLower();

                    //If the file is a jpeg, convert it to a Bitmap, add it to the result List, and delay for 2 seconds.

                    if (ext == ".jpg" || ext == "jpeg")

                    {

                        result.Add(ConvertJpgToBitmap(file));

                        await Task.Delay(2000, \_cts);

                    }

                });

            }

            catch (OperationCanceledException e)

            {

                //Display an error message if the operation was cancelled.

                MessageBox.Show(e.Message);

            }

            finally

            {

                //Dispose of the CancellationTokenSource.

                cts.Dispose();

            }

            //Return the result List.

            return result;

        }

        //This is the ConvertJpgToBitmap method that takes a file path for a jpeg and returns a Bitmap object.

        private static Bitmap ConvertJpgToBitmap(string fileName)

        {

            Bitmap bmp;

            //Open the file stream and create an Image object from it.

            using (Stream bmpStream = File.Open(fileName, FileMode.Open))

            {

                Image image = Image.FromStream(bmpStream);

                //Create a new Bitmap object from the Image object and assign it to bmp.

                bmp = new Bitmap(image);

            }

            //Return the bmp object.

            return bmp;

        }

    }

}

**Form1.cs**

// Import required namespaces

using System.Text;

// Declare namespace for the application

namespace WinFormsParallelLoopApp

{

    // Declare the main form class that inherits the Form class

    public partial class ParallelLoopForm : Form

    {

        // Declare a CancellationTokenSource instance variable

        private CancellationTokenSource \_cts;

        // Declare the class constructor

        public ParallelLoopForm()

        {

            // Call the InitializeComponent() method

            InitializeComponent();

        }

        // Declare the event handler for the FolderBrowseButton click event

        private void FolderBrowseButton\_Click(object sender, EventArgs e)

        {

            // Show the folder browser dialog and get the result

            var result = folderToProcessDialog.ShowDialog();

            // If the user clicked OK

            if (result == DialogResult.OK)

            {

                // Set the selected path to the text box

                FolderToProcessTextBox.Text = folderToProcessDialog.SelectedPath;

            }

        }

        // Declare the event handler for the FolderProcessButton click event

        private void FolderProcessButton\_Click(object sender, EventArgs e)

        {

            // Check if the folder text box is not empty and the folder exists

            if (!string.IsNullOrWhiteSpace(FolderToProcessTextBox.Text) &&

                Directory.Exists(FolderToProcessTextBox.Text))

            {

                // Get an array of file names from the selected folder

                string[] filesToProcess = Directory.GetFiles(FolderToProcessTextBox.Text);

                // Call the GetInfoForFilesThreadLocal() method to get file data

                FileData? results = FileProcessor.GetInfoForFilesThreadLocal(filesToProcess);

                // If the file data is null

                if (results == null)

                {

                    // Set the folder results text box to empty and return

                    FolderResultsTextBox.Text = "";

                    return;

                }

                // Declare a StringBuilder instance to store the result text

                StringBuilder resultText = new();

                // Append the total file count and file size to the result text

                resultText.Append($"Total file count: {results.FileInfoList.Count}; ");

                resultText.AppendLine($"Total file size: {results.TotalSize} bytes");

                // Append the name and time of last written file to the result text

                resultText.Append($"Last written file: {results.LastWrittenFileName} ");

                resultText.Append($"at {results.LastFileWriteTime}");

                // Set the folder results text box text to the result text

                FolderResultsTextBox.Text = resultText.ToString();

            }

        }

        // Declare the event handler for the ProcessJpgsButton click event

        private async void ProcessJpgsButton\_Click(object sender, EventArgs e)

        {

            // Check if the folder text box is not empty and the folder exists

            if (!string.IsNullOrWhiteSpace(FolderToProcessTextBox.Text) &&

                Directory.Exists(FolderToProcessTextBox.Text))

            {

                // Initialize a new CancellationTokenSource instance

                \_cts = new CancellationTokenSource();

                // Get a list of file names from the selected folder

                List<string> filesToProcess = Directory.GetFiles(FolderToProcessTextBox.Text).ToList();

                // Call the ConvertFilesToBitmapsAsync() method to convert files to bitmaps

                List<Bitmap> results = await FileProcessor.ConvertFilesToBitmapsAsync(filesToProcess, \_cts);

                // Declare a StringBuilder instance to store the result text

                StringBuilder resultText = new();

                // Iterate through each bitmap in the results list

                foreach (var bmp in results)

                {

                    // Append the bitmap height to the result text

                    resultText.AppendLine($"Bitmap height: {bmp.Height}");

                }

                // Set the folder results text box text to the result text\n

                FolderResultsTextBox.Text = resultText.ToString();

            }

        }

        // Declare the event handler for the CancelButton click event

        private void CancelButton\_Click(object sender, EventArgs e)

        {

            // If the CancellationTokenSource instance is not null

            if (\_cts != null)

            {

                // Cancel the asynchronous operation

                \_cts.Cancel();

            }

        }

    }

}

**Form1.Designer.cs**

// The namespace for the WinForms application

namespace WinFormsParallelLoopApp

{

    // The partial class for the main form

    partial class ParallelLoopForm

    {

         /// <summary>

        ///  Required designer variable.

        /// </summary>

        private System.ComponentModel.IContainer components = null;

         /// <summary>

        ///  Clean up any resources being used.

        /// </summary>

        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

        protected override void Dispose(bool disposing)

        {

            if (disposing && (components != null))

            {

                components.Dispose();

            }

            base.Dispose(disposing);

        }

         #region Windows Form Designer generated code

        /// <summary>

        ///  Required method for Designer support - do not modify

        ///  the contents of this method with the code editor.

        /// </summary>

        private void InitializeComponent()

        {

            // Create a new group box for the file processor

            this.FileProcessorGroup = new System.Windows.Forms.GroupBox();

            // Create a button for processing JPGs

            this.ProcessJpgsButton = new System.Windows.Forms.Button();

            // Create a text box for displaying results

            this.FolderResultsTextBox = new System.Windows.Forms.TextBox();

            // Create a label for the results text box

            this.label2 = new System.Windows.Forms.Label();

            // Create a button for processing a folder

            this.FolderProcessButton = new System.Windows.Forms.Button();

            // Create a button for browsing for a folder to process

            this.FolderBrowseButton = new System.Windows.Forms.Button();

            // Create a text box for the folder to be processed

            this.FolderToProcessTextBox = new System.Windows.Forms.TextBox();

            // Create a label for the folder to process text box

            this.label1 = new System.Windows.Forms.Label();

            // Create a new folder browser dialog for selecting a folder

            this.folderToProcessDialog = new System.Windows.Forms.FolderBrowserDialog();

            // Create a button for cancelling the operation

            this.CancelButton = new System.Windows.Forms.Button();

            this.FileProcessorGroup.SuspendLayout();

            this.SuspendLayout();

            // Configure the file processor group box

            this.FileProcessorGroup.Controls.Add(this.CancelButton);

            this.FileProcessorGroup.Controls.Add(this.ProcessJpgsButton);

            this.FileProcessorGroup.Controls.Add(this.FolderResultsTextBox);

            this.FileProcessorGroup.Controls.Add(this.label2);

            this.FileProcessorGroup.Controls.Add(this.FolderProcessButton);

            this.FileProcessorGroup.Controls.Add(this.FolderBrowseButton);

            this.FileProcessorGroup.Controls.Add(this.FolderToProcessTextBox);

            this.FileProcessorGroup.Controls.Add(this.label1);

            this.FileProcessorGroup.Location = new System.Drawing.Point(17, 20);

            this.FileProcessorGroup.Name = "FileProcessorGroup";

            this.FileProcessorGroup.Size = new System.Drawing.Size(1077, 344);

            this.FileProcessorGroup.TabIndex = 0;

            this.FileProcessorGroup.TabStop = false;

            this.FileProcessorGroup.Text = "File Processor";

            // Configure the process JPGs button

            this.ProcessJpgsButton.Location = new System.Drawing.Point(748, 28);

            this.ProcessJpgsButton.Name = "ProcessJpgsButton";

            this.ProcessJpgsButton.Size = new System.Drawing.Size(172, 46);

            this.ProcessJpgsButton.TabIndex = 6;

            this.ProcessJpgsButton.Text = "Process JPGs";

            this.ProcessJpgsButton.UseVisualStyleBackColor = true;

            this.ProcessJpgsButton.Click += new System.EventHandler(this.ProcessJpgsButton\_Click);

            // Configure the results text box

            this.FolderResultsTextBox.Location = new System.Drawing.Point(12, 188);

            this.FolderResultsTextBox.Multiline = true;

            this.FolderResultsTextBox.Name = "FolderResultsTextBox";

            this.FolderResultsTextBox.ReadOnly = true;

            this.FolderResultsTextBox.Size = new System.Drawing.Size(1042, 128);

            this.FolderResultsTextBox.TabIndex = 5;

            // Configure the results label

            this.label2.AutoSize = true;

            this.label2.Location = new System.Drawing.Point(12, 144);

            this.label2.Name = "label2";

            this.label2.Size = new System.Drawing.Size(88, 32);

            this.label2.TabIndex = 4;

            this.label2.Text = "Results";

            // Configure the folder process button

            this.FolderProcessButton.Location = new System.Drawing.Point(927, 81);

            this.FolderProcessButton.Name = "FolderProcessButton";

            this.FolderProcessButton.Size = new System.Drawing.Size(127, 46);

            this.FolderProcessButton.TabIndex = 3;

            this.FolderProcessButton.Text = "Process";

            this.FolderProcessButton.UseVisualStyleBackColor = true;

            this.FolderProcessButton.Click += new System.EventHandler(this.FolderProcessButton\_Click);

            // Configure the folder browse button

            this.FolderBrowseButton.Location = new System.Drawing.Point(748, 81);

            this.FolderBrowseButton.Name = "FolderBrowseButton";

            this.FolderBrowseButton.Size = new System.Drawing.Size(172, 46);

            this.FolderBrowseButton.TabIndex = 2;

            this.FolderBrowseButton.Text = "Browse";

            this.FolderBrowseButton.UseVisualStyleBackColor = true;

            this.FolderBrowseButton.Click += new System.EventHandler(this.FolderBrowseButton\_Click);

            // Configure the folder to process text box

            this.FolderToProcessTextBox.Location = new System.Drawing.Point(12, 88);

            this.FolderToProcessTextBox.Name = "FolderToProcessTextBox";

                        this.FolderToProcessTextBox.Size = new System.Drawing.Size(722, 39);

            this.FolderToProcessTextBox.TabIndex = 1;

            //

            // label1

            //

            this.label1.AutoSize = true;

            this.label1.Location = new System.Drawing.Point(12, 42);

            this.label1.Name = "label1";

            this.label1.Size = new System.Drawing.Size(196, 32);

            this.label1.TabIndex = 0;

            this.label1.Text = "Folder to Process";

            //

            // folderToProcessDialog

            //

            this.folderToProcessDialog.RootFolder = System.Environment.SpecialFolder.MyDocuments;

            this.folderToProcessDialog.ShowNewFolderButton = false;

            //

            // CancelButton

            //

            this.CancelButton.Location = new System.Drawing.Point(927, 28);

            this.CancelButton.Name = "CancelButton";

            this.CancelButton.Size = new System.Drawing.Size(127, 46);

            this.CancelButton.TabIndex = 7;

            this.CancelButton.Text = "Cancel";

            this.CancelButton.UseVisualStyleBackColor = true;

            this.CancelButton.Click += new System.EventHandler(this.CancelButton\_Click);

            //

            // ParallelLoopForm

            //

            this.AutoScaleDimensions = new System.Drawing.SizeF(13F, 32F);

            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;

            this.ClientSize = new System.Drawing.Size(1106, 384);

            this.Controls.Add(this.FileProcessorGroup);

            this.Name = "ParallelLoopForm";

            this.Text = "Parallel Loops";

            this.FileProcessorGroup.ResumeLayout(false);

            this.FileProcessorGroup.PerformLayout();

            this.ResumeLayout(false);

        }

        #endregion

        private GroupBox FileProcessorGroup;

        private Button FolderProcessButton;

        private Button FolderBrowseButton;

        private TextBox FolderToProcessTextBox;

        private Label label1;

        private TextBox FolderResultsTextBox;

        private Label label2;

        private FolderBrowserDialog folderToProcessDialog;

        private Button ProcessJpgsButton;

        private Button CancelButton;

    }

}

**Program.cs**

namespace WinFormsParallelLoopApp

{

    internal static class Program

    {

        /// <summary>

        /// The main entry point for the application.

        /// </summary>

        [STAThread]

        static void Main()

        {

            // Initialize the application configuration settings.

            // This method sets high DPI settings or default font.

            ApplicationConfiguration.Initialize();

            // Create an instance of the ParallelLoopForm and run it.

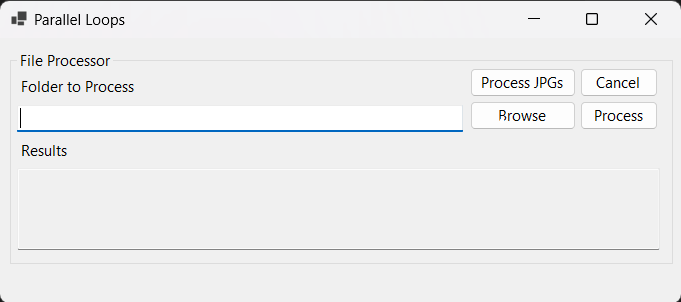
            Application.Run(new ParallelLoopForm());

        }

    }

}

**Output**

****

This software serves as a case study for WinForms applications that use parallel computing. Specifically, it shows how to implement a parallel loop that iterates over a group of data and applies some processing to each piece using the Parallel class in C#.

In this illustration, the software generates a WinForms application that allows the user to enter the desired level of parallelism and the number of repetitions. The software launches a parallel loop when the user selects the Start button, iterating through a set of random numbers and performing calculations on each one. The results are then displayed in the summary window of the application.

Through this program, we can learn the basics of parallel programming in C# and how to use the Parallel class to parallelize loops in our applications. We can also discover the advantages and difficulties of parallel computing, including how to control shared resources and ensure thread safety. Overall, this program serves as a helpful starting point for creating more complex parallel applications and provides a useful illustration of how to integrate parallel programming into a WinForms application.

**chapter08**

**LinqSnippets.cs**

namespace LINQandPLINQsnippets

{

    internal class LinqSnippets

    {

        // Method to query a list of cities that start with "S" and "T"

        internal void QueryCities(List<string> cities)

        {

            // Query is executed with ToList call

            List<string> citiesWithS = cities.Where(s => s.StartsWith('S')).ToList();

            // Query is not executed here

            IEnumerable<string> citiesWithT = cities.Where(s => s.StartsWith('T'));

            // Query is executed here when enumerating

            foreach (string city in citiesWithT)

            {

                // Do something with citiesWithT

            }

        }

        // Method to query a list of people and group them by last name if their age is over 17

        internal void QueryAndGroupPeople(List<Person> people)

        {

            var results = people.AsParallel().Where(p => p.Age > 17)

                .AsSequential().GroupBy(p => p.LastName);

            foreach (var group in results)

            {

                Console.WriteLine($"Last name {group.Key} has {group.Count()} people.");

            }

            // Sample output:

            // Last name Jones has 4220 people.

            // Last name Xu has 3434 people.

            // Last name Patel has 4798 people.

            // Last name Smith has 3051 people.

            // Last name Sanchez has 3811 people.

            // ...

        }

        // Method to demonstrate the query syntax and method syntax in LINQ

        internal void QuerySyntaxAndMethodSyntax(List<Person> people)

        {

            var peopleQuery1 = people.AsParallel().Where(p => p.Age > 17);

            var peopleQuery2 = from person in people.AsParallel()

                               where person.Age > 17

                               select person;

        }

        // Method to query a list of people whose last name starts with "H" in parallel, maintaining the order

        internal void QueryOrderedPeople(List<Person> people)

        {

            var results = people.AsParallel().AsOrdered()

                .Where(p => p.LastName.StartsWith("H"));

        }

        // Method to query a list of people whose last name starts with "H" in parallel, without maintaining the order

        internal void QueryUnorderedPeople(List<Person> people)

        {

            var results = people.AsParallel().AsUnordered()

                .Where(p => p.LastName.StartsWith("H"));

        }

        // Method to process a list of adults who can vote

        internal void ProcessAdultsWhoVote(List<Person> people)

        {

            foreach (var person in people)

            {

                if (person.Age < 18) continue;

                ProcessVoterActions(person);

            }

        }

        private void ProcessVoterActions(Person adult)

        {

            // Add adult to a voter database and process their data.

        }

        // Method to process a list of adults who can vote in parallel using Parallel.ForEach

        internal void ProcessAdultsWhoVoteInParallel(List<Person> people)

        {

            var adults = people.Where(p => p.Age > 17);

            Parallel.ForEach(adults, ProcessVoterActions);

        }

        // Method to process a list of adults who can vote in parallel using PLINQ's ForAll method

        internal void ProcessAdultsWhoVoteWithPlinq(List<Person> people)

        {

            var adults = people.AsParallel().Where(p => p.Age > 17);

            adults.ForAll(ProcessVoterActions);

        }

    }

}

**MergeSamples.cs**

namespace LINQandPLINQsnippets

{

    internal class MergeSamples

    {

        internal IEnumerable<Person> GetImportantChildrenNoMergeSpecified(List<Person> people)

        {

            // This method doesn't specify a merge option, so PLINQ will use the default behavior.

            // The query returns up to three important children who are under 18, executing in parallel.

            return people.AsParallel()

            .Where(p => p.IsImportant && p.Age < 18).Take(3);

        }

        internal IEnumerable<Person> GetImportantChildrenDefaultMerge(List<Person> people)

        {

            // This method sets the merge option to ParallelMergeOptions.Default.

            // PLINQ will use the default merge behavior, which may be different from the no-merge-specified behavior.

            // The query returns up to three important children who are under 18, executing in parallel.

            return people.AsParallel().WithMergeOptions(ParallelMergeOptions.Default)

                .Where(p => p.IsImportant && p.Age < 18).Take(3);

        }

        internal IEnumerable<Person> GetImportantChildrenAutoBuffered(List<Person> people)

        {

            // This method sets the merge option to ParallelMergeOptions.AutoBuffered.

            // PLINQ will execute the query in parallel and buffer intermediate results as necessary.

            // The query returns up to three important children who are under 18.

            return people.AsParallel().WithMergeOptions(ParallelMergeOptions.AutoBuffered)

                .Where(p => p.IsImportant && p.Age < 18).Take(3);

        }

        internal IEnumerable<Person> GetImportantChildrenNotBuffered(List<Person> people)

        {

            // This method sets the merge option to ParallelMergeOptions.NotBuffered.

            // PLINQ will execute the query in parallel and not buffer intermediate results.

            // The query may return fewer than three important children who are under 18.

            return people.AsParallel().WithMergeOptions(ParallelMergeOptions.NotBuffered)

                .Where(p => p.IsImportant && p.Age < 18).Take(3);

        }

        internal IEnumerable<Person> GetImportantChildrenFullyBuffered(List<Person> people)

        {

            // This method sets the merge option to ParallelMergeOptions.FullyBuffered.

            // PLINQ will execute the query in parallel and fully buffer intermediate results.

            // The query returns up to three important children who are under 18.

            return people.AsParallel().WithMergeOptions(ParallelMergeOptions.FullyBuffered)

                .Where(p => p.IsImportant && p.Age < 18).Take(3);

        }

    }

}

**OrderSamples.cs**

// This is a namespace called LINQandPLINQsnippets

namespace LINQandPLINQsnippets

{

    // This is a class called OrderSamples

    internal class OrderSamples

    {

        // This is a method called GetImportantChildrenNoOrder that takes in a List<Person> object called people and returns an IEnumerable<Person> object

        // The method uses LINQ to find all people that are marked as important and are under the age of 18, and returns them with no specific order

        internal IEnumerable<Person> GetImportantChildrenNoOrder(List<Person> people)

        {

            return people.AsParallel()

                .Where(p => p.IsImportant && p.Age < 18);

        }

        // This is a method called GetImportantChildrenUnordered that takes in a List<Person> object called people and returns an IEnumerable<Person> object

        // The method uses PLINQ to find all people that are marked as important and are under the age of 18, and returns them in an unordered list

        internal IEnumerable<Person> GetImportantChildrenUnordered(List<Person> people)

        {

             return people.AsParallel().AsUnordered()

                .Where(p => p.IsImportant && p.Age < 18);

        }

        // This is a method called GetImportantChildrenUnknownOrder that takes in a List<Person> object called people and returns an IEnumerable<Person> object

        // The method uses PLINQ to find all people that are marked as important and are under the age of 18, but it uses AsSequential when checking for age and thus the order is unknown

        internal IEnumerable<Person> GetImportantChildrenUnknownOrder(List<Person> people)

        {

            return people.AsParallel().Where(p => p.IsImportant)

                .AsSequential().Where(p => p.Age < 18);

        }

        // This is a method called GetImportantChildrenPreserveOrder that takes in a List<Person> object called people and returns an IEnumerable<Person> object\n        // The method uses PLINQ to find all people that are marked as important and are under the age of 18, and returns them in the original order of the list

        internal IEnumerable<Person> GetImportantChildrenPreserveOrder(List<Person> people)

        {

            return people.AsParallel().AsOrdered()

                .Where(p => p.IsImportant && p.Age < 18);

        }

        // This is a method called GetImportantChildrenReverseOrder that takes in a List<Person> object called people and returns an IEnumerable<Person> object

        // The method uses PLINQ to find all people that are marked as important and are under the age of 18, and returns them in reverse order of the original list

        internal IEnumerable<Person> GetImportantChildrenReverseOrder(List<Person> people)

        {

            return people.AsParallel().AsOrdered().Reverse()

                .Where(p => p.IsImportant && p.Age < 18);

        }

    }

}

**Person.cs**

//This is a namespace that groups a set of related classes for better organization and to avoid naming conflicts

namespace LINQandPLINQsnippets {

  //This is a class named Person which represents a person with properties such as FirstName, LastName, Age and IsImportant

  internal class Person {

    // The FirstName property represents the person's first name and it can be accessed and set by other classes

    public string FirstName { get; set; } = "";

    // The LastName property represents the person's last name and it can be accessed and set by other classes

    public string LastName { get; set; } = "";

    // The Age property represents the person's age as an integer value and it can be accessed and set by other classes

    public int Age { get; set; }

    // The IsImportant property represents whether or not the person is important and it can be accessed and set by other classes

    public bool IsImportant { get; set; }

  }

}

**PlinqExceptionsExample.cs**

// This namespace contains examples of LINQ and PLINQ snippets.

namespace LINQandPLINQsnippets

{

    // This class demonstrates how to handle exceptions that might occur

    // while processing voters with PLINQ.

    internal class PlinqExceptionsExample

    {

        // This method processes adults who are eligible for voting using PLINQ.

        // It catches AggregateException to handle any exceptions that might occur.

        internal void ProcessAdultsWhoVoteWithPlinq(List<Person> people)

        {

            try

            {

                var adults = people.AsParallel().Where(p => p.Age > 17);

                adults.ForAll(ProcessVoterActions);

            }

            catch (AggregateException ae)

            {

                foreach (var ex in ae.InnerExceptions)

                {

                    Console.WriteLine($"Exception encountered while processing voters. Message: {ex.Message}");

                }

            }

        }

        // This method processes an adult voter.

        // It throws an ArgumentException if the adult is too old.

        // Otherwise, it adds the adult to a voter database and processes their data.

        private void ProcessVoterActions(Person adult)

        {

            if (adult.Age > 120)

            {

                throw new ArgumentException("This person is too old!", nameof(adult));

            }

            // Add adult to a voter database and process their data.

        }

        private SpinLock \_spinLock = new SpinLock();

        // This method processes adults who are eligible for voting using PLINQ.

        // It uses a SpinLock to ensure that only one thread can access the age property of a voter at a time.

        // It sets the age of any voter who is too old to 120 without throwing an exception.

        internal void ProcessAdultsWhoVoteWithPlinq2(List<Person> people)

        {

            var adults = people.AsParallel().Where(p => p.Age > 17);

            adults.ForAll(ProcessVoterActions2);

        }

        // This method processes an adult voter.

        // It updates the age of any voter who is too old to 120 using a SpinLock.

        private void ProcessVoterActions2(Person adult)

        {

            var hasLock = false;

            if (adult.Age > 120)

            {

                try

                {

                    \_spinLock.Enter(hasLock);

                    adult.Age = 120;

                }

                finally

                {

                    if (hasLock) \_spinLock.Exit();

                }

            }

        }

    }

}

**Program.cs**

// Importing the LINQandPLINQsnippets namespace which contains classes and examples for LINQ and PLINQ

using LINQandPLINQsnippets;

// Creating a DateTime format

var timeFmt = "hh:mm:ss.fff tt";

//var orderExample = new OrderSamples();

//Console.WriteLine($"Start time: {DateTime.Now.ToString(timeFmt)}. AsParallel children:");

//OutputListToConsole(orderExample.GetImportantChildrenNoOrder(GetYoungPeople()).ToList());

//Console.WriteLine($"Start time: {DateTime.Now.ToString(timeFmt)}. AsUnordered children:");

//OutputListToConsole(orderExample.GetImportantChildrenUnordered(GetYoungPeople()).ToList());

//Console.WriteLine($"Start time: {DateTime.Now.ToString(timeFmt)}. Sequential after Where children:");

//OutputListToConsole(orderExample.GetImportantChildrenUnknownOrder(GetYoungPeople()).ToList());

//Console.WriteLine($"Start time: {DateTime.Now.ToString(timeFmt)}. AsOrdered children:");

//OutputListToConsole(orderExample.GetImportantChildrenPreserveOrder(GetYoungPeople()).ToList());

//Console.WriteLine($"Start time: {DateTime.Now.ToString(timeFmt)}. Reverse order children:");

//OutputListToConsole(orderExample.GetImportantChildrenReverseOrder(GetYoungPeople()).ToList());

//Console.WriteLine($"Finish time: {DateTime.Now.ToString(timeFmt)}");

//Console.ReadLine();

// Creating an instance of MergeSamples class and outputting the list of Important Children for the different merge options

var mergeExample = new MergeSamples();

Console.WriteLine($"Start time: {DateTime.Now.ToString(timeFmt)}. NoMerge children:");

OutputListToConsole(mergeExample.GetImportantChildrenNoMergeSpecified(GetYoungPeople()).ToList());

Console.WriteLine($"Start time: {DateTime.Now.ToString(timeFmt)}. DefaultMerge children:");

OutputListToConsole(mergeExample.GetImportantChildrenDefaultMerge(GetYoungPeople()).ToList());

Console.WriteLine($"Start time: {DateTime.Now.ToString(timeFmt)}. AutoBuffered children:");

OutputListToConsole(mergeExample.GetImportantChildrenAutoBuffered(GetYoungPeople()).ToList());

Console.WriteLine($"Start time: {DateTime.Now.ToString(timeFmt)}. NotBuffered children:");

OutputListToConsole(mergeExample.GetImportantChildrenNotBuffered(GetYoungPeople()).ToList());

Console.WriteLine($"Start time: {DateTime.Now.ToString(timeFmt)}. FullyBuffered children:");

OutputListToConsole(mergeExample.GetImportantChildrenFullyBuffered(GetYoungPeople()).ToList());

Console.WriteLine($"Finish time: {DateTime.Now.ToString(timeFmt)}");

Console.ReadLine();

// Creating two public functions to get people and young people lists

static List<Person> GetPeople()

{

    return new List<Person>

        {

        new Person { FirstName = "Bob", LastName = "Jones", Age = 23 },

        new Person { FirstName = "Sally", LastName = "Shen", Age = 2 },

        new Person { FirstName = "Joe", LastName = "Smith", Age = 45 },

        new Person { FirstName = "Lisa", LastName = "Samson", Age = 98 },

        new Person { FirstName = "Norman", LastName = "Patterson", Age = 121 },

        new Person { FirstName = "Steve", LastName = "Gates", Age = 40 },

        new Person { FirstName = "Richard", LastName = "Ng", Age = 18 }

    };

}

static List<Person> GetYoungPeople()

{

    return new List<Person>

    {

        new Person { FirstName = "Bob", LastName = "Jones", Age = 23 },

        new Person { FirstName = "Sally", LastName = "Shen", Age = 2, IsImportant = true },

        new Person { FirstName = "Joe", LastName = "Smith", Age = 5, IsImportant = true },

        new Person { FirstName = "Lisa", LastName = "Samson", Age = 9, IsImportant = true },

        new Person { FirstName = "Norman", LastName = "Patterson", Age = 17 },

        new Person { FirstName = "Steve", LastName = "Gates", Age = 20 },

        new Person { FirstName = "Richard", LastName = "Ng", Age = 16, IsImportant = true }

    };

}

// Creating a function to output List of people to the console

static void OutputListToConsole(List<Person> list)

{

    foreach (var item in list)

    {

        Console.WriteLine(item.FirstName);

    }

}

There are no executable files, only code for study or further use.

In summary, the files in the Chapter 8 subdirectory provide a useful introduction to LINQ and PLINQ, as well as illustrations of their use in real-world situations.

You can learn how to use LINQ and PLINQ for concurrent processing in C# by looking at these code samples and understanding how they work. In particular, you can learn how to use classes such as Partitioner to manipulate data for parallel processing, and how to build queries that take advantage of parallel processing to increase speed.

TASK 4. Answer the follwoing questions:

***a) Did ChatGPT by adding comments to your programs help you to understand the code? Yes***

***b) Was your experience with ChatGPT smooth/plesent? Yes***

***c) Do you think moving forward you might do this for other code that is new and hard to comprehend? Yes***