

Threads

Introduction

- The C# programming language allows us to write code that is executed in parallel through multi-threading.

Similarly to processes, that can execute concurrently on the same operating system, we can have multiple threads running concurrently within a single process. Unlike processes that are fully isolated from each other, threads might share the heap memory with each other.

- Using threads isn't always a good choice. In some cases, multiple threads damage the performance, instead of improving it.

Simple Threads

- When a C# program starts, a single thread is automatically created by the CLR and the operating system.
Unless we create more threads, our application will be a single threaded one.
- The simplest way to create a new thread in our application is to instantiate the 'Thread' object and call its 'Start' method. The 'Thread' constructor takes a 'ThreadStart' delegate.

Simple Threads

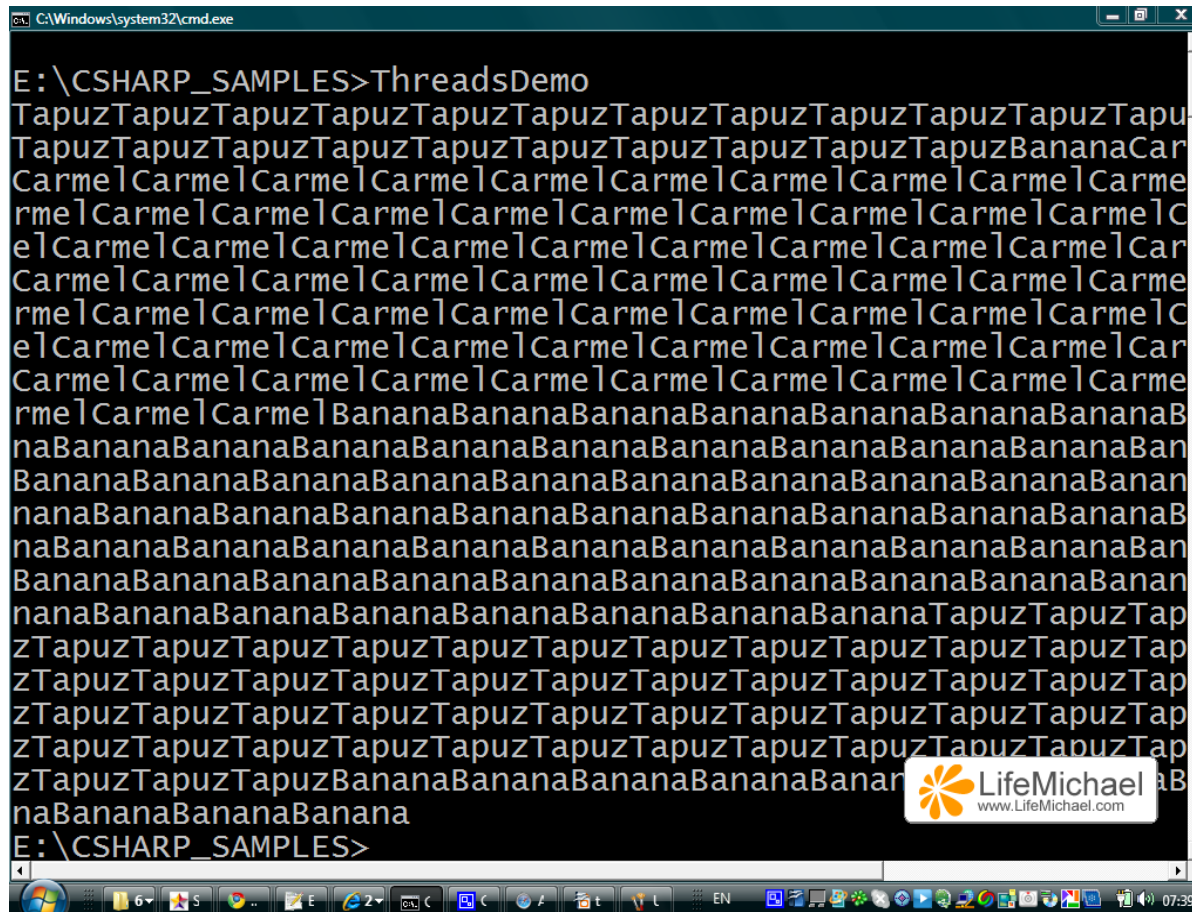
```
using System;
using System.Threading;

namespace abelski.csharp
{
    class ThreadsDemo
    {
        static void Main()
        {
            Thread t1 = new Thread(
                new ThreadStart(ThreadsDemo.WriteTapuz));
            Thread t2 = new Thread(
                new ThreadStart(ThreadsDemo.WriteBanana));
            t1.Start();
            t2.Start();
            for(int i=0; i<100; i++)
            {
                Console.Write("Carmel");
            }
        }
    }
}
```

Simple Threads

```
static void WriteBanana()  
{  
    for(int i=0; i<100; i++)  
    {  
        Console.Write("Banana");  
    }  
}  
  
static void WriteTapuz()  
{  
    for(int i=0; i<100; i++)  
    {  
        Console.Write("Tapuz");  
    }  
}  
}
```

Simple Threads



```
C:\Windows\system32\cmd.exe
E:\CSHARP_SAMPLES>ThreadsDemo
TapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapu
TapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzBananaCar
CarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarme
rmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelC
elCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCar
CarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarme
rmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelC
elCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCar
CarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarmelCarme
rmelCarmelCarmelBananaBananaBananaBananaBananaBananaBananaBananaB
naBananaBananaBananaBananaBananaBananaBananaBananaBananaBananaBan
BananaBananaBananaBananaBananaBananaBananaBananaBananaBananaBanana
nanaBananaBananaBananaBananaBananaBananaBananaBananaBananaBananaBan
naBananaBananaBananaBananaBananaBananaBananaBananaBananaBananaBanana
BananaBananaBananaBananaBananaBananaBananaBananaBananaBananaBanana
nanaBananaBananaBananaBananaBananaBananaBananaBananaBananaBananaTapuzTapuzTap
zTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTap
zTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTap
zTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTapuzTap
zTapuzTapuzTapuzBananaBananaBananaBananaBananaBananaBananaBanana
naBananaBananaBanana
E:\CSHARP_SAMPLES>
```

The ThreadStart Delegate

- The ThreadStart delegate we should pass over to the Thread constructor can be omitted. Passing over the name of specific method will automatically result in passing over a ThreadStart delegate.

```
Thread t1 = new Thread(new ThreadStart(ThreadsDemo.WriteTapuz));
```

is equivalent to

```
Thread t1 = new Thread(ThreadsDemo.WriteTapuz);
```

The Join Method

- Calling the 'Join' method on a Thread object will cause the current thread to pause its work and wait till that thread ends.

The Join Method

```
using System;
using System.Threading;

namespace abelski.csharp
{
    class JoinDemo
    {
        static void Main()
        {
            Thread t1 = new Thread(WriteOrange);
            Thread t2 = new Thread(WriteBanana);
            t1.Start();
            t2.Start();
            t1.Join();
            t2.Join();
            for(int i=0; i<100; i++)
            {
                Console.Write("Carmel");
            }
        }
    }
}
```

The Join Method

```
static void WriteBanana()  
{  
    for(int i=0; i<100; i++)  
    {  
        Console.Write("Banana");  
    }  
}  
  
static void WriteOrange()  
{  
    for(int i=0; i<100; i++)  
    {  
        Console.Write("Orange");  
    }  
}  
}
```

The Join Method

[illegible]

The IsAlive Property

- The IsAlive property, each thread has, returns 'true' as long as the thread is still running. Once the thread ends, IsAlive property returns 'false'.

Once a thread ends we cannot restart it. The thread ends once the method referenced in its constructor ends.

The IsAlive Property

```
using System;
using System.Threading;

namespace abelski.csharp
{
    class JoinDemo
    {
        static void Main()
        {
            Thread t1 = new Thread(WriteOrange);
            Thread t2 = new Thread(WriteBanana);
            t1.Start();
            t2.Start();
            t1.Join();
            t2.Join();
            Console.WriteLine("\nt1.IsAlive="+t1.IsAlive);
            Console.WriteLine("\nt2.IsAlive="+t2.IsAlive);
            Console.WriteLine("\n\n");
            for(int i=0; i<100; i++)
            {
                Console.WriteLine("Carmel");
            }
        }
    }
}
```

The IsAlive Property

```
static void WriteBanana()  
{  
    for(int i=0; i<100; i++)  
    {  
        Console.Write("Banana");  
    }  
}  
  
static void WriteOrange()  
{  
    for(int i=0; i<100; i++)  
    {  
        Console.Write("Orange");  
    }  
}  
}
```

The IsAlive Property

[illegible]

The Thread.Sleep Method

- The `Thread.Sleep` method pauses the current thread for a specified period.
- Calling `Thread.Sleep(0)` pauses the current thread long enough to allow other threads to execute. Passing over 0 is a good practice to avoid threads starvation.

The Thread.Sleep Method

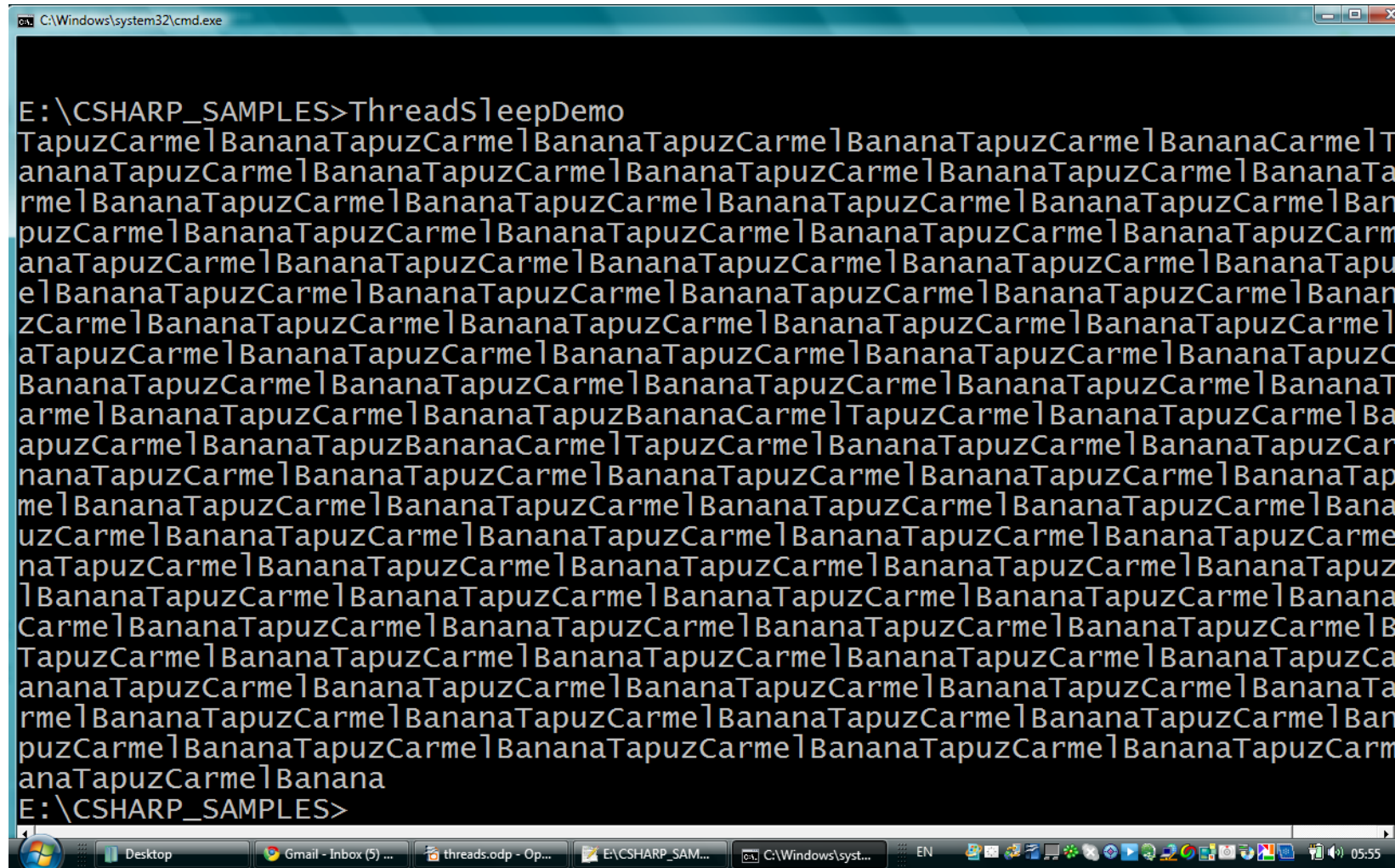
```
using System;
using System.Threading;

namespace abelski.csharp
{
    class ThreadSleepDemo
    {
        static void Main()
        {
            Thread t1 = new Thread(
                new ThreadStart(ThreadSleepDemo.WriteTapuz));
            Thread t2 = new Thread(
                new ThreadStart(ThreadSleepDemo.WriteBanana));
            t1.Start();
            t2.Start();
            for(int i=0; i<100; i++)
            {
                Console.Write("Carmel");
                Thread.Sleep(100);
            }
        }
    }
}
```

The Thread.Sleep Method

```
static void WriteBanana()  
{  
    for(int i=0; i<100; i++)  
    {  
        Console.Write("Banana");  
        Thread.Sleep(100);  
    }  
}  
  
static void WriteTapuz()  
{  
    for(int i=0; i<100; i++)  
    {  
        Console.Write("Tapuz");  
        Thread.Sleep(100);  
    }  
}  
}
```

The Thread.Sleep Method



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The prompt is at "E:\CSHARP_SAMPLES>". The user has entered "ThreadSleepDemo", which has triggered a rapid succession of text being printed to the screen. The text consists of many lines of "BananaTapuzCarme" and "BananaCarme" separated by vertical bars, with some lines ending in "Banana". The text is printed so fast that it appears as a dense block of characters. The command prompt window is open over a desktop background, and the taskbar at the bottom shows several open applications: "Desktop", "Gmail - Inbox (5) ...", "threads.odp - Op...", "E:\CSHARP_SAM...", and "C:\Windows\sys...". The system clock in the bottom right corner shows "05:55".

```
E:\CSHARP_SAMPLES>ThreadSleepDemo
BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaCarme|T
ananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTa
rme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|Ban
puzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarm
anaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapu
e|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|Banar
zCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|
aTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzC
BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaT
arme|BananaTapuzCarme|BananaTapuzBananaCarme|TapuzCarme|BananaTapuzCarme|Ba
apuzCarme|BananaTapuzBananaCarme|TapuzCarme|BananaTapuzCarme|BananaTapuzCar
nanaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTap
me|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|Bana
uzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme
naTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuz
|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|Banana
Carme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|B
TapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCa
ananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTa
rme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|Ban
puzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarme|BananaTapuzCarm
anaTapuzCarme|Banana
E:\CSHARP_SAMPLES>
```

The Thread.Name Property

- The `Thread.Name` property allows us to set a name for each thread we work with.
- When debugging threads it is a good practice setting a meaningful name for each one of the threads.

The `Thread.CurrentThread` Property

- The `Thread.CurrentThread` static property holds the reference for the currently executing thread object.
- Calling `Thread.CurrentThread.Name` should return the name of the currently executing thread object.

The Thread.CurrentThread Property

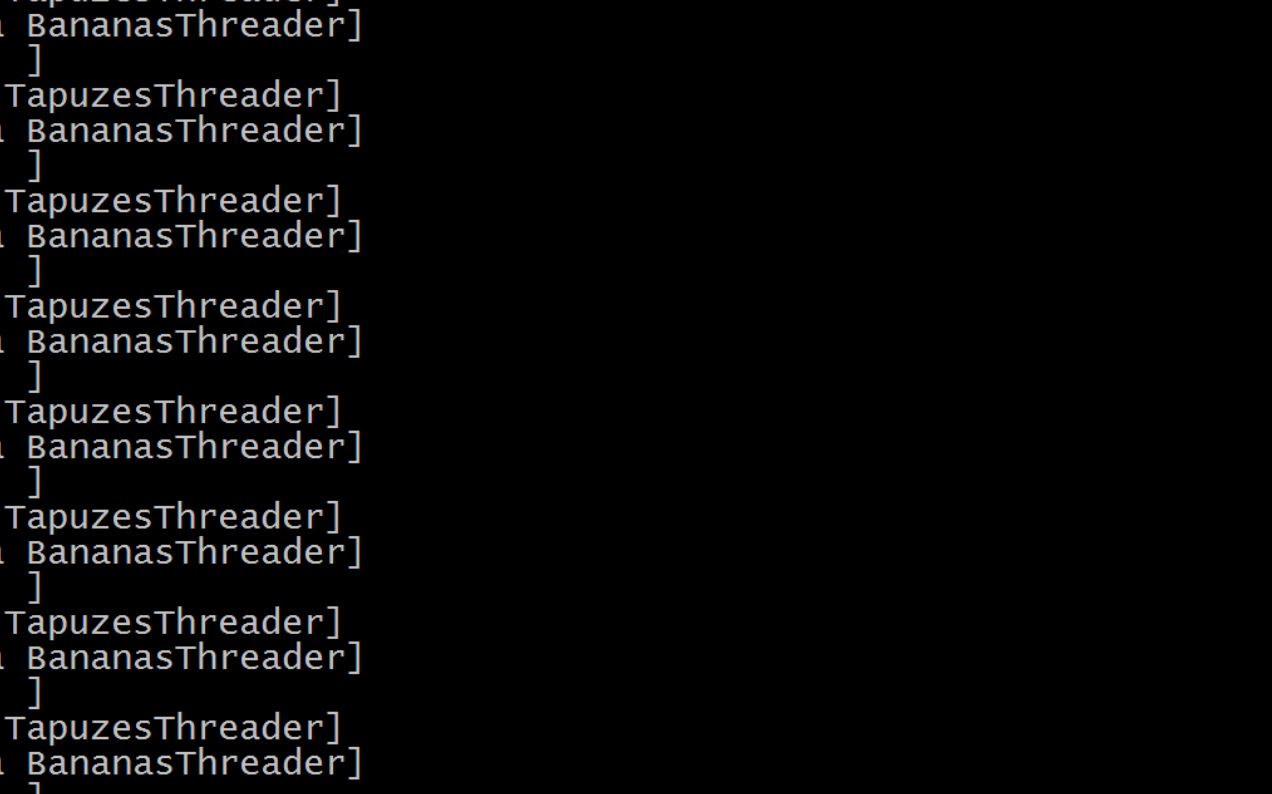
```
using System;
using System.Threading;

namespace abelski.csharp
{
    class CurrentThreadDemo
    {
        static void Main()
        {
            Thread t1 = new Thread(new ThreadStart(WriteTapuz));
            t1.Name = "TapuzesThreader";
            Thread t2 = new Thread(new ThreadStart(WriteBanana));
            t2.Name = "BananasThreader";
            t1.Start();
            t2.Start();
            for(int i=0; i<100; i++)
            {
                Console.WriteLine("\n[Carmel " + Thread.CurrentThread.Name + "]");
                Thread.Sleep(100);
            }
        }
    }
}
```

The Thread.CurrentThread Property

```
static void WriteBanana()  
{  
    for(int i=0; i<100; i++)  
    {  
        Console.WriteLine("\n[Banana " + Thread.CurrentThread.Name + "]);  
        Thread.Sleep(100);  
    }  
}  
  
static void WriteTapuz()  
{  
    for(int i=0; i<100; i++)  
    {  
        Console.WriteLine("\n[Tapuz " + Thread.CurrentThread.Name + "]);  
        Thread.Sleep(100);  
    }  
}  
}
```

The Thread.CurrentThread Property



A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Windows\system32\cmd.exe". The command prompt shows a repeating sequence of three lines: "[Carme]", "[Tapuz TapuzesThreader]", and "[Banana BananasThreader]". This sequence is repeated 10 times. At the bottom, the prompt "E:\CSHARP_SAMPLES>" is visible. The Windows taskbar at the bottom shows several open applications: "Desktop", "Gmail - Inbox (5) ...", "threads.odp - Op...", "E:\CSHARP_SAM...", and "C:\Windows\sys...". The system clock in the bottom right corner displays "06:13".

Passing Data

- When creating a new `Thread` object we pass over a `ThreadStart` delegate to the constructor.
- When the represented method has a parameter we can pass over an argument to that parameter when calling the `Start` method on our `Thread` object.
- The represented method can have one parameter only and it must be of type `object`. For that reason, in most cases there is a need to do casting.

Passing Data

- The Thread constructor is overloaded to allow accepting either of the following two delegates:

```
public delegate void ThreadStart();
```

```
public delegate void ParameterizedThreadStart(object ob);
```

Passing Data

```
using System;
using System.Threading;

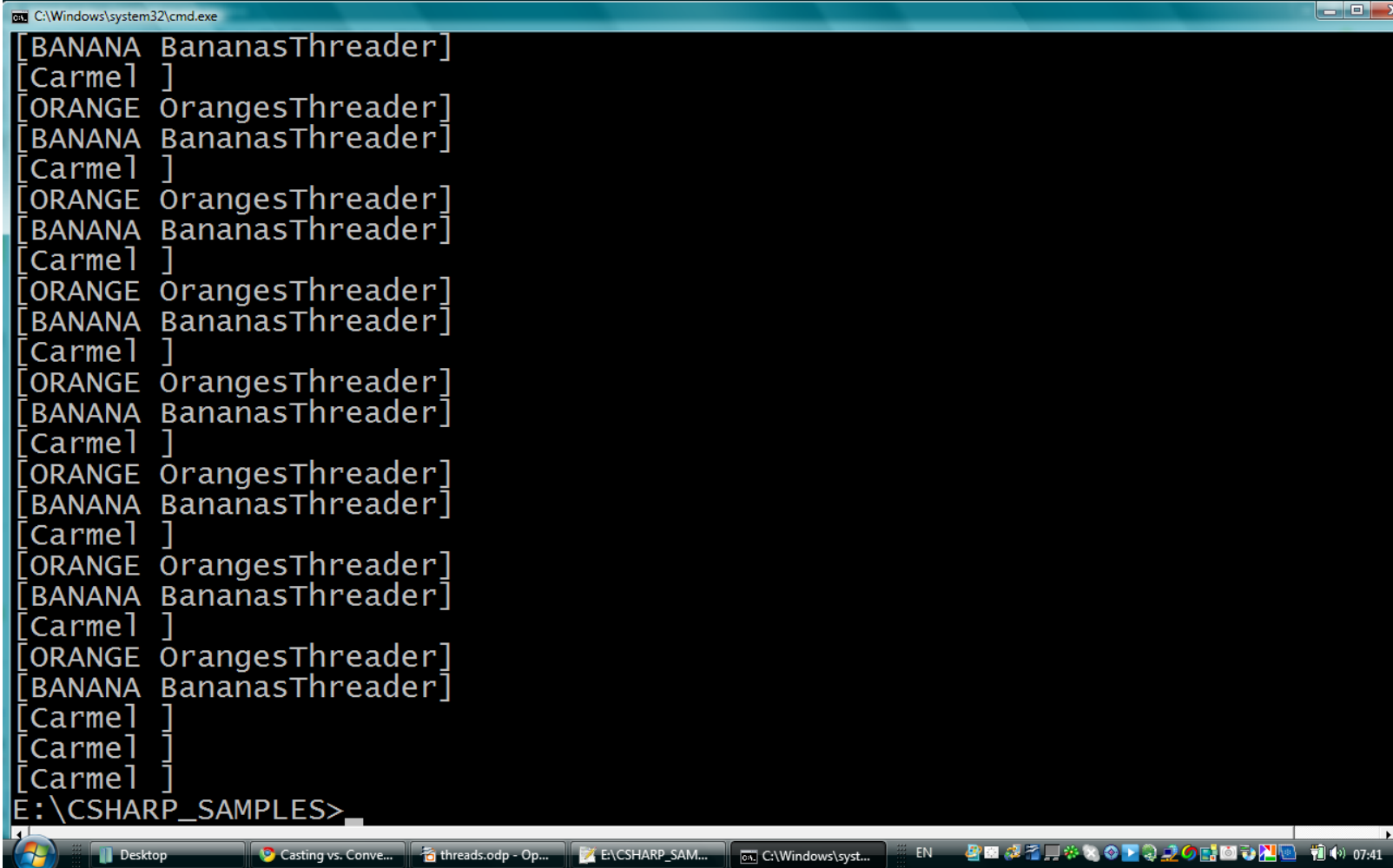
namespace abelski.csharp
{
    class PassingDataDemo
    {
        static void Main()
        {
            Thread t1 = new Thread(WriteOrange);
            t1.Name = "OrangesThreader";
            Thread t2 = new Thread(WriteBanana);
            t2.Name = "BananasThreader";
            t1.Start("ORANGE");
            t2.Start("BANANA");
            for(int i=0; i<10; i++)
            {
                Console.WriteLine("\n[Carmel " + Thread.CurrentThread.Name + "]");
                Thread.Sleep(100);
            }
        }
    }
}
```

Passing Data

```
static void WriteBanana(object ob)
{
    for(int i=0; i<8; i++)
    {
        Console.WriteLine("\n["+ob+" "+Thread.CurrentThread.Name+"]");
        Thread.Sleep(100);
    }
}

static void WriteOrange(object ob)
{
    for(int i=0; i<8; i++)
    {
        Console.WriteLine("\n["+ob+" "+Thread.CurrentThread.Name+"]");
        Thread.Sleep(100);
    }
}
}
```

Passing Data



```
C:\Windows\system32\cmd.exe
[BANANA BananasThreader]
[Carme] ]
[ORANGE OrangesThreader]
[BANANA BananasThreader]
[Carme] ]
[ORANGE OrangesThreader]
[BANANA BananasThreader]
[Carme] ]
[ORANGE OrangesThreader]
[BANANA BananasThreader]
[Carme] ]
[ORANGE OrangesThreader]
[BANANA BananasThreader]
[Carme] ]
[ORANGE OrangesThreader]
[BANANA BananasThreader]
[Carme] ]
[ORANGE OrangesThreader]
[BANANA BananasThreader]
[Carme] ]
[Carme] ]
[Carme] ]
E:\CSHARP_SAMPLES>
```

The screenshot shows a Windows command prompt window with a black background and white text. The title bar at the top reads 'C:\Windows\system32\cmd.exe'. The main area contains a series of log entries, each consisting of a thread name in brackets followed by the thread's name. The threads alternate between 'BANANA BananasThreader' and 'ORANGE OrangesThreader', with 'Carme' appearing as a separator. The sequence ends with three 'Carme' entries. The command prompt is currently at the 'E:\CSHARP_SAMPLES>' prompt. The taskbar at the bottom shows several open applications: 'Desktop', 'Casting vs. Conve...', 'threads.odp - Op...', 'E:\CSHARP_SAM...', and 'C:\Windows\syst...'. The system clock in the bottom right corner shows '07:41'.

Anonymous Method

- Alternatively for using the parameterless

`ParameterizedThreadStart` delegate is to pass over an anonymous method that includes the call to the method we want to execute in the separated thread.

- Doing so, the target method can accept any number of arguments and we don't need to do casting.

Anonymous Method

```
using System;
using System.Threading;

namespace abelski.csharp
{
    class PassingDataDemo
    {
        static void Main()
        {
            Thread t1 = new Thread(delegate() { WriteBanana("banana", 4); });
            Thread t2 = new Thread(delegate() { WriteOrange("orange", 3); });
            t1.Start();
            t2.Start();
        }

        static void WriteBanana(object ob, int num)
        {
            for(int i=0; i<num; i++)
            {
                Console.WriteLine("\n["+ob+" "+Thread.CurrentThread.Name+"]");
                Thread.Sleep(100);
            }
        }
    }
}
```

Anonymous Method

```
static void WriteOrange(object ob,int num)
{
    for(int i=0; i<num; i++)
    {
        Console.WriteLine("\n["+ob+" "+Thread.CurrentThread.Name+"]");
        Thread.Sleep(100);
    }
}
```


Data Sharing

- Each thread is assigned with its own private memory stack for local variables.
- Fields declared as static are shared between threads. Sharing data between threads using static variables is the recommended approach.
- Objects references are shared between threads as well.

Threads Pooling

- Creating a new thread consumes resources. There is a need to allocate a private local variables stack separately for each thread. In addition, each thread consumes memory required for its execution. Allocating that memory consumes resources as well.
- We can improve our application performance by using a Threads Pool. The easiest way is calling the `ThreadPool.QueueUserWorkItem` method each time a new thread is required.

Threads Pooling

- The target method must accept (at the minimum) a single object argument.
- Calling the `SetMinThreads` method we can set the number of idle threads the thread pool maintains when anticipating new requests for threads.

Threads Pooling

```
using System;
using System.Threading;

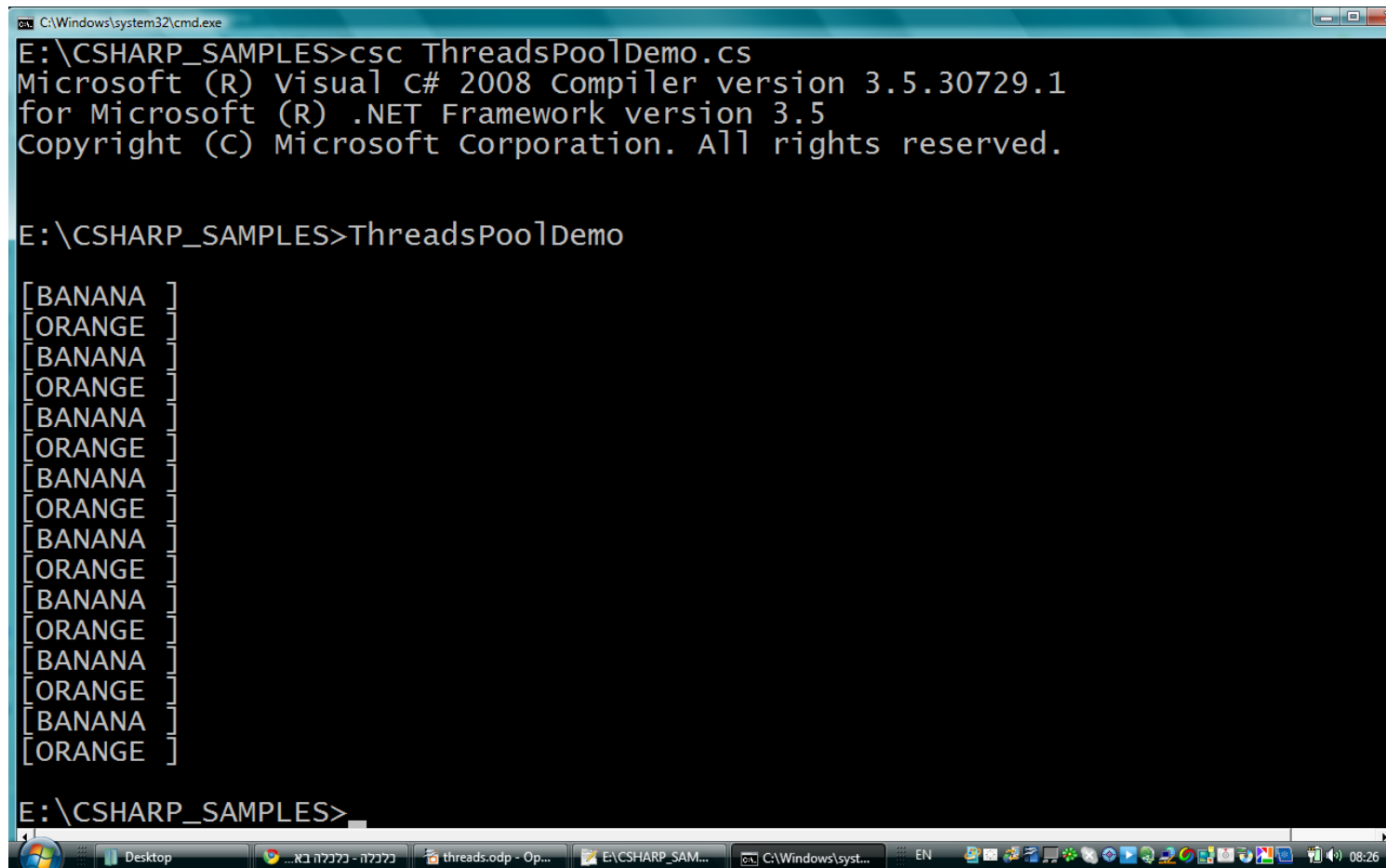
namespace abelski.csharp
{
    class ThreadPoolDemo
    {
        static void Main()
        {
            ThreadPool.QueueUserWorkItem(WriteBanana);
            ThreadPool.QueueUserWorkItem(WriteOrange);
            Console.ReadLine();
        }

        static void WriteBanana(object ob)
        {
            for(int i=0; i<8; i++)
            {
                Console.Write(
                    "\n[BANANA " + Thread.CurrentThread.Name + "]");
                Thread.Sleep(100);
            }
        }
    }
}
```

Threads Pooling

```
static void WriteOrange(object ob)
{
    for(int i=0; i<8; i++)
    {
        Console.Write(
            "\n[ORANGE "+Thread.CurrentThread.Name+"]");
        Thread.Sleep(100);
    }
}
```

Threads Pooling



```
C:\Windows\system32\cmd.exe
E:\CSHARP_SAMPLES>csc ThreadsPoolDemo.cs
Microsoft (R) Visual C# 2008 Compiler version 3.5.30729.1
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

E:\CSHARP_SAMPLES>ThreadsPoolDemo

[ BANANA ]
[ ORANGE ]
[ BANANA ]
[ ORANGE ]
[ BANANA ]
[ ORANGE ]
[ BANANA ]
[ ORANGE ]
[ BANANA ]
[ ORANGE ]
[ BANANA ]
[ ORANGE ]
[ BANANA ]
[ ORANGE ]
[ BANANA ]
[ ORANGE ]
E:\CSHARP_SAMPLES>
```

Foreground & Background Threads

- By default, each time we create explicitly a new thread it is a foreground thread. Working with pool of threads, each new thread we get is a background thread.
- The application is kept alive as long as there is a foreground thread that is still running. Once all foreground threads finish the application ends, and if there are any background threads that are still running they are abruptly terminated.

Foreground & Background Threads

- The main thread is (by default) a foreground thread.
- We can query or even change a thread's background status using its `IsBackground` property.

Foreground & Background Threads

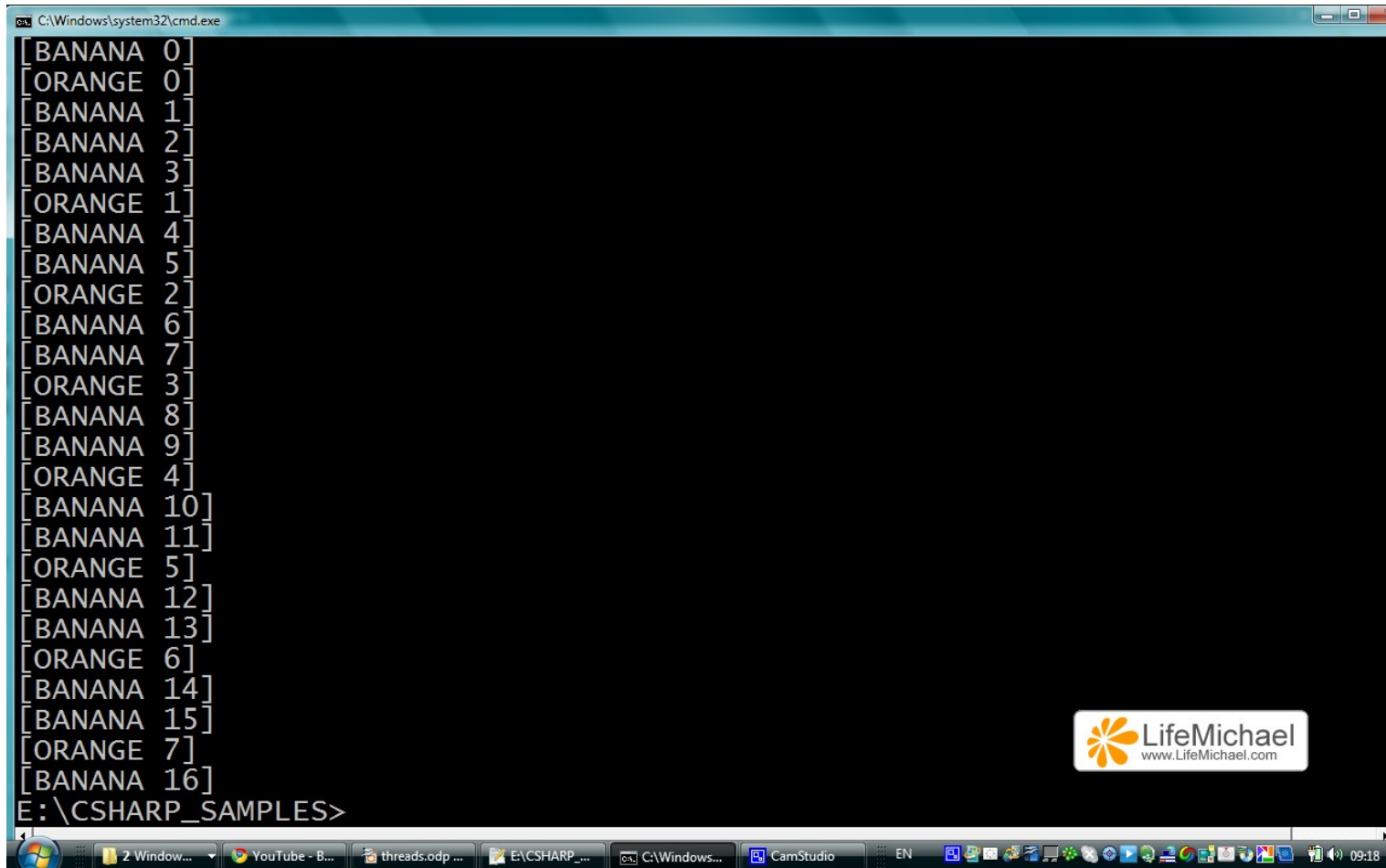
```
using System;
using System.Threading;

namespace abelski.csharp
{
    class ForegroundBackgroundThreadsDemo
    {
        static void Main()
        {
            Thread t1 = new Thread(WriteBanana);
            Thread t2 = new Thread(WriteOrange);
            t1.IsBackground = true;
            t2.IsBackground = false;
            t1.Start();
            t2.Start();
        }
    }
}
```

Foreground & Background Threads

```
static void WriteBanana()  
{  
    for(int i=0; i<800; i++)  
    {  
        Console.Write("\n[BANANA "+i+"]");  
        Thread.Sleep(2);  
    }  
}  
  
static void WriteOrange()  
{  
    for(int i=0; i<8; i++)  
    {  
        Console.Write("\n[ORANGE "+i+"]");  
        Thread.Sleep(10);  
    }  
}  
}
```

Foreground & Background Threads



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays a list of fruit names and numbers, alternating between "BANANA" and "ORANGE" with indices from 0 to 16. The text is as follows:

```
[BANANA 0]  
[ORANGE 0]  
[BANANA 1]  
[BANANA 2]  
[BANANA 3]  
[ORANGE 1]  
[BANANA 4]  
[BANANA 5]  
[ORANGE 2]  
[BANANA 6]  
[BANANA 7]  
[ORANGE 3]  
[BANANA 8]  
[BANANA 9]  
[ORANGE 4]  
[BANANA 10]  
[BANANA 11]  
[ORANGE 5]  
[BANANA 12]  
[BANANA 13]  
[ORANGE 6]  
[BANANA 14]  
[BANANA 15]  
[ORANGE 7]  
[BANANA 16]  
E:\CSHARP_SAMPLES>
```

The window also features a "LifeMichael" logo in the bottom right corner with the website "www.LifeMichael.com". The taskbar at the bottom shows several open applications, including "2 Window...", "YouTube - B...", "threads.odp ...", "E:\CSHARP_...", "C:\Windows...", and "CamStudio". The system clock in the bottom right corner indicates the time is 09:18.

Threads Priority

- The thread's `Priority` property determines the execution time it gets relatively to the other active threads within the same process.
- The value type of this property is of the following enum:

```
enum ThreadPriority {  
    Lowest,  
    BelowNormal,  
    Normal,  
    AboveNormal,  
    Highest}
```

Asynchronous Delegates

- Using asynchronous delegates we can get returned values back from a thread when it finishes its execution.

First Step

Define a delegate its signature matches the method we want to run concurrently with the main method.

Second Step

Instantiating the delegate with a specific method defined separately.

Third Step

Calling the `BeginInvoke` on our delegate and saving its `IAsyncResult` returned value. Calling `BeginInvoke` returns immediately allowing us to continue and perform other activities while the thread is working.

Asynchronous Delegates

Fourth Step

Calling `EndInvoke` on the delegate, passing over the saved `IAsyncResult` object, will block the current thread till the concurrently executed method completes. If the concurrently executed method has already completed then calling `EndInvoke` returns immediately. Calling `EndInvoke` returns the value the delegated method returns.

Asynchronous Delegates

- When calling the `BeginInvoke` function we can also pass over a callback delegate. Doing so, its represented method will be automatically called upon completion of the method that was started when we called the `BeginInvoke` function.
- The callback delegate should be for a method that has one argument of type `IAsyncResult`.

Asynchronous Delegates

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
namespace abelski.csharp
{
    delegate int Calculate(int numA, int numB);
    class AsynchronousAnotherDemoSimple
    {
        static void Main(string[] args)
        {
            Calculate ob = CalcSum;
            IAsyncResult asynchronousCall_5_14 = ob.BeginInvoke(5, 14, null, null);
            IAsyncResult asynchronousCall_9_24 = ob.BeginInvoke(9, 24, null, null);
            IAsyncResult asynchronousCall_3_8 = ob.BeginInvoke(3, 8, null, null);
        }
    }
}
```


Asynchronous Delegates

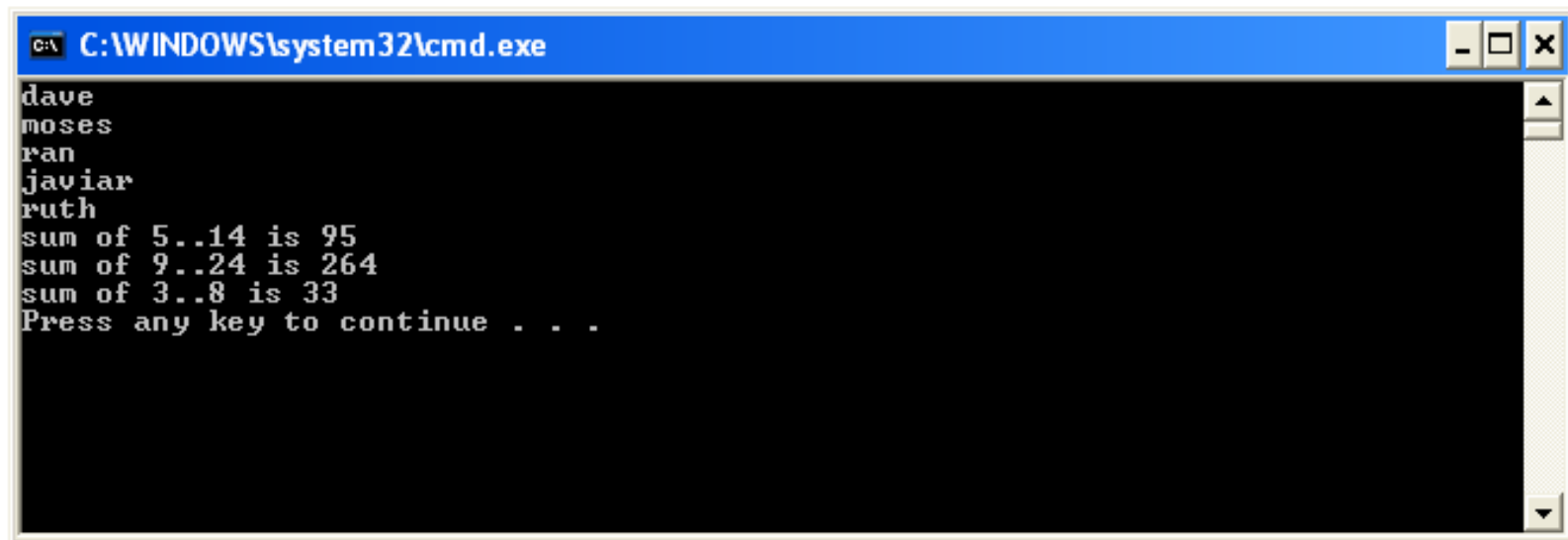
```
//int sum = CalcSum(5, 14);
string[] vec = { "dave", "moses", "ran", "javiar", "ruth" };
for (int i = 0; i < vec.Length; i++)
{
    Console.WriteLine(vec[i]);
    Thread.Sleep(1800);
}

Console.WriteLine("sum of 5..14 is "+ob.EndInvoke(asynchronousCall_5_14));
Console.WriteLine("sum of 9..24 is "+ob.EndInvoke(asynchronousCall_9_24));
Console.WriteLine("sum of 3..8 is "+ob.EndInvoke(asynchronousCall_3_8));
}
```

Asynchronous Delegates

```
static int CalcSum(int numberA, int numberB)
{
    int total = 0;
    for (int i = numberA; i <= numberB; i++)
    {
        total += i;
        Thread.Sleep(1000);
    }
    return total;
}
}
```

Asynchronous Delegates



```
C:\WINDOWS\system32\cmd.exe
dave
moses
ran
javiar
ruth
sum of 5..14 is 95
sum of 9..24 is 264
sum of 3..8 is 33
Press any key to continue . . .
```

Asynchronous Delegates

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
namespace abelski.csharp
{
    delegate int Calculate(int numA, int numB);
    class AsynchronousAnotherDemoLessSimple
    {
        static void Main(string[] args)
        {
            Calculate ob = CalcSum;
            //Calculate ob = new Calculate(AsynchronousAnotherDemo.CalcSum);
            IAsyncResult asynchronousCall_5_14 = ob.BeginInvoke(5, 14,
                PrintResult, ob);
            IAsyncResult asynchronousCall_9_24 = ob.BeginInvoke(9, 24,
                PrintResult, ob);
            IAsyncResult asynchronousCall_3_8 = ob.BeginInvoke(3, 8,
                PrintResult, ob);
        }
    }
}
```

Asynchronous Delegates

```
//int sum = CalcSum(5, 14);  
string[] vec = { "dave", "moses", "ran", "javiar", "ruth" };  
for (int i = 0; i < vec.Length; i++)  
{  
    Console.WriteLine(vec[i]);  
    Thread.Sleep(1800);  
}  
  
Console.ReadLine();  
}
```

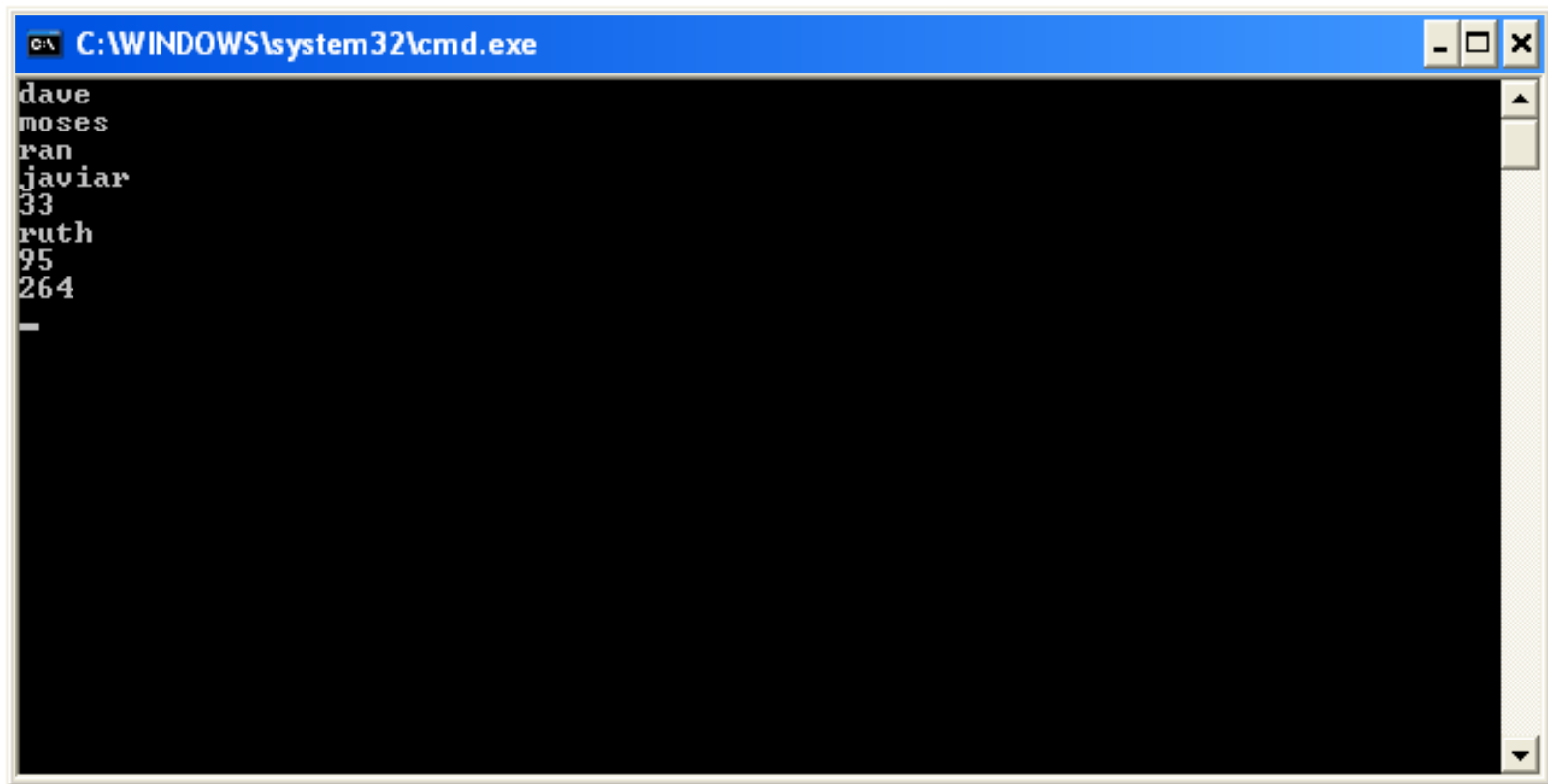
Asynchronous Delegates

```
static int CalcSum(int numberA, int numberB)
{
    int total = 0;
    for (int i = numberA; i <= numberB; i++)
    {
        total += i;
        Thread.Sleep(1000);
    }
    return total;
}

static void PrintResult(IAsyncResult param)
{
    Calculate method = (Calculate)param.AsyncState;
    Console.WriteLine(method.EndInvoke(param));
}

}
```

Asynchronous Delegates



```
C:\WINDOWS\system32\cmd.exe
dave
moses
ran
javiar
33
ruth
95
264
-
```

Synchronization

- When threads access the same data it is highly important to synchronize them.
- There are different techniques for synchronizing between threads.

Blocked Thread

- When the execution of a specific thread is paused (for some reason.. as when waiting for another thread to end when calling `Join` or `EndInvoke...`) that thread is deemed blocked.
- A blocked thread consumes very low resources. The CLR is aware of the blocked thread and wake it up when the blocking condition is satisfied.
- We can check whether a specific thread is blocked by accessing its `ThreadState` property.

The ThreadState Property

- The `ThreadState` is a flags enum, that combines different 'layers' of data in a bitwise fashion.
- Once a thread is created and till it is terminated, it is in at least one of the possible states, the `ThreadState` enum describes.
<http://msdn.microsoft.com/en-us/library/system.threading.threadstate.aspx>
- It is useful to use the `ThreadState` property for diagnostic purposes. It is unwise to use it for synchronization. The thread state may change in between testing the `ThreadState` and the operation been taken.

The ThreadState Property

```
using System;
using System.Threading;

class ThreadStateDemo
{
    public static void Countdown()
    {
        for (int i = 100; i > 0; i--)
        {
            Console.WriteLine("\n"+i);
        }
    }

    public static void PrintThreadState(Thread thread )
    {
        Console.WriteLine("\nCurrent Thread State is ");
        if ((thread.ThreadState & ThreadState.Aborted) == ThreadState.Aborted)
            Console.WriteLine("Aborted ");
        if ((thread.ThreadState & ThreadState.AbortRequested) ==
            ThreadState.AbortRequested)
            Console.WriteLine("AbortRequested ");
        if ((thread.ThreadState & ThreadState.Background) ==
            ThreadState.Background)
            Console.WriteLine("Background ");
    }
}
```

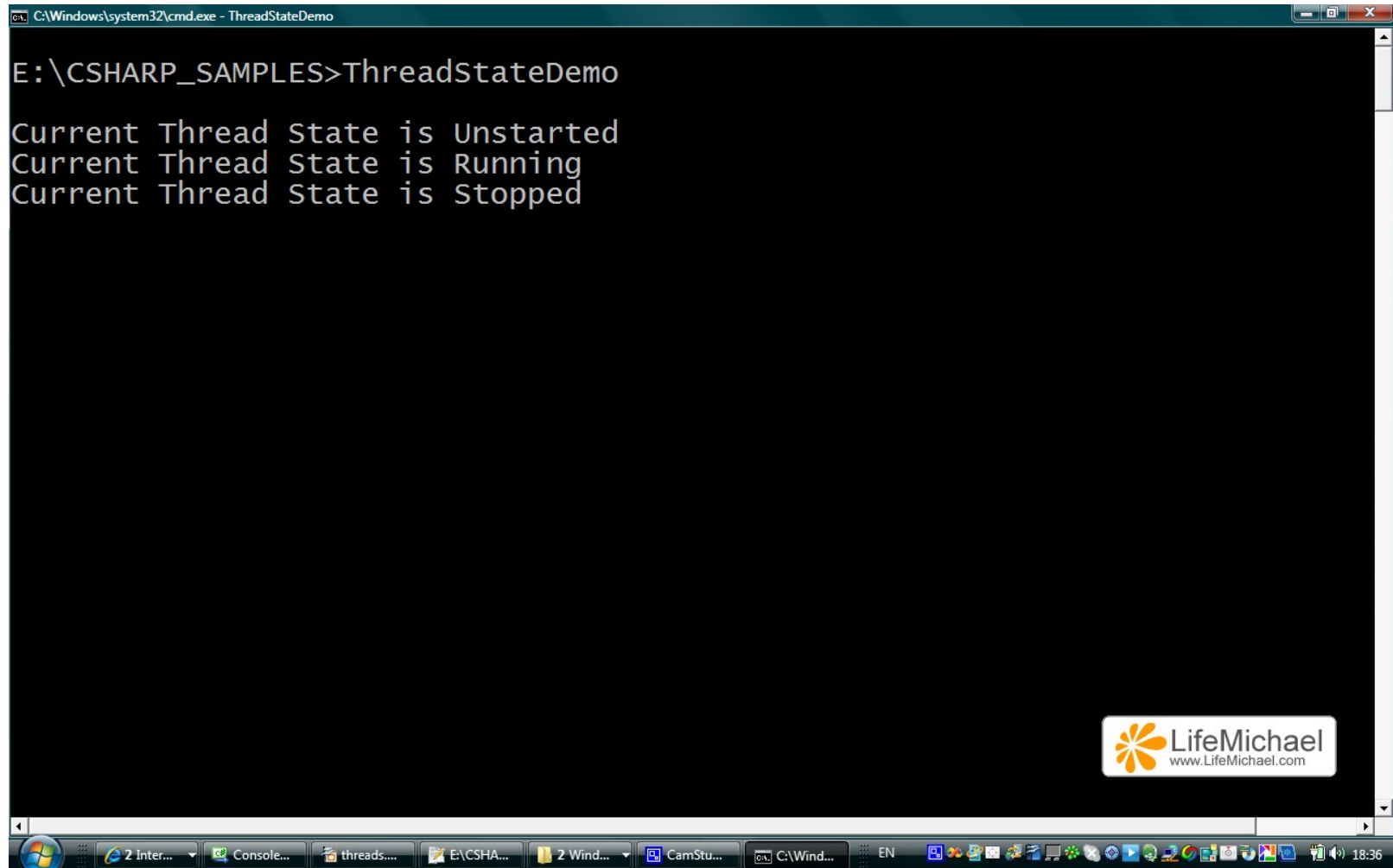
The ThreadState Property

```
if ((thread.ThreadState &
    (ThreadState.Stopped | ThreadState.Unstarted | ThreadState.Aborted))
    == 0)
    Console.Write("Running ");
if ((thread.ThreadState & ThreadState.Stopped) == ThreadState.Stopped)
    Console.Write("Stopped ");
if ((thread.ThreadState & ThreadState.StopRequested) ==
    ThreadState.StopRequested)
    Console.Write("StopRequested ");
if ((thread.ThreadState & ThreadState.Suspended) ==
    ThreadState.Suspended)
    Console.Write("Suspended ");
if ((thread.ThreadState & ThreadState.SuspendRequested) ==
    ThreadState.SuspendRequested)
    Console.Write("SuspendRequested ");
if ((thread.ThreadState & ThreadState.Unstarted) ==
    ThreadState.Unstarted)
    Console.Write("Unstarted ");
if ((thread.ThreadState & ThreadState.WaitSleepJoin) ==
    ThreadState.WaitSleepJoin)
    Console.Write("WaitSleepJoin ");
}
```

The ThreadState Property

```
public static void Main()
{
    Thread t1 = new Thread(Countdown);
    PrintThreadState(t1);
    t1.Start();
    PrintThreadState(t1);
    t1.Abort();
    PrintThreadState(t1);
    Console.Read();
}
```

The ThreadState Property



```
C:\Windows\system32\cmd.exe - ThreadStateDemo

E:\CSHARP_SAMPLES>ThreadStateDemo

Current Thread State is Unstarted
Current Thread State is Running
Current Thread State is Stopped
```

The `lock` Construct

- The `lock` construct ensures that only one thread can enter a particular section of code.
- Using the `lock` construct we should specify a specific object the `lock` refers and place the code within brackets.

```
static object ob = new object();  
static void DoSomething()  
{  
    lock(ob)  
    {  
        ...  
    }  
}
```

The lock Construct

- When more than one thread contends the lock they are queued on a FIFO based queue.
- When a thread is blocked while awaiting a contended lock its ThreadState is WaitSleepJoin.

The lock Construct

- Using the lock construct is in fact a shortcut for calling the `Monitor.Enter()` and `Monitor.Exit()` methods.

```
static void DoSomething()  
{  
    Monitor.Enter(ob);  
    try  
    {  
        ...  
    }  
    finally {Monitor.Exit(ob);}  
}
```

The `lock` Construct

- Calling `Monitor.TryEnter` we can specify a timeout so that if the lock isn't obtained within the specified timeout limit then it returns `false`.
- The synchronization object can be any object as long as it is a reference type.
It is recommended to have that object privately scoped in order to prevent others from interacting with it.
- When the `lock` construct isn't in use, the synchronized object can be accessed without any limitation.

The lock Construct

- We can have nested lock statements. Each one of them can refer another object.

Impoverished Concurrency

- When too much code is placed within lock statements we might get an impoverished concurrency.

Dead Lock

- When having two threads while each one of them is waiting for a lock held by the other so neither can proceed we shall get a dead lock.

Lock Race

- When having more than one thread racing for obtaining a lock and the wrong thread obtains it first we get a problem known as 'Lock Race'.

Mutex

- The `Mutex` class provides a functionality similar to the one we get when using the `Lock` construct.
- Unlike `Lock`, when using the `Mutex` class we can use it across multiple processes.
- Acquiring and releasing a `Mutex` is a bit slower comparing with `Lock`.

Mutex

- We acquire a `Mutex` by calling the `WaitOne` method. We release it by calling the `ReleaseMutex` method.
- The next code sample shows how to use `Mutex` in order to ensure that only one instance of the program can run at any given time.
- If an application terminates without calling the `ReleaseMutex` method, the CLR releases `Mutex` automatically.

Mutex

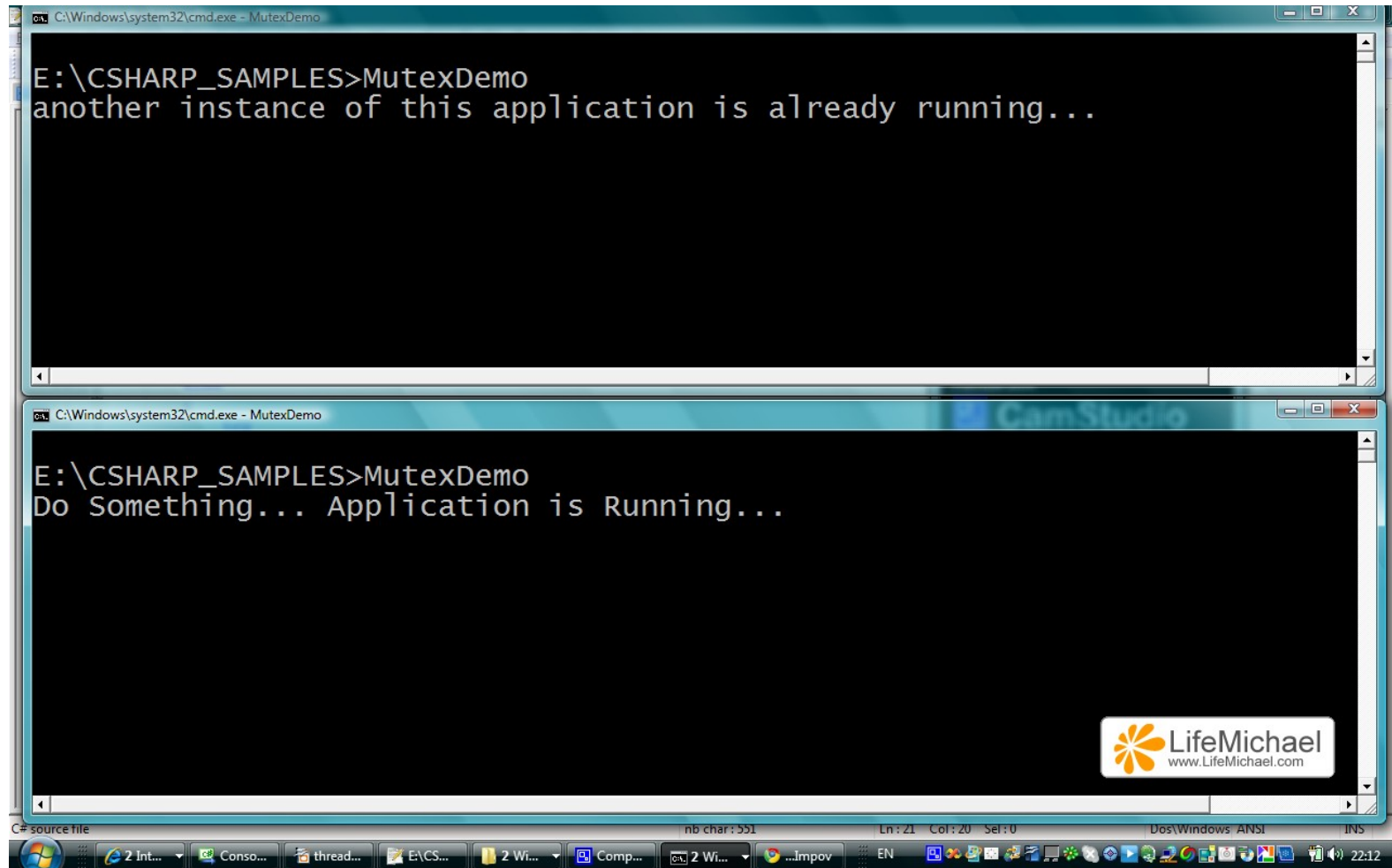
```
using System;
using System.Threading;

class MutexDemo
{
    static Mutex mutex = new Mutex(false, "demo");
    public static void Main()
    {
        if(!mutex.WaitOne(2000))
        {
            Console.WriteLine(
                "another instance of this application is already running...");
            Console.ReadLine();
        }
    }
}
```

Mutex

```
else
{
    try
    {
        //run the application
        Run();
    }
    finally
    {
        mutex.ReleaseMutex();
    }
}
}
public static void Run()
{
    Console.WriteLine("Do Something... Application is Running...");
    Console.ReadLine();
}
}
```

Mutex



```
C:\Windows\system32\cmd.exe - MutexDemo
E:\CSHARP_SAMPLES>MutexDemo
another instance of this application is already running...

C:\Windows\system32\cmd.exe - MutexDemo
E:\CSHARP_SAMPLES>MutexDemo
Do Something... Application is Running...
```

LifeMichael
www.LifeMichael.com

Semaphore

- Instantiating Semaphore, we get an object that can assist us synchronizing threads.
- The Semaphore object functions similarly to a restaurant. Once created we specify the maximum number of people that can enter the restaurant. An additional queue is created to hold those people that couldn't enter to the restaurant because it was full.
- Each time a person leaves the restaurant, the one on top of the waiting queue is entered.

Semaphore

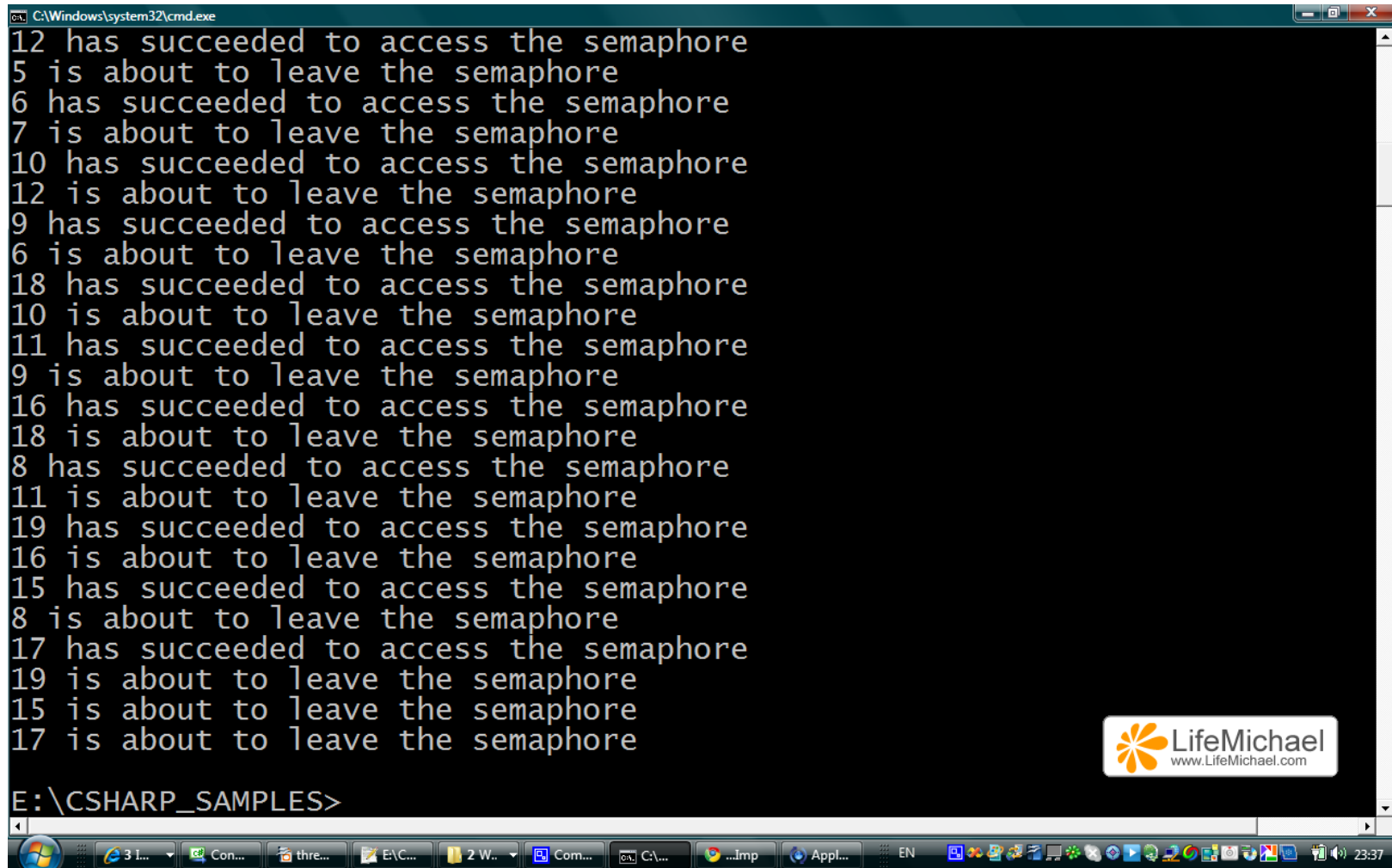
- Semaphore is useful when trying to ensure a maximum number of threads being capable of executing a specific code at the same time.
- As with Mutex, a Semaphore can span over separated processes. The only requirement is having the Semaphore named, just as with Mutex.

Semaphore

```
using System;
using System.Threading;

class SemaphoreDemo
{
    static Semaphore semaphore = new Semaphore(3,3);
    public static void Main()
    {
        for(int i=0; i<20; i++)
        {
            new Thread(SemaphoreDemo.DoSomething).Start(i);
        }
    }
    static void DoSomething(object id)
    {
        Console.WriteLine(id+" wants to access the semaphore");
        semaphore.WaitOne();
        Console.WriteLine(id+"
            has succeeded to access the semaphore");
        Thread.Sleep(3000);
        Console.WriteLine(id+" is about to leave the semaphore");
        semaphore.Release();
    }
}
```

Semaphore



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays a series of messages indicating semaphore operations. The messages are as follows:

```
12 has succeeded to access the semaphore
5 is about to leave the semaphore
6 has succeeded to access the semaphore
7 is about to leave the semaphore
10 has succeeded to access the semaphore
12 is about to leave the semaphore
9 has succeeded to access the semaphore
6 is about to leave the semaphore
18 has succeeded to access the semaphore
10 is about to leave the semaphore
11 has succeeded to access the semaphore
9 is about to leave the semaphore
16 has succeeded to access the semaphore
18 is about to leave the semaphore
8 has succeeded to access the semaphore
11 is about to leave the semaphore
19 has succeeded to access the semaphore
16 is about to leave the semaphore
15 has succeeded to access the semaphore
8 is about to leave the semaphore
17 has succeeded to access the semaphore
19 is about to leave the semaphore
15 is about to leave the semaphore
17 is about to leave the semaphore
```

The prompt is currently at "E:\CSHARP_SAMPLES>". In the bottom right corner of the command prompt window, there is a logo for "LifeMichael" with the website "www.LifeMichael.com". The Windows taskbar is visible at the bottom of the screen, showing various open applications and the system clock at 23:37.

Semaphore

- When instantiating `Semaphore` we pass over two numbers. The second number is the total number of permissions the new `Semaphore` object will hold. The first number is the initial number of permissions that will be available for threads that ask for permission by calling the `WaitOne()` method on the `Semaphore` object.
- The difference between the two is the number of permissions that are handed over to the thread through which `Semaphore` is instantiated.

Semaphore

- In the coming code sample, 3 out of the 5 available permissions are handed over to the main thread. Only once the `Release()` method is called (through the execution of the 'main' thread) on our `Semaphore` object permissions are returned back to the `Semaphore` object.

Semaphore

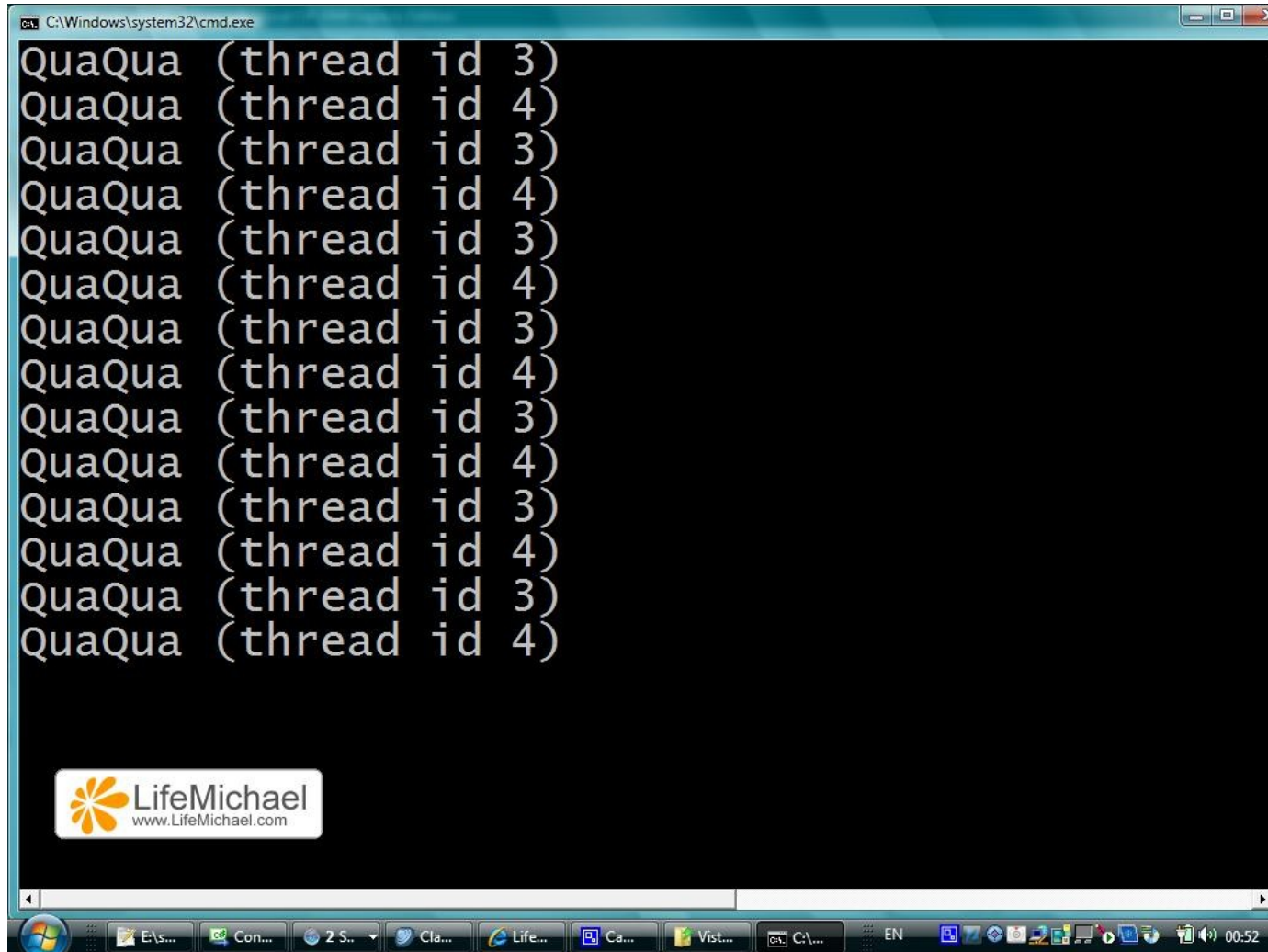
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace Com.Abeliski.Samples
{
    class Program
    {
        static Semaphore mainSemaphore;
        static void Main(string[] args)
        {
            Thread t1 = new Thread(QuaQua);
            Thread t2 = new Thread(QuaQua);
            Thread t3 = new Thread(QuaQua);
            Thread t4 = new Thread(QuaQua);
            Thread t5 = new Thread(QuaQua);
            Thread t6 = new Thread(QuaQua);
            mainSemaphore = new Semaphore(2, 5);
        }
    }
}
```

Semaphore

```
t1.Start();
t2.Start();
t3.Start();
t4.Start();
t5.Start();
t6.Start();
Thread.Sleep(10000);
mainSemaphore.Release();
mainSemaphore.Release();
mainSemaphore.Release();
}
static void QuaQua()
{
    mainSemaphore.WaitOne();
    for (int i = 0; i <= 30; i++)
    {
        Thread.Sleep(1000);
        Console.WriteLine("QuaQua (thread id " +
            Thread.CurrentThread.GetHashCode()+") ");
    }
    mainSemaphore.Release();
}
}
```

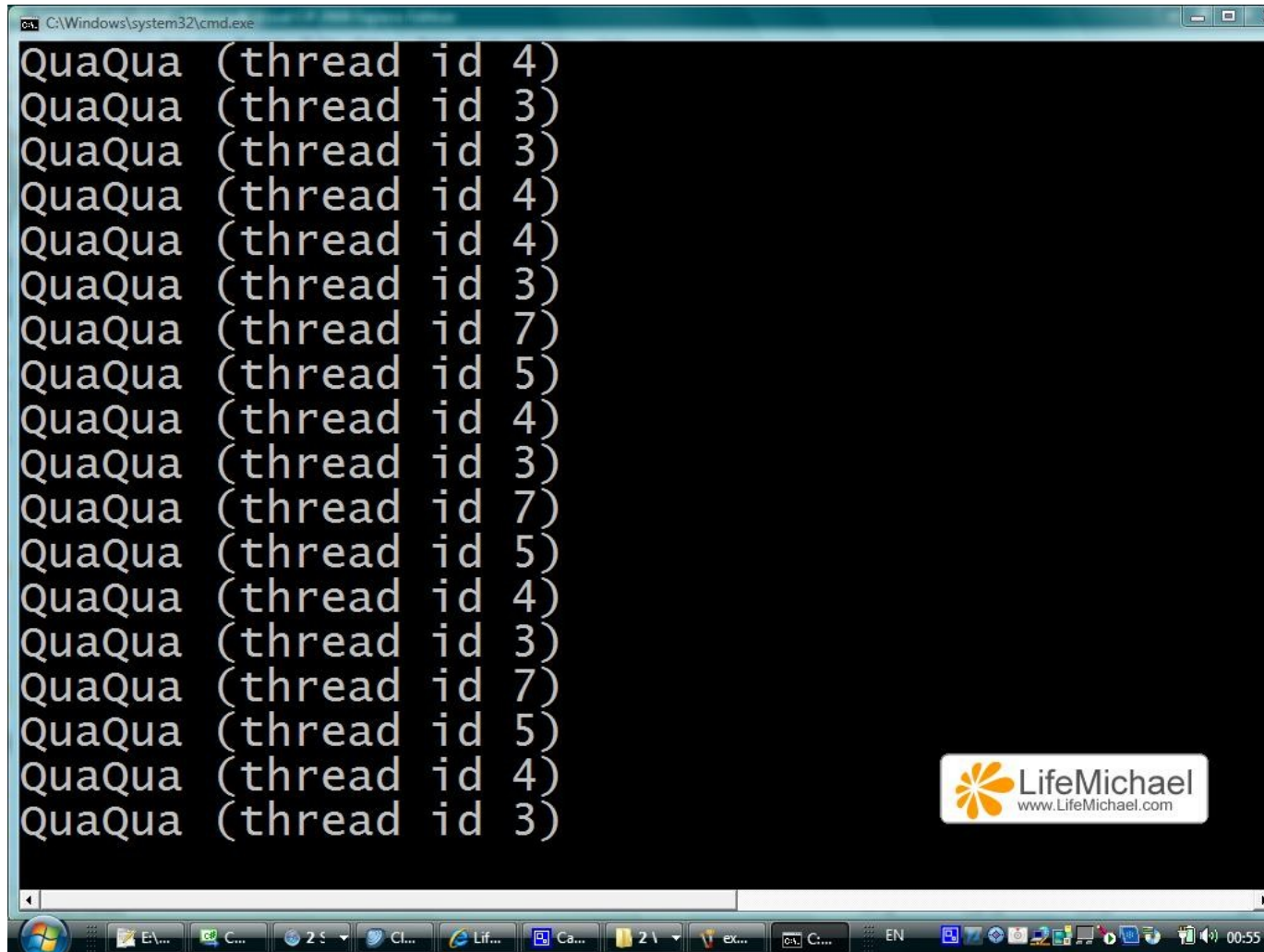
Semaphore



```
C:\Windows\system32\cmd.exe
QuaQua (thread id 3)
QuaQua (thread id 4)
QuaQua (thread id 3)
QuaQua (thread id 4)
QuaQua (thread id 3)
QuaQua (thread id 4)
QuaQua (thread id 3)
QuaQua (thread id 4)
QuaQua (thread id 3)
QuaQua (thread id 4)
QuaQua (thread id 3)
QuaQua (thread id 4)
QuaQua (thread id 3)
QuaQua (thread id 4)

LifeMichael
www.LifeMichael.com
```

Semaphore



```
C:\Windows\system32\cmd.exe
QuaQua (thread id 4)
QuaQua (thread id 3)
QuaQua (thread id 3)
QuaQua (thread id 4)
QuaQua (thread id 4)
QuaQua (thread id 3)
QuaQua (thread id 7)
QuaQua (thread id 5)
QuaQua (thread id 4)
QuaQua (thread id 3)
QuaQua (thread id 7)
QuaQua (thread id 5)
QuaQua (thread id 4)
QuaQua (thread id 3)
QuaQua (thread id 7)
QuaQua (thread id 5)
QuaQua (thread id 4)
QuaQua (thread id 3)
```

Threads Safety

- We consider an application or a method as 'thread safe' if it doesn't have any indeterminacy in multi threading scenarios.
- In most cases, we can turn our application and/or method into a 'thread safe' one by using locking mechanism. Another alternative approach involves with minimizing the shared data.

Threads Safety

- Turning an application or a method into a 'thread safe' one might have a significant development burden. In addition, it can damage the performance.

Static Members

- When dealing with a specific type's static members, externally locking every access to these members can be performed by locking on the `Type` object that represents the type we are dealing with.

```
...  
lock (typeof (SomeClass) )  
{  
    ...  
}  
...
```


Static Members

- Each and every code segment that tries to access a lock block that refers the same type will fail.
- Nevertheless, it is important to understand that code segments that try to access the static member without doing it from within a lock block won't have any difficulty doing so.

Static Members

- Unlike other similar programming languages C# doesn't allow to access static members through an object reference.
Therefore, once we lock on the type we guarantee that no other code that tries to access that very same static member will succeed.
- By default, all static members throughout the .NET framework are thread-safe.

Static Members

```
using System;
using System.Threading;
namespace abelski.samples
{
    public class Program
    {
        public static int num;
        public static void Main()
        {
            new Thread(DoGoGo).Start();
            new Thread(DoBoBo).Start();
        }
        public static void DoGoGo()
        {
            lock (typeof(Program))
            {
                for (int i = 0; i < 20; i++)
                {
                    Thread.Sleep(1000);
                    Console.WriteLine("DoGoGo bing "+Program.num);
                }
            }
        }
    }
}
```

Static Members

```
public static void DoBoBo()  
{  
    for (int i = 0; i < 5; i++)  
    {  
        lock (typeof(Program))  
        {  
            Thread.Sleep(1000);  
        }  
        Program.num++;  
        Console.WriteLine("DoBoBo bing "+Program.num);  
    }  
}  
}
```

Static Members

[illegible]

Atomic Statements

- A statement is considered as an atomic statement if it executes as a single indivisible instruction on the underlying processor.

Simple reads and simple assignments on fields their type is of 32 bits (or smaller) are considered as atomic statements.

Atomic Statements

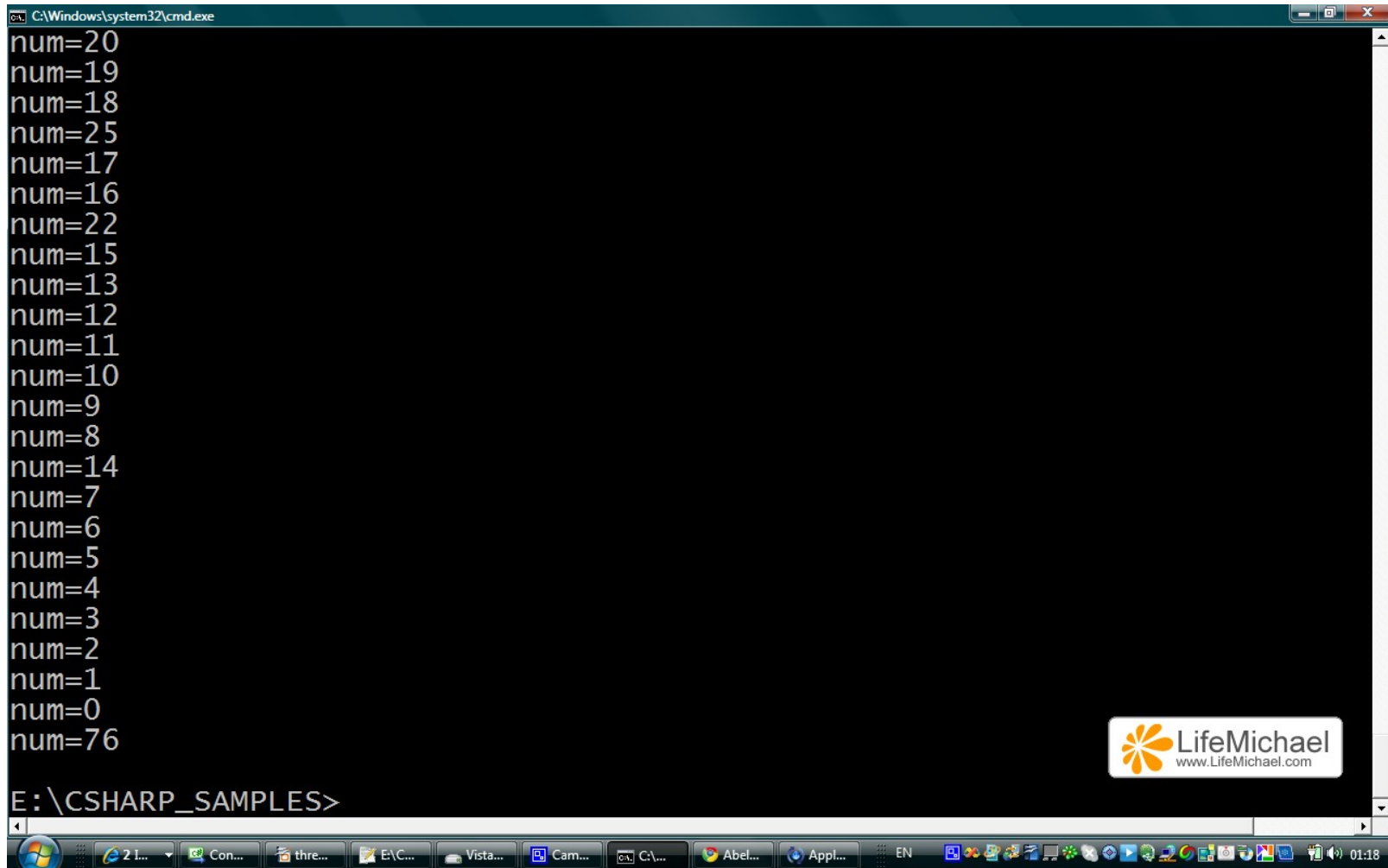
```
using System;
using System.Threading;

namespace abelski.csharp
{
    class AtomicityDemo
    {
        public static int num = 100;
        public static void Main()
        {
            for(int i=0; i<10; i++)
            {
                new Thread(AtomicityDemo.DoSomething).Start();
            }
        }
    }
}
```

Atomic Statements

```
static void DoSomething()  
{  
    for(int i=0; i<10; i++)  
    {  
        num=num-1;  
        Console.WriteLine("num="+num);  
    }  
}  
}
```


Atomic Statements



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background with white text. It displays a list of 20 atomic statements for a variable named 'num', starting with 'num=20' and ending with 'num=76'. The statements are: num=20, num=19, num=18, num=25, num=17, num=16, num=22, num=15, num=13, num=12, num=11, num=10, num=9, num=8, num=14, num=7, num=6, num=5, num=4, num=3, num=2, num=1, num=0, and num=76. The prompt 'E:\CSHARP_SAMPLES>' is visible at the bottom left. A watermark for 'LifeMichael' with the website 'www.LifeMichael.com' is located in the bottom right corner of the window. The Windows taskbar is visible at the bottom of the screen, showing various application icons and the system clock displaying '01:18'.

```
C:\Windows\system32\cmd.exe
num=20
num=19
num=18
num=25
num=17
num=16
num=22
num=15
num=13
num=12
num=11
num=10
num=9
num=8
num=14
num=7
num=6
num=5
num=4
num=3
num=2
num=1
num=0
num=76
E:\CSHARP_SAMPLES>
```

The 'Interlocked' Class

- The `Interlocked` class provides a simple and an easier way for locking simple statements that include separated atomic operations.
- Calling the `Interlocked.Decrement` method we can decrement the value of a variable in an atomic operation.

The 'Interlocked' Class

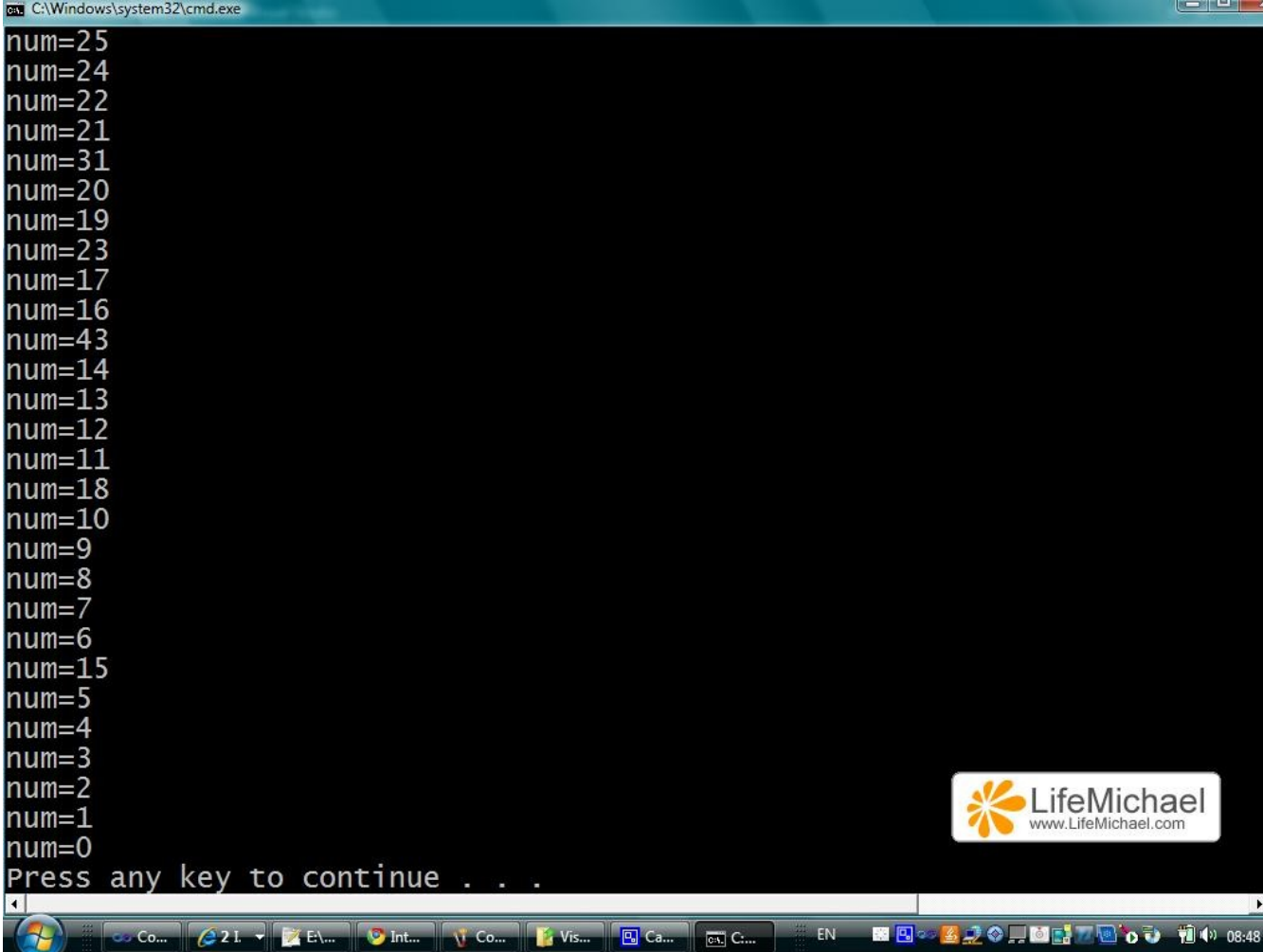
```
using System;
using System.Threading;

namespace abelski.csharp
{
    class AtomicityDemo
    {
        public static int num = 100;
        public static void Main()
        {
            for (int i = 0; i < 10; i++)
            {
                new Thread(AtomicityDemo.DoSomething).Start();
            }
        }
    }
}
```

The 'Interlocked' Class

```
public static void DoSomething()
{
    for (int i = 0; i < 10; i++)
    {
        Interlocked.Decrement(ref num);
        //num = num - 1;
        Console.WriteLine("num=" + num);
    }
}
```

The 'Interlocked' Class



```
C:\Windows\system32\cmd.exe
num=25
num=24
num=22
num=21
num=31
num=20
num=19
num=23
num=17
num=16
num=43
num=14
num=13
num=12
num=11
num=18
num=10
num=9
num=8
num=7
num=6
num=15
num=5
num=4
num=3
num=2
num=1
num=0
Press any key to continue . . .
```

LifeMichael
www.LifeMichael.com

The 'Interlocked' Class

- The following code sample shows how to use the `Interlocked.Add` method in order to add value in an atomic operation.

The 'Interlocked' Class

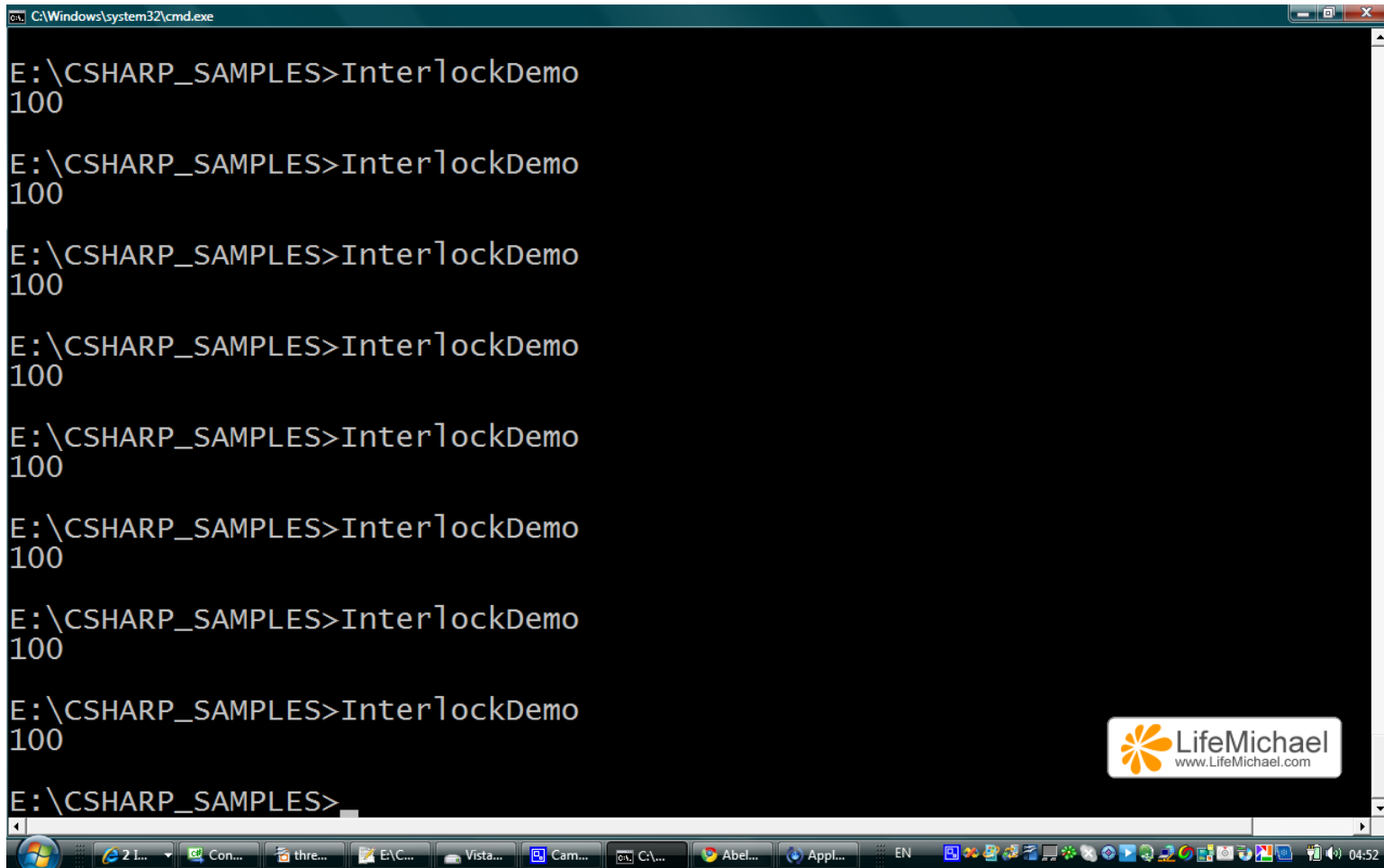
```
using System;
using System.Threading;

namespace abelski.csharp
{
    class InterlockDemo
    {
        public static int num = 0;
        public static void Main()
        {
            for(int i=0; i<10; i++)
            {
                new Thread(InterlockDemo.DoSomething).Start();
            }
            Thread.Sleep(2000);
            Console.WriteLine(num);
        }
    }
}
```

The 'Interlocked' Class

```
static void DoSomething()
{
    for(int i=0; i<10; i++)
    {
        Interlocked.Add(ref num,1);
        //num = num + 1;
    }
}
}
```


The 'Interlocked' Class



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window shows the execution of a program named "InterlockDemo" from the directory "E:\CSHARP_SAMPLES". The program is run multiple times, and each time it outputs the number "100". The command prompt shows the following sequence of commands and outputs:

```
E:\CSHARP_SAMPLES>InterlockDemo
100

E:\CSHARP_SAMPLES>InterlockDemo
100

E:\CSHARP_SAMPLES>InterlockDemo
100

E:\CSHARP_SAMPLES>InterlockDemo
100

E:\CSHARP_SAMPLES>InterlockDemo
100

E:\CSHARP_SAMPLES>InterlockDemo
100

E:\CSHARP_SAMPLES>InterlockDemo
100

E:\CSHARP_SAMPLES>InterlockDemo
100

E:\CSHARP_SAMPLES>
```

The Windows taskbar is visible at the bottom, showing various application icons and the system clock displaying "04:52". A watermark for "LifeMichael" with the website "www.LifeMichael.com" is visible in the bottom right corner of the command prompt window.

The 'volatile' Modifier

- When executing our program on a multi processors machine, we might get into unexpected results due to having our variables values hosted within CPU registers.

The platform might choose to host our variables values within CPU registers in order to improve the performance.

- When a variable value is hosted within a CPU register, there might be a delay when we try to change that variable value.

The delay might take place till both the value hosted within the CPU register and the value hosted within the memory are updated.

The 'volatile' Modifier

- We can add the 'volatile' modifier when declaring a field.
- Using the 'volatile' modifier ensures that one thread retrieves the most up-to-date value written by another thread.

Timers

- When there is a need to execute a specific method repeatedly, at regular intervals, concurrently with the other threads our application includes, the simplest way would be to use a Timer.
- The .NET framework provides four timers.

`System.Threading.Timer`

`System.Timers.Timer`

`System.Windows.Forms.Timer`

`System.Windows.Threading.DispatcherTimer`

Timers

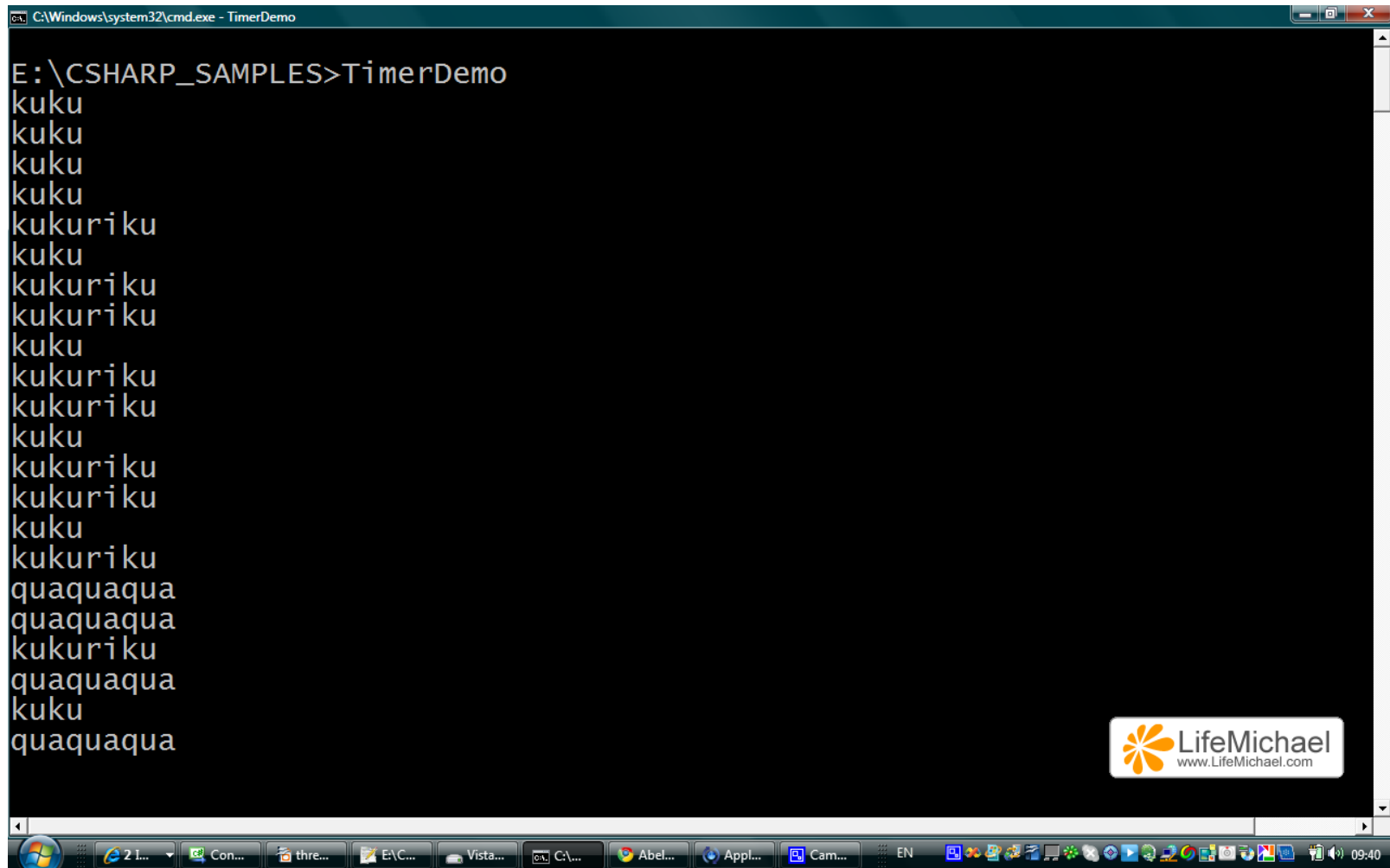
```
using System;
using System.Threading;

namespace abelski.csharp
{
    class TimerDemo
    {
        public static void Main()
        {
            Timer timerKuKu =
                new Timer(SayHello, "kuku", 2000, 800);
            Timer timerKuKuRiku =
                new Timer(SayHello, "kukuriku", 5000, 400);
            Timer timerQuaQua =
                new Timer(SayHello, "quaquaqua", 8000, 200);
            Console.ReadLine();
            timerKuKu.Dispose();
            timerKuKuRiku.Dispose();
            timerQuaQua.Dispose();
        }
    }
}
```

Timers

```
static void SayHello(object data)
{
    Console.WriteLine(data);
}
}
```

Timers



```
C:\Windows\system32\cmd.exe - TimerDemo

E:\CSHARP_SAMPLES>TimerDemo
kuku
kuku
kuku
kuku
kukuriku
kuku
kukuriku
kukuriku
kuku
kukuriku
kukuriku
kuku
kukuriku
kukuriku
kuku
kukuriku
kukuriku
kuku
quaquaqua
quaquaqua
kukuriku
quaquaqua
kuku
quaquaqua
```

Multi Threaded Timers

- The `System.Threading.Timer` and the `System.Timers.Timer` use the thread pool to generate timer events.
- They are considered to be multi threaded timers.

Single Threaded Timers

- The `System.Windows.Forms.Timer` and the `System.Windows.Threading.DispatcherTimer` rely on the same thread that started them. The same thread that started them is the one that calls the scheduled method.
- Using each one of these single threaded timers, the same thread that manages the user interface elements and controls is the one that executes the scheduled method.

Threads Signaling

- When one thread is waiting for another... waiting till it receives a notification from it... waiting till it receives a signal... we can use the `AutoResetEvent` and the `ManualResetEvent` classes.
- These two classes extend the `EventWaitHandle` class.

The 'AutoResetEvent' Class

- Using the `AutoResetEvent` class we can easily signal threads.
- Calling the `WaitOne()` method on the `AutoResetEvent` object from within a thread will pause it till a signal is received.
- Calling the `Set()` method on the `AutoResetEvent` object from within any other thread will signal the waiting thread and wakes it up.

The 'AutoResetEvent' Class

```
using System;
using System.Threading;

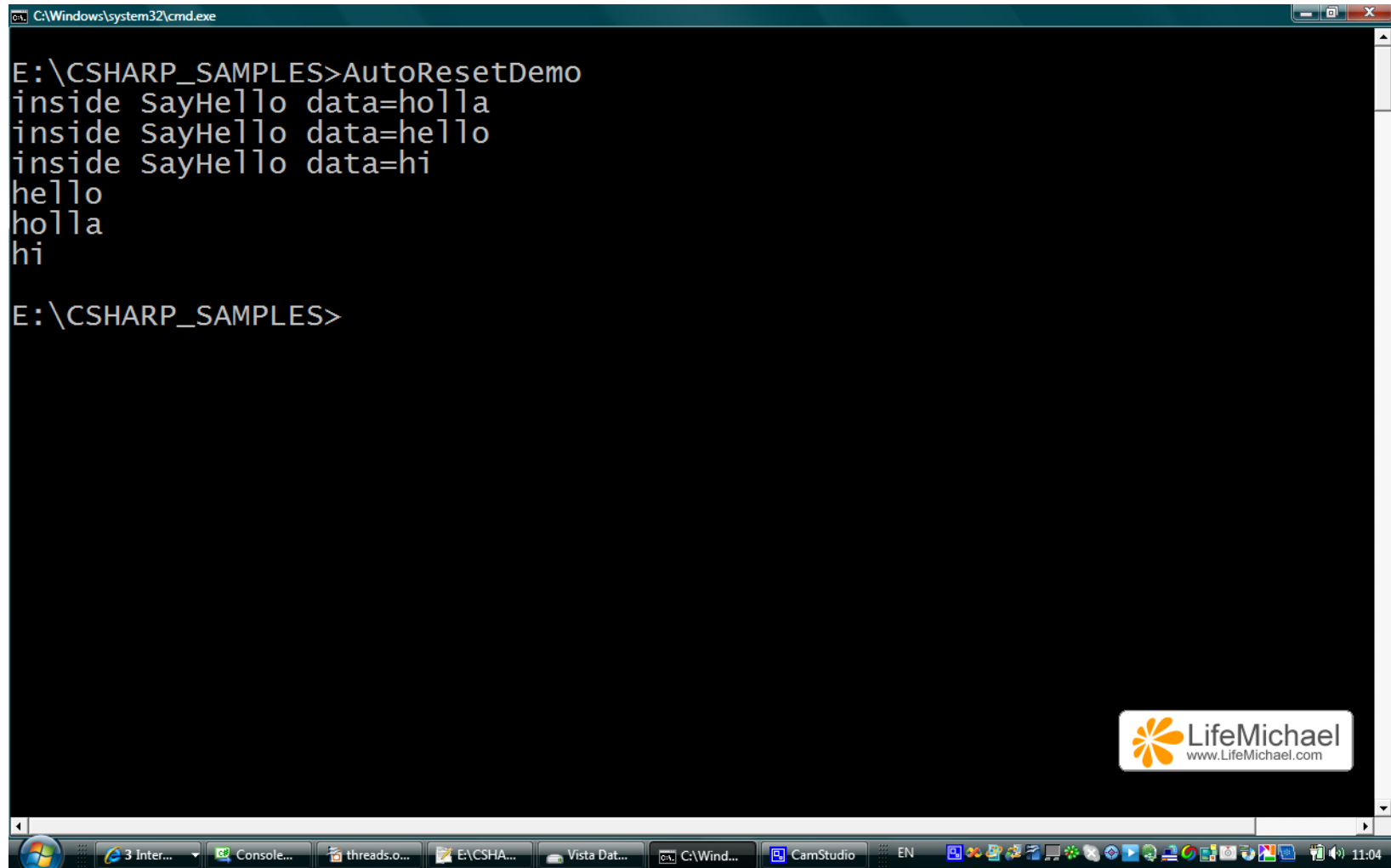
namespace abelski.csharp
{
    class AutoResetDemo
    {
        static EventWaitHandle handle = new AutoResetEvent(false);

        public static void Main()
        {
            new Thread(SayHello).Start("holla");
            new Thread(SayHello).Start("hello");
            new Thread(SayHello).Start("hi");
            Thread.Sleep(2000);
            handle.Set();
            Thread.Sleep(2000);
            handle.Set();
            Thread.Sleep(2000);
            handle.Set();
        }
    }
}
```

The 'AutoResetEvent' Class

```
static void SayHello(object data)
{
    Console.WriteLine("inside SayHello data="+data);
    handle.WaitOne();
    Console.WriteLine(data);
}
}
```

The 'AutoResetEvent' Class



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window shows the execution of a program named "AutoResetDemo" from the directory "E:\CSHARP_SAMPLES". The program's output consists of three lines of "inside SayHello data=" followed by "holla", "hello", and "hi" respectively. Below these, the words "hello", "holla", and "hi" are printed on separate lines. The command prompt then returns to the "E:\CSHARP_SAMPLES>" prompt. In the bottom right corner of the command prompt window, there is a logo for "LifeMichael" with the website "www.LifeMichael.com". The Windows taskbar is visible at the bottom, showing several open applications including "3 Inter...", "Console...", "threads.o...", "E\CSHA...", "Vista Dat...", "C:\Wind...", and "CamStudio". The system clock in the bottom right corner of the taskbar shows "11:04".

```
C:\Windows\system32\cmd.exe
E:\CSHARP_SAMPLES>AutoResetDemo
inside SayHello data=holla
inside SayHello data=hello
inside SayHello data=hi
hello
holla
hi
E:\CSHARP_SAMPLES>
```

The 'AutoResetEvent' Class

- If more than one thread calls the `WaitOne` method then a queue of waiting threads is built up behind the scene.
- A signal can come from any thread. Any thread (unblocked one) that can access the `AutoResetEvent` object can call the `Set` method on it, in order to release one of the blocked threads.

The 'AutoResetEvent' Class

- If `Set` is called and there isn't any thread waiting, the handle will stay open for as long as it takes till some thread calls `WaitOne`. The next code sample shows that.
- Calling `Set` multiple times doesn't create a queue of signals. Multiple threads that try to call `WaitOne` won't get an automatic signal. Only the first one will get an automatic signal.

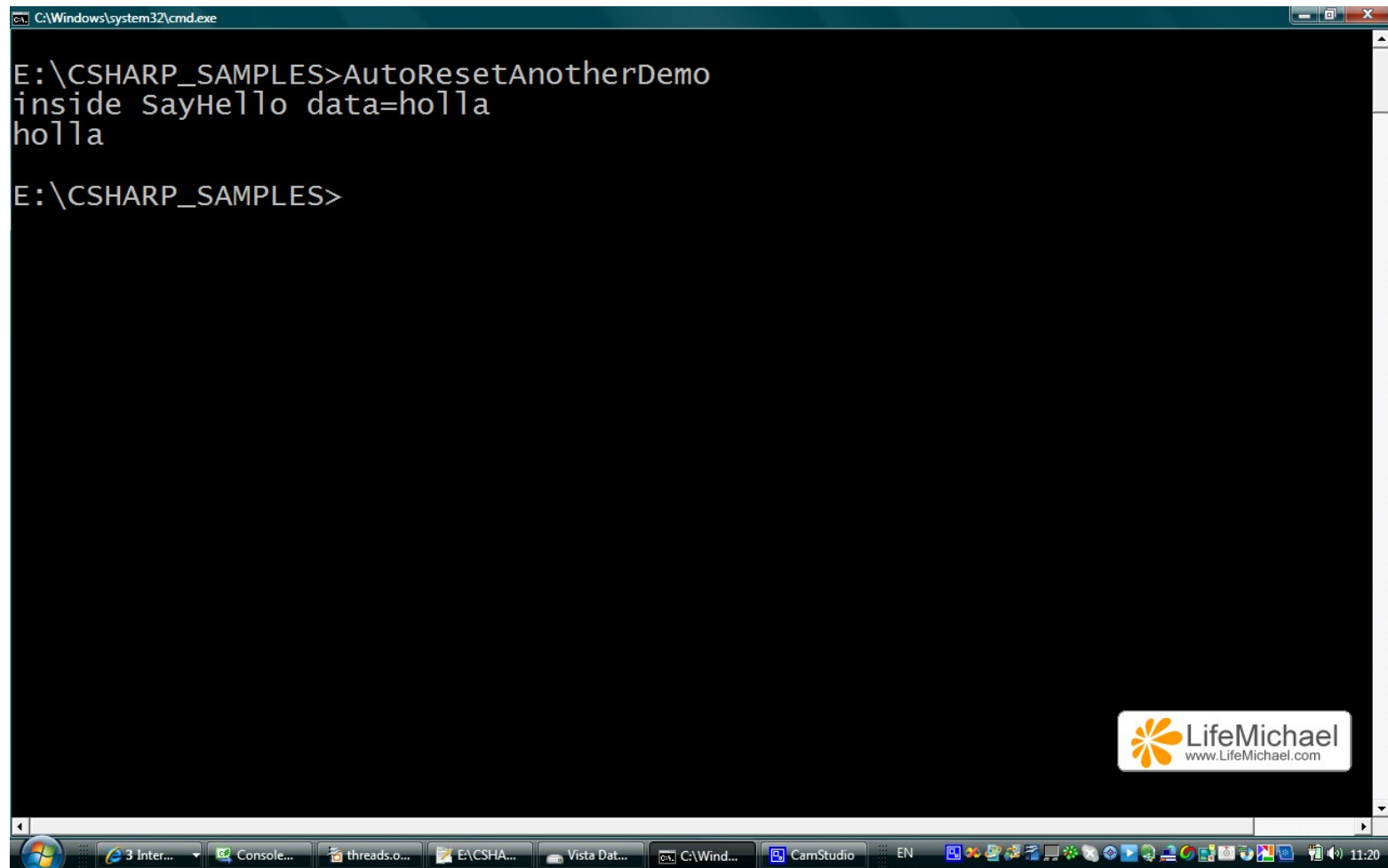
The 'AutoResetEvent' Class

```
using System;
using System.Threading;

namespace abelski.csharp
{
    class AutoResetAnotherDemo
    {
        static EventWaitHandle handle = new AutoResetEvent(false);

        public static void Main()
        {
            handle.Set();
            new Thread(SayHello).Start("hollla");
        }
        static void SayHello(object data)
        {
            Console.WriteLine("inside SayHello data="+data);
            handle.WaitOne();
            Console.WriteLine(data);
        }
    }
}
```

The 'AutoResetEvent' Class



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window shows the following text:

```
E:\CSHARP_SAMPLES>AutoResetAnotherDemo  
inside SayHello data=holla  
holla  
E:\CSHARP_SAMPLES>
```

The window has a standard Windows taskbar at the bottom with several open applications: "3 Inter...", "Console...", "threads.o...", "E\CSHA...", "Vista Dat...", "C\Wind...", and "CamStudio". A watermark for "LifeMichael" with the website "www.LifeMichael.com" is visible in the bottom right corner of the command prompt window.

The 'ManualResetEvent' Class

- The `ManualResetEvent` class functions similarly to the `AutoResetEvent` class.
- Unlike `AutoResetEvent` class, when working with the `ManualResetEvent` class calling the `Set()` method functions as if we opened the door so that every thread that calls `WaitOne()` on the very same `ManualResetEvent` object is immediately signaled with a permission to continue. Calling `Reset()` closes that door.

The 'ManualResetEvent' Class

```
using System;
using System.Threading;

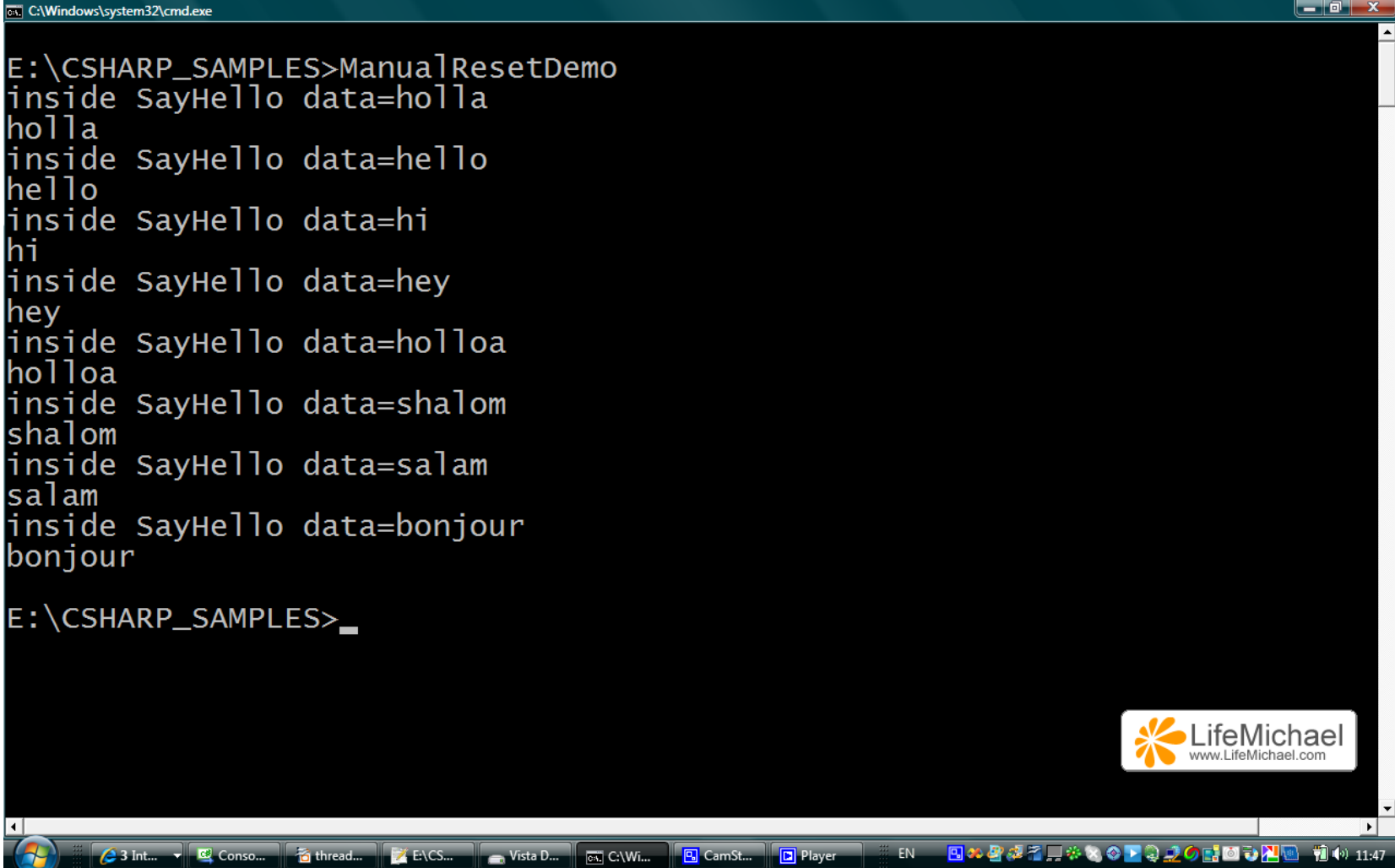
namespace abelski.csharp
{
    class ManualResetDemo
    {
        static EventWaitHandle handle = new ManualResetEvent(false);

        public static void Main()
        {
            handle.Set();
            new Thread(SayHello).Start("holla");
            new Thread(SayHello).Start("hello");
            new Thread(SayHello).Start("shalom");
            new Thread(SayHello).Start("salam");
            Thread.Sleep(2000);
            handle.Reset();
            new Thread(SayHello).Start("bonjour");
            Thread.Sleep(4000);
            handle.Set();
        }
    }
}
```

The 'ManualResetEvent' Class

```
static void SayHello(object data)
{
    Console.WriteLine("inside SayHello data="+data);
    handle.WaitOne();
    Console.WriteLine(data);
}
}
```

The 'ManualResetEvent' Class



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window shows the execution of a program named "ManualResetDemo" from the directory "E:\CSHARP_SAMPLES". The program outputs a series of "inside SayHello" messages with different data values: "holla", "hello", "hi", "hey", "holloa", "shalom", "salam", and "bonjour". The prompt "E:\CSHARP_SAMPLES>" is visible at the bottom of the command window. A watermark for "LifeMichael" with the website "www.LifeMichael.com" is located in the bottom right corner of the command window. The Windows taskbar is visible at the bottom of the screen, showing various open applications and the system clock.

```
C:\Windows\system32\cmd.exe

E:\CSHARP_SAMPLES>ManualResetDemo
inside SayHello data=holla
holla
inside SayHello data=hello
hello
inside SayHello data=hi
hi
inside SayHello data=hey
hey
inside SayHello data=holloa
holloa
inside SayHello data=shalom
shalom
inside SayHello data=salam
salam
inside SayHello data=bonjour
bonjour

E:\CSHARP_SAMPLES>
```

Two Way Signaling

- When we want two threads to signal each other we cannot signal few times in a row and expect the other thread to receive each one of the signals. When calling `Set()` several times rapidly the second or third signals may get lost.
- We can have two threads communicating with each other using two `AutoResetEvent` objects.

Two Way Signaling

```
using System;
using System.Threading;

namespace abelski.csharp
{
    class TwoWaySignalingDemo
    {
        static EventWaitHandle handleA = new AutoResetEvent(false);
        static EventWaitHandle handleB = new AutoResetEvent(false);
        static string message;

        public static void Main()
        {
            new Thread(DoSomething).Start();

            handleA.WaitOne(); //wait till DoSomething is ready to reply
            message = "Hello";
            handleB.Set();     //indicate DoSomething it can proceed

            handleA.WaitOne(); //wait till DoSomething is ready to reply
            message = "Good Morning";
            handleB.Set();
        }
    }
}
```

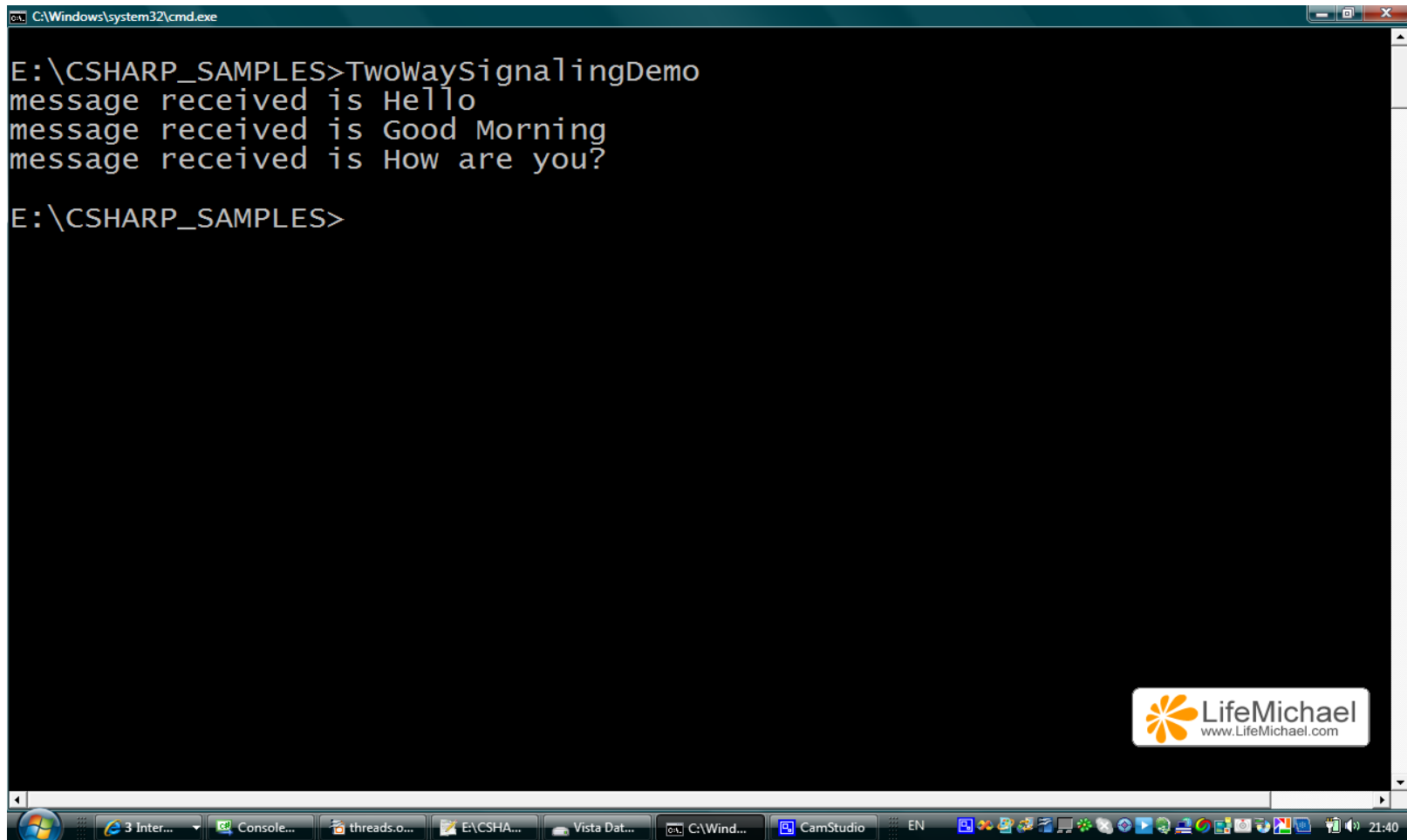

Two Way Signaling

```
handleA.WaitOne(); //wait till DoSomething is ready to reply
message = "How are you?";
handleB.Set();
```

```
handleA.WaitOne(); //wait till DoSomething is ready to reply
message = "exit";
handleB.Set();
```

```
}
static void DoSomething()
{
    while(true)
    {
        handleA.Set(); //indicate DoSomething is ready
        handleB.WaitOne(); //wait for getting a message
        if(message=="exit") return;
        Console.WriteLine("message received is "+message);
    }
}
}
```

Two Way Signaling



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window shows the execution of a program named "TwoWaySignalingDemo" from the directory "E:\CSHARP_SAMPLES". The program outputs three lines of text: "message received is Hello", "message received is Good Morning", and "message received is How are you?". The prompt "E:\CSHARP_SAMPLES>" is visible at the bottom of the command input area. In the bottom right corner of the command prompt window, there is a logo for "LifeMichael" with the website "www.LifeMichael.com". The Windows taskbar is visible at the bottom of the screen, showing several open applications including "3 Inter...", "Console...", "threads.o...", "E\CSHA...", "Vista Dat...", "C\Wind...", and "CamStudio". The system clock in the bottom right corner of the taskbar shows "21:40".

```
C:\Windows\system32\cmd.exe
E:\CSHARP_SAMPLES>TwoWaySignalingDemo
message received is Hello
message received is Good Morning
message received is How are you?
E:\CSHARP_SAMPLES>
```

Two Way Signaling

- The `EventWaitHandle`'s constructor allows us to create a named `EventWaitHandle` object, we can use across multiple processes.

The assigned name is a simple string and it can be of any value that doesn't unintentionally conflict with a name of another `EventWaitHandle`.

```
EventWaitHandle handle = new EventWaitHandle(  
    false,  
    EventResetMode.AutoReset,  
    "com.abelski.samples");
```

The 'BackgroundWorker' Class

- The 'Background Worker' is a helper class for managing a thread running in the background.
- It provides a standard protocol for reporting its progress.

The 'BackgroundWorker' Class

- It implements the `IComponent` interface, which allows it to be sited within the Visual Studio's designer.
- It provides an exception handling mechanism for the working thread. There is no need to use try & catch block within the worker method.
- A `BackgroundWorker` object uses `ThreadPool`.

The 'BackgroundWorker' Class

```
using System;
using System.Threading;
using System.ComponentModel;

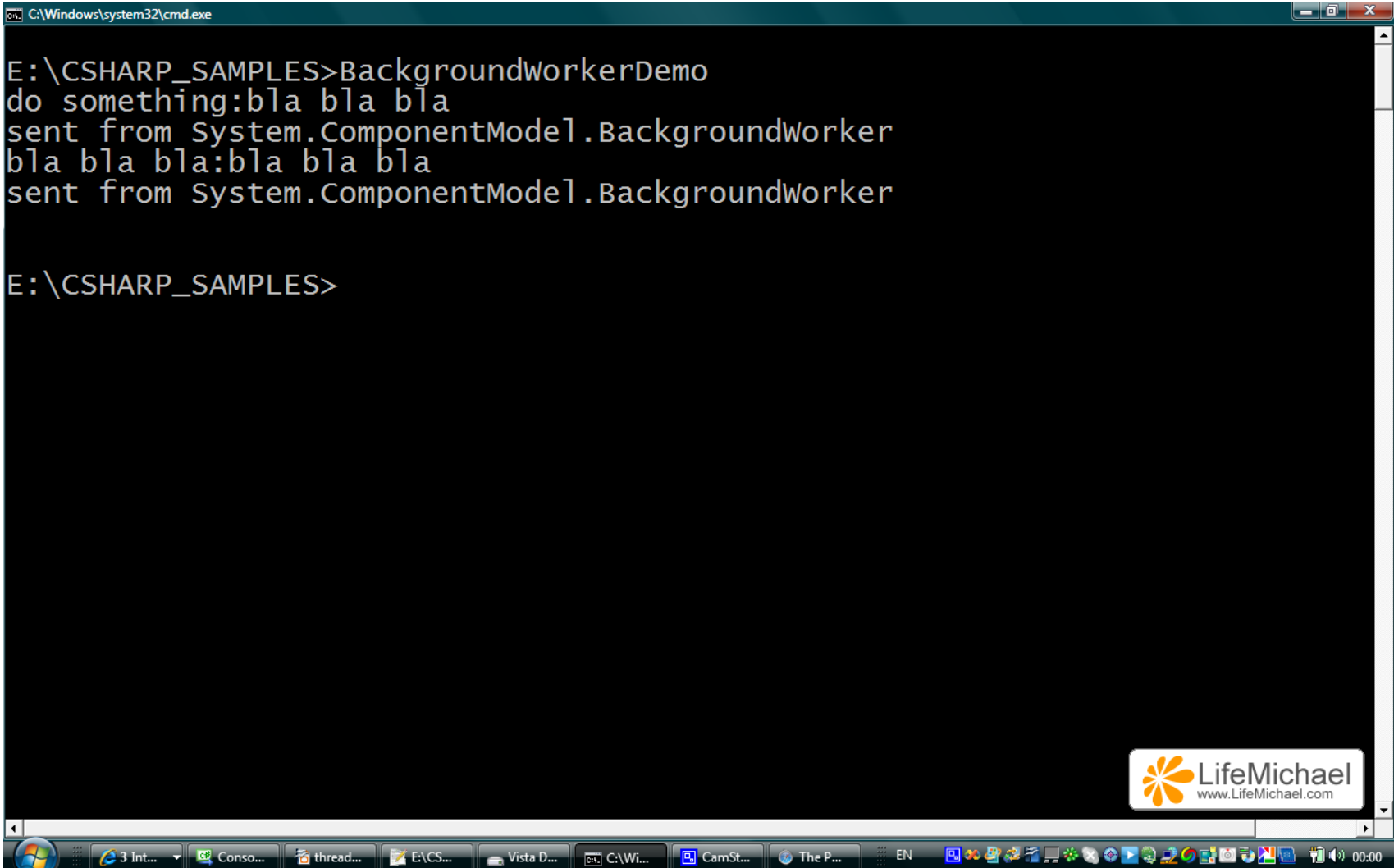
namespace abelski.csharp
{
    class BackgroundWorkerDemo
    {
        static BackgroundWorker worker = new BackgroundWorker();

        public static void Main()
        {
            worker.DoWork += DoSomething;
            worker.DoWork += DoBlaBla;
            worker.RunWorkerAsync("bla bla bla");
            Console.ReadLine();
        }
    }
}
```

The 'BackgroundWorker' Class

```
static void DoSomething(object sender, DoWorkEventArgs ev)
{
    Console.WriteLine("do something:"+ev.Argument);
    Console.WriteLine("sent from "+sender);
}
static void DoBlaBla(object sender, DoWorkEventArgs ev)
{
    Console.WriteLine("bla bla bla:"+ev.Argument);
    Console.WriteLine("sent from "+sender);
}
}
```

The 'BackgroundWorker' Class



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window shows the following text:

```
E:\CSHARP_SAMPLES>BackgroundWorkerDemo  
do something:bla bla bla  
sent from System.ComponentModel.BackgroundWorker  
bla bla bla:bla bla bla  
sent from System.ComponentModel.BackgroundWorker  
  
E:\CSHARP_SAMPLES>
```

The window has a taskbar at the bottom with several open applications: "3 Int...", "Conso...", "thread...", "E:\CS...", "Vista D...", "C:\Wi...", "CamSt...", "The P...", and "EN". A watermark for "LifeMichael" with the website "www.LifeMichael.com" is visible in the bottom right corner of the command prompt window.

The 'BackgroundWorker' Class

- The `BackgroundWorker` class also enables to set a `RunWorkerCompleted` event that fires once the `BackgroundWorker` object completes its job.

We can use this event in order to query about exceptions as well as for updating the user interface. Code within the event handler method (each one of the threads) isn't allowed to update the user interface.

- Calling periodically the `ReportProgress` from within the event handler we can update about its progress.

The `BackgroundWorker` should be registered with method to handle the report progress reports. The registration should be done using the `ProgressChanged` property of the `BackgroundWorker` object.

The 'BackgroundWorker' Class

- In order to add support for canceling a running thread we should set the `WorkerSupportsCancellation` property to 'true'.

The 'BackgroundWorker' Class

```
using System;
using System.Threading;
using System.ComponentModel;

namespace abelski.csharp
{
    class BackgroundWorkerSophisticatedDemo
    {
        static BackgroundWorker worker = new BackgroundWorker();

        public static void Main()
        {
            worker.WorkerReportsProgress = true;
            worker.WorkerSupportsCancellation = true;
            worker.DoWork += DoSomething;
            worker.DoWork += DoNothing;
            worker.ProgressChanged += ProgressChangeMethod;
            worker.RunWorkerCompleted += WorkerCompletedMethod;
            worker.RunWorkerAsync("bla bla bla");
            Console.ReadLine();
        }
    }
}
```

The 'BackgroundWorker' Class

```
static void WorkerCompletedMethod(  
    Object sender,  
    RunWorkerCompletedEventArgs ev)  
{  
    if (ev.Cancelled)  
    {  
        Console.WriteLine(sender+" canceled by the thread");  
    }  
    else if (ev.Error!=null)  
    {  
        Console.WriteLine("exception within the "+sender  
            +" thread : "+ev.Error.ToString());  
    }  
    else  
    {  
        Console.WriteLine(sender+" was completed");  
    }  
}
```

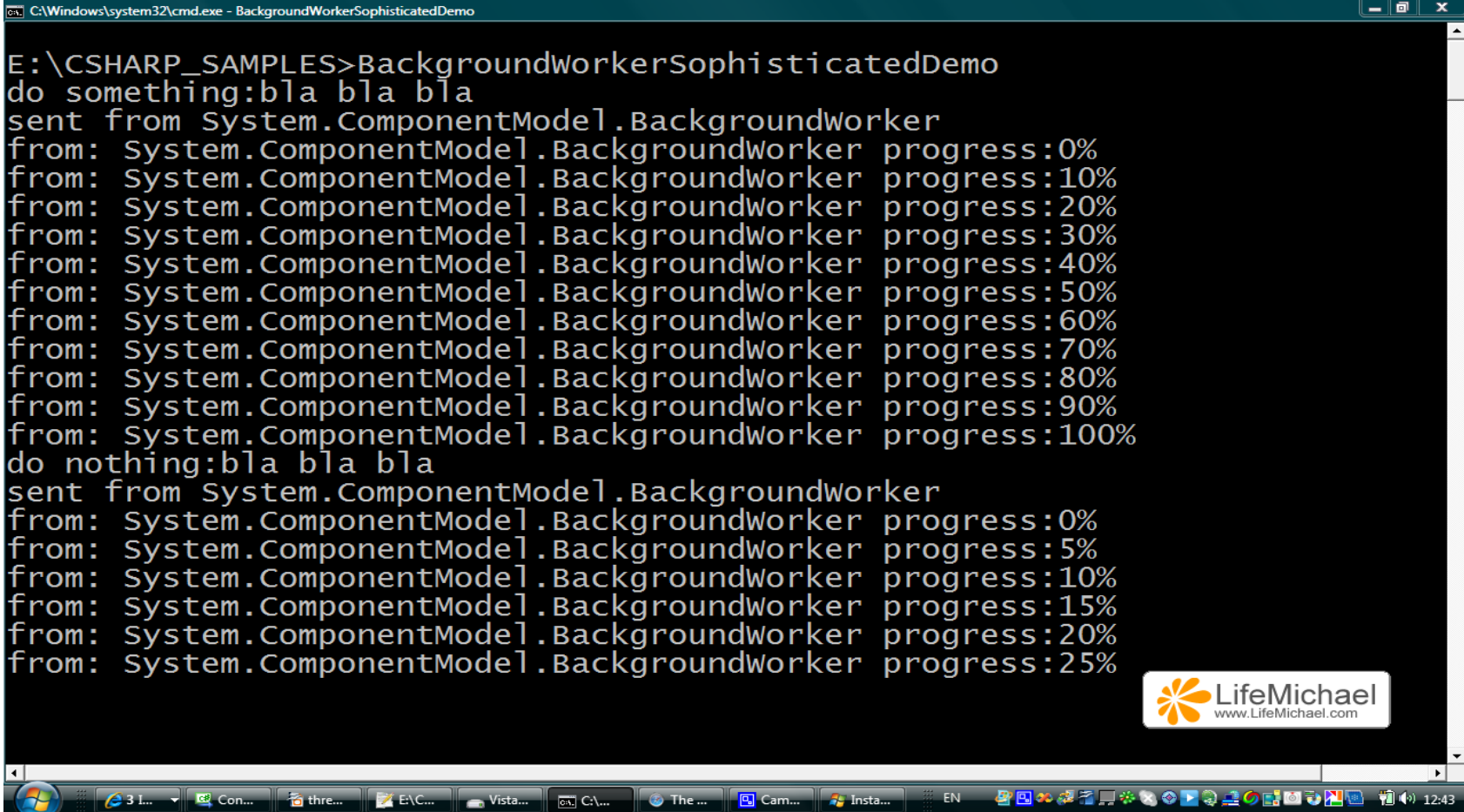
The 'BackgroundWorker' Class

```
static void ProgressChangeMethod(  
    Object sender,  
    ProgressChangedEventArgs ev)  
{  
    Console.WriteLine("from: "+sender  
        +" progress:"+ev.ProgressPercentage+"%");  
}  
static void DoSomething(object sender, DoWorkEventArgs ev)  
{  
    Console.WriteLine("do something:"+ev.Argument);  
    Console.WriteLine("sent from "+sender);  
    for(int i=0;i<=100; i+=10)  
    {  
        worker.ReportProgress(i);  
        Thread.Sleep(500);  
    }  
}
```

The 'BackgroundWorker' Class

```
static void DoNothing(object sender, DoWorkEventArgs ev)
{
    Console.WriteLine("do nothing:"+ev.Argument);
    Console.WriteLine("sent from "+sender);
    for(int i=0;i<=100; i+=5)
    {
        worker.ReportProgress(i);
        Thread.Sleep(400);
    }
}
```

The 'BackgroundWorker' Class

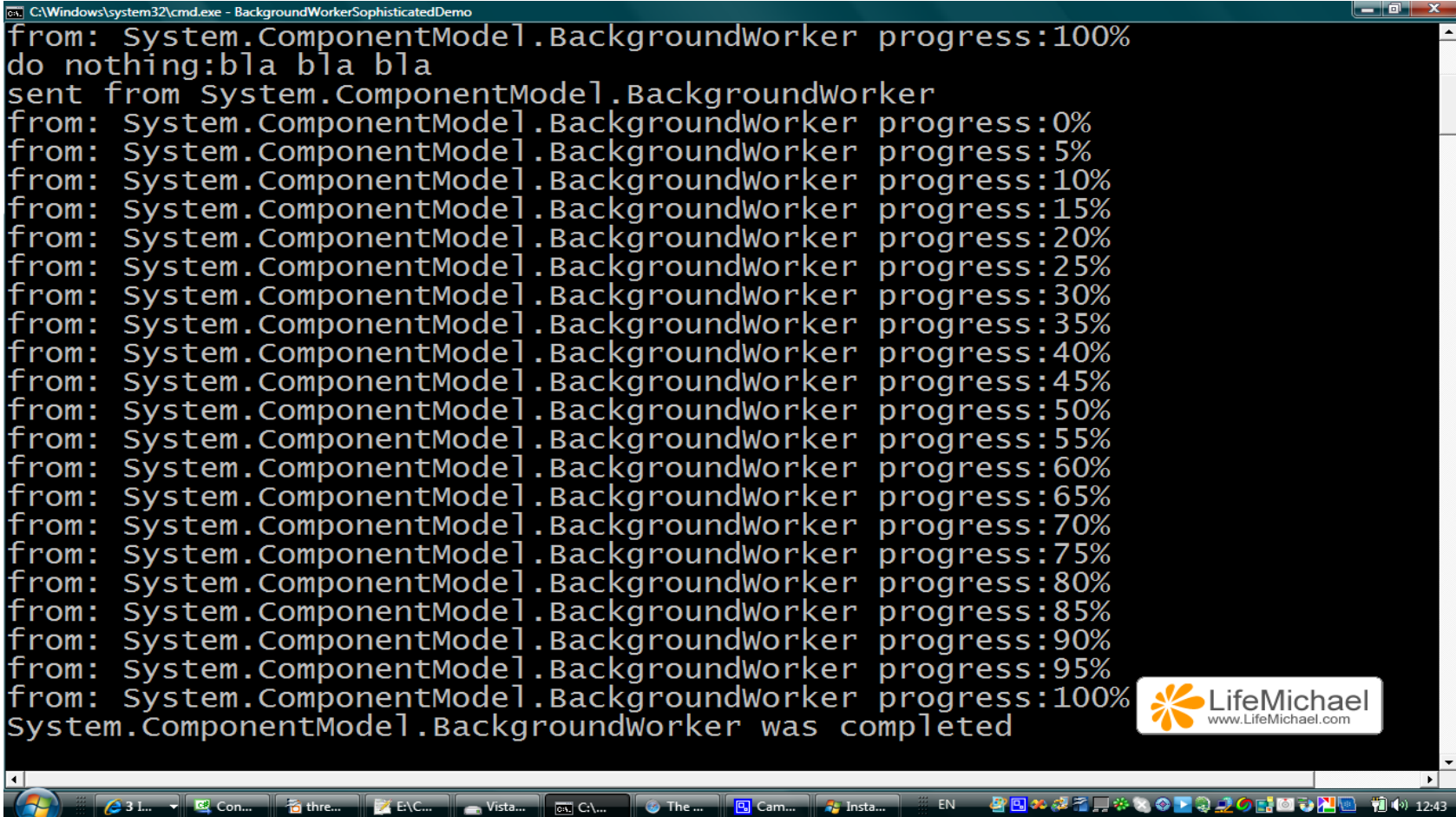


```
C:\Windows\system32\cmd.exe - BackgroundWorkerSophisticatedDemo

E:\CSHARP_SAMPLES>BackgroundWorkerSophisticatedDemo
do something:bla bla bla
sent from System.ComponentModel.BackgroundWorker
from: System.ComponentModel.BackgroundWorker progress:0%
from: System.ComponentModel.BackgroundWorker progress:10%
from: System.ComponentModel.BackgroundWorker progress:20%
from: System.ComponentModel.BackgroundWorker progress:30%
from: System.ComponentModel.BackgroundWorker progress:40%
from: System.ComponentModel.BackgroundWorker progress:50%
from: System.ComponentModel.BackgroundWorker progress:60%
from: System.ComponentModel.BackgroundWorker progress:70%
from: System.ComponentModel.BackgroundWorker progress:80%
from: System.ComponentModel.BackgroundWorker progress:90%
from: System.ComponentModel.BackgroundWorker progress:100%
do nothing:bla bla bla
sent from System.ComponentModel.BackgroundWorker
from: System.ComponentModel.BackgroundWorker progress:0%
from: System.ComponentModel.BackgroundWorker progress:5%
from: System.ComponentModel.BackgroundWorker progress:10%
from: System.ComponentModel.BackgroundWorker progress:15%
from: System.ComponentModel.BackgroundWorker progress:20%
from: System.ComponentModel.BackgroundWorker progress:25%
```

LifeMichael
www.LifeMichael.com

The 'BackgroundWorker' Class



```
C:\Windows\system32\cmd.exe - BackgroundWorkerSophisticatedDemo
from: System.ComponentModel.BackgroundWorker progress:100%
do nothing:bla bla bla
sent from System.ComponentModel.BackgroundWorker
from: System.ComponentModel.BackgroundWorker progress:0%
from: System.ComponentModel.BackgroundWorker progress:5%
from: System.ComponentModel.BackgroundWorker progress:10%
from: System.ComponentModel.BackgroundWorker progress:15%
from: System.ComponentModel.BackgroundWorker progress:20%
from: System.ComponentModel.BackgroundWorker progress:25%
from: System.ComponentModel.BackgroundWorker progress:30%
from: System.ComponentModel.BackgroundWorker progress:35%
from: System.ComponentModel.BackgroundWorker progress:40%
from: System.ComponentModel.BackgroundWorker progress:45%
from: System.ComponentModel.BackgroundWorker progress:50%
from: System.ComponentModel.BackgroundWorker progress:55%
from: System.ComponentModel.BackgroundWorker progress:60%
from: System.ComponentModel.BackgroundWorker progress:65%
from: System.ComponentModel.BackgroundWorker progress:70%
from: System.ComponentModel.BackgroundWorker progress:75%
from: System.ComponentModel.BackgroundWorker progress:80%
from: System.ComponentModel.BackgroundWorker progress:85%
from: System.ComponentModel.BackgroundWorker progress:90%
from: System.ComponentModel.BackgroundWorker progress:95%
from: System.ComponentModel.BackgroundWorker progress:100%
System.ComponentModel.BackgroundWorker was completed
```

LifeMichael
www.LifeMichael.com

Local Storage

- Using the `GetData` and `SetData` methods declared within the `Thread` class we can store isolated data. Each thread will have its own isolated separated data.
- Both methods required a `LocalDataStoreSlot` object that will identify the slot they refer. The same slot can be used across all threads. Each thread will have its separated value.

Local Storage

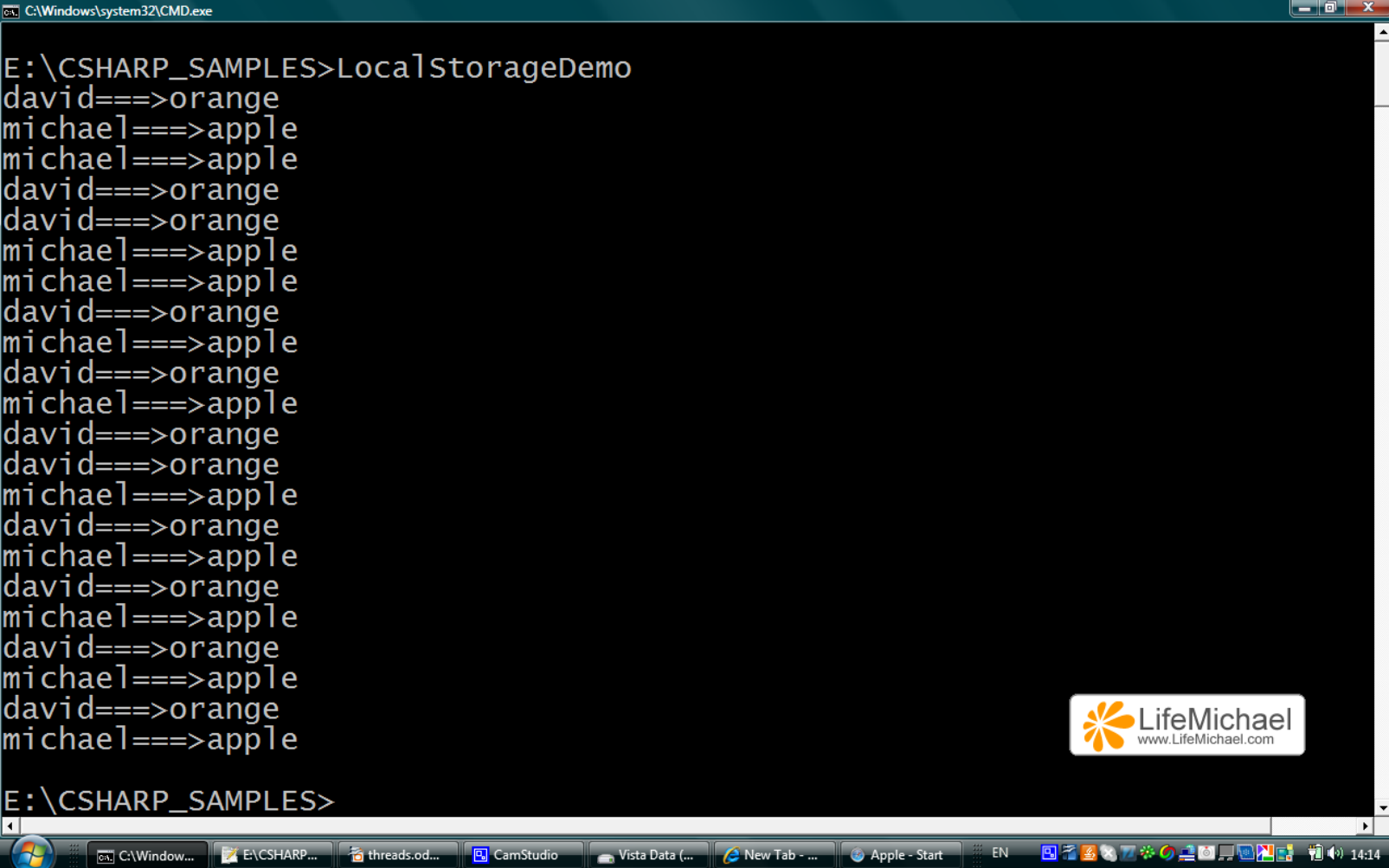
```
using System;
using System.Threading;
using System.ComponentModel;

namespace abelski.csharp
{
    public class LocalStorageDemo
    {
        public static LocalDataStoreSlot slot =
            Thread.GetNamedDataSlot("yarkan_name");
        public static void Main()
        {
            Thread t1 = new Thread(
                new Yarkan("david", "orange").DoSomething);
            t1.Start();
            Thread t2 = new Thread(
                new Yarkan("michael", "apple").DoSomething);
            t2.Start();
        }
    }
}
```

Local Storage

```
public class Yarkan
{
    private string name;
    private string product;
    public Yarkan(string nameVal, string productVal)
    {
        name = nameVal;
        product = productVal;
    }
    public void DoSomething()
    {
        Thread.SetData(LocalStorageDemo.slot, name);
        for(int i=0; i<=100; i+=10)
        {
            Console.WriteLine(
                Thread.GetData(LocalStorageDemo.slot) +
                "===>" + product);
            Thread.Sleep(500);
        }
    }
}
```

Local Storage



A screenshot of a Windows command prompt window titled "C:\Windows\system32\CMD.exe". The window shows a directory path "E:\CSHARP_SAMPLES" and a command "LocalStorageDemo". The output of the command is a series of alternating names: "david==>orange", "michael==>apple", "michael==>apple", "david==>orange", "david==>orange", "michael==>apple", "michael==>apple", "david==>orange", "michael==>apple", "david==>orange", "michael==>apple", "david==>orange", "david==>orange", "michael==>apple", "david==>orange", "michael==>apple", "david==>orange", "michael==>apple", "david==>orange", "michael==>apple", "david==>orange", "michael==>apple". The window also features a "LifeMichael" logo with the website "www.LifeMichael.com" in the bottom right corner. The taskbar at the bottom shows various open applications including "C:\Window...", "E:\CSHARP...", "threads.od...", "CamStudio", "Vista Data (...)", "New Tab - ...", "Apple - Start", and "EN". The system clock shows "14:14".

```
C:\Windows\system32\CMD.exe

E:\CSHARP_SAMPLES>LocalStorageDemo
david==>orange
michael==>apple
michael==>apple
david==>orange
david==>orange
michael==>apple
michael==>apple
david==>orange
michael==>apple
david==>orange
michael==>apple
david==>orange
david==>orange
michael==>apple
david==>orange
michael==>apple
david==>orange
michael==>apple
david==>orange
michael==>apple
david==>orange
michael==>apple

E:\CSHARP_SAMPLES>
```

The 'ReaderWriterLockSlim' Class

- This class enables us to create two different types of locks, a read lock and a write lock. When a thread holds a write lock all other threads trying to obtain a read or a write lock shall fail. When a thread holds a read lock, any number of threads may concurrently obtain a read lock as well.

The 'ReaderWriterLockSlim' Class

- This class defines the following methods for obtaining and releasing read and write locks:

```
public void EnterReadLock()  
public void TryEnterReadLock()  
public void ExitReadLock()  
public void EnterWriteLock()  
public void TryEnterWriteLock()  
public void ExitWriteLock()
```

The 'ReaderWriterLockSlim' Class

```
using System;
using System.Threading;
using System.ComponentModel;
using System.Collections.Generic;

namespace abelski.csharp
{
    public class ReaderWriterLockSlimDemo
    {
        static ReaderWriterLockSlim locker = new ReaderWriterLockSlim();
        static List<int> numbers = new List<int>();
        static int[] vec = {12,123,512,21,535,6,
                           3,74654,233,4,1,2,3,
                           4,5,6,7,8,8,65,4,3,3,2};

        public static void Main()
        {
            new Thread(Read).Start();
            new Thread(Read).Start();
            new Thread(Read).Start();
            new Thread(Write).Start();
        }
    }
}
```

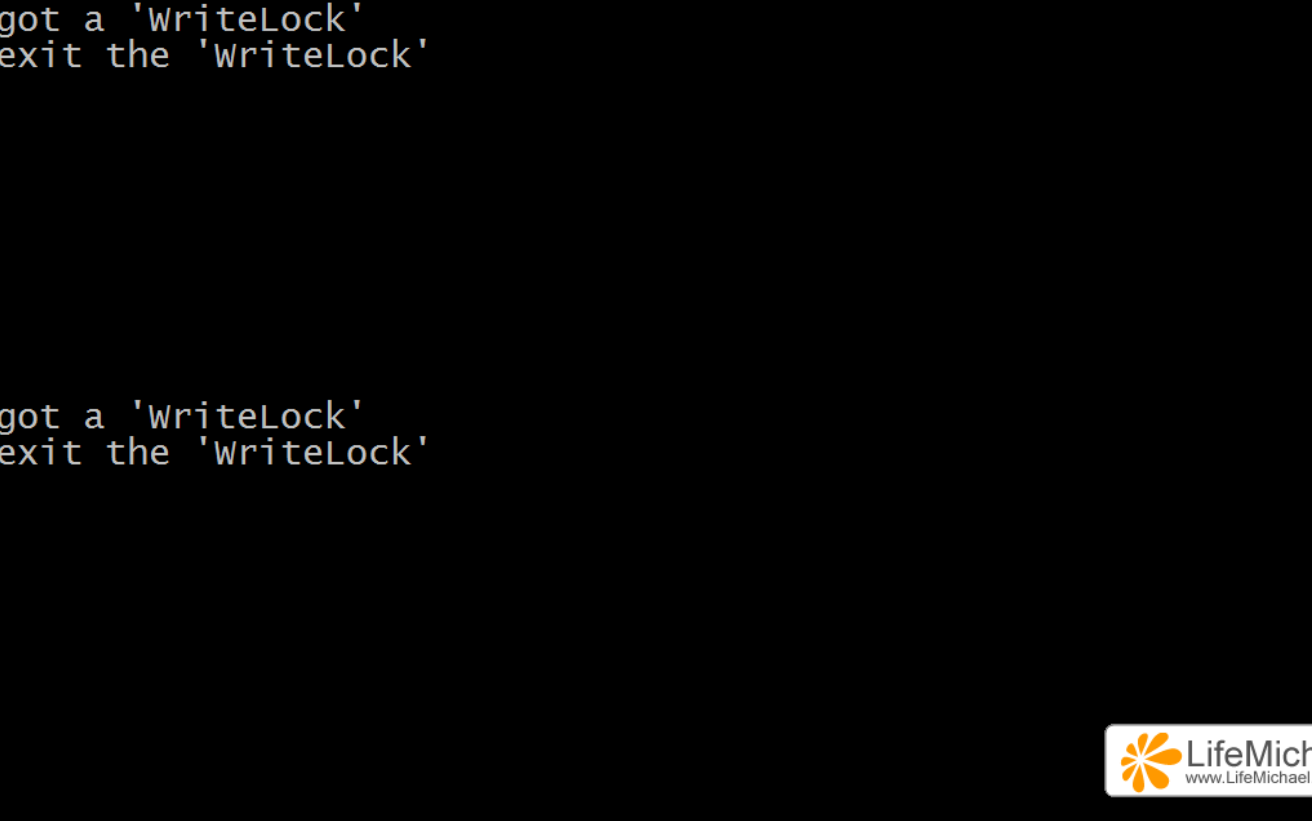
The 'ReaderWriterLockSlim' Class

```
public static void Read()
{
    while(true)
    {
        locker.EnterReadLock();
        foreach(int i in numbers)
        {
            Console.WriteLine(i);
        }
        Thread.Sleep(2000);
        locker.ExitReadLock();
    }
}
```


The 'ReaderWriterLockSlim' Class

```
public static void Write()
{
    for(int i=0; i<vec.Length; i++)
    {
        locker.EnterWriteLock();
        Console.WriteLine("writer got a 'WriteLock'");
        numbers.Add(vec[i]);
        Thread.Sleep(4000);
        Console.WriteLine("writer exit the 'WriteLock'");
        locker.ExitWriteLock();
        Thread.Sleep(4000);
    }
}
}
```

The 'ReaderWriterLockSlim' Class



```
C:\Windows\system32\CMD.exe - ReaderWriterLockSlimDemo

E:\CSHARP_SAMPLES>ReaderWriterLockSlimDemo
writer got a 'WriteLock'
writer exit the 'WriteLock'
12
12
12
12
12
12
12
12
12
12
writer got a 'WriteLock'
writer exit the 'WriteLock'
12
123
12
123
12
123
```

LifeMichael
www.LifeMichael.com

The Abort Method

- We can terminate the life of a blocked thread by calling the `Thread.Abort` method.

A thread becomes a blocked thread as a result of calling any of the following methods: `Sleep`, `Join`, `EndInvoke`, `WaitOne` or `Wait`.

- Calling the `Thread.Abort` method we can also end the life of a non-blocked thread, such as a thread that reaches an infinite loop.

The Interrupt Method

- When calling `Thread.Interrupt` on a blocked thread the `ThreadInterruptedException` is thrown.

```
...  
try  
{  
    Thread.Sleep(...);  
}  
catch (ThreadInterruptedException ex)  
{  
    ...  
}  
...
```

The Interrupt Method

- Calling the `Thread.Interrupt` on a thread that is not blocked, the thread continues to execute till it blocks and once that happens the `ThreadInterruptedException` is thrown.

Threads Termination

- Terminating a thread in a safe way involves with using a boolean flag variable within the thread loop.

```
...  
while(flag)  
{  
    ...  
}  
...
```

Threads Termination

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Diagnostics;

namespace ConsoleApplication1
{
    public class Program
    {
        private static bool flag = true;
        public static void Main(string[] args)
        {
            new Thread(GenerateMagicNumbers).Start();
            Thread.Sleep(10000);
            StopGeneratingMagicNumbers();
        }
    }
}
```

Threads Termination

```
public static void GenerateMagicNumbers()
{
    Random random = new Random();
    while (flag)
    {
        for (int i = 0; i < 10; i++)
        {
            Console.Write(random.Next(1,1000) + " ");
        }
        Console.WriteLine();
        Thread.Sleep(200);
    }
}

public static void StopGeneratingMagicNumbers()
{
    flag = false;
}

}
```


The Wait & Pulse Methods

- The `Monitor` class allows us to signal in between two threads by calling the `Monitor.Wait` and the `Monitor.Pulse` static methods.
- Unlike using the event wait handles, the `Wait` and `Pulse` methods cannot span application domains or processes.
- Using `Wait` and `Pulse` is much faster than using an event wait handle.

The Wait & Pulse Methods

```
using System;
using System.Threading;
using System.ComponentModel;
using System.Collections.Generic;

namespace abelski.csharp
{
    public class SimpleWaitPulseDemo
    {
        static object locker = new object();
        static Queue<int> numbers = new Queue<int>();
        static int[] vec = {12,123,512,21,535,6,3,74654,233,
                           4,1,2,3,4,5,6,7,8,8,65,4,3,3,2};

        public static void Main()
        {
            new Thread(Write).Start();
            new Thread(Read).Start();
        }
    }
}
```

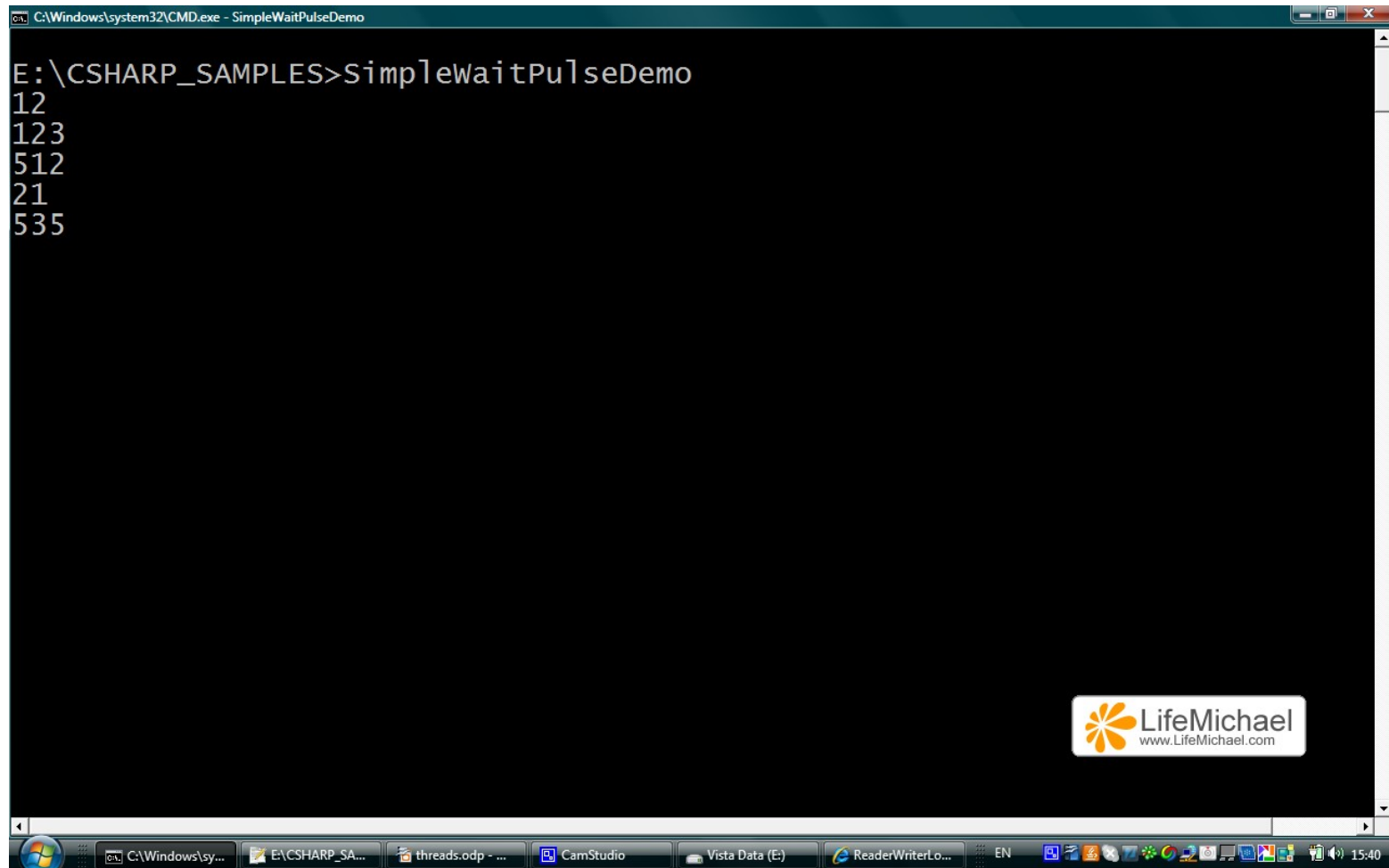
The Wait & Pulse Methods

```
public static void Read()
{
    while(true)
    {
        lock(locker)
        {
            while(numbers.Count==0)
            {
                Monitor.Wait(locker);
            }
            Console.WriteLine(numbers.Dequeue());
        }
        Thread.Sleep(1000);
    }
}
```

The Wait & Pulse Methods

```
public static void Write()
{
    for(int i=0; i<vec.Length; i++)
    {
        lock(locker)
        {
            numbers.Enqueue(vec[i]);
            Monitor.Pulse(locker);
        }
        Thread.Sleep(2000);
    }
}
}
```

The Wait & Pulse Methods

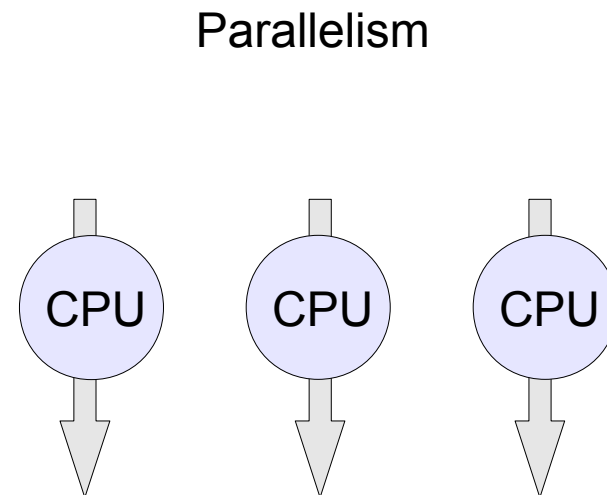
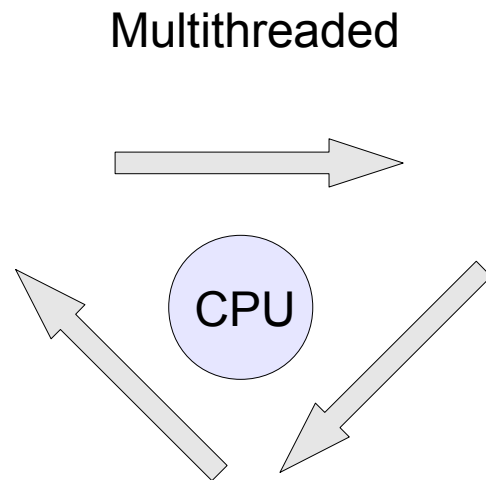


```
C:\Windows\system32\CMD.exe - SimpleWaitPulseDemo

E:\CSHARP_SAMPLES>SimpleWaitPulseDemo
12
123
512
21
535
```

Parallelism isn't Multithreaded

- We can have multithreading on a single core machine. We can have parallelism on a multi core machine only.



Parallelism isn't Multithreaded

“On a single core you can use threads and you can have concurrency, but to achieve parallelism on a multi-core box you have to identify in your code the exploitable concurrency: the portions of your code that can truly run at the same time.”

Daniel Moth, Microsoft

When Can We Benefit from Parallelism?

- Code we can benefit from running it in parallelism will most likely have the following characteristic:
 1. We can break it down into self encapsulated units.
 2. It has no dependencies or shared data.

The Parallel Libraries

- Starting with .NET 4.0 we can benefit using parallelism in our code. Using the new libraries isn't limited to multi-core machines only.
- Using these new libraries our code will automatically scale saving us from spending time on altering our code to target single core environments in a different way.

The Amdahl's Law

“The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.”

Gene Amdahl

The Parallel Loops

- One of the easiest ways to parallelize our code is by using the Parallel Loop construct.
- The parallel loop construct includes two types of loops:

`Parallel.For()`

`Parallel.ForEach()`

The Parallel.For Loop

- The `Parallel.For` static method allows us to pass over a method and have it executed in a parallel way.

The Parallel.For Loop

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;

namespace ConsoleApplication1
{
    public class Account
    {
        public double Balance { get; set; }
        public Account(double sum)
        {
            Balance = sum;
        }
        public int CalcCreditPoints()
        {
            Thread.Sleep(400);
            return (int)Balance/10;
        }
    }
}
```

The Parallel.For Loop

```
public class Program
{
    public static List<Account> accounts = new List<Account>();
    public static void Main(string[] args)
    {
        CreateAccounts();
        Console.WriteLine("parallel... "+CalcParallel());
        Console.WriteLine("serial... "+CalcSerial());
    }
    public static double CalcSerial()
    {
        Stopwatch stopper = new Stopwatch();
        stopper.Start();
        for (int i = 0; i < accounts.Count; i++)
        {
            Console.WriteLine("serially processing {0}",
                               accounts[i].CalcCreditPoints());
        }
        stopper.Stop();
        return stopper.ElapsedMilliseconds;
    }
}
```

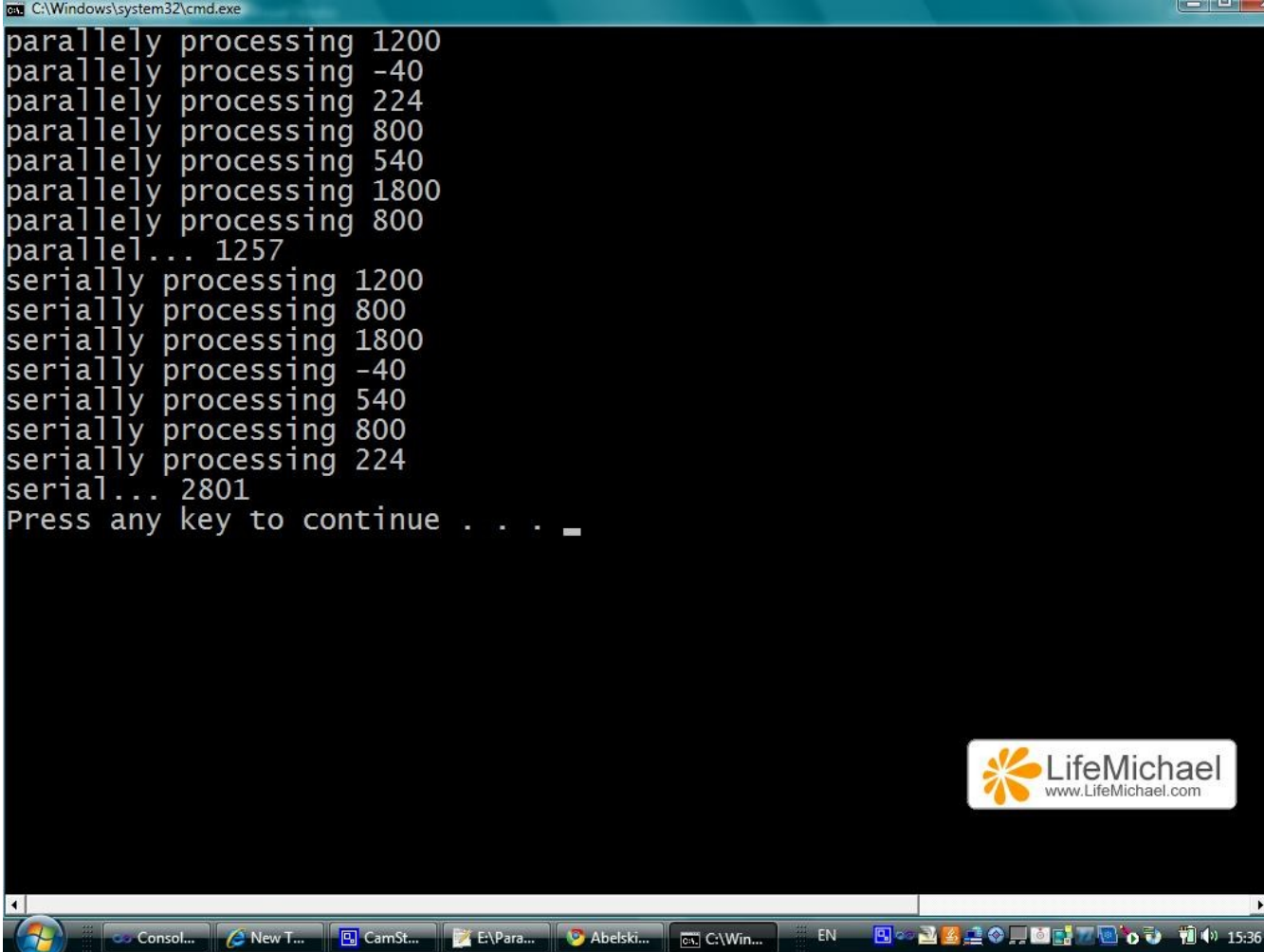
The Parallel.For Loop

```
public static double CalcParallel()
{
    Stopwatch stopper = new Stopwatch();
    stopper.Start();
    Parallel.For(0, accounts.Count, i =>
    {
        Console.WriteLine("parallely processing {0}",
            accounts[i].CalcCreditPoints());
    } );
    stopper.Stop();
    return stopper.ElapsedMilliseconds;
}

public static void CreateAccounts()
{
    double[] sums = { 12000, 8000, 18000, -400,
                     5400, 8000, 2240 };
    foreach (double sum in sums)
    {
        accounts.Add(new Account(sum));
    }
}

}
```

The Parallel.For Loop



```
C:\Windows\system32\cmd.exe
parallelly processing 1200
parallelly processing -40
parallelly processing 224
parallelly processing 800
parallelly processing 540
parallelly processing 1800
parallelly processing 800
parallel... 1257
serially processing 1200
serially processing 800
serially processing 1800
serially processing -40
serially processing 540
serially processing 800
serially processing 224
serial... 2801
Press any key to continue . . . _
```


The `Parallel.ForEach` Loop

- Similarly to the `Parallel.For` static method we can find the `Parallel.ForEach`.
- Both methods allow us to pass over a specific function and have it executed in a parallel way.

The Parallel.ForEach Loop

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;

namespace ConsoleApplication1
{
    public class Account
    {
        public double Balance { get; set; }
        public Account(double sum)
        {
            Balance = sum;
        }
        public int CalcCreditPoints()
        {
            Thread.Sleep(400);
            return (int)Balance/10;
        }
    }
}
```

The Parallel.ForEach Loop

```
public class Program
{
    public static List<Account> accounts = new List<Account>();
    public static void Main(string[] args)
    {
        CreateAccounts();
        Console.WriteLine("parallel... "+CalcParallel());
        Console.WriteLine("serial... "+CalcSerial());
    }
    public static double CalcSerial()
    {
        Stopwatch stopper = new Stopwatch();
        stopper.Start();
        foreach (Account account in accounts)
        {
            Console.WriteLine("serially processing {0}",
                               account.CalcCreditPoints());
        }
        stopper.Stop();
        return stopper.ElapsedMilliseconds;
    }
}
```

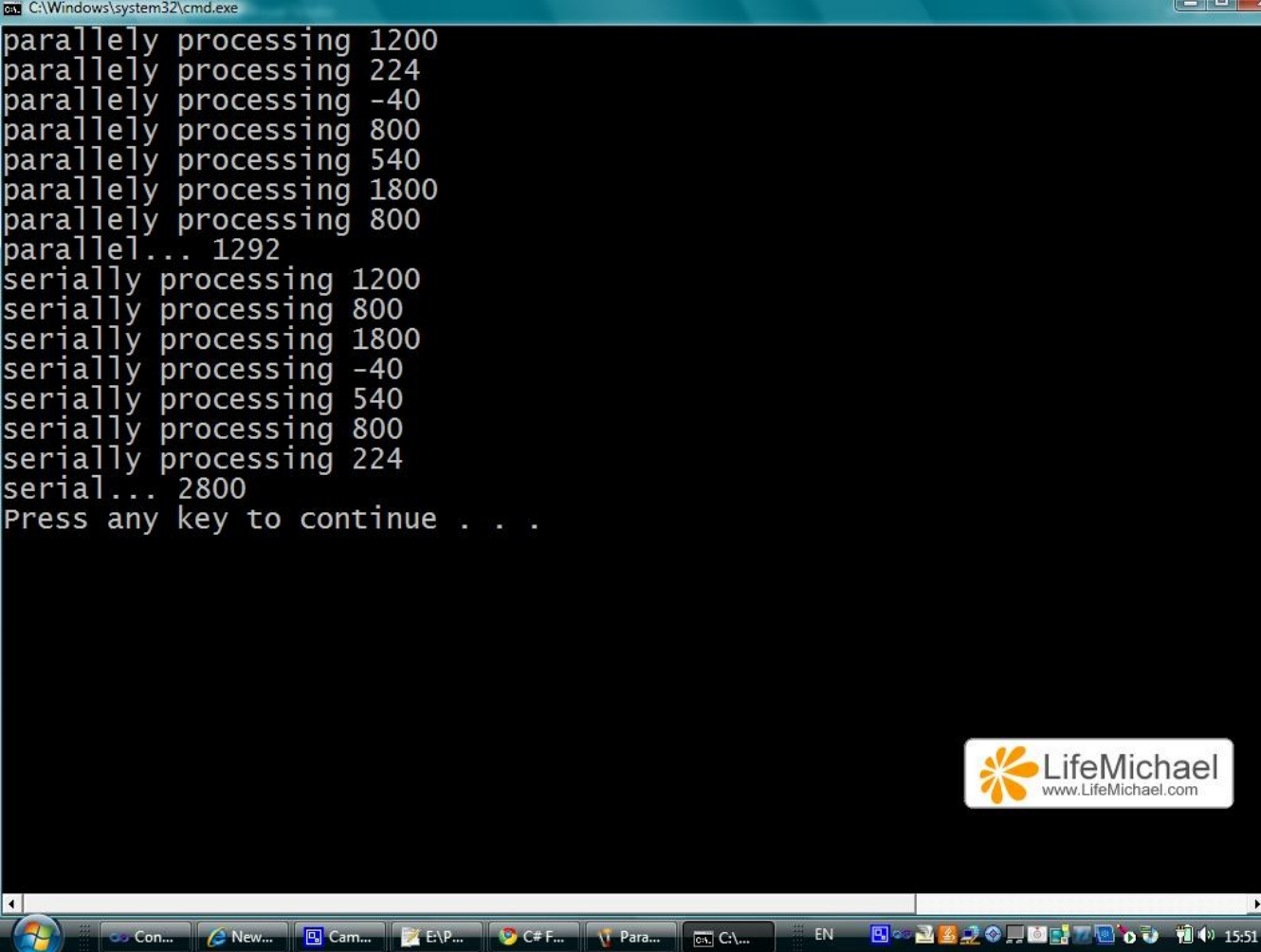
The Parallel.ForEach Loop

```
public static double CalcParallel()
{
    Stopwatch stopper = new Stopwatch();
    stopper.Start();
    Parallel.ForEach(accounts, account =>
    {
        Console.WriteLine("parallely processing {0}",
            account.CalcCreditPoints());
    });
    stopper.Stop();
    return stopper.ElapsedMilliseconds;
}
```

The Parallel.ForEach Loop

```
public static void CreateAccounts()
{
    double[] sums = { 12000, 8000, 18000,
                      -400, 5400, 8000, 2240 };
    foreach (double sum in sums)
    {
        accounts.Add(new Account(sum));
    }
}
```

The Parallel.ForEach Loop



```
C:\Windows\system32\cmd.exe
parallelly processing 1200
parallelly processing 224
parallelly processing -40
parallelly processing 800
parallelly processing 540
parallelly processing 1800
parallelly processing 800
parallel... 1292
serially processing 1200
serially processing 800
serially processing 1800
serially processing -40
serially processing 540
serially processing 800
serially processing 224
serial... 2800
Press any key to continue . . .
```

Parallelism & Performance

- It isn't always faster to work in a parallel way. The overhead involved in partitioning the data and the cost of invoking a delegate on each loop iteration doesn't necessarily pay in getting a better performance.

The Parallel.Invoke() Method

- Calling this method we can pass over those methods we want to execute in a parallel way.
- The implementation of this method uses tasks.

The Parallel.Invoke() Method

```
namespace ConsoleApplication1
{
    public class Account
    {
        public double Balance { get; set; }
        public Account(double sum)
        {
            Balance = sum;
        }
        public int CalcCreditPoints()
        {
            Thread.Sleep(400);
            return (int)Balance/10;
        }
        public double CalcUSBalance(double currency)
        {
            Thread.Sleep(400);
            return currency * Balance;
        }
    }
}
```

The Parallel.Invoke() Method

```
public class Program
{
    public static List<Account> accounts = new List<Account>();
    public static void Main(string[] args)
    {
        CreateAccounts();
        CalcParallel();
        CalcSerial();
    }
}
```

The Parallel.Invoke() Method

```
public static void CalcSerial()  
{  
    Stopwatch stopper = new Stopwatch();  
    stopper.Start();  
    double accountCreditPointsAverage = 0;  
    double accountBalanceAverage = 0;  
    foreach (Account account in accounts)  
    {  
        AccountCreditPointsAverage +=  
            account.CalcCreditPoints();  
    }  
    foreach (Account account in accounts)  
    {  
        accountBalanceAverage += account.CalcUSBalance(4.3);  
    }  
    int num = accounts.Count;  
    accountBalanceAverage = accountBalanceAverage / num;  
    accountCreditPointsAverage = accountCreditPointsAverage / num;  
    Console.WriteLine("serial... average balance is " +  
        accountBalanceAverage);  
    Console.WriteLine("serial... average credit points is " +  
        accountCreditPointsAverage);  
    stopper.Stop();  
    Console.WriteLine("serial... " + stopper.ElapsedMilliseconds);  
}
```

The Parallel.Invoke() Method

```
public static void CalcParallel()  
{  
    Stopwatch stopper = new Stopwatch();  
    stopper.Start();  
    double accountCreditPointsAverage = 0;  
    double accountBalanceAverage = 0;  
    Parallel.Invoke(() =>  
        {  
            foreach (Account account in accounts)  
            {  
                AccountCreditPointsAverage +=  
                    account.CalcCreditPoints();  
            }  
        },
```

The Parallel.Invoke() Method

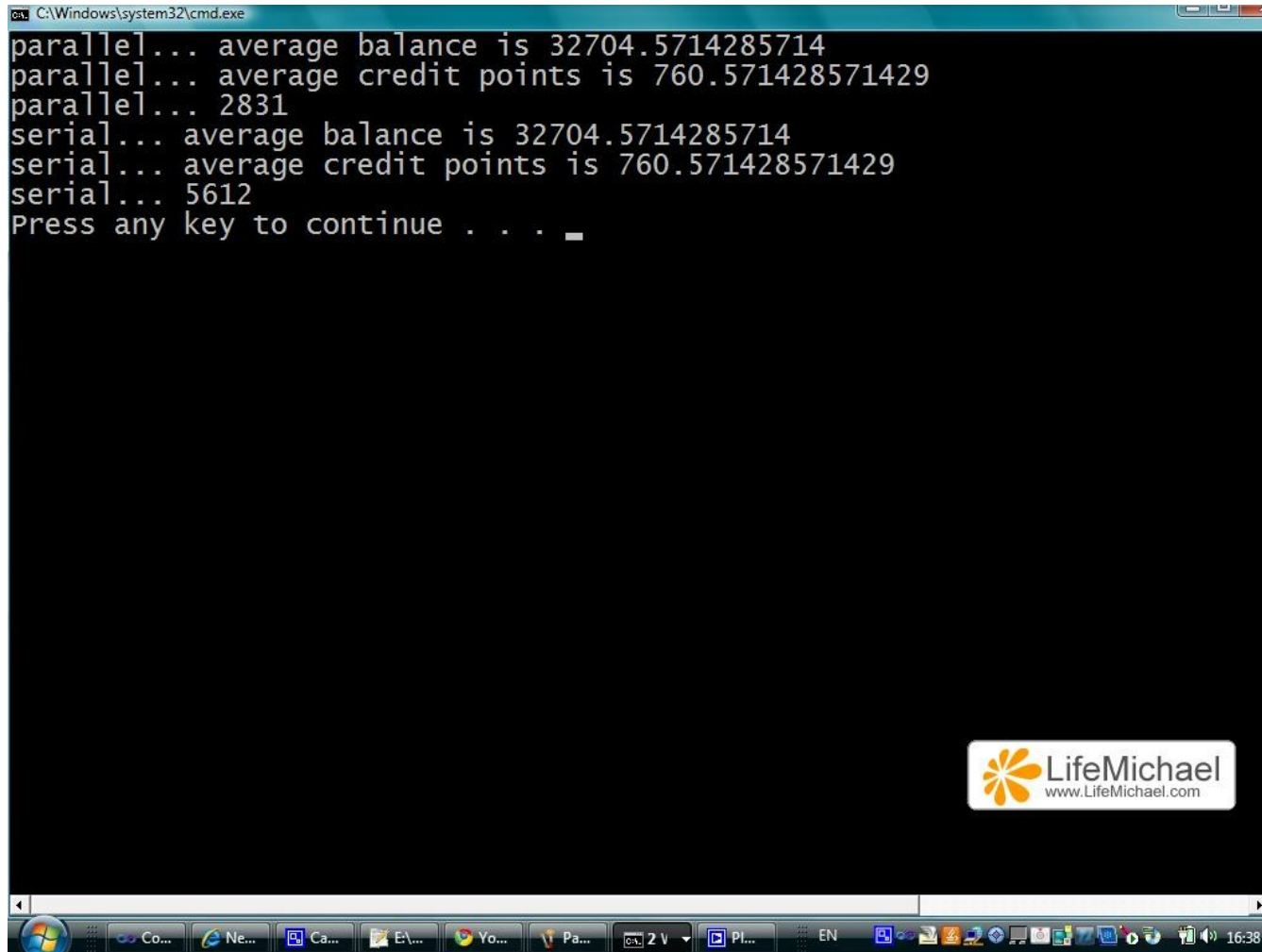
```
        () =>
        {
            foreach (Account account in accounts)
            {
                AccountBalanceAverage +=
                    account.CalcUSBalance(4.3);
            }
        });

int num = accounts.Count;
accountBalanceAverage = accountBalanceAverage / num;
accountCreditPointsAverage = accountCreditPointsAverage / num;
Console.WriteLine("parallel... average balance is " +
    accountBalanceAverage);
Console.WriteLine("parallel... average credit points is " +
    accountCreditPointsAverage);
stopper.Stop();
Console.WriteLine("parallel... " + stopper.ElapsedMilliseconds);
}
```

The Parallel.Invoke() Method

```
public static void CreateAccounts()  
{  
    double[] sums = { 12000, 8000, 18000, -400, 5400, 8000, 2240 };  
    foreach (double sum in sums)  
    {  
        accounts.Add(new Account(sum));  
    }  
}  
}
```

The Parallel.Invoke() Method



```
C:\Windows\system32\cmd.exe
parallel... average balance is 32704.5714285714
parallel... average credit points is 760.571428571429
parallel... 2831
serial... average balance is 32704.5714285714
serial... average credit points is 760.571428571429
serial... 5612
Press any key to continue . . . _
```

LifeMichael
www.LifeMichael.com

Tasks Overview

- The Task class represents the work we want completed.
- Various methods allow us to create, schedule and synchronize tasks in our code.

The Tasks Scheduler

- The tasks scheduler handles all of the complexity involved with running tasks. It is kind of a wrapper for the threading pool.
- When we create new tasks they are added to the global tasks queue.

The Tasks Scheduler

- The .NET threading pool creates a number of threads in accordance with the number of tasks that were added to the global tasks queue.
- Each working thread picks tasks from the global queue and moves them into its local queue.
- Each working thread processes the tasks on its own queue.
- If a thread finishes the tasks on its own local queue it helps other threads to complete their tasks.

The Tasks Scheduler

- Threads assisting others to complete their tasks will do so by stealing tasks from the end of their local queues. This way, the chance for a collision between the threads is minimized.

Creating Tasks

- The simplest way to create a new task involves with calling the `Task.Factory.StartNew()` method. This method receives an `Action` delegate and immediately starts it.

...

```
Task t = Task.Factory.StartNew(()=>Console.WriteLine("hello"));
```

...

Creating Tasks

- We can alternatively pass over the Action delegate to the Task constructor and later when calling `Start()` have that task executed.

...

```
Task t = new Task(()=>Console.WriteLine("hello"));
```

```
t.Start();
```

...

The `Wait()` and `WaitAll()` Methods

- Calling these methods causes the current thread to wait till the specified task (or tasks) are finished.

The Wait () and WaitAll () Methods

```
namespace ConsoleApplication1
{
    public class Account
    {
        public double Balance { get; set; }
        public Account(double sum)
        {
            Balance = sum;
        }
        public int CalcCreditPoints()
        {
            Thread.Sleep(400);
            return (int)Balance/10;
        }
        public double CalcUSBalance(double currency)
        {
            Thread.Sleep(400);
            return currency * Balance;
        }
    }
}
```

The Wait () and WaitAll () Methods

```
public class Program
{
    public static List<Account> accounts = new List<Account>();
    public static void Main(string[] args)
    {
        Task creatingAccountsTask = new Task(CreateAccounts);
        Task printTotalTask = new Task(PrintAccountsTotal);
        creatingAccountsTask.Start();
        creatingAccountsTask.Wait();
        printTotalTask.Start();
        Console.ReadLine();
    }
    public static void PrintAccountsTotal()
    {
        double total = 0;
        foreach (Account account in accounts)
        {
            total += account.Balance;
        }
        Console.WriteLine(total);
    }
}
```


The Wait () and WaitAll () Methods

```
public static void CreateAccounts()
{
    double[] sums = { 12000, 8000, 18000, -400,
                      5400, 8000, 2240 };
    foreach (double sum in sums)
    {
        accounts.Add(new Account(sum));
        Thread.Sleep(200);
    }
}
```

The `WaitAny()` Method

- Calling this static method will cause the current thread to wait till one of the specified tasks finishes.

...

```
Task.WaitAny({taskA, taskB, taskC});
```

...

The Task.IsCompleted Property

- Calling this specific property we can know whether a specific task is completed or not.

```
...  
while(!taskA.IsCompleted)  
{  
    Console.WriteLine("taskA is still running...");  
}  
...
```

The ContinueWith () Method

- Calling this method we can specify the order in which we want our tasks to be completed.

...

```
taskA.ContinueWith(taskB);
```

```
taskB.ContinueWith(taskC);
```

...

The ContinueWith () Method

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;

namespace ConsoleApplication1
{
    public class Account
    {
        public double Balance { get; set; }
        public Account(double sum)
        {
            Balance = sum;
        }
        public int CalcCreditPoints()
        {
            Thread.Sleep(400);
            return (int)Balance/10;
        }
    }
}
```

The ContinueWith () Method

```
public double CalcUSBalance(double currency)
{
    Thread.Sleep(400);
    return currency * Balance;
}

public class Program
{
    public static List<Account> accounts = new List<Account>();
    public static void Main(string[] args)
    {
        Task creatingAccountsTask = new Task(CreateAccounts);
        Task printTotalTask =
            creatingAccountsTask.ContinueWith(
                (t)=>PrintAccountsTotal());
        creatingAccountsTask.Start();
        Console.ReadLine();
    }
}
```

The ContinueWith () Method

```
public static void PrintAccountsTotal()
{
    double total = 0;
    foreach (Account account in accounts)
    {
        total += account.Balance;
    }
    Console.WriteLine(total);
}
public static void CreateAccounts()
{
    double[] sums = {12000,8000,18000,-400,5400,8000,2240 };
    foreach (double sum in sums)
    {
        accounts.Add(new Account(sum));
        Thread.Sleep(200);
    }
}
}
```

The Result Property

- We can get the result returned from executing a task by referring the `Result` property.

...

```
int num = task.Result;
```

...

- If the task still hasn't completed the execution will block and wait.

The Result Property

```
namespace ConsoleApplication1
{
    public class Account
    {
        public double Balance { get; set; }
        public Account(double sum)
        {
            Balance = sum;
        }
        public int CalcCreditPoints()
        {
            Thread.Sleep(400);
            return (int)Balance/10;
        }
        public double CalcUSBalance(double currency)
        {
            Thread.Sleep(400);
            return currency * Balance;
        }
    }
}
```

The Result Property

```
public class Program
{
    public static List<Account> accounts = new List<Account>();
    public static void Main(string[] args)
    {
        Task creatingAccountsTask = new Task(CreateAccounts);
        Task<double> calculateTotalTask =
            new Task<double>(CalculateAccountsTotal);
        creatingAccountsTask.Start();
        creatingAccountsTask.Wait();
        calculateTotalTask.Start();
        double result = calculateTotalTask.Result;
        Console.WriteLine("result=" + result);
        Console.ReadLine();
    }
    public static double CalculateAccountsTotal()
    {
        double total = 0;
        foreach (Account account in accounts)
        {
            Thread.Sleep(500);
            total += account.Balance;
        }
        return total;
    }
}
```

The Result Property

```
public static void CreateAccounts()  
{  
    double[] sums = {12000,8000,18000,-400,5400,8000,2240};  
    foreach (double sum in sums)  
    {  
        accounts.Add(new Account(sum));  
        Thread.Sleep(200);  
    }  
}  
}
```

The Barrier Class

- We will use this class when having a set of (two or more) threads that a predefined number of them first needs to complete or reach a specific point in their execution before moving forward.

The Barrier Class

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;

namespace ConsoleApplication1
{
    public class Account
    {
        public double Balance { get; set; }
        public Account(double sum)
        {
            Balance = sum;
        }
        public int CalcCreditPoints()
        {
            Thread.Sleep(400);
            return (int)Balance/10;
        }
    }
}
```

The Barrier Class

```
public double CalcUSBalance(double currency)
{
    Thread.Sleep(400);
    return currency * Balance;
}
}
public class Program
{
    public static List<Account> accounts = new List<Account>();
    public static Barrier myBarrier = new Barrier(3);
    public static double balanceTotal = 0;
    public static double creditPointsTodal = 0;
    public static void Main(string[] args)
    {
        CreateAccounts();
        new Thread(CalculateAccountsTotal).Start();
        new Thread(CalculateTotalCreditPoints).Start();
        myBarrier.SignalAndWait();
        Console.WriteLine("total balance = " + balanceTotal);
        Console.WriteLine("total credit points = " +
            creditPointsTodal);
        Console.ReadLine();
    }
}
```

The Barrier Class

```
public static void CalculateAccountsTotal()
{
    double total = 0;
    foreach (Account account in accounts)
    {
        Thread.Sleep(500);
        total += account.Balance;
    }
    balanceTotal = total;
    myBarrier.SignalAndWait();
}

public static void CalculateTotalCreditPoints()
{
    double total = 0;
    foreach (Account account in accounts)
    {
        Thread.Sleep(500);
        total += account.CalcCreditPoints();
    }
    creditPointsTotal = total;
    myBarrier.SignalAndWait();
}
```

The Barrier Class

```
public static void CreateAccounts()  
{  
    double[] sums = {12000,8000,18000,-400,5400,8000,2240};  
    foreach (double sum in sums)  
    {  
        accounts.Add(new Account(sum));  
    }  
}  
}
```


The CountdownEvent Class

- When instantiating this class we set an integer value. Each thread that blocks on the object we got will resume its execution when the total number of paused threads reached the predefined integer value.
- Each time the `Signal()` method is called the value of the inner counter decreases by one. When that counter reaches 0 all paused threads resume their execution.

The CountdownEvent Class

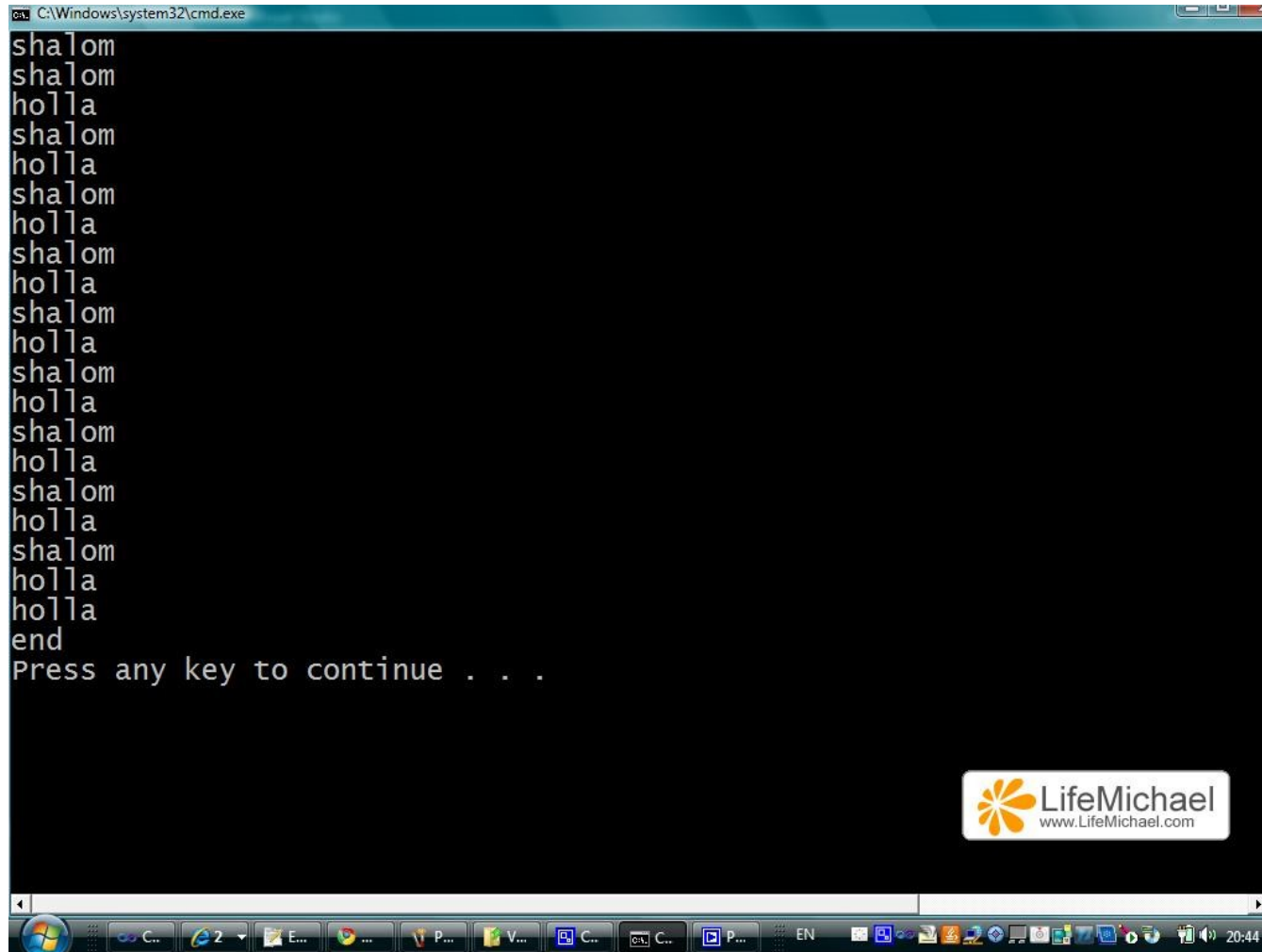
```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;

namespace ConsoleApplication1
{
    public class Program
    {
        private static CountdownEvent counter = new CountdownEvent(2);
        public static void Main(string[] args)
        {
            new Thread(Say).Start("shalom");
            new Thread(Say).Start("holla");
            counter.Wait();
            Console.WriteLine("end");
        }
    }
}
```

The CountdownEvent Class

```
public static void Say(object str)
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(str);
        Thread.Sleep(200);
    }
    counter.Signal();
}
}
```

The CountdownEvent Class



```
C:\Windows\system32\cmd.exe
shalom
shalom
holla
shalom
holla
shalom
holla
shalom
holla
shalom
holla
shalom
holla
shalom
holla
shalom
holla
shalom
holla
holla
end
Press any key to continue . . .
```

LifeMichael
www.LifeMichael.com