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

\*

\* Part 16C: Threading

\*

\* Topic: 1. Uses an array to store multiple instances of the

\* ComplicatedCalculator so several secondary foreground

\* threads will be started.

\* 2. Uses the Random object to generate different numbers

\* for each instance of the ComplicatedCalculator.

\* 3. Uses the AutoResetEvent object as an alternative to using

\* the Join method to force the main thread to wait until

\* all secondary threads are done:

\* a. WaitOne method of the AutoResetEvent object

\* is used to cause the Main thread to wait until all

\* secondary threads are done.

\* b. Set method is used by the last secondary thread

\* to wake up the Main thread.

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

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

\*

\* 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 delegate will be used by each thread to call the callback

// method in the Program class.

internaldelegatevoidCallbackDelegate**();**

// 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**;**

**}**

**}**

**}**

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

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

// application AND that the original console foreground color is set

// only once.

internalstaticclassConsoleColorObject

**{**

// The actual object that will be used to synchronize

// the console foreground color.

privatestaticobject\_colorSync**;**

privatestaticConsoleColor\_originalColor**;**

// This method will only allow the foreground console color

// to be set only once.

publicstaticvoidSetConsoleColor**()**

**{**

if **(**null==\_colorSync**)**

**{**

\_colorSync=newobject**();**

\_originalColor=Console.ForegroundColor**;**

**}**

**}**

// A read-only property that returns the original foreground

// color of the console.

publicstaticConsoleColorConsoleForegroundColor

**{**

get

**{**

return\_originalColor**;**

**}**

**}**

**}**

internalclassComplicatedCalculator

**{**

// Member variable that represent the number of milliseconds

// to pause the thread.

privateint\_millisecondsToPause**;**

// Member variables that are used for the calculation.

privatedouble\_firstNumber**;**

privatedouble\_secondNumber**;**

privatedouble\_results**;**

// Member variable that stores the CallbackDelegate.

CallbackDelegate\_completedMethod**;**

// Provide another constructor that takes a CallbackDelegate

// in addition to the milliseconds.

publicComplicatedCalculator

**(**intmillisecondsToPause**,** CallbackDelegatecallback**)**

**:** this**(**millisecondsToPause**)**

**{**

Callback=callback**;**

**}**

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**)**

**{**

try

**{**

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

// doubles.

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

// This is new because of a bug found in the code.

FirstNumber=inputValues**[**0**];**

SecondNumber=inputValues**[**1**];**

// 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**(**FirstNumber**,** SecondNumber**);**

**}**

**}**

catch **(**Exceptionex**)**

**{**

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

**{**

// 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=ConsoleColorObject.ConsoleForegroundColor**;**

**}**

**}**

finally

**{**

if **(**null!=Callback**)**

**{**

Callback**();**

**}**

**}**

**}**

// 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**;**

// 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 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=ConsoleColorObject.ConsoleForegroundColor**;**

**}**

// 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=ConsoleColorObject.ConsoleForegroundColor**;**

**}**

// 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 access to the first number.

internaldoubleFirstNumber

**{**

get **{** return\_firstNumber**; }**

set **{** \_firstNumber=value**; }**

**}**

// Provide access to the second number.

internaldoubleSecondNumber

**{**

get **{** return\_secondNumber**; }**

set **{** \_secondNumber=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**; }**

**}**

// This property is used to access the callback delegate

// object. This is private because it's only used in this

// class

privateCallbackDelegateCallback

**{**

get **{** return\_completedMethod**; }**

set **{** \_completedMethod=value**; }**

**}**

**}**

classProgram

**{**

// This constant indicates how many different operations to

// use. This equals the number of threads to create.

privateconstintMAX\_OPERATIONS=10**;**

// This variable indicates how many threads have completed.

// The highest value this variable will contain is equal to

// the MAX\_OPERATIONS constant above.

privatestaticint\_completedThreads=0**;**

// Create an AutoResetEvent object to be used to synchronize

// the primary thread and all the ThreadPool threads. The

// boolean indicates that this object IS NOT in a signaled

// state.

privatestaticAutoResetEventsignalPrimaryThread=

newAutoResetEvent**(**false**);**

// This method is called by each thread to indicate they are done.

publicstaticvoidThreadDoneCallback**()**

**{**

// Beware that this variable is a common resource among multiple

// threads and its update could be interrupted that will cause

// this application to not end as the main thread will be

// waiting forever to be woke up.

\_completedThreads++;

// There is also a possibility that the code in the if statement

// below will be run more than once (it is only supposed to run

// once for the last thread coming through). However, it is possible

// for the next to last thread to lose its time slice at this exact

// moment allowing for the last thread to come through. Then when

// the next to last thread gets its time slice back, it will go

// through. Both will execute the code in the if statement.

// TO FIX: Place the code line above inside the lock block below and

// see the difference.

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

**{**

// If all the threads completed, it's time to wake up the

// primary thread.

if **(**\_completedThreads>=MAX\_OPERATIONS**)**

**{**

// Change the foreground color in the console. Since this

// code is being executed on one of the secondary threads

// the color is being changed to reflect that.

Console.ForegroundColor=ConsoleColor.Red**;**

Console.WriteLine

**(**"\n\t{0}: Signalling the primary thread to wake up."

**,** Thread.CurrentThread.ManagedThreadId**);**

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

Console.ForegroundColor=ConsoleColorObject.ConsoleForegroundColor**;**

// Set signals the AutoResetEvent object.

signalPrimaryThread.Set**();**

**}**

**}**

**}**

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**;**

**}**

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**;**

// Set the original color of the color.

ConsoleColorObject.SetConsoleColor**();**

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**]);**

// Declare an array of ComplicatedCalculator objects

ComplicatedCalculator**[]** ccList=

newComplicatedCalculator**[**MAX\_OPERATIONS**];**

// Populate the array with object.

for **(**inti=0**;** i<ccList.Length**;** i++**)**

**{**

// Pass an instance of the callback delegate defined above

// to the asynchronous type.

CallbackDelegatecd=

newCallbackDelegate**(**Program.ThreadDoneCallback**);**

ComplicatedCalculatorcc=

newComplicatedCalculator**(**milliseconds**,** cd**);**

// Store several instances of the asynchronous type into

// an array.

ccList**[**i**]** =cc**;**

**}**

// Create a couple of number generators for the data.

RandomfirstNum=newRandom**(**DateTime.Now.Millisecond**);**

RandomsecondNum=newRandom**(**DateTime.Now.Minute**);**

// Start up a set of foreground threads for each instance of the

// asynchronous type stored in the array.

for **(**inti=0**;** i<ccList.Length**;** i++**)**

**{**

double**[]** numbers=

**{** firstNum.Next**(**1**,** 30**),** secondNum.Next**(**1**,** 7**) };**

ParameterizedThreadStartthreadedMethod=

newParameterizedThreadStart**(**ccList**[**i**]**.CalculateValue**);**

// Create the thread object and start the thread. In this

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

// as an argument.

ThreadsecondaryThread=newThread**(**threadedMethod**);**

// Set the name of the secondary thread.

secondaryThread.Name="Calculation Thread #"+i.ToString**();**

// Start the threads.

secondaryThread.Start**(**numbers**);**

**}**

// 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**);**

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

**{**

Console.WriteLine

**(**"\n{0}: Waiting for threads to complete."**,**

threadId**);**

**}**

// Put the primary thread into a wait state. This

// will wait until the AutoResetEvent object is

// signalled through a call to Set().

signalPrimaryThread.WaitOne**();**

// Now display the results of all the threads.

for **(**inti=0**;** i<ccList.Length**;** i++**)**

**{**

Console.WriteLine**(**"\n{0}: {1} raised to {2} = {3}"**,**

threadId**,**

ccList**[**i**]**.FirstNumber**,**

ccList**[**i**]**.SecondNumber**,**

ccList**[**i**]**.Results**);**

**}**

**}**

catch **(**Exceptione**)**

**{**

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**();**

**}**

**}**

**}**