/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Part 15: Threading

\*

\* Topic: 1. The singleton object.

\* 2. Use Monitor class with a singleton object to control the serial

\* use of a common resource (Console window in this example)

\* by multiple threads to avoid collisions.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Be sure to provide an integer value on the command line. To do this:

\*

\* 1) Right-click on the project in Solution Explorer and click

\* Properties.

\*

\* 2) In the Properties window, click the Debug tab.

\*

\* 3) In the "Command line arguments" field, enter a whole number.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

using System;

using System.Collections.Generic;

using System.Text;

using System.Threading;

namespace ThreadsDemo

{

// This static class implements the Singleton pattern. This pattern

// ensures that there is one and only one object throughout the entire

// application.

internal static class SyncObject

{

// The actual object that will be used to synchronize.

private static object \_sync;

// A read-only property that returns the sync object. If the

// object was not yet created, it is created here.

public static object Sync

{

get

{

if (null == \_sync)

{

\_sync = new object();

}

return \_sync;

}

}

}

internal class ComplicatedCalculator

{

// Member variable that represent the number of milliseconds

// to pause the thread.

private int \_millisecondsToPause;

// Member variables that are used for the calculation.

private double \_results;

public ComplicatedCalculator(int millisecondsToPause)

{

MillisecondsToPause = millisecondsToPause;

}

// Provide yet another method that takes a single object

// argument. This will parse the object and get the input values

// from it.

internal void CalculateValue(object input)

{

try

{

// Attempt to convert the input object to an array of

// doubles.

double[] inputValues = input as double[];

// If the conversion worked and there are at least two elements

// in the double array, run the calculation.

if (null != inputValues && inputValues.Length >= 2)

{

Results = CalculateValue(inputValues[0], inputValues[1]);

}

}

catch (Exception ex)

{

Monitor.Enter(SyncObject.Sync);

try

{

// Save the foreground color of the console window.

ConsoleColor originalcolor = Console.ForegroundColor;

// Change the foreground color in the console.

Console.ForegroundColor = ConsoleColor.Red;

// Display a message that we're starting the task.

Console.WriteLine

("\n\t{0}: {1} - The following exception occurred:\n{2}.",

Thread.CurrentThread.ManagedThreadId,

Thread.CurrentThread.Name,

ex.Message);

// Set the console color back to the original value.

Console.ForegroundColor = originalcolor;

}

finally

{

// End synchronization

Monitor.Exit(SyncObject.Sync);

}

}

}

// This method represents a task that could potentially run for

// a long period of time.

internal double CalculateValue

(double firstNumber, double secondNumber)

{

double answer = 0;

// Get the currently-running thread object.

Thread threadObject = Thread.CurrentThread;

ConsoleColor originalcolor;

// Synchronize the following code using the Singleton object.

Monitor.Enter(SyncObject.Sync);

try

{

// Save the foreground color of the console window.

originalcolor = Console.ForegroundColor;

// Change the foreground color in the console.

Console.ForegroundColor = ConsoleColor.Red;

// Display a message that we're starting the task.

Console.WriteLine

("\n\t{0}: {1} - Starting the calculation task.",

threadObject.ManagedThreadId, threadObject.Name);

// Set the console color back to the original value.

Console.ForegroundColor = originalcolor;

}

finally

{

// End synchronization

Monitor.Exit(SyncObject.Sync);

}

// Pause for a moment.

System.Threading.Thread.Sleep(MillisecondsToPause);

// Perform the calculation.

answer = Math.Pow(firstNumber, secondNumber);

// Pause for another moment.

System.Threading.Thread.Sleep(MillisecondsToPause);

// Synchronize the following code using the Singleton object.

Monitor.Enter(SyncObject.Sync);

try

{

// Change the foreground color in the console.

Console.ForegroundColor = ConsoleColor.Red;

// Display a message that we're done with the task.

Console.WriteLine

("\n\t{0}: {1} - Done with the calculation task.",

threadObject.ManagedThreadId, threadObject.Name);

// Set the console color back to the original value.

Console.ForegroundColor = originalcolor;

}

finally

{

// End synchronization

Monitor.Exit(SyncObject.Sync);

}

// Return the answer. The risk here is that the parent thread

// may wake up and finish before we can return from here. Not

// a very good synchronization mechanism.

return answer;

}

// Make this available to code outside of this class.

internal int MillisecondsToPause

{

get { return \_millisecondsToPause; }

set

{

if (value < 0)

{

throw new ArgumentException

("Milliseconds must be greater than or equal to 0.");

}

\_millisecondsToPause = value;

}

}

// Provide a way to allow code outside this class to access

// the results. This is read-only to code outside this class

// (notice the private accessor on set).

internal double Results

{

get { return \_results; }

private set { \_results = value; }

}

}

class Program

{

private static int GetMilliseconds(string s)

{

int milliseconds = 0;

// If this call fails, milliseconds will be set to zero.

if (int.TryParse(s, out milliseconds))

{

// If the user types in a low number, let's assume

// that they entered in the number of seconds and

// convert the value to milliseconds.

if (milliseconds < 250)

{

milliseconds = 1000;

}

}

return milliseconds;

}

private static void JoinThread(int mainThreadId, Thread thread)

{

// Join the secondary thread, but don't wait forever.

Monitor.Enter(SyncObject.Sync);

try

{

Console.WriteLine("\n{0}: Joining thread {1}: {2}.",

mainThreadId, thread.ManagedThreadId, thread.Name);

}

finally

{

Monitor.Exit(SyncObject.Sync);

}

if (!thread.Join(10000))

{

Monitor.Enter(SyncObject.Sync);

try

{

Console.WriteLine

("\n{0}: Thread {1}: {2} " +

"is still alive. Calling Abort().",

mainThreadId, thread.ManagedThreadId, thread.Name);

}

finally

{

Monitor.Exit(SyncObject.Sync);

}

// If it is, abort the thread and then Join it again

// to give the processor time to abort the thread.

thread.Abort();

if (!thread.Join(10000))

{

Monitor.Enter(SyncObject.Sync);

try

{

Console.WriteLine

("\n{0}: Thread {1}: {2} is still running!",

mainThreadId, thread.ManagedThreadId, thread.Name);

}

finally

{

Monitor.Exit(SyncObject.Sync);

}

}

}

}

static void Main(string[] args)

{

// Get the currently-running thread object.

Thread primaryThreadObject = Thread.CurrentThread;

// Set the name of the thread. This will help with debugging

// when looking at the Threads window.

primaryThreadObject.Name = "The Main Thread";

// Get the thread ID so that we can use it in output statements.

int threadId = primaryThreadObject.ManagedThreadId;

try

{

// Display a message to show we're in Main().

Console.WriteLine("{0}: Starting the program.", threadId);

// Get the number of milliseconds from the arguments

// passed in from the command line.

int milliseconds = GetMilliseconds(args[0]);

// Create the ComplicatedCalculator objects.

ComplicatedCalculator cc1 =

new ComplicatedCalculator(milliseconds);

ComplicatedCalculator cc2 =

new ComplicatedCalculator(milliseconds);

// Create the ParameterizedThreadStart delegate. This

// delegate will be used to pass an array of doubles

// to the method on the secondary thread.

double[] numbers1 = { 10.4, 7.451 };

double[] numbers2 = { 18.7, 3.6 };

ParameterizedThreadStart threadedMethod1 =

new ParameterizedThreadStart(cc1.CalculateValue);

ParameterizedThreadStart threadedMethod2 =

new ParameterizedThreadStart(cc2.CalculateValue);

// Create the thread objects and start the threads. In this

// case, when we call Start(), we pass in the double array

// as an argument.

Thread secondaryThread1 = new Thread(threadedMethod1);

Thread secondaryThread2 = new Thread(threadedMethod2);

// Set the name of the secondary threads.

secondaryThread1.Name = "Calculation Thread #1";

secondaryThread2.Name = "Calculation Thread #2";

// Start the threads.

secondaryThread1.Start(numbers1);

secondaryThread2.Start(numbers2);

// Notice now that each Console.WriteLine() call is now

// in a critical section. This is here to synchronize with

// the threads when they are writing their output in a

// different color.

Monitor.Enter(SyncObject.Sync);

try

{

Console.WriteLine

("\n{0}: Now I'm going to go do something else.",

threadId);

}

finally

{

Monitor.Exit(SyncObject.Sync);

}

System.Threading.Thread.Sleep(1500);

Monitor.Enter(SyncObject.Sync);

try

{

Console.WriteLine("\n{0}: Like talk about the weather.",

threadId);

}

finally

{

Monitor.Exit(SyncObject.Sync);

}

System.Threading.Thread.Sleep(1500);

Monitor.Enter(SyncObject.Sync);

try

{

Console.WriteLine("\n{0}: Or the latest news.",

threadId);

}

finally

{

Monitor.Exit(SyncObject.Sync);

}

System.Threading.Thread.Sleep(1500);

Monitor.Enter(SyncObject.Sync);

try

{

Console.WriteLine("\n{0}: You know, my foot hurts.",

threadId);

}

finally

{

Monitor.Exit(SyncObject.Sync);

}

System.Threading.Thread.Sleep(1500);

Monitor.Enter(SyncObject.Sync);

try

{

Console.WriteLine("\n{0}: I love hotdogs!",

threadId);

}

finally

{

Monitor.Exit(SyncObject.Sync);

}

System.Threading.Thread.Sleep(1500);

Monitor.Enter(SyncObject.Sync);

try

{

Console.WriteLine

("\n{0}: How much is a shake at Burgermaster?",

threadId);

}

finally

{

Monitor.Exit(SyncObject.Sync);

}

System.Threading.Thread.Sleep(1500);

Monitor.Enter(SyncObject.Sync);

try

{

Console.WriteLine("\n{0}: Ok, now I'm getting hungry!",

threadId);

}

finally

{

Monitor.Exit(SyncObject.Sync);

}

System.Threading.Thread.Sleep(1500);

// Join one of the secondary threads.

JoinThread(threadId, secondaryThread1);

// Join the other secondary thread.

JoinThread(threadId, secondaryThread2);

// We don't need to synchronize here because the threads

// should already be done.

Console.WriteLine("\n{0}: The result from {1} is: {2}",

threadId, secondaryThread1.ManagedThreadId, cc1.Results);

Console.WriteLine("\n{0}: The result from {1} is: {2}",

threadId, secondaryThread2.ManagedThreadId, cc2.Results);

}

catch (Exception e)

{

Monitor.Enter(SyncObject.Sync);

try

{

Console.WriteLine("\n{0}: EXCEPTION: {1}.",

threadId, e.Message);

}

finally

{

Monitor.Exit(SyncObject.Sync);

}

}

// Pause so we can look at the console window.

Console.Write("\n\n{0}: Press <ENTER> to end: ",

threadId);

Console.ReadLine();

}

}

}