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

\*

\* Part 16: Threading

\*

\* Topic: 1. Use "lock" blocks with a singleton object to control the serial

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

\* by multiple threads to avoid collisions.

\* 2. Show the CIL code that really uses the Monitor class with

\* Enter and Exit methods in try/finally blocks.

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

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

\*

\* 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.

\*

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

usingSystem**;**

usingSystem.Collections.Generic**;**

usingSystem.Text**;**

usingSystem.Threading**;**

namespaceThreadsDemo

**{**

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

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

// application.

internalstaticclassSyncObject

**{**

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

privatestaticobject\_sync**;**

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

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

publicstaticobjectSync

**{**

get

**{**

if **(**null==\_sync**)**

**{**

\_sync=newobject**();**

**}**

return\_sync**;**

**}**

**}**

**}**

internalclassComplicatedCalculator

**{**

// Member variable that represent the number of milliseconds

// to pause the thread.

privateint\_millisecondsToPause**;**

// Member variables that are used for the calculation.

privatedouble\_results**;**

publicComplicatedCalculator**(**intmillisecondsToPause**)**

**{**

MillisecondsToPause=millisecondsToPause**;**

**}**

// Provide yet another method that takes a single object

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

// from it.

internalvoidCalculateValue**(**objectinput**)**

**{**

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

// doubles.

double**[]** inputValues=inputasdouble**[];**

// 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**]);**

**}**

**}**

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

// a long period of time.

internaldoubleCalculateValue

**(**doublefirstNumber**,** doublesecondNumber**)**

**{**

doubleanswer=0**;**

// Get the currently-running thread object.

ThreadthreadObject=Thread.CurrentThread**;**

// Save the foreground color of the console window.

ConsoleColororiginalcolor**;**

// Synchronize the following code using the Singleton object.

lock **(**SyncObject.Sync**)**

**{**

// 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**;**

**}**

// 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.

lock **(**SyncObject.Sync**)**

**{**

// 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**;**

**}**

// 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.

returnanswer**;**

**}**

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

internalintMillisecondsToPause

**{**

get **{** return\_millisecondsToPause**; }**

set

**{**

if **(**value<0**)**

**{**

thrownewArgumentException

**(**"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).

internaldoubleResults

**{**

get **{** return\_results**; }**

privateset **{** \_results=value**; }**

**}**

**}**

classProgram

**{**

privatestaticintGetMilliseconds**(**strings**)**

**{**

intmilliseconds=0**;**

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

if **(**int.TryParse**(**s**,** outmilliseconds**))**

**{**

// 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**;**

**}**

**}**

returnmilliseconds**;**

**}**

privatestaticvoidJoinThread**(**intmainThreadId**,** Threadthread**)**

**{**

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

lock **(**SyncObject.Sync**)**

**{**

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

mainThreadId**,** thread.ManagedThreadId**,** thread.Name**);**

**}**

if **(**!thread.Join**(**10000**))**

**{**

lock **(**SyncObject.Sync**)**

**{**

Console.WriteLine

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

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

mainThreadId**,** thread.ManagedThreadId**,** thread.Name**);**

**}**

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

thread.Abort**();**

if **(**!thread.Join**(**10000**))**

**{**

lock **(**SyncObject.Sync**)**

**{**

Console.WriteLine

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

mainThreadId**,** thread.ManagedThreadId**,** thread.Name**);**

**}**

**}**

**}**

**}**

staticvoidMain**(**string**[]** args**)**

**{**

// Get the currently-running thread object.

ThreadprimaryThreadObject=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.

intthreadId=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.

intmilliseconds=GetMilliseconds**(**args**[**0**]);**

// Create the ComplicatedCalculator objects.

ComplicatedCalculatorcc1=

newComplicatedCalculator**(**milliseconds**);**

ComplicatedCalculatorcc2=

newComplicatedCalculator**(**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 **};**

ParameterizedThreadStartthreadedMethod1=

newParameterizedThreadStart**(**cc1.CalculateValue**);**

ParameterizedThreadStartthreadedMethod2=

newParameterizedThreadStart**(**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.

ThreadsecondaryThread1=newThread**(**threadedMethod1**);**

ThreadsecondaryThread2=newThread**(**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.

lock **(**SyncObject.Sync**)**

**{**

Console.WriteLine

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

threadId**);**

**}**

System.Threading.Thread.Sleep**(**1500**);**

lock **(**SyncObject.Sync**)**

**{**

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

threadId**);**

**}**

System.Threading.Thread.Sleep**(**1500**);**

lock **(**SyncObject.Sync**)**

**{**

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

threadId**);**

**}**

System.Threading.Thread.Sleep**(**1500**);**

lock **(**SyncObject.Sync**)**

**{**

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

threadId**);**

**}**

System.Threading.Thread.Sleep**(**1500**);**

lock **(**SyncObject.Sync**)**

**{**

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

threadId**);**

**}**

System.Threading.Thread.Sleep**(**1500**);**

lock **(**SyncObject.Sync**)**

**{**

Console.WriteLine

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

threadId**);**

**}**

System.Threading.Thread.Sleep**(**1500**);**

lock **(**SyncObject.Sync**)**

**{**

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

threadId**);**

**}**

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 **(**Exceptione**)**

**{**

lock **(**SyncObject.Sync**)**

**{**

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

threadId**,** e.Message**);**

**}**

**}**

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

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

threadId**);**

Console.ReadLine**();**

**}**

**}**

**}**